Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors

by

Patrick Gregory Sobalvarro

S.B., Massachusetts Institute of Technology (1988)
S.M., Massachusetts Institute of Technology (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1997

Author . . . .
Department of Electrical Engineering and Computer Science
January 10, 1997

Certified by . . . .
William E. Weihl
Associate Professor
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors

by

Patrick Gregory Sobalvarro

Submitted to the Department of Electrical Engineering and Computer Science
on January 10, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis describes *demand-based coscheduling*, a new approach to scheduling parallel computations on multiprogrammed multiprocessors. In demand-based coscheduling, rather than making the pessimistic assumption that all the processes constituting a parallel job must be simultaneously scheduled in order to achieve good performance, information about which processes are communicating is used in order to coschedule only these; the resulting scheme is well-suited to implementation on a workstation cluster because it is naturally decentralized. I present an analytical model and simulations of demand-based coscheduling, an implementation on a cluster of workstations connected by a high-speed network, and a set of experimental results. An analysis of the results shows that demand-based coscheduling successfully coschedules parallel processes in a timeshared workstation cluster, significantly reducing the response times of parallel computations.

Thesis Supervisor: William E. Weihl
Title: Associate Professor
Department of Electrical Engineering and Computer Science

Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors

by

Patrick Gregory Sobalvarro

Submitted to the Department of Electrical Engineering and Computer Science
on January 10, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis describes *demand-based coscheduling*, a new approach to scheduling parallel computations on multiprogrammed multiprocessors. In demand-based coscheduling, rather than making the pessimistic assumption that all the processes constituting a parallel job must be simultaneously scheduled in order to achieve good performance, information about which processes are communicating is used in order to coschedule only these; the resulting scheme is well-suited to implementation on a workstation cluster because it is naturally decentralized. I present an analytical model and simulations of demand-based coscheduling, an implementation on a cluster of workstations connected by a high-speed network, and a set of experimental results. An analysis of the results shows that demand-based coscheduling successfully coschedules parallel processes in a timeshared workstation cluster, significantly reducing the response times of parallel computations.

Thesis Supervisor: William E. Weihl
Title: Associate Professor
Department of Electrical Engineering and Computer Science

For Marie

Acknowledgments[1]

Sometimes when looking through a thesis I find that it contains a very long acknowledgments section, which more often than not proves upon examination to be a very personal document. When the writer is a friend, I enjoy reading the acknowledgments, especially if my name is mentioned. When the writer is someone I know only professionally, my reaction is typically embarassment tinged with distaste. I am reminded of a Peanuts cartoon I read when I was a child, in which Snoopy saw a dog riding in a car with its head out the window and its tongue flapping in the breeze. Snoopy thought that if he were in a car he would never behave in so undignified a manner; rather, he would sit up straight and wear a seat belt.

It will be clear, then, that in order to maintain a dignified posture and spare the reader embarassment, I should like to write a brief and formal acknowledgments section, but when I look back on the work that has occupied me for most of three years, I realize that that I am so indebted to colleagues, family, and friends, that to do so would be out of the question. Thus to the reader who does not know me, and is now preparing himself or herself for a distasteful experience, I suggest by way of fortification a stiff drink, taken perhaps with a pickled walnut for the good of the liver.

Among my colleagues I owe the greatest debt to my thesis advisor Bill Weihl, who was a teacher, co-worker, mentor, and friend to me throughout this project. The original idea behind dynamic coscheduling was his, and as I've worked to develop it, he has been a constant source of good ideas and constructive criticism. I've had the good fortune to work with Bill on other projects as well, and I've benefited from and been inspired by his technical acuity, his clear thinking and his generous spirit.

In the implementation and measurement stages of the project, I had two other collaborators as well: Scott Pakin and Andrew Chien of the University of Illinois at Urbana-Champaign.

The implementation of dynamic coscheduling on Illinois Fast Messages was achieved partly through Scott's hard work on the message processor code and the FM messaging layer. Scott not only implemented the FM support for FM-DCS, but he ran our earlier experiments, putting in many hours writing scripts, supervising runs, and compiling the data. Scott and I spent a great deal of time "in the trenches" together, debugging, measuring, and figuring, and I very much enjoyed working with him. His unfailing sense of humor always put frustrating phases of the project into a more sensible perspective.

Andrew Chien provided the impetus for the FM-DCS implementation, suggesting it at IPPS in 1995. He worked hard to make it possible, organizing its computational resources and inviting me to UIUC where I worked with him and Scott; but also he provided good technical insights into the implementation and critical analysis during the measurement stage of the project. His technical perceptiveness was very valuable

[1]This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985, by grants from IBM and AT&T, and by an equipment grant from DEC.

and his advice on all aspects of the project was invariably measured and judicious.

My thesis readers were Bert Halstead and Steve Ward. Bert Halstead has been a mentor and friend to me for many years now, starting when he advised my undergraduate thesis on the Lucid Ephemeral Garbage Collector, and continuing into an internship in which I worked with him at Digital's Cambridge Research Laboratory. It was indeed fortunate for me that Bert agreed early on to be a reader of my thesis, and both early and late technical discussions with him were illuminating and fruitful. In addition to his technical acumen and insight Bert is simply a delightful person to work with and be with.

I know Steve Ward mostly as a sharp thinker who asks hard questions, as when he chaired my oral qualifying examination. He also asked hard questions during my thesis defense. The two cluster-months' worth of experimental data presented here are due to Steve's wanting to know which artifacts in my earlier, less exhaustive data were noise and which were systematic; I believe the thesis was strengthened by other hard questions he asked as well.

My thesis work benefited from technical discussions with many other people besides my immediate co-workers and thesis committee. Standing out among these individuals is Larry Rudolph of the Hebrew University of Jerusalem. Larry Rudolph came to visit M.I.T. the year after Bill and I began this project, and brought with him a wealth of knowledge born from work he had done with Dror Feitelson. In particular, Larry and Dror's work on runtime activity working set identification was a close cousin of our work on demand-based coscheduling, and I benefited from many technical discussions with him on scheduling and other aspects of parallel computation.

Other people who come to mind when I think of technical discussions about aspects of this work are Butler Lampson, Bobby Blumofe, Chandu Thekkath, Ed Lee, Frans Kaashoek, Charles Leiserson, Kavita Bala, Shel Kaphan, Chris Joerg, Carl Waldspurger, Mike Burrows, Sanjay Ghemawat, Volker Strumpen, and Deborah Wallach.

Looking back over my thesis work is a mixed experience. One sees that there are real results; but one also sees many shortcomings. Here I must admit that the usual formula applies in my case as well: what is valuable in this work owes a great deal to my colleagues; the shortcomings are my own.

Not being a libertarian, I believe that society is real and not a fiction. Similarly I believe in the reality of institutions as more than collections of individuals: I believe that the best institutions accomplish what they do also because of the knowledge, morals, standards and traditions they cultivate over time. I owe a debt of gratitude to one institution in particular, the Massachusetts Institute of Technology, which was several things to me: a school, a community, and a place of work. I first came to the Institute as an undergraduate student who was very grateful for a generous scholarship; I worked as a member of the research staff at the Artificial Intelligence Laboratory for a number of years, and eventually after working in California for several years I chose to do my graduate work at M.I.T. Like any institution, M.I.T. has its flaws. A friend said to me many years ago that in M.I.T. I had found the one

place in the world where nothing I did would ever be sufficient, and there is some truth to that: the Institute is generous with opportunity but niggardly with praise. Still, I don't think I overstate the case when I say that M.I.T.'s culture also exemplifies much of what is right and good in engineering and science: a lively curiosity coupled with an aesthetic of cynicism and iconoclasm; a sense of humor; and an occasionally fatal desire to Implement The Right Thing.

Over the years, a number of people at M.I.T. besides those I have already mentioned have impressed me as exemplifying the positive aspects of that culture. Some of these in particular are Rod Brooks, who I worked with when I was a staff member at the Artificial Intelligence Laboratory and again at Lucid, Inc.; W. Eric L. Grimson, who was my academic counselor; Tom Knight, who was briefly my undergraduate advisor and eventually my master's thesis advisor; and Hal Abelson, for whom I worked on the Logo project when I was just a little hacker.

Administrative personnel at M.I.T. are also exceptional, and I would be remiss if I did not mention two of them who went out of their way to be helpful to me on many occasions. Irena Kveraga, the administrative support person for my research group, was unfailingly pleasant and cheerful while arranging my many trips to UIUC and to conferences, some of which had to be booked in considerable haste. I am grateful also to Marilyn Pierce, who runs the EECS Graduate Office, and who bailed me out on the many occasions when I registered late, or when I was across the country and needed to submit some essential document in time for a deadline. 'But Marilyn does that for everyone,' some of my fellow graduate students might say. I agree; and I think everyone should express gratitude for it.

I am also indebted to the Digital Systems Research Center, to its former director Bob Taylor, its current director Roy Levin, and again to Bill Weihl, for SRC's support of me during these six months of finishing the writing of my thesis.

A doctoral dissertation is often the sort of work that exacts a toll on family members and friends as well as on the writer, and my dissertation was no exception. My friends who have listened patiently over the years when I spoke of the long project of my doctorate are too numerous to list, but my closest friends Leigh Klotz, Gregor Kiczales, and Laurel Simmons have all heard quite a bit about it. Visits with my brothers Jim and Eric and my sister Laurie have often been put off while I struggled with one bit of work or another. My good friend Kavita Bala and my officemates Bobby Blumofe, Jan-Willem Maessen, and Alex Caro deserve special mention for their patience because I complained more loudly to them than to anyone else, and to add insult to injury, they were suffering through the same process themselves!

But I have saved the best for last: most of all I am grateful to my wife, Marie Crowley Sobalvarro, for her many years of love, support, patience, good sense and companionship while I was engaged in the long undertaking of my graduate work. This dissertation is dedicated to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Demand-based Coscheduling

This thesis describes *demand-based coscheduling,* a new approach to scheduling parallel computations on timeshared multiprogrammed multiprocessors. The approach is a simple one: under demand-based coscheduling, processes are scheduled simultaneously only if they communicate; communication is treated as a demand for coordinated scheduling. The key idea behind demand-based coscheduling is also simple: processes that do not communicate or communicate only rarely need not be coscheduled for efficient execution.

The main effect of uncoordinated scheduling and fine-grain communication under timesharing is increased latency. Without coscheduling, a process performing request-reply communication with a descheduled process will not receive a reply until the descheduled process is scheduled. We found in our experiments (see Chapter 5) that under conditions of heavy load, the effects on time to completion of the job can be significant. If the operating system performs some sort of priority boosting that serves to schedule a process when the message eventually arrives, then it effectively implements a limited form of demand-based coscheduling; latency on the experiments we performed is typically twenty to forty percent greater than with DCS. If such a mechanism is not used, as when polling or spinning message receipt is used or the priority boosting mechanism is disabled, latency can be a hundred times greater than the latency with a full implementation of demand-based coscheduling.

In order to significantly reduce the number of additional context switches due to failed attempts to synchronize communicating processes, demand-based coscheduling requires a locality property of the processes it manages. The property is this: processes that have communicated recently will communicate again within a timeslice. Here an analogy may be drawn with demand paging. Just as efficient use of demand-based coscheduling requires that communications be clustered into times when the process is scheduled, under demand paging, in order to reduce the number of page faults suffered by a process referencing a page, the page must be referenced more than once before being ejected from main memory.

We say that demand-based coscheduling is a *dynamic* approach to scheduling, not because it differs from static schedules of the sort that might be drawn for multithreaded computations running on dedicated hardware — such static approaches are not useful for general interactive applications and so we do not consider them further — but because it coschedules processes that are currently communicating or have done so recently and thus are expected to do so again in the near future. Because demand-based coscheduling uses more information than does Ousterhout's form of coscheduling [18], it can reduce the difficulty of the scheduling problem and exploit opportunities for coscheduling that traditional coscheduling cannot. Because it does not rely on a particular programming technique, such as task-queue-based multithreading, demand-based coscheduling is applicable in domains where process control [24] is not.

Demand-based coscheduling is intended for scheduling timeshared loads of parallel jobs or mixed loads of parallel and serial jobs. While our implementation runs only on workstation clusters, we expect forms of demand-based coscheduling to be useful on a variety of platforms: large message-passing multiprocessors, message-passing- and shared-memory-based departmental servers, desktop shared-memory multiprocessors with a low degree of parallelism, as well as workstation clusters. We concentrate in this thesis on one kind of demand-based coscheduling, *dynamic* coscheduling, which is intended for message-passing processors. However, we describe in Appendix A a scheme that might be used to implement demand-based coscheduling on a shared-memory processor.

1.2 The Problem of Timesharing Multiprocessors

To date, parallel computers have been used mostly for the solution of scientific and engineering problems, as dedicated platforms for transaction processing, and as testbeds for research in parallel computation.

In these problem domains, the problem of scheduling parallel jobs is simplified. Batch scheduling may be appropriate if the problems are large and the I/O and synchronization blocking rates are low. If the I/O and synchronization blocking rates are low and the multiprocessor has a larger number of nodes than are demanded by the problems, simple space partitioning and a batch queue may be the best choice. Such job-scheduling policies are particularly appropriate for very expensive computers, where economics will dictate careful planning of the job load, and users should be encouraged to perform their debugging off-line, under emulation if possible.

1.2.1 Problems with batch processing and space partitioning

However, as multiprocessors continue to follow the course set by uniprocessors over the past forty years, they have begun to move out of laboratories and computing centers and into offices. Already multiprocessors with relatively small numbers of nodes (4 – 16) have become popular as departmental servers, and we have begun to see desktop machines with two and four nodes. In business environments, these machines do not typically run explicitly parallel jobs, although we can expect that explicitly parallel computation-intensive jobs will appear once the platforms have become sufficiently popular. Instead, they are purchased because the typical departmental server and many desktop machines run large numbers of processes, and some of the processes are sufficiently compute-intensive that sharing the memory, disk and display resources of the machine is the most economical solution.

In these office environments, with mixed loads of client/server jobs, serial jobs, and (in the future) parallel jobs, the scheduling problem becomes more complex. Batch scheduling is inappropriate, because response times must be low. Simple space-partitioning will not be sufficient in such an environment, because the number of processes will be high compared to the number of processors. Furthermore, it can be difficult to know in advance how many processes a job will require or which processes will communicate with other processes — in an interactive environment, these might depend on user input.

1.2.2 Independent timesharing results in poor performance

Crovella *et al.* have presented results in [5] that show that independent timesharing without regard for synchronization produced significantly greater slowdowns than coscheduling, in some cases a factor of two worse in total runtime of applications.[1] Chandra *et al.* have reported similar results in [4]: in some cases independent timesharing is as much as 40% slower than coscheduling. In [7], Feitelson and Rudolph compared the performance of gang scheduling using busy-waiting synchronization to that of independent (uncoordinated) timesharing using blocking synchronization. They found that for applications with fine-grain synchronization, performance could degrade severely under uncoordinated timesharing as compared to gang scheduling. In an example where processes synchronized about every 160μsec on a NUMA multiprocessor with 4-MIPS processing nodes, applications took roughly twice as long to execute under uncoordinated scheduling as they did under gang scheduling.

In general, the results cited above agree with the claims advanced by Ousterhout in [18]: under independent timesharing, multiprogrammed parallel job loads will suffer large numbers of context switches, with attendant overhead due to cache and TLB reloads. The extra context switches result from attempts to synchronize with descheduled processes resulting in blocking. As Gupta *et al.* have shown in [10], the use of non-blocking (spinning) synchronization primitives will result in even worse performance under moderate multiprogrammed loads, because, while the extra context switches are avoided, the spinning time is large.

Although the literature to date has described experiments with relatively small numbers of jobs timesharing a multiprocessor, we may expect (and know, from experience) that departmental servers in practice will be heavily loaded for some portion of their lifetime. The reason is a simple economic one: a system that is not heavily loaded is not fully utilized; an underutilized system is a waste of resources. We

[1] Crovella *et al.* found that hardware partitions gave the best performance in their experiments, but, as we have discussed above, these are not feasible when one has a large number of jobs to run on a small number of processors.

may expect that more heavily loaded systems will suffer even higher synchronization blocking rates under independent timesharing, and commensurately higher context switching overhead.

1.3 Goals

Ousterhout compared parallel scheduling and virtual memory systems in [18]. He suggested that coscheduling is necessary on timeshared multiprocessors running parallel jobs in order to avoid a kind of process thrashing that is analogous to virtual memory thrashing. This kind of process thrashing arises because each process can run for only a short period before blocking on an attempt to synchronize with another process that is not currently scheduled; the result is greatly increased numbers of context switches.

Later work has shown that the context switches themselves can be expensive due to the cache reloads they entail; also the overhead of spinning while awaiting a message can consume a large amount of CPU time [23]. In the present work, we shall see that in an environment with serial jobs running in competition with a parallel job, a parallel job using fine-grain communication can suffer greatly increased time to completion unless some means of coscheduling is used — at the very least, the operating system must perform some priority-boosting to increase the probability that processes will be scheduled when they have blocked awaiting message arrival. In our own experiments, we have seen this increased time to completion of the job show up as CPU time spent spinning, in the case of spinning message receipt; and as time spent blocked awaiting message arrival, in the case of spin-block message receipt.

If coscheduling is necessary, how is it to be provided? To motivate our approach, we return to Ousterhout's analogy of parallel scheduling to virtual memory, but rather than building a mechanism that resembles swapping, as traditional coscheduling does, we seek to produce a mechanism that resembles demand paging. To take the analogy somewhat further, our goals are to produce a scheduler that is *non-intrusive* in the same way that demand paging is non-intrusive: we do not want to impose on the programmer a particular programming model. For example, while demand-based coscheduling could be compatible with a task-queue-based multithreaded approach like process control [24], we do not want to *require* that all parallel applications be coded in a multithreaded fashion in order not to suffer excessive context-switching.

Again as with demand paging, we want an approach that is *flexible*: we wish to free the programmer and the compiler writer from consideration of exactly how many processors are present on the target machine, in the same way that demand paging frees the programmer and the compiler writer from considering exactly how much physical memory is present on the target machine. This is as distinct from traditional coscheduling [18], in which there is no clear means for scheduling jobs with more processes than there are nodes on the multiprocessor.

Finally, we want an approach that is *dynamic*, and can adapt to changing conditions of load and communication between processes. For example, we expect that client/server applications will be particularly important on multiprocessor systems.

In such applications, it may not be known ahead of time which client processes will communicate with which servers, but if the rate of communication is sufficiently high, coscheduling will be important. Examples might include SQL front ends communicating with a parallel database engine, or window system clients and servers, or different modules in a microkernel operating system running on a multiprocessor.

1.4 Demand-based Coscheduling

Demand-based coscheduling should meet the goals described above: it is by its nature non-intrusive, as it accomplishes coscheduling simply by monitoring communication. The programmer should not even need to identify the processes that constitute a parallel job. We expect demand-based coscheduling implementations to be flexible, in that they only need to coschedule communicating processes, and thus if communication is not all-to-all, they simplify the scheduling problem, possibly allowing a job with more processes than there are nodes on the machine to run efficiently. And finally, we expect demand-based coscheduling to quickly adapt to changing loads: because the algorithms we will describe have very little state, if they work at all for a set of loads, they should adapt quickly to changes between them.

In following chapters, we will often draw an analogy between demand-based coscheduling and demand paging. In this analogy, if process A makes a request of process B, we may think of process B as a page that process A accesses. If process B is currently scheduled, it is as though the page were resident in main memory; but if process B is descheduled, it is as though the page were on backing store.

The analogy is not exact, because in modern systems the time to schedule a descheduled process is very small, at most hundreds of microseconds even when cache reloads are taken into account, by comparison with about ten milliseconds for fetching a page from backing store. What is more, in demand-based coscheduling, because the intention is to coordinate scheduling, we do not immediately block the process sending the message, whereas in demand paging it would be desirable to do so in order to utilize the CPU during the very long disk access time.

Inexact though it may be, however, the analogy preserves much of the intuition behind demand-based coscheduling: it is a low-level mechanism that attempts to reduce the expense of an operation by exploiting locality of reference.

In what follows, we will concentrate on a particular form of demand-based coscheduling called dynamic coscheduling. Our analytical model and simulations will show that under some circumstances dynamic coscheduling can cause strong coscheduling behavior in a message-passing processor. They will also point up potential problems with the simplest form of this idea and allow us to explore means of surmounting these problems. Our implementation and experiments clarify some of the issues underlying communication and scheduling, and show that dynamic coscheduling can significantly improve performance for a parallel application running on a timeshared workstation cluster.

1.5 Terminology

We use the word *job* to describe a distinct application running on a computer. The application may be a single serial process that does not communicate with any other control thread but the kernel; or it may be a multithreaded application consisting of separate processes sharing a single address space; it could be a single-program, multiple-data application communicating with message-passing; or it could even be a client/server application consisting of one or more server processes and one or more client processes communicating with each other. The important point is that a job is a logically distinct application consisting of one or more processes that communicate.

We use the word *process* to mean the state of a serially executed program with an address space (possibly wholly or partly shared with other processes) and a process control block. A process may be executed on one processing node of a multiprocessor at a time.

In order to hide the latency of certain operations or to allow more clear expression of natural parallelism, a process may have one or more *threads* of control; these have separate stacks and register states, but share the address space of the process and some parts of its process control block. Depending on the threads implementation, they may be dispatched by the kernel or by a user scheduler, or some combination thereof. Demand-based coscheduling does not require multithreading, but may enhance the performance of multithreaded applications.

1.6 Overview

The remainder of this thesis is structured as follows: in Chapter 2 we present related work.

In Chapter 3 we present dynamic coscheduling, the particular kind of demand-based coscheduling that we implemented, and some results of modeling and simulation. We used a simple analytical model, and a slightly more complex simulation to investigate the feasibility of dynamic coscheduling. Our results led us to the conclusion that dynamic coscheduling could provide strong coscheduling behavior when processes communicate often. The rate of communication caused the degree of coscheduling to vary, so that when processes communicated very rarely, almost no coscheduling resulted from dynamic coscheduling. We attempted in our model and simulations to make weak assumptions; we nonetheless found that the region in which coscheduling happened was a useful one for fine-grain parallel processes.

In Chapter 4 we describe our prototype implementation of dynamic coscheduling on a workstation cluster. Because of the limitations of the current version of the messaging layer we used, Illinois Fast Messages, the prototype did not allow us to experiment with more than one parallel job. Thus in all of our experiments we ran a single parallel job under timesharing in competition with serial jobs. Although we believe based on the results of our analytical modeling and simulation that these results will extend to multiple parallel jobs, this was nonetheless a serious limitation, because we did not have the opportunity to confirm this experimentally and because we were unable to exercise some mechanisms with which we wished to experiment. Another limitation of our implementation was the need to work with a Unix priority-decay scheduler, which entangles in a single parameter, process priority, both execution order and CPU share. Because coscheduling requires changing execution order while still maintaining fairness, this conflation in Unix priority-decay schedulers was problematic for us. Our compromise was to use a fairness mechanism that required some tuning. Despite these limitations, however, our implementation was sufficient for us to find some very useful experimental results.

Our experiments and experimental results are described in Chapter 5. We found that dynamic coscheduling does indeed result in coordinated scheduling of a parallel process across the nodes of a timeshared workstation cluster. The performance for fine-grained programs using spin-block message receipt is close to ideal. With spinning message receipt, the performance is much better than under the unmodified scheduler, but an efficiency/fairness tradeoff emerges — if fairness is maintained, efficiency suffers; small decreases in fairness lead to large improvements in efficiency. We also confirmed that dynamic coscheduling works better for fine-grained programs than for coarse-grained programs, as we had found through modeling and simulation in Chapter 3. We conclude that dynamic coscheduling improves performance sufficiently to make it worthwhile to implement; but performance is not ideal in all cases and further research on methods of improving it is necessary.

In Chapter 6 we draw conclusions and describe directions for future work. The conclusions we have already reviewed here, in describing the contents of the individual chapters. Our description of directions for future work concentrates on improvements over our current prototype that could be realized in a future implementation and on exploring other kinds of demand-based coscheduling. In the case of our implementation of dynamic coscheduling, we would like to build an implementation that allows us to experiment with multiple parallel jobs, and we would like to find a more flexible mechanism for ensuring fairness. The utility and practicability of a demand-based coscheduler for shared-memory multiprocessors is an area that would also be interesting to explore; a scheme for implementing demand-based coscheduling on a bus-based shared-memory multiprocessor is sketched in Appendix A.

Chapter 2

Related Work

We saw in the previous chapter that some form of coscheduling in necessary for good performance on multiprogrammed multiprocessors. We now review some of the other work in the field.

2.1 Traditional Coscheduling

Ousterhout's pioneering scheduler is described in [18]. Under this traditional form of coscheduling, the processes constituting a parallel job are scheduled simultaneously across as many of the nodes of a multiprocessor as they require. Some fragmentation may result from attempts to pack jobs into the schedule; in this case, and also in the case of blocking due to synchronization or I/O, alternate jobs are selected and run.

Relatively good performance has been reported for competent implementations of traditional coscheduling. Gupta *et al.* report in [10] that when coscheduling was used with 25-millisecond timeslices on a simulated system, it achieved 71% utilization, as compared to 74% for batch scheduling (poorer performance is reported with 10-millisecond timeslices). Chandra *et al.* conclude in [4] that coscheduling and process control achieve similar speedups running on the Stanford DASH distributed-shared-memory multiprocessor as compared to independent timesharing.

However, traditional coscheduling suffers from two problems. The first is that, without information about which processes are communicating, it is not clear how to extend any of Ousterhout's three algorithms to work on jobs where the number of processes is larger than the number of processors — the best one might do would be an oblivious round-robin among the processes during a timeslice in which the job was allocated the entire machine. The second is that the selection of alternate jobs to run, either when the process allotted a node is not runnable or because of fragmentation, is not in any way coordinated under Ousterhout's coscheduling.

We may expect the first problem to become significant as multiprocessors become more prevalent. Manufacturers wishing to provide systems of varying expense and power already vary the number of nodes on the multiprocessors they sell, so that one may buy bus-based symmetric multiprocessors with as few as two or as many as six processors from some manufacturers. The application programmer must then be concerned with somehow keeping the number of processes that constitute a parallel application flexible. This is easy if the application is a multithreaded one using a task queue. But if the application uses a client/server model, or if it consists of independent processes communicating through message-passing or some smaller amount of shared memory, the extra heavyweight context switches required in the case of frequent synchronization will result in considerable overhead.

The second problem is a performance problem. Although the loads examined in the works we have cited have typically been highly parallel ones, many parallel jobs have relatively long sections in which many of the processes are blocked. In these sections alternate processes must be selected to run on the nodes where the blocked processes reside. Additionally, the internal fragmentation in Ousterhout's most popular algorithm (the matrix algorithm) results in some nodes not having processes assigned to them by the algorithm during some timeslices; these nodes will also need to perform this "alternate selection." Unfortunately, traditional coscheduling presents no means of coscheduling these alternates. The result is that even in the two-job case examined by Crovella *et al.* in [5], when approximately 25% of the cycles in the multiprocessor were devoted to running alternates, their use decreased the runtime of the application to which they were devoted only about 1%.

2.2 Distributed Hierarchical Control

Distributed hierarchical control was presented by Feitelson and Rudolph in [8]. The algorithm logically structures the multiprocessor as a binary tree in which the processing nodes are at the leaves and all the children of a tree node are considered a partition. Jobs are handled by a controller at the level of the smallest partition larger than the number of processes required by the job. The placement algorithm strives to balance loads and keep fragmentation low.

Unlike Ousterhout's coscheduling, distributed hierarchical control has a mechanism for the coordinated scheduling of alternates. Suppose K of the nodes allocated to a job cannot run the job's processes, because these processes are blocked. Then the placement algorithm will attempt to find a job with K or fewer processes to run on these K nodes.

If a partition holds processes belonging to different parallel jobs, then the parallel jobs are gang-scheduled within the partition. Distributed hierarchical control thus strikes a middle ground between space-partitioning and coscheduling. It is particularly attractive for larger multiprocessors, where it removes the bottleneck inherent in the centrally-controlled traditional coscheduling of Ousterhout. However, distributed hierarchical control was not designed for smaller machines, such as the desktop machines and departmental servers we have described, on which we expect that it would suffer from the same problems as traditional coscheduling.

2.3 Process Control

Tucker and Gupta suggested in [24] a strategy called *process control*, which has some of the characteristics of space partitioning and some of the characteristics of time-sharing. Under process control, parallel jobs must be written as multithreaded applications keeping their threads in a task queue. The scheduler divides the number of processors on the system by the number of parallel jobs to calculate the "number of available processors." The system dynamically makes known to each parallel application the number of available processors, and the application maintains as many processes as there are available processors. The processes simply dequeue threads from the application's task queue and run them until they block, at which point they take another thread. If more parallel jobs exist than there are processors, the scheduler timeshares processor sets among the parallel jobs.

One advantage of this approach is that when the processes of a parallel job switch among threads, the switch performed is a low-overhead one that does not cross address-space boundaries, because the multiple threads of an application share an address space. Thus fewer heavyweight context switches need be performed. Tucker and Gupta also cite as an advantage what they call the *operating point effect* — the fact that many parallel jobs will run more efficiently on a smaller number of nodes than on a larger number of nodes, due to the overhead of communication among larger numbers of processes.

Several published works [4, 10, 23] cite good performance for process control, but these works also find that coscheduling can be modified to have equivalently good performance.

It will be clear in what follows that demand-based coscheduling is not at all incompatible with a multithreaded approach; it might even be made to work with process control. But we find process control *alone* to be insufficient for the office environment we have described for two reasons: the requirement that applications be programmed in a particular way, and the high variability of runtimes of memory-intensive applications.

We have already discussed the first problem, that of intrusiveness, to some extent above. For many parallel applications, especially data-parallel applications, a multithreaded approach is entirely appropriate. But for others, applications composed of subtasks that perform distinct and logically autonomous functions, the multithreaded approach may be inappropriate or even impracticable. Examples might include clients and servers that require high rates of communication, but where for security reasons the client is not allowed access to all of the server's data.

Thus process control alone is insufficient as a scheduling approach in the environment we have described, because in requiring that all parallel applications be coded in a task-queue multithreaded fashion, it would require that an important abstraction be given up by the programmer in order to achieve good performance: the abstraction of a process with its own address space. But processes offer modularity and security, and application writers will be loath to give up these qualities in applications where the process abstraction is the natural one.

The second problem, that of high variability of runtimes for some sorts of processes under process control, results from certain parallel jobs requiring more resources than are available on a single node in order to execute efficiently. Under process control, the arrival of new jobs into the system can cause the "number of available processors" to fall below a critical level at which the performance on some jobs will begin to deteriorate worse than linearly.

This implies that in fact the jobs in question show superlinear speedup. In fact this is true in two examples in published works on process control. In [10], the LU application is found to perform very poorly under process control when run on three processors, and the authors point out that a drastically increased cache miss rate is to blame. Similarly, in [4], the Ocean application suffers a twofold decrease in efficiency when run on eight processors as compared to when it is run on sixteen processors. Some of this decrease in efficiency is attributed by the authors to data distribution optimizations being performed in the sixteen-processor case, but not in the eight-processor case. The implication is that, if the data distribution optimizations had not been performed in the sixteen-processor case, the Ocean application would have performed nearly as inefficiently in the sixteen-processor case as in the eight-processor case. So far as one can tell from the published work alone, this attribution of cause may be mistaken, because the same work shows a coscheduling experiment in which data distribution optimizations were not performed. In this experiment, coscheduling among two jobs suffered only a five-percent decrease in efficiency compared to the standalone sixteen-processor case with data distribution optimizations — thus it seems that we can bound above the effect of data distribution optimizations by five percent. Because the authors state that Ocean has a larger working set than the other applications tested, we suspect that the actual cause of the inefficiency here may be the larger number of cache misses that result from the application being executed on a collective cache of half the size as in the sixteen-processor case.

Helmbold and McDowell have documented this sort of "superunitary speedup due to increasing cache size" in [11]. Because of this property of certain parallel applications, their ideal "operating point" is larger than one — possibly considerably larger than one. Thus forcing them to run on fewer processors will be very inefficient. This is not a problem under coscheduling, because under coscheduling the arrival of new jobs does not cause fewer processors to be devoted to the execution of a parallel job.

We believe that the phenomenon of increasing inefficiency with higher loads under process control may be an important problem in practice. This is because software tends to perform near the memory boundaries available on most users' processors. The reason for this pressure is simply economic: purchasers of computer hardware will tend to buy as little memory as possible while still maintaining satisfactory performance on applications; to purchase more would be wasteful. Purveyors of software tend to use more memory to add new features to their applications in order to gain competitive advantage. Programming so as to conserve memory requires more effort and thus costs more, and will be done only insofar as is necessary to keep customers happy.

This pushing at the boundaries of available memory will probably mean that many commercial applications will show superlinear speedup. If process control as it is described in [24] were used as the only means of timesharing a multiprocessor, we would expect that such applications would show poor performance when the job load was high.

2.4 Runtime Activity Working Set Identification

Feitelson and Rudolph describe in [9] an algorithm called "runtime activity working set identification" for scheduling parallel programs on a timeshared multiprocessor (we shall call this algorithm RAWSI, for brevity's sake). While demand-based coscheduling was developed independently from RAWSI,[1] the two have significant similarities: in both approaches, runtime mechanisms are used to identify communicating processes so that they can be coscheduled. However, RAWSI differs significantly from both dynamic coscheduling (described in Chapter 3 and [21]) and predictive coscheduling (described in Appendix A), the two approaches to demand-based coscheduling we present here.

One major difference between RAWSI in a message-passing system and dynamic coscheduling is that RAWSI does not make the decision of what process to schedule immediately upon receiving a message. Instead, RAWSI uses the sending and receiving of messages as a means of identifying to the system the rate of communication between processes, and then uses this information to determine whether to coschedule them. Another difference is that RAWSI is more similar to Ousterhout's original matrix coscheduling algorithm in that processes are assigned to nodes and multi-context-switching is used: that is, the assumption is that it is possible to effect simultaneous context-switches across the processing nodes. This requires closer coordination than dynamic coscheduling, in which scheduling decisions are made by each node entirely independently of other nodes, and process placement is not part of the scheduling algorithm; this is because dynamic coscheduling was originally conceived for use in networks of workstations, where close coordination may be difficult and the assignment of processes to processors or their migration may be impossible or inappropriate.

The most significant difference between RAWSI in a shared-memory system and predictive coscheduling is that, rather than using virtual memory system information to automatically detect the sharing of information, RAWSI relies on the identification by the programmer of distinguished data structures, or *communication objects*, which are shared between processes. The compiler then inserts with every read or write of these structures code that makes known to the system the fact that communication has taken place. Thus RAWSI on a shared-memory multiprocessor will be more intrusive than predictive coscheduling, because it will require the use of particular compilers and the advance identification by the programmer of shared data structures. Also, we believe that use of virtual memory system structures as described in Appendix A will be less expensive than the execution of instruction sequences to record accesses to shared data structures.

[1] A description of dynamic coscheduling was first published in [21].

2.5 Implicit Scheduling

Implicit scheduling is the name given by Dusseau *et al.*[6] to their algorithms for adaptively modifying the spin times in spin-block message receipt to achieve good performance on "bulk-synchronous" applications (those which perform regular barriers, possibly with other communication taking place in between barriers).[2] This work has some bearing on our own, because of the similarity of problems and experimental platforms, and so we will treat it here at some length and revisit it in sections describing our experimental results.

The experiments described in [6] were performed used the Solaris 2.4 scheduler code in a simulation of 32 workstations running 3 parallel jobs, each having one process residing on each of the 32 workstations.

The workloads were SPMD, consisting of loops of four phases each: in the first phase, a variable amount of computation was performed; then, second, an "opening" barrier synchronization was performed; third, some optional communication in the form of request-reply exchanges was performed; and, fourth, a closing barrier synchronization was performed.

Message receipt was by the popular spin-block mechanism described in [18], [10], and others. Dusseau *et al.* found that performance with fixed-spin-time spin-block messaging was quite good under the Solaris 2.4 scheduler when fixed spin times on the order of a context switch were used. Spin times of two context-switch times performed better; and varying the spin time using an adaptive algorithm that attempted to measure the variation of the length of the computation phase worked best.

Performance was evaluated by comparing workload completion times of the entire workload to completion times under an idealized gang scheduler that used spinning message receipt and 500-millisecond timeslices.[3] Results are presented for workloads consisting of jobs with the same computational granularity. The conclusion of the paper reports, however, that more coarse-grain jobs are favored by the scheduler over more fine-grain jobs, and that fairness can be a problem.

The contribution of this paper that has the greatest relevance to our own work is its presentation of experimental data that show the surprisingly strong performance of spin-block message receipt under the Solaris 2.4 scheduler. The paper correctly states that the priority-boosting mechanism of the Solaris 2.4 scheduler is responsible (as we confirm experimentally in Section 5.4.1), but provides an account of this that states

[2]There is some ambiguity in [6] about whether the term "implicit scheduling" is also intended to cover all approaches for achieving coordinated scheduling in a network of workstations by making local decisions based on information about communication; but because so broad a description would also cover demand-based coscheduling, described here and in [22], we will use the term "implicit scheduling" to describe only the combination of spin-block message receipt with the algorithm for adaptively determining spin times described in [6].

[3]Given the variable amount of computation that was performed in each cycle of these processes, it seems possible that the use of spinning message receipt was not the most optimistic choice for the idealized gang scheduler. Ousterhout suggested spin-block message receipt in [18] for cases where the workload included jobs with very coarse-grain computation, because alternates might be able to perform additional computation before blocking for message receipt.

that the priority-boosting happens whenever a process is returned from a sleep queue to a run queue. This is incorrect; we examine these issues further in Section 5.4.1.

A later section in [6] describes an experiment with a "round-robin" scheduler (simulating only one run queue for the Solaris 2.4 scheduler rather than the normal sixty),[4] which found very poor performance for spin-block message receipt. This was attributed to the absence of priorities in round-robin schedulers, which is an overly narrow statement. The important issue in achieving coscheduling with independent schedulers is the ability to run a process when a message arrives. We repeat the experiment in our own work (see Section 5.4.1), and by using dynamic coscheduling show that the use of priorities is not necessary to achieve coordinated scheduling even with a single-queue scheduler; all that is necessary is that the scheduler run the process when the message arrives.

[4]The modified scheduler is not a strict round-robin scheduler, because when a process returns from a sleep queue to a run queue in Solaris 2.4, it is always placed at the end of the run queue (even if it has received a priority boost), and so execution order is varied even with a single run queue.

Chapter 3

Dynamic Coscheduling

In this chapter we describe *dynamic coscheduling,* the demand-based coscheduling algorithm we implemented [21, 22]. Dynamic coscheduling is well suited to implementation on message-passing processors, because it uses the arrival of a message as a means of signalling the scheduler that the process to which the message is addressed should be scheduled immediately.

Clearly, dynamic coscheduling is straightforward to implement; an arriving message not addressed to the currently running process can trigger an interrupt on the network interface device.[1] Alternatively, if protection is not an important issue and the network interface is manipulated directly in user mode, the detection of an arriving message not addressed to the currently running process can be performed by a library routine which can execute a system call in the case when a scheduling decision must be made.

Dynamic coscheduling should also work on many distributed-shared-memory multiprocessors. In a cache-coherence scheme such as the software schemes presented by Chaiken *et al.* in [3], cache line invalidations can be treated in the same fashion as arriving messages. We can do even better on systems with network interface processors, such as FLASH [15] or Typhoon [20]. In these systems, some of the scheduler state can be cached in the interface processor, so that the scheduling decision can be made without consulting the computation processor. The computation processor could be interrupted only when a preemption was needed. In this case the number of exceptions could be kept to the minimum necessary.

It is more difficult to envision applying this scheme to a shared-memory multiprocessor with hardware-only cache-coherence protocols; for such processors predictive coscheduling will be more appropriate.

We now develop a dynamic coscheduling algorithm by taking the simplest possible implementation of this idea and successively modifying it to achieve fair scheduling while maintaining good coscheduling.

[1] Of course, with current operating systems, arriving messages should not trigger interrupts if the process to which they are addressed is currently running — the overhead incurred would be too great. Instead, the messaging layer can poll for messages.

3.1 The "always-schedule" dynamic coscheduling algorithm

The first version of the dynamic coscheduling algorithm is the simplest possible one, in which the job for which the arriving message was destined is always immediately scheduled. We have modeled this case analytically with a Markov process for two symmetric jobs of N processes running on N nodes, using the weak assumptions that messages are uniformly addressed, that the processes generating them are memoryless, and that the run-time of processes before they block spontaneously is exponentially distributed. We call the assumptions "weak" because we expect that real processes exhibit greater regularity that would in fact improve the performance of such a scheduler.

The two-job Markov process is a skip-free birth-death process, and a closed-form solution for the steady-state probabilities is possible. The multiprocessor has N nodes. The states of the process are defined as follows: in state i, $N-i$ nodes are running the first job and i are running the second job. If we call the jobs A and B, in our model we make use of the quantities q_{SA} and q_{SB}, the rates of spontaneous context switching of processes for jobs A and B. The spontaneous switching rate is intended to capture at once the notion of timeslice expiration and blocking due to I/O or synchronization requirements. A node running a process will switch from running it to the next resident process at this rate. We also use the quantities q_{MA} and q_{MB}, the rates of message-sending for processes of jobs A and B — these are the rates at which the running processes generate uniformly-addressed messages to other processes that make up their jobs.

The Markov process is shown in Figure 3-1. In summary, state 0 is the state in which all the nodes are running job A and no nodes are running job B. In state N, all the nodes are running job B and no nodes are running job A. In state $N/2$, half of the nodes are running each job.

The steady-state probabilities are then given by

$$p_k = p_0 \prod_{i=0}^{k-1} \frac{(N-i)q_{SB} + i\frac{N-i}{N}q_{MB}}{(i+1)q_{SA} + (N-i-1)\frac{(i+1)}{N}q_{MA}} \tag{3.1}$$

where

$$p_0 = \frac{1}{1 + \sum_{k=1}^{N} \prod_{i=0}^{k-1} \frac{(N-i)q_{SB} + i\frac{N-i}{N}q_{MB}}{(i+1)q_{SA} + (N-i-1)\frac{(i+1)}{N}q_{MA}}} \tag{3.2}$$

Results for this case are shown in Figure 3-2. Here we have taken $N = 64$, $q_{SA} = q_{SB} = Q_S$ and $q_{MA} = q_{MB} = Q_M$. The vertical axis is steady-state probability. The deep axis is $log_{10}(Q_s/Q_m)$. The horizontal axis along the front gives state number.

Towards the front of the graph, we see that the probabilities of being in the states where all the nodes are running one job or the other are high, and the probabilities of being in states where some nodes are running one job and some running the other are low. We see then that the ratio of the rate of sending messages to the rate of

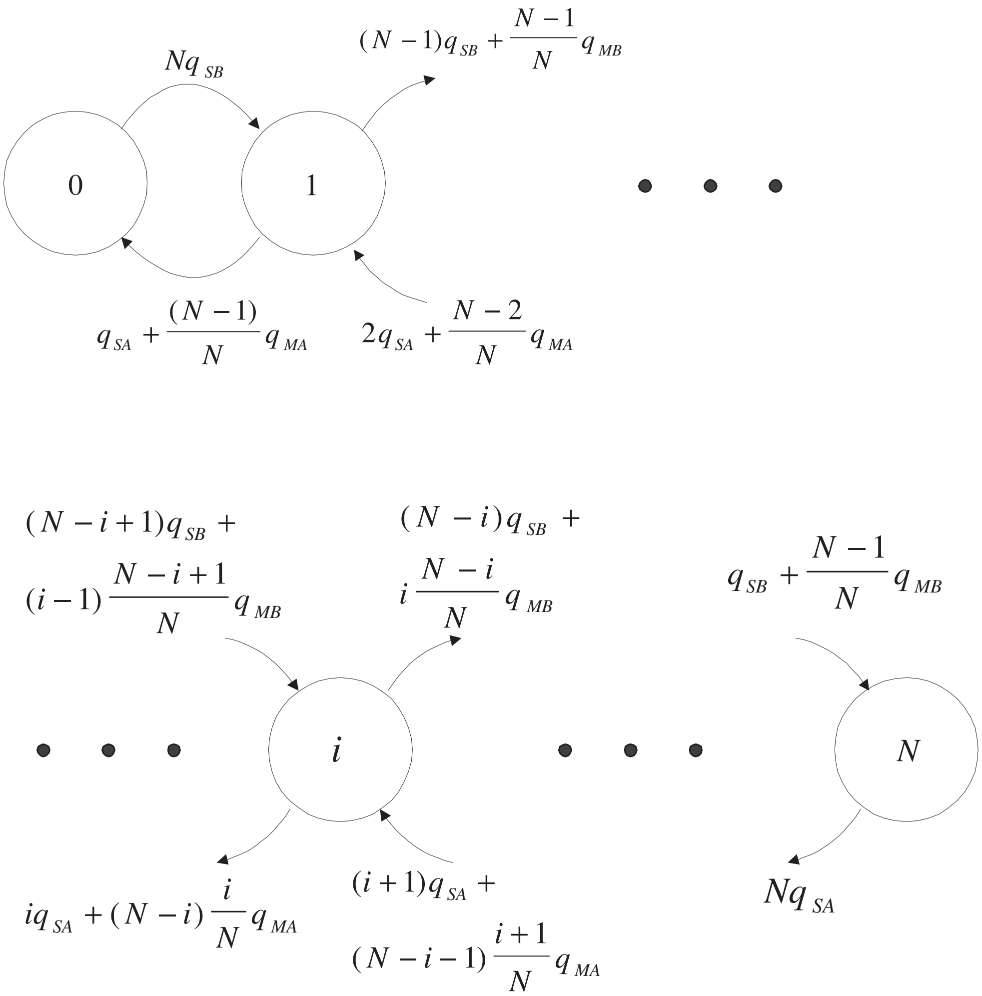

Figure 3-1: A Markov process whose states represent the degree of coscheduling on a N-processor system running two jobs, under the always-schedule dynamic coscheduling algorithm. See the text for further details.

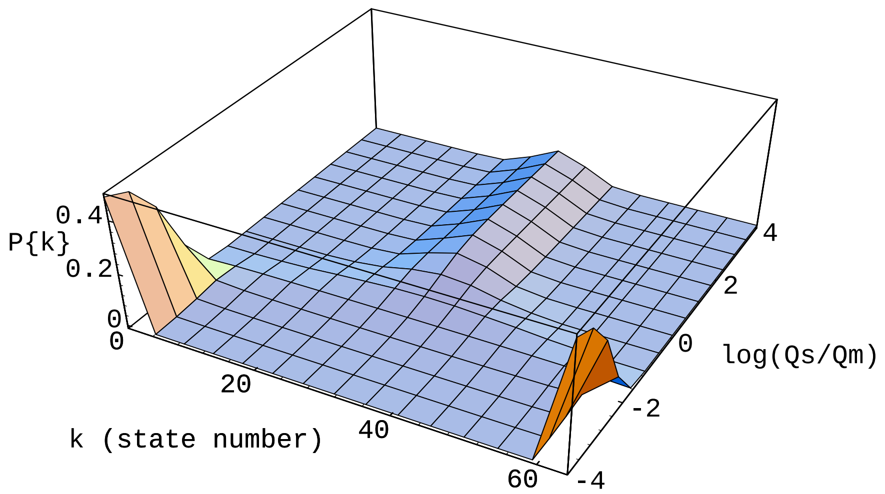

Figure 3-2: Steady-state probabilities found using a Markov model to calculate dynamic coscheduling performance on a 64-processor system running two jobs. See the text for further details.

spontaneous switching of processes determines the steady-state probability that all processors in the modeled system are running a single job. We found that if several hundred or more messages are sent on average between the spontaneous context switches, then the steady-state probability that either all processors are running one job or all processors are running the other job is about one-half. If fewer messages are sent between spontaneous context switches, then a binomial behavior begins to emerge, so that when only one message is being sent on average between spontaneous context switches, about half of the processors are running one job, and half running another. It is to be noted, though, that when very few messages are being sent, coscheduling is unlikely to be important.

It is encouraging that with such a simple rule, we find strong coscheduling behavior under such weak assumptions. Unfortunately, this coscheduling algorithm has a fatal flaw. The flaw is that it is completely unfair, tending to very strongly favor jobs that send a lot of messages. Also, even if message-sending rates are equal, this algorithm may take a very long time to switch out of a state in which most processors are running one job, although this dynamic behavior of the algorithm cannot be seen from the steady-state probabilities alone.

Figure 3-3 illustrates the unfairness of this algorithm, showing the steady-state probabilities for the case where $q_{SA} = q_{SB} = 0.005$ but $q_{MA} = 0.49$ and $q_{MB} = 0.5$. It can be seen that, despite the fact that the message-sending rates are very close, job A achieves full coscheduling only about 2% of the time whereas job B achieves full coscheduling about three times as often.

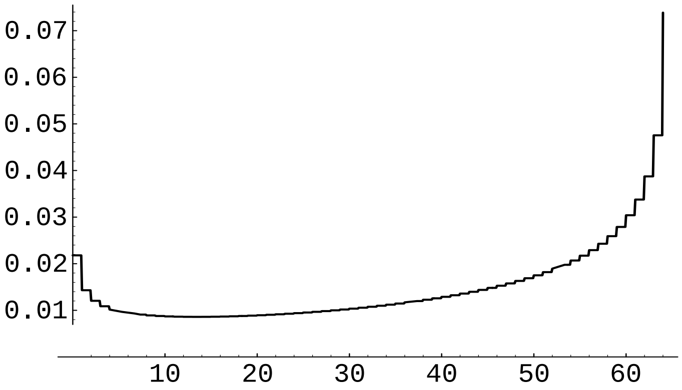

Figure 3-3: Steady-state probabilities in the case where message-sending rates differ very slightly – see the text for further details.

3.2 The "equalizing" dynamic coscheduling algorithm

We modified the "always-schedule" dynamic coscheduling algorithm to require that runnable processes receive equal shares of the CPU, within some constant difference. We called this policy "run-time equalization." Because it was more difficult to analytically model the new algorithm, we wrote a discrete event simulator for it, and ran experiments in which we modeled a 64-node multiprocessor running for 100,000 scheduler cycles.

We maintain for each process i a quantity r_i, the number of scheduler cycles for which it has run since the process that most recently joined the scheduler run queue started running. We define a global quantity h, which can be modified to affect the "volatility" of scheduling: a larger value of h causes the scheduler to take longer to switch due to arriving messages.

Run-time equalization works as follows: when a message destined for process j arrives at its node, which is running process i, $i \neq j$, we switch to process j if and only if $r_j + h < r_i$, that is, if and only if process j *lags* process i by more than h scheduler cycles. This definition of h means that if the system is run for no more than H scheduler cycles, and $h = -H$, the "equalizing" algorithm will always behave the same as the "always-schedule" algorithm. This is because r_j cannot be greater than H if the system is run for no more than H cycles, and so necessarily $r_j + h \leq 0$, and in this scenario process i has run for at least 1 scheduling cycle. With this very negative value of h, then, the scheduler will always context-switch due to arriving messages.

On the other hand, if $h = H$ and the system is run for no more than H cycles, a process i will never accumulate more than H scheduling cycles, and it will always be the case that $r_j + h \geq r_i$ (until possibly the H^{th} cycle, when the experiment ends). Thus with this large positive value of h, the scheduler will never context-switch due

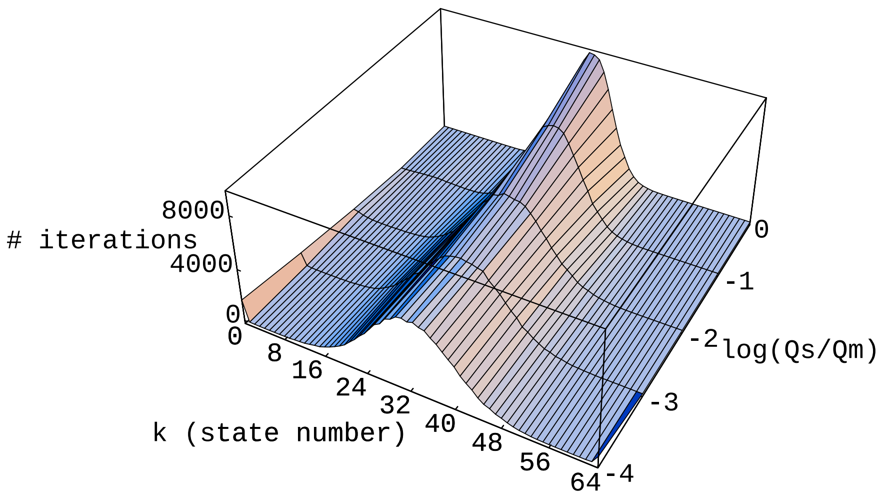

Figure 3-4: Degree of coscheduling in the case where message-sending rates differ by a factor of two, but equalization is used. The vertical axis approximates steady-state probability – the scale is the number of iterations out of 100,000 in which the process was found in the indicated state.

to arriving messages.

We found that, for values of h near $-1,000$, reasonably fair performance was attained over the running of an experiment; however, little coscheduling was achieved. The results for the more radical case of message-sending rates of .25 and .5 may be seen in Figure 3-4.

Our intuition about the failure to coschedule under simple equalization is that, by disregarding more opportunities to coschedule processes, we caused more thrashing. In general, the higher the value of h, the less coscheduling was achieved. One possible solution was to further reduce h, but in fact, we already had a mechanism that proved to work better in practice at recovering strong coscheduling behavior, by ensuring that the scheduler makes progress from job to job.

3.3 The "epochs and equalization" dynamic coscheduling algorithm

Consider a scenario in which about half of the nodes on a multiprocessor are running one parallel job, and half the other. In our simulation, when a node running parallel job A spontaneously switches to parallel job B, there is a probability of close to $1/2$ that the next message it receives will be destined for a process belonging to job A, provided that message-sending rates for the two jobs are equal. Thus there is a substantial probability that the node will switch quickly back to job A without job B ever having achieved full coscheduling. This probability is greater if switching is mostly spontaneous.

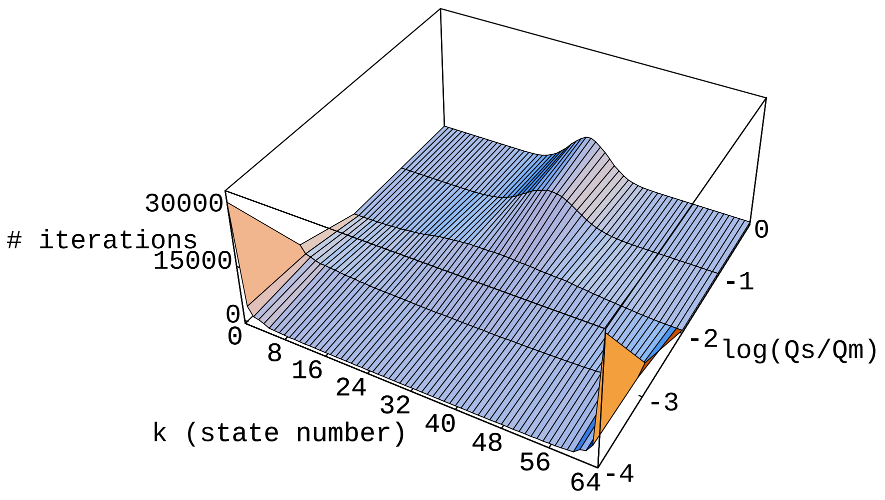

Figure 3-5: Degree of coscheduling achieved in the case where message-sending rates differ by a factor of two, and both equalization and epochs are used – the vertical axis approximates steady-state probability, being the number of iterations out of 100,000 in which the simulator was in the indicated state.

Epoch values are used to reduce this sort of thrashing. The epoch value is maintained in a counter at each node. The counter is incremented at each spontaneous context switch. When a node sends a message, the epoch value is included in the message; when receiving a message, the node considers switching only if the equalization criteria are met and the epoch number is higher than its own. If the node does switch processes, it adopts the higher epoch number as its own.

The result is that nodes "defecting" from a parallel job will not return to the job due to messages being sent by nodes remaining with the job. Progress must be made to the new job before the node will consider switching back.

The results for this strategy can be seen in Figure 3-5, and are quite encouraging — coscheduling behavior is achieved for more than about 300 messages per timeslice, even given our pessimistic assumptions. As in Figure 3-4, message-sending rates of .25 and .5 are used, so that we see that unbalanced message-sending rates are still effectively handled by the equalization mechanism and fairness is preserved.

The actual composition of the epoch number in an implementation would be slightly different than in this simulation. In an actual implementation, because the number of runnable processes on different nodes in a multiprocessor would typically differ, the same epoch number could be reached by two nodes running different processes. In order to avoid this possibility, when incrementing the epoch number, we embed as the least significant bits of the epoch number the unique node number of the node on which the epoch number is being increased. However, this is the only occasion on which the node number is embedded in the epoch number — when adopting an epoch number in an incoming message or when comparing it to a local epoch number, the epoch number is treated as a whole. The result is that a group of nodes running the same job will typically share an epoch number whose least significant bits are the node number of the first node in the group to have switched to the job.

3.4 Discussion of results of simulation and modeling

Simulation and modeling showed that a crucial quantity in dynamic coscheduling is the ratio of the message-passing rate to the rate at which processes spontaneously block. We will find some confirmation of this in later experiments on a workstation cluster, where we will see that a job performing repeated barrier synchronizations at low communication rates does not become so strongly coscheduled as one with higher communication rates.

We also saw that the simplest form of dynamic coscheduling suffered from a problem with fairness — jobs performing more communication could monopolize the processing nodes on which they executed. Our imposition of a simple fairness criterion resulted in additional thrashing — because a processing node would often refuse to switch to a new process for fairness reasons, the region in which strong coscheduling behavior emerged moved further towards high communication rates or long scheduling quanta.

The addition of epoch numbers to our scheme reduced thrashing behavior, by reducing the frequency of opportunities for nodes to defect from a group running a single job. The epoch number scheme assures progress from job to job occurs in an orderly fashion.

In conclusion, our model and simulations showed dynamic coscheduling to be promising. Because we had made weak assumptions about communication patterns and message interarrival times and were still able to cause strong coscheduling behavior at realistic communication rates, we hoped that a real workstation cluster would perform well under dynamic coscheduling.

Chapter 4

An Implementation of Dynamic Coscheduling

We describe in this chapter the version of dynamic coscheduling (DCS) we implemented to run with Illinois Fast Messages, a user-level messaging layer developed at the University of Illinois at Urbana-Champaign [19].

4.1 Experimental Platform: Illinois Fast Messages and Myrinet

Illinois Fast Messages is a high-performance messaging layer that uses a user-level library to provide messaging primitives without the overhead of domain crossing that would be required by a kernel-resident device driver.

The implementation of Illinois Fast Messages we used ran on a Myrinet network [2] connecting eight SPARCstation-2 workstations. The Myrinet switch provides a relatively low-latency, high-bandwidth interconnection for workstations. The Myrinet 2.3 interface boards we used had a slow CISC processor operating at the 20 MHz speed of the SPARCstation's SBUS peripheral bus; however, even with these obsolete workstations and slow network interface processors, we achieved user-space to user-space latencies of 40 μsec with 128-byte messages, and bandwidths of 13 MB/sec.

The SPARCstation-2 processors we used were equipped with 16 MB of main memory, had 40 MHz processors, and ran the Solaris 2.4 operating system. The Myrinet interface board memory and control registers are mapped into both kernel and user space. In FM, the mapping into kernel space enables initialization and control of the device in response to system calls, but is not used in the common case of sending and receiving messages. Instead, the mapping into user space allows the user-level FM library to control the device directly, without kernel intervention. A schematic diagram showing the memory inclusion is shown in Figure 4-1.

One of our goals in this implementation of DCS was to implement a version that did not require building a kernel. We hoped to distribute our version of DCS with FM, and for legal reasons it would not have been possible to distribute a modified kernel. Thus the functionality, but not the code, of the kernel was modified. We

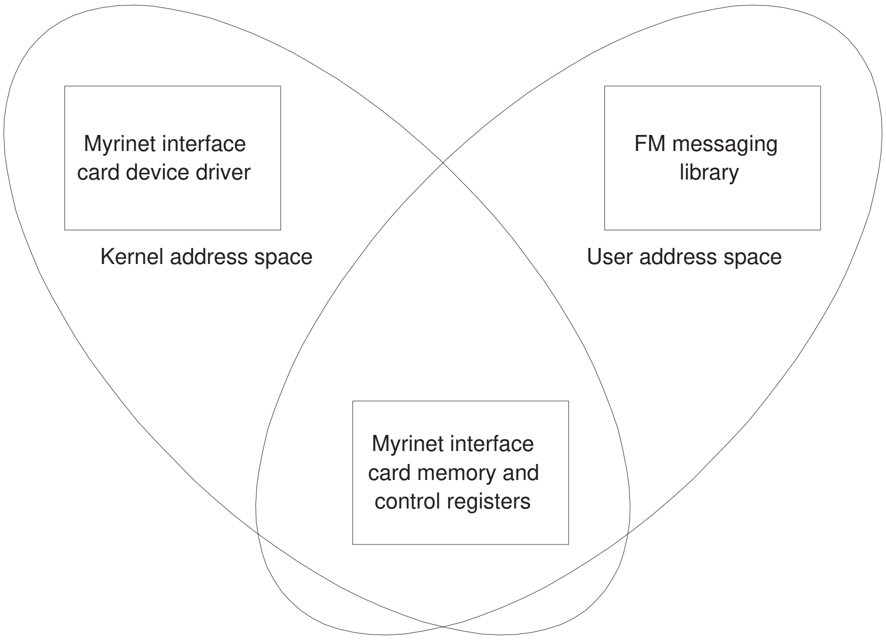

Figure 4-1: Address spaces in the FM implementation on Myrinet, running on Sun SPARC processors.

modified only a loadable device driver that manipulated the Solaris 2.4 scheduler data structures to cause the scheduler to preempt and run processes as necessary. One consequence of this approach was that we had to work with the Unix priority decay scheduler, with the result that fairness had to be achieved through a less direct mechanism that might have been possible had we been able to use a mechanism that directly specified the percentages of CPU time allocated to different processes. A particular disadvantage of the mechanism was that we never automated it, although we believe it would not have been difficult to do so; instead we manually modified its parameters for good performance.

4.1.1 Disadvantages of the experimental platform

At the inception of the project, one disadvantage of FM as a platform for testing demand-based coscheduling was obvious. The Myrinet implementation of FM did not allow multiple parallel jobs to be run simultaneously. The reason for this was the nature of the implementation. Because the Myrinet device's memory was simply mapped into the user space of the process that was using the device, the most obvious means of having multiple processes make use of the interface simultaneously (by mapping its memory into each user's address space and using mutexes for the buffers and control registers) would have entailed a security risk. It would still have been possible to run multiple parallel jobs if we had had multiple Myrinet interface cards for each processor. Unfortunately, resources were limited and this was not possible; thus we measured the performance of DCS with a single parallel job running with serial competitors.

As a result, we were unable to test the efficacy of the epoch number scheme for avoiding thrashing between multiple parallel jobs.

Another disadvantage of FM became obvious as the project proceeded. FM did not include any means of implementing spin-block message receipt; a strict polling model was assumed. As a result, spinning message receipt was used for earlier experiments. This had two negative effects. The first was that, as other researchers have reported [23], spinning message receipt results in very poor performance for fine-grain programs in a timeshared environment. Thus the Solaris 2.4 scheduler with spinning message receipt served as implausible competition for DCS. The second was that DCS itself was conceived and modeled for use with spin-block or blocking message receipt. We did not know what the effects of using DCS with purely spinning message receipt would be.

Thus we found it necessary to implement spin-block message receipt for FM. However, as will be seen in the following chapter, DCS with spinning message receipt performs almost as well as DCS with spin-block message receipt, and both perform better than the unmodified Solaris 2.4 scheduler.

4.2 Implementation

A diagram of the implementation is given in Figure 4-2. This diagram is simplified, in that it does not include a description of the spin-block code.

Our implementation of dynamic coscheduling involved modifications to three parts of the system: the device driver for the Myrinet network interface card; the control program running on the Myrinet network interface processor; and the FM library invoked by the user process.

4.2.1 Modifications to the device driver

The code that actually manipulates the scheduler data structures resides in the device driver for the Myrinet interface card. This is the code referred to as "DCS policy and interrupt handler" in Figure 4-2.

Because FM does not allow multiplexing of the network interface, only one process belonging to a parallel job can be resident on each node in the workstation cluster. Thus, although in what follows, the phrase "the process that is to receive the message" can refer to only one process, the process that opened the FM device, we refer to it as the process that is to receive the message because this would be correct even if FM allowed multiple parallel processes.

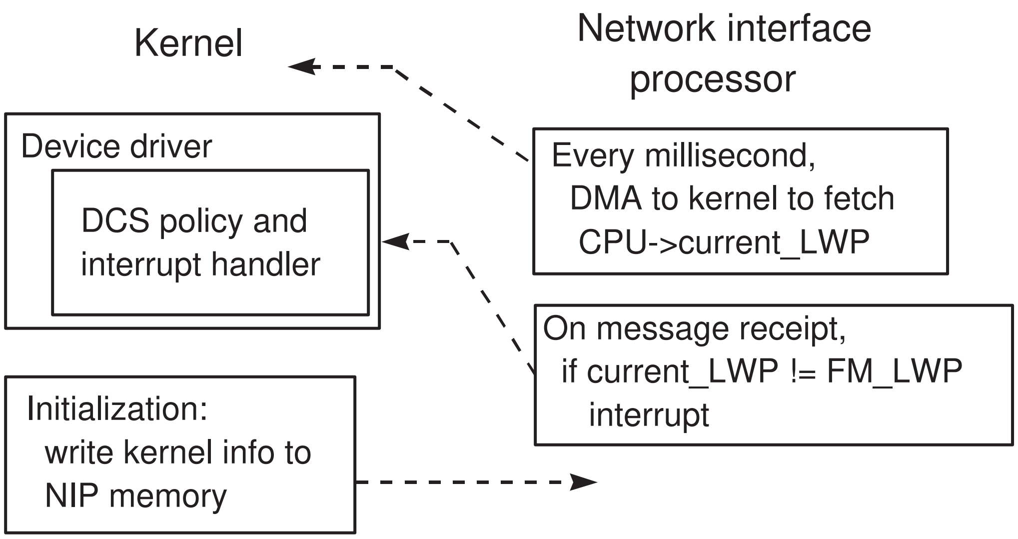

Figure 4-2: Simplified DCS implementation schematic (spin-block implementation not shown).

In the normal course of receiving a message on a host that is currently running the process to which the message is directed, the scheduling code in the device driver is not invoked at all. However, if the message is received when the setting of a variable in the network interface processor's memory indicates that the process that is to receive it is not currently running, the control program running in the Myrinet interface card will cause an interrupt to be generated. When this happens, the interrupt handler in the device driver is invoked.

The interrupt handler first determines whether in fact the currently running process is the one to which the message is directed, because, as will be explained later, the variable in the network interface processor's memory contains only an approximate value of the process that is currently running on the host. If the currently running process is the one to which the message is directed, the interrupt routine simply returns; but if it is another process, then if it would be fair to schedule the process that is to receive the message, an attempt is made to preempt the currently running process and immediately run the process that is to receive the message.

Because Solaris 2.4 processes can be multithreaded, the mechanism that attempts to schedule the process for which the message is intended makes all of the runnable threads belonging to the process that is to receive the message equally likely to be scheduled next, because FM messages do not contain any information about precisely which thread is intended to receive them. If the network interface were multiplexed among multiple processes, then such information would have to be present in the messages, and it would be best in multithreaded operating systems like Solaris 2.4 if it could uniquely identify the particular thread that was to receive the message, as in that case that thread could be scheduled immediately. The algorithm currently used is sketched in Figure 4-3.

We attempt to run the process receiving the message by raising its priority to

```
In interrupt handler:

if (running_LWP != FM_LWP) {
  if (fair to preempt) {
    for each kernel thread belonging to FM_LWP {
      raise priority to maximum for user mode;
    }
    preempt currently running thread;
  }
}
```

Figure 4-3: Sketch of the algorithm used in the device driver interrupt handler for implementation of dynamic coscheduling.

the maximum allowable priority for user-mode timesharing proceses; with the default Solaris 2.4 dispatcher table, this is 59. It is also placed at the front of the dispatcher queue for that priority. Then flags are set that will cause the Solaris 2.4 scheduler to run before the interrupt handler returns to the process that was running when the interrupt occurred. Unless the process that was running when the interrupt occurred had a higher priority than the maximum allowable priority for user mode, the process receiving the message will run immediately upon return from the interrupt routine.

Achieving fairness under Solaris 2.4

As mentioned above, under Unix priority-decay scheduling, fairness must be achieved by a less direct means than the equalization mechanism described in Section 3.2. A limitation of our prototype was that we never implemented a mechanism to automatically modify the fairness parameters, although we believe it would have been straightforward to do so; instead we tuned the parameters for good performance on each of the experiments we ran. The mechanism we used allowed us to reduce the frequency with which we preempt other processes in favor of the parallel process that is receiving the arriving message, and, in particular, to reduce it more as the number of runnable processes in the system increased.

Our mechanism used a predicate for "fair to preempt" that was an inequality:

$$2^E(T_c - T_p) + C \geq T_q R \tag{4.1}$$

where E, T_c, T_p, C, T_q, and R are defined as:

$$\begin{aligned} E &= \text{Exponent. We chose this value empirically.} \\ T_c &= \text{Current time.} \\ T_p &= \text{Time of previous priority boost and preemption attempt.} \end{aligned}$$

$$C = \text{Constant, in milliseconds. We chose this value empirically.}$$
$$T_q = \text{Length of time quantum – set to 20 msec under Solaris.}$$
$$R = \text{Number of jobs in the run queue.}$$

The intention here is to limit the frequency of the preemptions to some number of times for each cycle the scheduler makes through the run queue, assuming that all jobs on the run queue are running at the highest priority. That is, RT_q is the "length of the run queue in time" if the jobs on the run queue are assumed to run to the completion of their timeslices before blocking and their timeslices are assumed to be of duration T_q. $T_c - T_p$ is the time since the last preemption. For example, an empirically-determined value of $1/2$ for 2^E would mean that we required that the time since the last preemption attempt be at least twice the length of the run queue in time; if this criterion was not met, we would not attempt to schedule the process that was to receive the message.

This mechanism is crude, but as will be seen in the next chapter, it can typically be tuned to achieve fairness very close to that achieved by the default Unix priority-decay scheduler on serial loads. As mentioned above, a less clumsy mechanism for achieving fairness could be implemented by writing a new scheduler that did not use the Unix priority-decay mechanism. We also believe that it would be straightforward to dynamically monitor the recent CPU shares of runnable processes every 1 or 2 seconds and modify the fairness parameters to achieve fairness. Because of time constraints, we were unable to implement this approach.

Initialization; spin-block implementation

The device driver also contains routines (which can be invoked via the `.ioctl` interface) to initialize the dynamic scheduling mechanism and set parameters in the fairness mechanism. Initialization requires writing various variables in the network interface processor's memory: in particular, the address of the memory location in which the kernel maintains the address of the current lightweight process, or LWP (which is what the entities usually called "processes" are called in Solaris 2.4), and also the address of the LWP that opened the Myrinet network interface device.

The kernel side of spin-block message receipt is implemented by providing an additional call that can be invoked by the user process to remove the calling process from the run queue and place it on a sleep queue until a message arrives. The interrupt routine in the driver is modified to awaken the sleeping process upon message arrival if it was sleeping; this mechanism works regardless of whether DCS is being used, because spin-block message receipt must be available without DCS for performance comparisons.

4.2.2 Modifications to the network interface processor control program

Because the processor that controls the Myrinet network interface is called the "LANai," the control program that runs on it is called the "LANai Control Program," or LCP.

The default FM LCP is very simple: it merely dequeues outgoing messages and sends them, performing scatter operations as necessary, and gathers incoming packets into messages and writes them by DMA into the host's memory.

We modified the LCP to periodically read by DMA from the host's kernel memory the address of the currently running LWP. The value is read once per millisecond; because of the expense of the operation (tens of microseconds), reading the value on each message arrival would have imposed unacceptable overhead on message receipt.

As a result of reading the address of the current LWP only periodically, interrupts will occasionally be issued on message arrival when the currently running process is in fact the process to which the message was directed. The test at the beginning of the interrupt handler, described in Section 4.2.1, serves to handle this case. In testing, we found that this happened very rarely, so that few interrupts were spurious, and the overhead from these was negligible. Chapter 5 will present information on the number of interrupts and the number of priority modifications in each of the experiments.

The dual of this problem is slightly more complex to handle. In this case, a message may arrive when the process for which it is intended is no longer scheduled, but the variable in the LCP has not yet been updated to show this. In this case, an interrupt would not be generated, although it should be. We handle this case by checking whether any unqueued messages are in the buffers whenever we make the transition from the state in which the LCP variable holds the address of the LWP for which the message is intended to the state in which it holds some other LWP's address. If unqueued messages remain, the LCP generates an interrupt at this time. In one experiment with a load of two competing processes and fine-grain communication, these additional interrupts increased the total number of interrupts by about 35%; however, we did not attempt to determine how many of these interrupts succeed in causing rescheduling once the fairness mechanism described in Section 4.2.1 was implemented. Possibly many of them are discarded by the fairness mechanism, in which case it would perhaps be best to ignore the problem and not generate an interrupt when the variable makes its transition to the state in which it indicates the host is no longer running the program to which the message was directed.

4.2.3 Modifications to the FM messaging library

Few modifications to the FM messaging library were required. Upon initialization, calls are made to set values in the device driver portion of the DCS implementation; these are parameters to the fairness predicate, as described in Section 4.2.1. A call is also made to cause the device driver to set the variable in the LANai memory that holds the address of the kernel variable that identifies the currently running LWP.

The spin-block message receipt interface adds a call to await the arrival of data to the previously existing interface. The code for the call contains a polling loop. We timed iterations of the polling loop to find how many iterations corresponded to a given spin time. If the message polling call is invoked unsuccessfully for this number of iterations, the call to await data makes the system call into the DCS code in the device driver to block the process. When the message arrives, the network interface processor signals an interrupt which causes the DCS code in the device driver to awaken the process.

Chapter 5

Experimental Results

5.1 Overview

5.1.1 Introduction

In this chapter, we present the results of running mixed parallel and serial jobs on the experimental implementation of dynamic coscheduling described in Chapter 4. Through our experiments, we sought to determine whether dynamic coscheduling could bring about coscheduling in a real system; to determine the effects of different synchronization mechanisms; and to probe the limitations of this method of achieving coscheduling. The parameters we used in measuring the success of our implementation were CPU time, response time (wall-clock time to completion), and fairness.

We ran three parallel applications in separate tests, with varying background workloads. As discussed at greater length in Section 4.1.1, because the current version of Illinois Fast Messages does not allow multiple processes to share a network interface, we were unable to run experiments in which multiple parallel jobs timeshared the cluster; instead our parallel applications timeshared the nodes of the cluster with serial competitors. The applications were a latency test in which two nodes exchanged a token repeatedly, a barrier test in which all the nodes in the cluster ran a simulated SPMD workload, and a two-dimensional Laplace equation solver that used successive over-relaxation.

We found that DCS does indeed coschedule the processes constituting a parallel job in the experiments we ran. If the granularity of communication is relatively fine and spin-block message receipt is used, DCS can achieve essentially perfect results. With spinning message receipt, there is a tradeoff between efficiency and fairness; slight decreases in fairness lead to significant gains in efficiency, but efficiency under timesharing is never quite as good as the ideal case of batch processing. We encountered limitations of DCS: as the granularity of communication increases, with spinning message receipt, efficiency decreases; with spin-block message receipt, response time increases. The limitations show the need for continued work on better coscheduling algorithms; but we conclude that DCS performs well enough to be a useful approach until better algorithms are found.

5.1.2 Goals of the experiments

As mentioned above, our primary goal was to establish whether dynamic coscheduling could in fact achieve coscheduling in a workstation cluster. That is, we wanted to determine whether the approach of scheduling a process immediately when a message arrives can have the effect of coordinating the scheduling of a collection of processes constituting a parallel job across the nodes of a workstation. We wanted particularly to do this in the case of our DCS implementation, but when we found that the effect of the priority boosting upon process wakeup performed by Solaris 2.4 (and some other Unix systems) was sometimes to schedule processes using spin-block synchronization immediately when a message arrived, we also wished to understand the performance of that mechanism and to compare its effects to those of DCS.

In addition, we wished to find the limitations of DCS, and to determine in particular whether the relation between frequency of communication and degree of coscheduling that we had observed in the analysis and simulation described in Chapter 3 could be observed in a real system. We did not measure degree of coscheduling directly, although with hindsight comes the realization that it might have been useful to do so. Instead we observed the performance of the system and histograms of the elapsed time between communication and response to indirectly determine the degree of coscheduling.

Finally, we wished to experimentally modify aspects of our implementation as a result of testing. The most important of these modifications was our selection of a fixed spin time for use in spin-block message receipt, based on our experimental results.

Performance measures

We used three measures of performance: job response time, CPU time, and fairness.

Job response time is the total wall-clock time from the inception of the job until its completion. If perfect fairness could be achieved, and all jobs on the system ran to the completion of the test, then the response time would simply be the number of jobs on the system multiplied by the total (system and user) CPU time consumed by the job.

CPU time is the sum of system and user CPU time used by the job. We report only the sum because in our experiments we found that system time consumed by the jobs we ran was very low, typically less than 1%, and so it would not be visible in the graphs we use to present our results. We sometimes refer to a closely related quantity we call efficiency, which we take to be the ratio of the CPU time consumed by a program under timesharing to the CPU time it consumes in the ideal case where it is the only program running on the machine.

Fairness is the quality of using a fair share of the processor. If the CPU time consumed by a process over its lifetime is called T_C, and its job response time is called T_E, the share of the processor it consumes is T_C/T_E. If N jobs are running on the processor, the ideal fair fraction of the processor for a particular job is $1/N$, and we compute the fairness ratio F as the ratio of the CPU share it consumes to its ideal fair share:

$$F = N\frac{T_C}{T_E} \tag{5.1}$$

Thus if $F = 1$, we have ideal fairness; if $F > 1$, the process is consuming more than its fair share; and if $F < 1$, the process is consuming less than its fair share of the CPU. This relation shows that elapsed time and fairness are linked in a way that will be relevant to our experimental results. In particular, if CPU time is held constant, as it is in the case of comparisons between DCS with spin-block message receipt and the default Solaris 2.4 scheduler with spin-block message receipt, then fairness determines elapsed time to completion of the job, and vice-versa.

In our graphs, we did not attempt to show time consumed by operating system tasks and demons running on the nodes we used. In our latency and barrier tests these tasks accounted for less than 1% of the total elapsed time of the test. In the mixed workload test, these loads were more significant, but typically less than 10%. Further work is needed to measure these loads precisely in the case of the mixed workload test, but for the latency and barrier test we consider them insignificant.

5.1.3 Overview of the experiments

We performed three experiments. The first of these was a latency test, in which two nodes repeatedly exchanged a virtual token; a message was sent from one node to the other, with each node awaiting a reply before sending another message. We used the latency test as a microbenchmark to gain some initial insight into the implementation, in addition to using it as an overall measure of performance.

The second was a barrier test, in which all the nodes of the cluster performed repeated barrier synchronizations, with a variable amount of simulated computation between successive iterations of the test. The barrier test allowed us to evaluate the result of varying the granularity of communication performed by a parallel job.

The third test was the mixed workload test, which sought to use a somewhat realistic mix of processes to determine whether DCS could coschedule a parallel process against competitors that performed I/O, rather than simple spin loops.

In the case of the latency and barrier tests, the background loads were always identical processes that ran a simple busy loop until the end of the experiment. In the case of the Laplace equation solver, the background loads were real application processes, as will be discussed below.

We did not attempt to duplicate in our prototype implementation the experiment we modeled and simulated in Chapter 3. This would have been impossible in any case, because we were unable to run multiple parallel jobs; but additionally, we did not feel that such an experiment was as realistic as those we did perform.

5.1.4 Limitations and restrictions of the experiments

As mentioned above and in Chapter 4, our prototype implementation had two major limitations. The first of these was an inability to run multiple parallel jobs, which is a limitation of the current version of Illinois Fast Messages. This meant that we were unable to evaluate the epoch number mechanism described in Section 3.3, and also that we were unable to discover any unforeseen problems associated with the use of DCS on multiple parallel jobs.

The second limitation of our prototype was that we did not implement a means of automatically achieving fairness. We believe that a straightforward implementation of a feedback control mechanism would have allowed the fairness parameters to be modified with minimal overhead; however, limitations of time did not allow us to pursue this. Instead, we simply manually modified the fairness parameters C and E described in Section 4.2.1 to achieve good fairness. For each set of experimental results we present below, we show the fairness parameter settings we used.

Our cluster was not a large one. As mentioned above, we started with eight nodes, and ended with seven; the results presented here used at most seven nodes. The result is that we were unable to evaluate how DCS scales. Scaling is potentially important; for example, due to propagation delays, DCS might not coschedule a job running on a large cluster using a virtual ring for communication, because the time for a message to travel around the cluster could be larger than a timeslice length.

We did not experiment with varying the timeslice lengths on the cluster. Because DCS boosts the priority of the job to the highest value possible, which as shown in Table 5.2 corresponds to the shortest timeslice length, and we saw in Chapter 3 that the number of messages per timeslice was an important determinant of performance, it is possible that DCS could have benefited from a dispatch table using longer timeslice lengths. We believe that the use of the default dispatcher table also had the effect of exacerbating the inefficiency of spinning message receipt under Solaris 2.4, by causing even processes that started their timeslices in synchrony to suffer long spinning phases when one ends its timeslice before others. However, limitations of time did not allow us to experiment with varying timeslice lengths.

We also used a fixed spin time across all our experiments, and did not attempt to fit it to individual applications. There were several other possible refinements of our techniques which we considered, but did not attempt. These included predictive coscheduling techniques in which processes would attempt to "prefetch" peers with which they often communicated, by sending them messages; as well as a technique in which the residual lifetime of the timeslice would be included in messages and a process scheduled as a result of message receipt would only receive a timeslice length equal to this residual lifetime.

5.1.5 Overview of the experimental results

Our experiments showed that dynamic coscheduling can indeed realize nearly ideal performance in the case of spin-block message receipt and relatively fine-grained loads. In the case of spinning message receipt, DCS also achieves much better coscheduling and efficiency than the default Solaris 2.4 scheduler, but efficiency becomes significantly worse than the ideal case of batch processing unless parallel processes are allowed to use from 1.2 to 1.5 times their ideal fair share of the processor. As mentioned briefly above, we believe this is partly due to the differing timeslice lengths used in the default Solaris 2.4 dispatcher table.

As the granularity of communication increases, DCS does a worse job of coordinating scheduling. In the case of spin-block synchronization, this is manifested as increased latency; in the case of spinning synchronization, it is manifested as decreased efficiency. In both cases DCS continues to do better than the unmodified Solaris 2.4 scheduler, but worse than ideal.

The default Solaris 2.4 scheduler, through priority boosting on process wakeup, is able to accomplish some coscheduling for processes that use spin-block message receipt, but parallel jobs running under it have greater response times than under DCS. Parallel jobs using spinning message receipt under the default Solaris 2.4 scheduler have very poor performance, and do considerably better under DCS, although typically worse than ideal.

The results suggest to us that, while a full implementation of DCS with an automatic fairness mechanism would be a significant improvement over the default Solaris 2.4 scheduler for those running parallel jobs on workstation clusters, further development would be worthwhile, as DCS does not achieve ideal performance in all cases. Chapter 6 will discuss some possible directions for future research in this area.

5.2 Descriptions of the test workloads

5.2.1 Latency test

Our first test was the latency test, which we used initially to gain some insight into the behavior of the implementation. The name of this test is a historical artifact; this test is the direct descendant of one used in the Illinois Fast Messages project to measure user-space to user-space messaging latency.

The latency test is a simple token-passing benchmark in which two nodes repeatedly exchange a 128-byte packet. If the nodes are called node 0 and node 1, then node 0 sends a packet to node 1 and waits for a response; upon receipt of node 0's packet node 1 sends a packet to node 0 and waits for a response, until a specified number of exchanges have been made. The exchanges do not happen as quickly as possible, because the nodes record the wall-clock elapsed time for each round trip, including the sending and receiving of the packet. System calls are performed to get the time of day before and after the round trip, adding approximately three microseconds to the total; also the routines that record the information about the round-trip time perform floating-point arithmetic for rounding so that the elapsed times can be stored in a small data structure. This is why the shortest round-trip times in our tests show up as being approximately 90 μsec rather than 80 μsec.

Figure 5-1 shows the pattern of communications and wait periods in the latency test.

The competitors we used in the latency test were all processes that ran in a simple spin loop for the duration of the test.

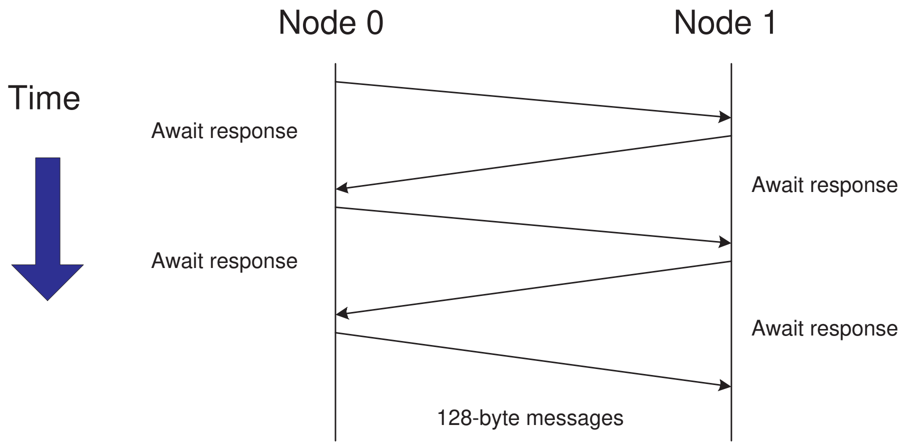

Figure 5-1: Communication and wait periods in the latency test.

5.2.2 Barrier test

The barrier test provides not only a different pattern of communication than does the latency test, but also a different granularity, in that messages are sent less often.

Because the Myrinet hardware does not support broadcast communication, we implemented our barrier test using sequential messages between a root node and six leaf nodes (only seven of our eight nodes were working towards the conclusion of the work described in this thesis). The root node initially broadcast a "pass barrier" message to all the leaves; the leaf nodes would then enter a simple spin loop intended to mimic local computation before each sent an "at barrier" message to the root node. When the root node received all six "at barrier" messages, the loop would begin anew. The algorithm is schematically depicted in Figure 5-2.

Each iteration of the spin loop took approximately 78 nanoseconds, or about three instructions on the processors we used. In most of the tests whose results we present here, $1,000$ delay iterations were used between barrier synchronizations; however, we also ran some tests with larger numbers of delay iterations. $100,000$ barrier synchronizations were performed in each of the experiments.

The competitors we used in the barrier test were all processes that ran in a simple spin loop for the duration of the test.

5.2.3 Mixed workload test

We also ran an MPI FORTRAN application kernel, a two-dimensional Laplace equation solver using a successive over-relaxation technique, using an FM implementation of MPI [16]. Because a rectangular grid of nodes was required, we used only six nodes in the cluster for this test. Each of six nodes in the workstation cluster ran a workload consisting of the applications shown in Table 5.1. Here "SOR" refers to the Laplace equation solver, which performs successive overrelaxation on a 128×128-element matrix. GNU tar is an archiving program that combines a set of files into a single archive. We used a collection of 97 files, totalling 2.1 MB. The `-z` option specifies that

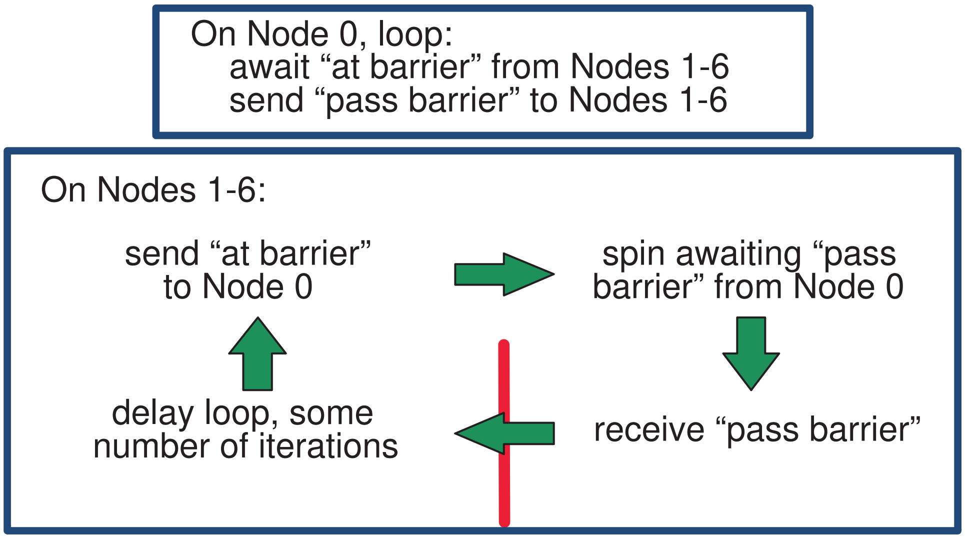

Figure 5-2: Barrier test performed by six leaf nodes and one root node. Delay loop simulates computation.

Program	Command line
SOR	`sor`
GNU tar (+ GNU zip)	`gtar -czhvf /dev/null /usr/local/Gnu/lib/gnuemacs/etc`
Ghostscript	`gs -q -dNODISPLAY -dNOPAUSE inputfiles/pakin-ms.ps inputfiles/quit.ps`

Table 5.1: Applications used in the mixed workload benchmark

GNU zip, a compression program, should be run on the result. Finally, Ghostscript is a PostScript interpreter. The input file is a 1.7 MB, 103-page PostScript file.

All files were read from a remote NFS filesystem.

5.3 Experimental results

5.3.1 Latency test results

We show the response time of the latency test with and without DCS, and under spinning and spin-block synchronization, in Figure 5-3. In all the graphs we will present, the mean of several runs is shown, and we show 90% confidence intervals computed using Student's T-distribution; however, sometimes the confidence intervals are too small to be seen on the graph.

The most obvious feature of the graph is that spinning message receipt without DCS performs very poorly, taking enough additional time to complete so that a log

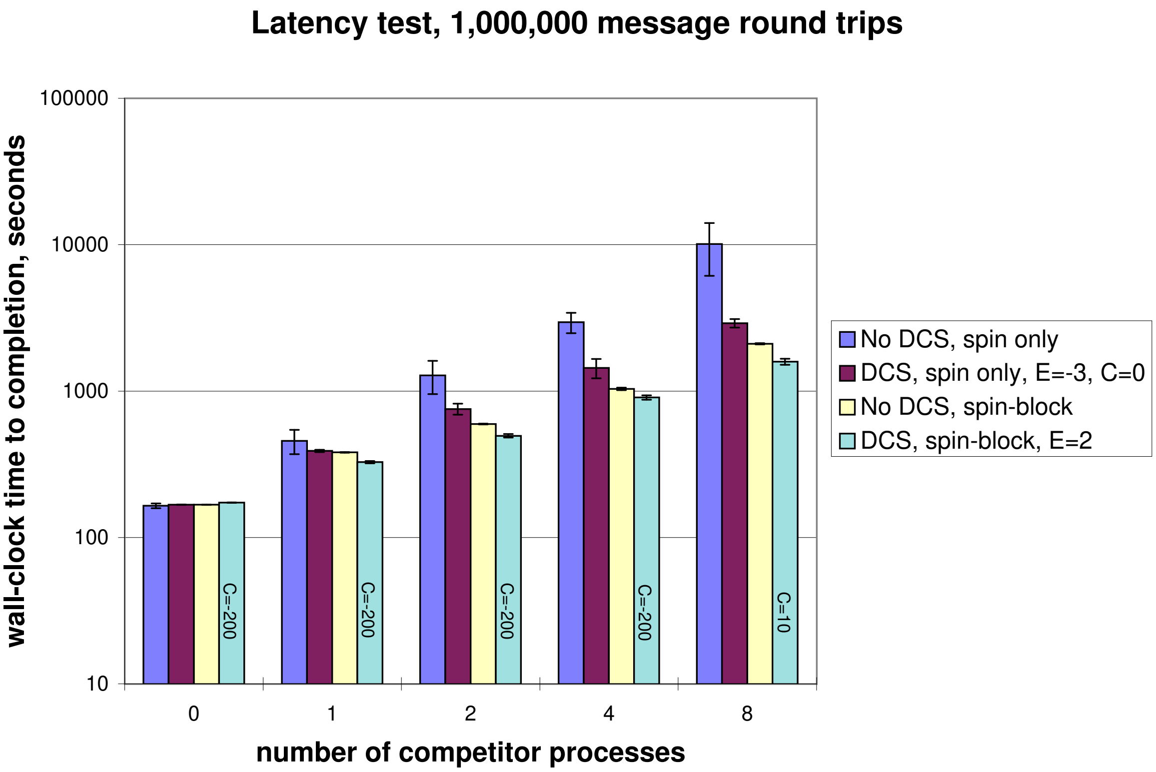

Figure 5-3: Total wall-clock time consumed in the latency test under spinning message receipt and spin-block message receipt, with and without DCS. In this and in all other graphs presented here, 90% confidence intervals computed using Student's T-distribution are shown.

scale has been used here to allow this case to be depicted on the same graph as the others.

Response time for the latency test with spinning message receipt under DCS is considerably better, but much worse than with spin-block message receipt either with or without DCS. As we will see below, there is a tradeoff here; performance with spinning message receipt under DCS can be improved significantly for a small additional penalty in fairness; but spinning message receipt is clearly a poor choice in this experiment under either the unmodified Solaris 2.4 scheduler or DCS. As mentioned above, we believe this is partly due to the differing timeslice lengths used in the Solaris scheduler.

Before dismissing spinning message receipt althogether, however, let us note that it is an important synchronization method for parallel computations on workstation

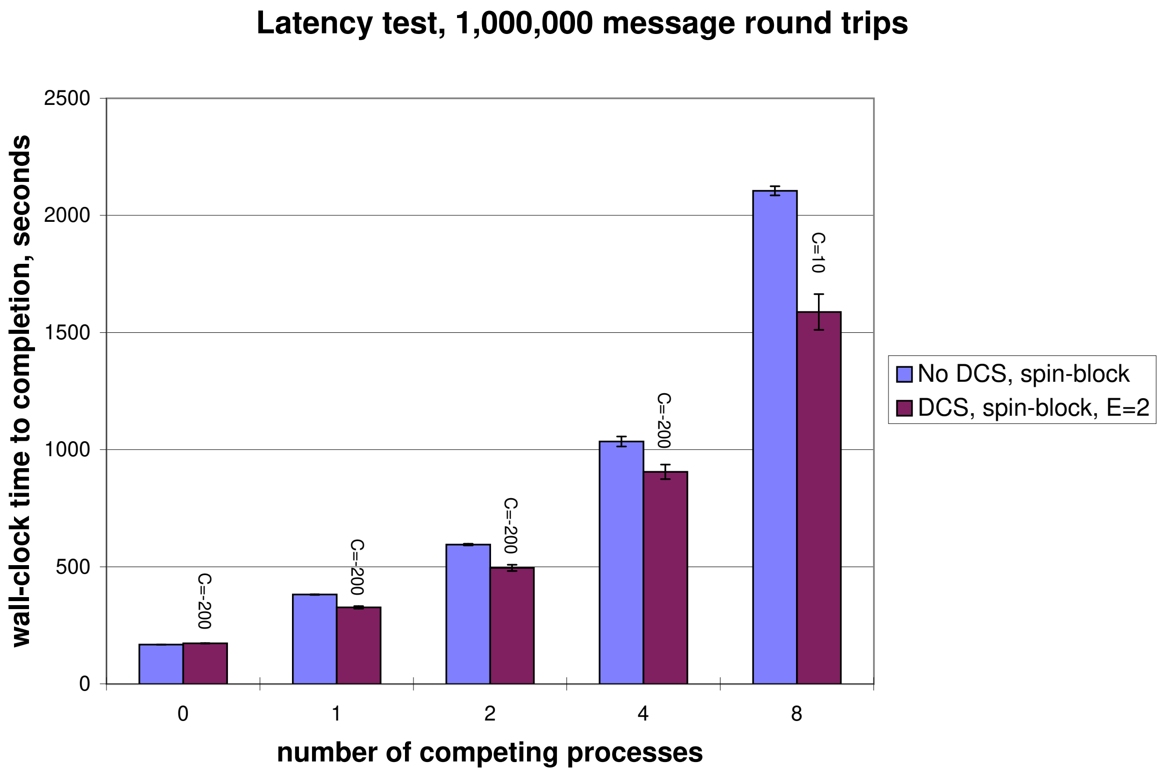

Figure 5-4: Total wall-clock time consumed in the latency test under spin-block message receipt, with and without DCS. The data are the same as in Figure 5-3, but presented here with a linear scale to show detail.

clusters. Many parallel programs are written to use polling for message receipt; some of these may perform other work while awaiting the arrival of messages. As we have mentioned above, these programs will not be coscheduled under the normal Solaris 2.4 scheduler, because they do not block. Our tests with spinning message receipt give some notion of the degree of response time penalty these programs will suffer as a result — the penalty is quite severe. However, DCS will attempt to coschedule programs whether they use spinning or spin-block message receipt; thus it is a better default scheduling mechanism for parallel programs than the normal Solaris 2.4 scheduler.

That said, the choice on which we shall mainly concentrate is spin-block message receipt, with and without DCS. In order to show detail that cannot be seen on a log scale, we depict again in Figure 5-4 the spin-block cases shown in Figure 5-3.

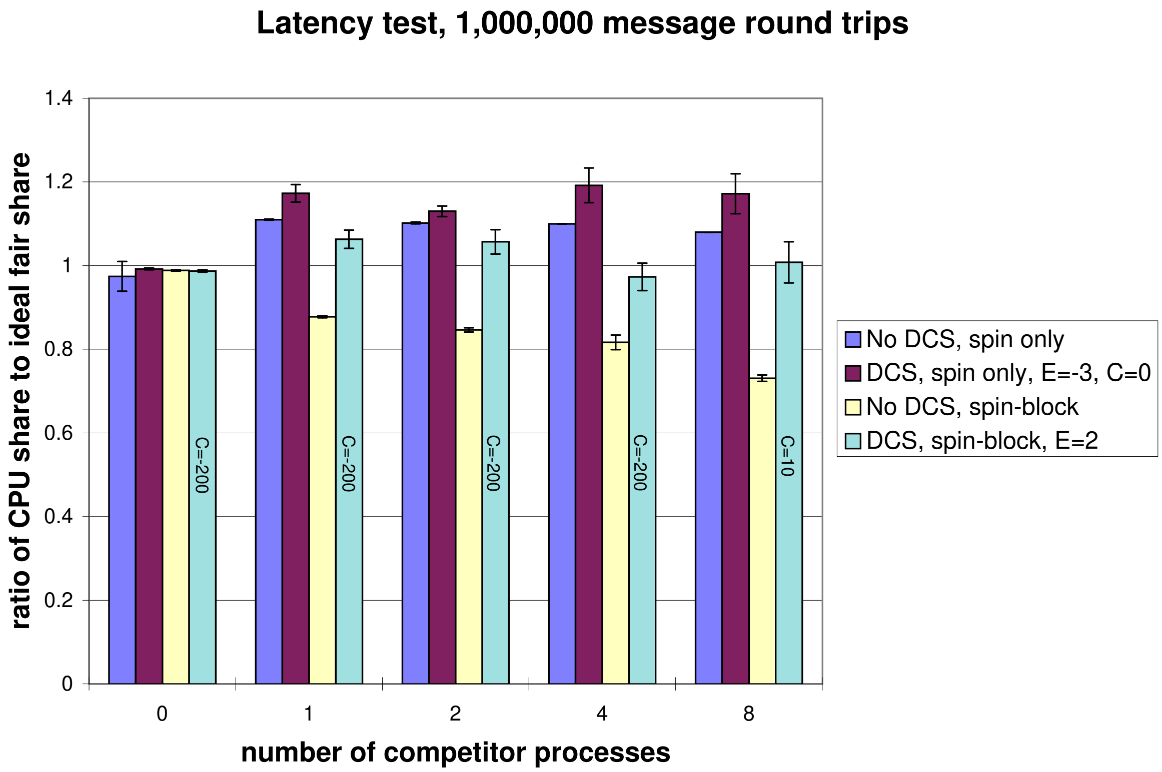

Figure 5-5: Fairness in the latency test. The value shown is the ratio of the fraction of the CPU used by the parallel process to its ideal fair share of the CPU.

Interpretation of the results: fairness and performance

We show the fraction of ideal CPU time shares consumed on one node by the latency process in Figure 5-5. The value plotted in the case of n competitor processes is the mean CPU fraction used by the process, including both user and system CPU time, divided by the ideally fair CPU share $1/(n+1)$. If this value is greater than 1, the process has used more than its fair share of the CPU; if it is less than 1, the process has used less than its fair share.

The results for the case of spinning message receipt are straightforward to understand. DCS with the fairness parameters $E = -3, C = 0$ is less fair than the unmodified scheduler, using slightly less than 20% more than its fair share in the worst case, so that with 8 competitors it was using 13% of the CPU rather than the 11.1% to which it was entitled. However, examination of the graph in Figure 5-6 reveals that in this case the latency test running under DCS completes in 379 CPU-

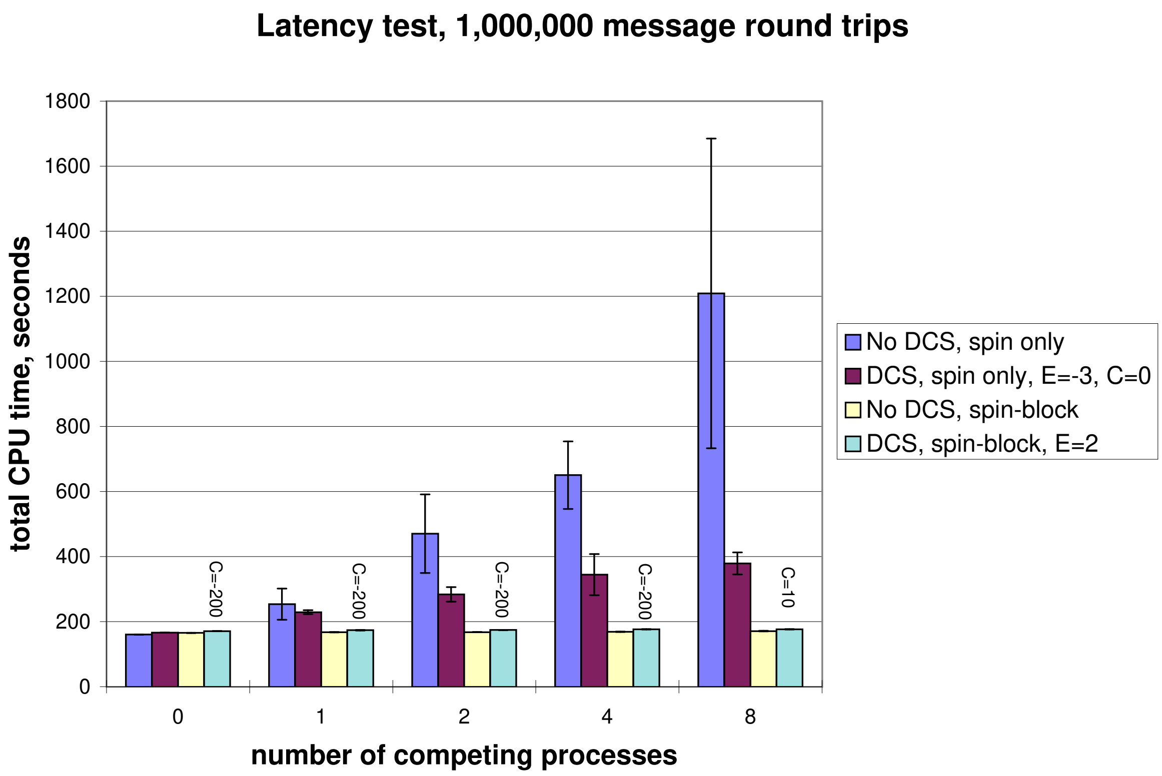

Figure 5-6: CPU usage in the latency test. The value shown is the total of user and system CPU time, but the system CPU time was in all cases less than 1% of the total.

seconds, rather than the 1209 CPU-seconds taken by the latency test running under the unmodified Solaris 2.4 scheduler, so that it is much more efficient.

However, there is a tradeoff here between fairness and efficiency. We can drive the efficiency of the latency test with spinning synchronization up considerably by using the parameters $E = -2, C = 0$, so that it completes in only 289.8 CPU-seconds; however, this requires using 14.5% of the CPU instead of the 11.1% to which the process is entitled. Clearly the "equalization" we are performing here by refusing to preempt when it would be too unfair to do so is causing us to suffer poorer coscheduling. This phenomenon was predicted by our simulation, described in Section 3.2. An interesting question for future inquiry is whether we could increase efficiency in the case of spinning synchronization without hurting fairness, possibly by matching timeslice lengths for the parallel jobs, by paying more attention to which messages are discarded and which are not, or by performing some sort of predictive coscheduling.

In summary, though, while we would prefer greater efficiency, in every run we performed, DCS was much more efficient for the case of spinning synchronization than the unmodified Solaris 2.4 scheduler, and is clearly preferable to it.

The issue of fairness is more complex in the case of spin-block synchronization. This is because CPU time is not significantly different with and without DCS under spin-block synchronization for the tests we ran. As can be seen in Figure 5-6, both are essentially the same as for the same program run in batch mode, because the amount of spinning performed under the two schedulers is limited, so that the unmodified Solaris 2.4 scheduler is already essentially perfectly efficient for programs using spin-block message receipt. As described in Chapter 2, other researchers have reported increased context-switch and cache-reload times due to poor coscheduling, but, perhaps because spin-block message receipt both with and without DCS results in some coscheduling under Solaris 2.4, we were unable to find any significant difference under spin-block message receipt between CPU times for the ideal case of 0 competitors and under timesharing.

Therefore, under spin-block message receipt DCS cannot make the program more efficient; it can only change the execution order of jobs on the processor so that the program completes sooner. As can be seen from Equation 5.1, any reduction in the amount of time it takes for the program to complete is time taken from competitor processes, so our goal is for DCS to change the execution order in such a way that the parallel program receives exactly its fair share of the CPU. Because in the latency test DCS is able to come closer to doing this than the unmodified Solaris 2.4 scheduler, which spends more time blocked awaiting message arrivals, its elapsed time to completion is better.

Round-trip times with spinning message receipt in the latency test

We show the wall-clock round-trip times achieved in the latency test with spinning message receipt, with and without DCS, in Figure 5-7. In each of several runs of the experiment, $1,000,000$ messages were sent by each of the two nodes. Each message

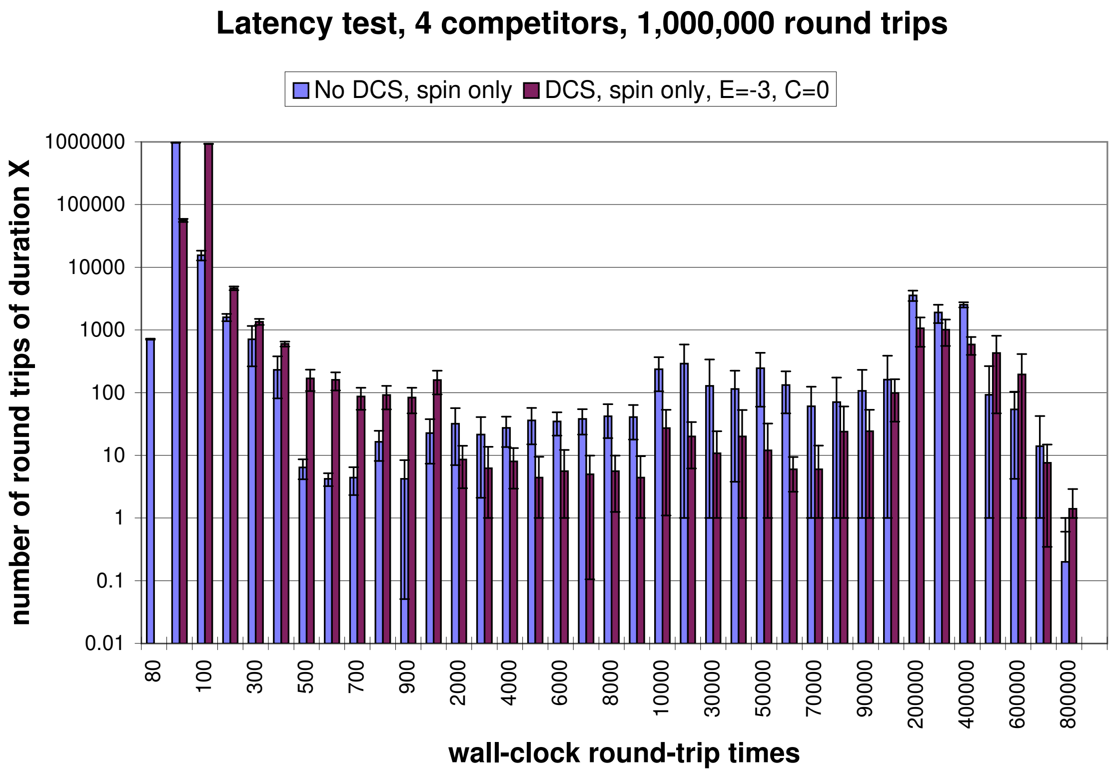

Figure 5-7: Histogram of number of message round trips taking a given time to complete in the latency test, with four competing processes and spinning message receipt.

was 128 bytes long, which is the maximum length that a message can have in this version of FM without necessitating scatter-gather operations. Four competing jobs were run on each of the two nodes; these competing jobs were simple spin loops that ran for the duration of the experiment. The mean round-trip times seen on one of the nodes were recorded. A log scale is used on the vertical axis because of the widely varying numbers of messages falling into each category.

It is worth noting that some of the features of the graph are artifacts of the data collection method, which was intended to be fast (so as not to distort results) and to use only a small amount of memory. Specifically, the buckets into which incidences are gathered have one significant digit each, and rounding is performed, with incidences being placed in the most precise bucket possible. Thus the bucket labeled 900 contains all incidences with round-trip times between 851 microseconds and 950 microseconds; but the bucket labeled 1000 contains all incidences with round-trip times between 951 and 1500 microseconds — a much larger range into which more incidences fall, showing up as a significant jump in the graph.

It can be seen that our implementation of DCS slowed down the best mean message round-trip time by about 10 microseconds, or 5 microseconds per message. This is because of several extra instructions performed on message receipt under DCS in the LANAI control program control loop.[1] The overall effect is to add some 5 seconds of execution time to the base case of 0 competitors, as will be seen in Figure 5-3. However, if we examine job response time under timesharing, the better coscheduling of DCS more than makes up for this additional CPU time, as can be seen in Figure 5-6.

This improved job response time results from DCS's significant reduction of the number of very long delays. The effect of these delays can be seen in Figure 5-8, where the number of round trips falling into each category has been multiplied by the length of the round trip in question. We see that most of the time in the case where DCS was not used is expended in very long delays.

Selecting a maximum spin time for spin-block message receipt

Returning to Figure 5-7, the context switches that result under DCS from sending messages to a descheduled process can be seen when one examines the number of messages with round-trip times in the 500 μsec to 1500 μsec range. We see that under DCS there are an increased number of such round-trip times, and a decreased number of long delays on the order of 2 msec or more, and to a larger extent, those of 20 msec or more. This is particularly significant because 20 msec is the minimum timeslice length under Solaris 2.4 with the default dispatch table; thus we see that DCS limits the number of round trips in which the sender has to wait a full timeslice length or more for a response.

The 500 μsec to 1500 μsec round-trip times, on the other hand, correspond well to the context switch times we found empirically for the processors we were using, when cache reload effects were taken into account. These times varied between 400 μsec

[1] If we had used Myricom's newer RISC-based interface boards, rather than the older CISC-based boards we used, this extra time would have been considerably smaller.

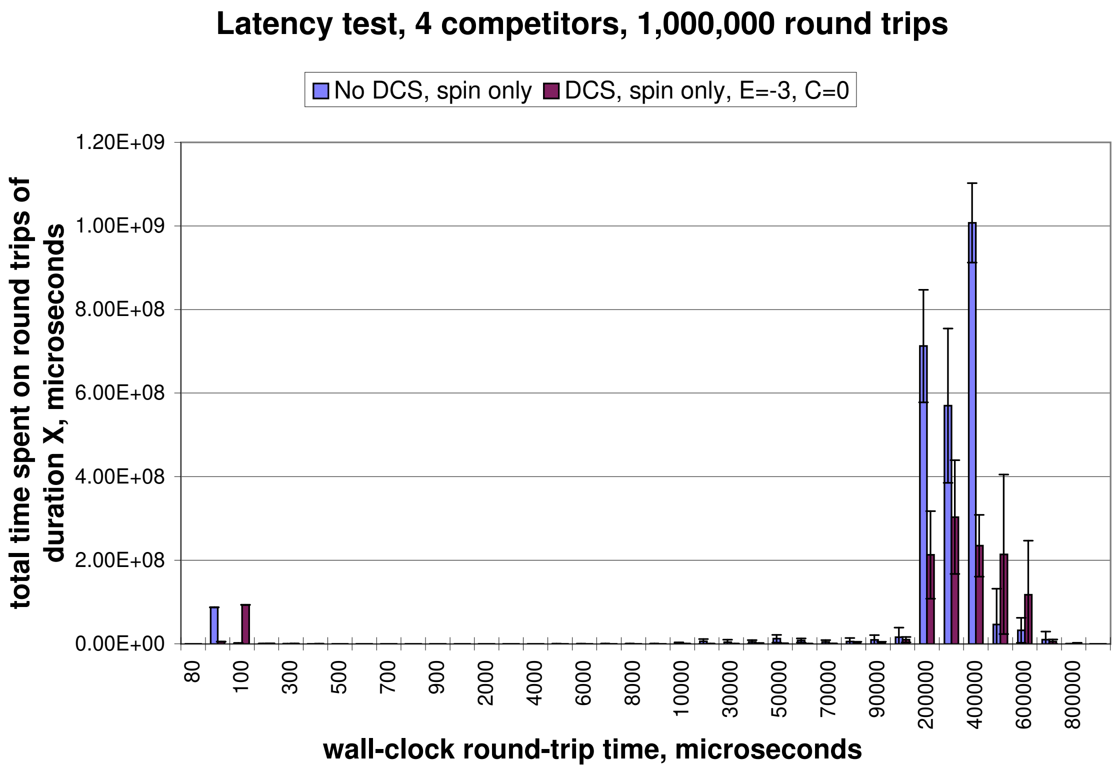

Figure 5-8: Histogram of total wall clock time consumed in message round trips taking a given time in the latency test, for the case of four competing processes and spinning message receipt. This is the same experiment as is shown in Figure 5-7; one can see that the long delays are responsible for most of the time consumed by spinning message receipt without DCS.

and 1400 μsec, with a mean somewhere around 700 μsec. We tried varying fixed spin times between 200 μsec and 3000 μsec in runs of the barrier test, with and without DCS; while performance degraded at the extremes, it was not very sensitive to changes in the range of values from 1 to 2 milliseconds. Within the resolution of the experiment, the fixed spin time we chose provided the best performance for both spin-block with DCS and spin-block without DCS; but the differences were small.

Thus we chose our maximum spin time of 1600 μsec based on the empirical evidence of our experiments, which showed us that the maximum delay we saw for response in the case where a context switch was required was approximately 1500 μsec. In fact, we may have been overly conservative in our choice, because we did not gather data at a sufficiently fine granularity to allow us to determine where in the range 951 μsec to 1500 μsec the cutoff actually occurs, but the choice of 1600μsec would avoid edge effects in which we would often switch just before the message arrived, and, as mentioned above, it gave us the best barrier-test times we saw, within the resolution of the experiment.

It is also to be noted that 1600 μsec is slightly greater than twice the mean context-switch time plus the message round-trip time. Ousterhout claims in [18] that a two-context-switch fixed spin time is competitive (he calls the spin time the *pause* in his description of two-phase *waiting*). The competitive arguments presented in [14] can be used to show that this spin time is indeed competitive, with a competitive ratio of at worst $3+M/C$ times the optimal spin time, for M the message round-trip time and C the context-switch time. This worst-case performance is of course worse than one could do with a spin time equal to the context-switch time. However, as Karlin *et al.* note in [14], the competitive ratio says nothing about the mean cost of spinning, which we found to be higher with a spin time of approximately the context switch time than with the 1600 μsec time we picked.

Dusseau *et al.* also argue for this fixed spin time in [6], on the basis that two context-switch times might be required for a processor to respond to a message if the message arrives at the beginning of a context switch to a process that is not the one to which the message is directed.

Latency test results with spin-block message receipt

Figure 5-9 shows the round trip times for spin-block message receipt with and without DCS. This is the same experiment as the one whose results are depicted in Figure 5-7, with only the exception that spin-block message receipt is used, rather than spinning message receipt.

Once again, DCS reduces the number of very long delays of 20 msec or more for round trip message exchanges, and instead has more delays from about 500 μsec to about 1500 μsec.

The effects of the choice of scheduling algorithm and message-receipt policy on elapsed time to completion of the job can be seen more clearly in Figure 5-10. As in Figure 5-9, messages have been categorized according to rounded round-trip time, but in this case the number of messages in each category has been multiplied by the round-trip time. The result is the amount of wall-clock time consumed by messages

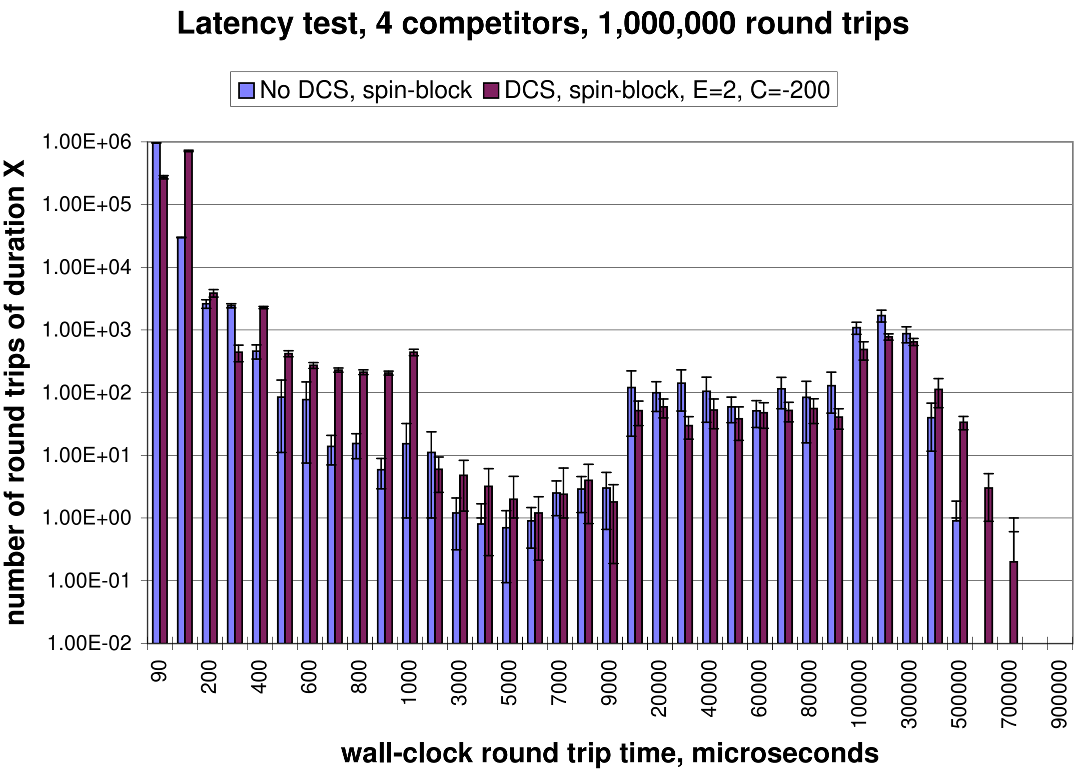

Figure 5-9: Histogram of number of message round trips taking a given time to complete in the latency test, with four competing processes and 1600 μsec maximum spin times on spin-block message receipt.

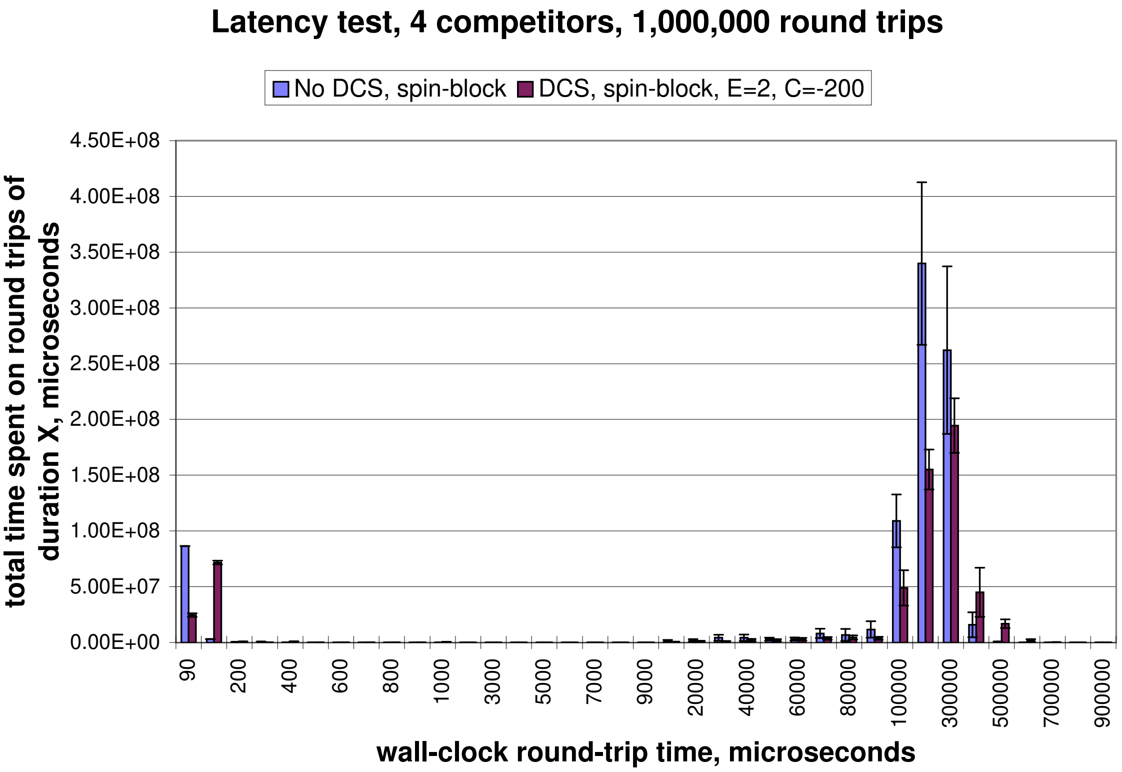

Figure 5-10: Histogram of total wall clock time consumed in message round trips taking a given time in the latency test, for the case of four competing processes and spin-block message receipt. This is the same experiment as shown in Figure 5-9; one can see that the long delays are responsible for most of the time consumed in this test.

falling into each round-trip-time category. It can be seen that the very long delays of more than 20 msec consumed most of the elapsed time in both cases where DCS was not used. Where DCS was used, however, the coordinated scheduling of the two nodes led to far fewer messages suffering very long delays.

5.3.2 Barrier test results

As described in Section 5.2.2, these experiments were run on a seven-node cluster, due to the failure of one of the nodes in our original eight-node cluster. We ran the same number of competing processes on each of the seven nodes. The results of the performance tests are shown in Figure 5-11. We use a logarithmic scale because the time to completion of the barrier test with spinning synchronization and without DCS is very large.

The times shown here are those experienced on the root node. However, these times are representative of those on the leaves as well, because the barrier test requires repeated synchronizations between the root node and the leaf nodes and so the root and leaves complete at the same times (except for message transmission delays).

Barrier test results in the case of spinning message receipt

It can be seen from Figures 5-11 and 5-12 that, as in the case of the latency test, spinning message receipt without DCS results in both decreased efficiency and increased response time. With DCS, efficiency is drastically improved by comparison with the case where DCS is not used, but suffers by comparison with spin-block message receipt. The same tradeoff applies as in the latency test: we were able to drive up efficiency in other barrier test experiments (not shown here) by decreasing fairness.

It is interesting that the results for spinning synchronization without DCS are so much worse in the barrier test than in the latency test. One possible reason for this is the increased probability as the number of nodes increases that some node will not be scheduled simultaneously with the others because of a fairness constraint. We also conjecture that the differing timeslice lengths used in the Solaris 2.4 scheduler (shown in Table 5.2) mean that with an increased number of nodes, the probability that the participants have the same timslice lengths when running decreases, and so the time spent spinning after a peer's shorter timeslice has expired increases. This effect will obtain to some extent for DCS with spinning message receipt, but here it is most likely that only one of the nodes (the one that awakened first) has a different timeslice length than the others, which will have been awakened by message receipts and will run at the highest priority, and so will all have 20-millisecond timeslices

Also interesting is the fact that, in the case of spinning message receipt without DCS, CPU time decreases slightly as the number of competitor processes increases. We believe that priority compression is the reason for this. As the number of runnable jobs increases, their mean priority also increases, because priorities are increased once per second of wall-clock time, but are decayed only on timeslice expirations, which happen less frequently as load increases. If the mean priority of jobs increases, then in Solaris 2.4 this means that their mean timeslice length decreases, as shown in

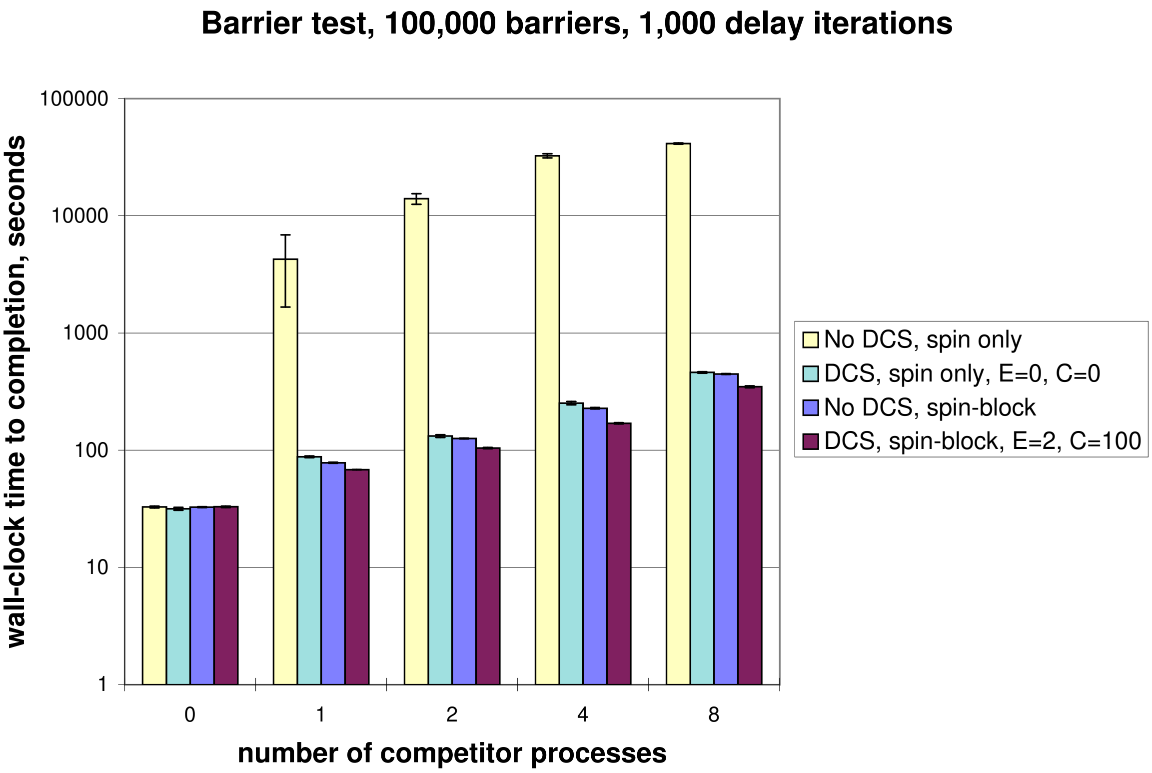

Figure 5-11: Barrier test wall-clock times to completion under a variety of scheduling and synchronization methods. Loads were balanced; 1,000 delay iterations totalling 78 μsec were performed by all nodes between successive barriers.

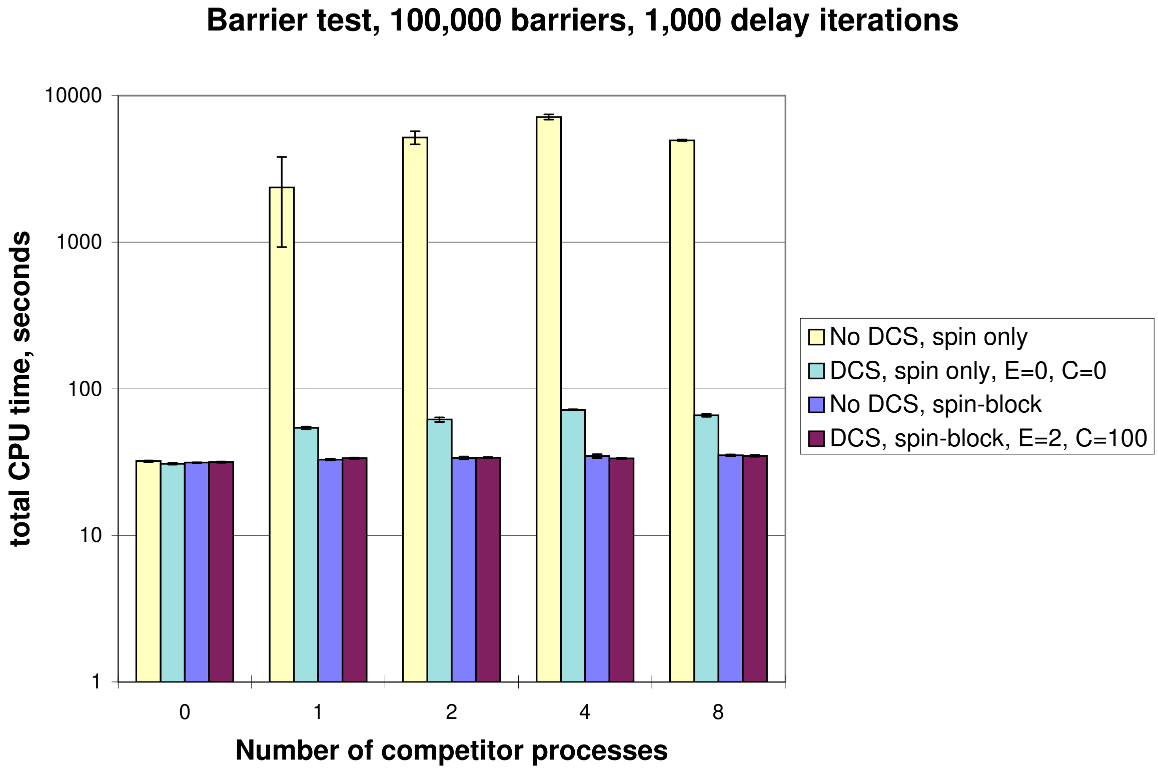

Figure 5-12: Barrier test CPU usage for the experiment of Figure 5-11. The value shown is the total of user and system CPU time, but the system CPU time was in all cases less than 1% of the total.

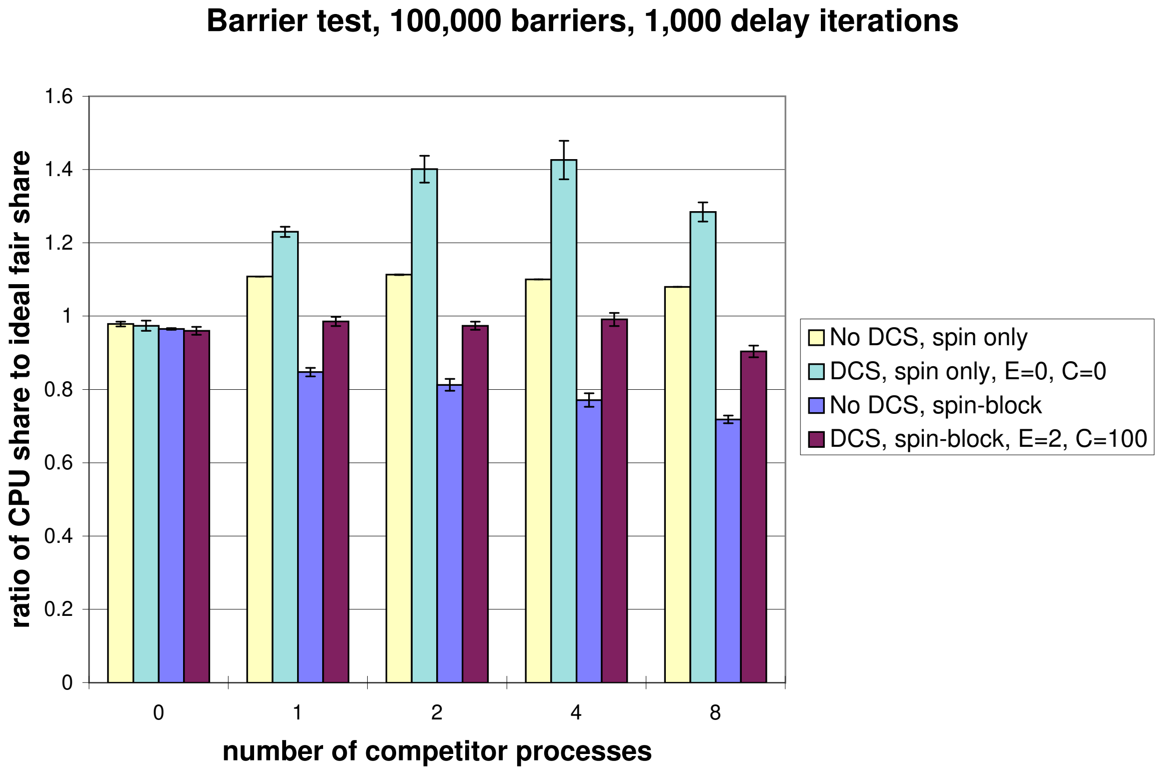

Figure 5-13: Fairness in the barrier test experiment of Figure 5-11. The value shown is the ratio of the fraction of the CPU used by the parallel process to its ideal fair share of the CPU.

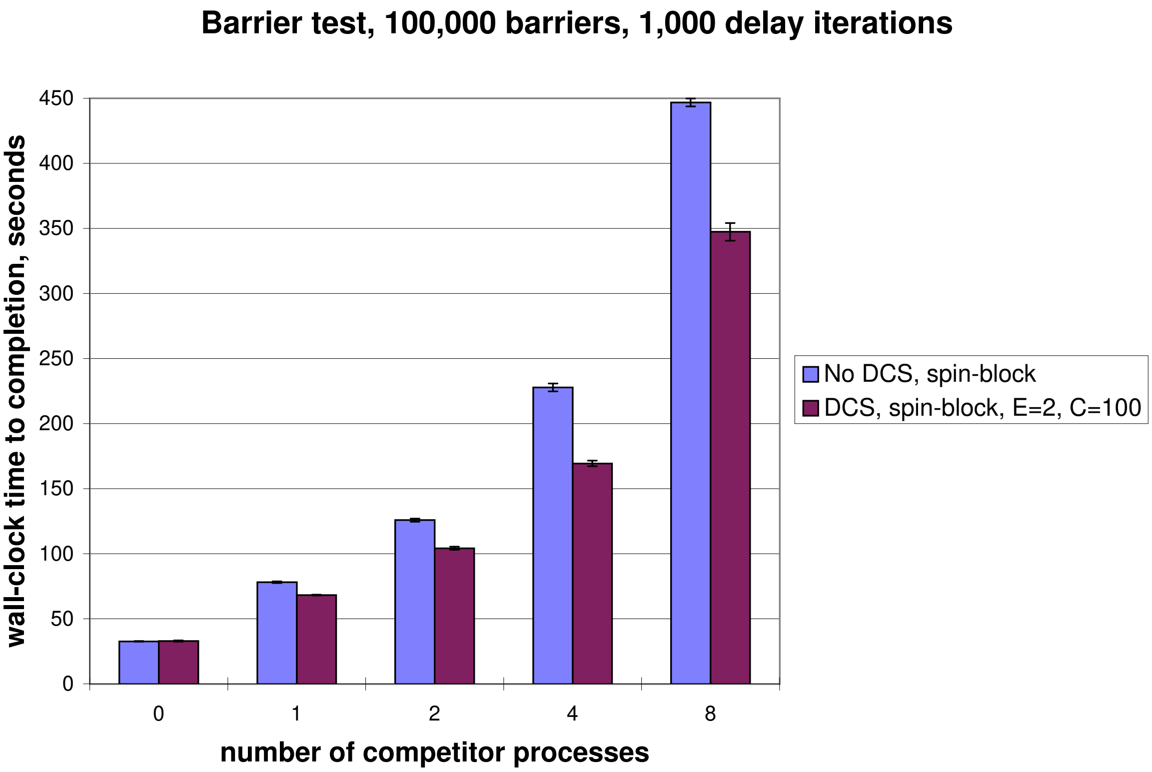

Figure 5-14: Wall-clock times to completion in the barrier test with spin-block message receipt only. These are the same results presented in Figure 5-11, but with a linear scale so more detail can be seen.

Table 5.2. As a result, the length of time for which a process will spin before yielding the processor to other jobs decreases.

Results in the case of spin-block message receipt

The wall-clock times to completion in the case of spin-block message receipt only can be seen repeated in Figure 5-14, where more detail is visible than was possible with the logarithmic scale of Figure 5-11. As can be seen from the fairness results in Figure 5-13, it would have been possible to improve somewhat on the DCS results for the 8-competitor case by using more aggressive fairness parameters, but we did not pursue the search.

As with the latency test, it can be seen in Figure 5-12 that both cases of spin-block message receipt, with and without DCS, are completely efficient: neither uses

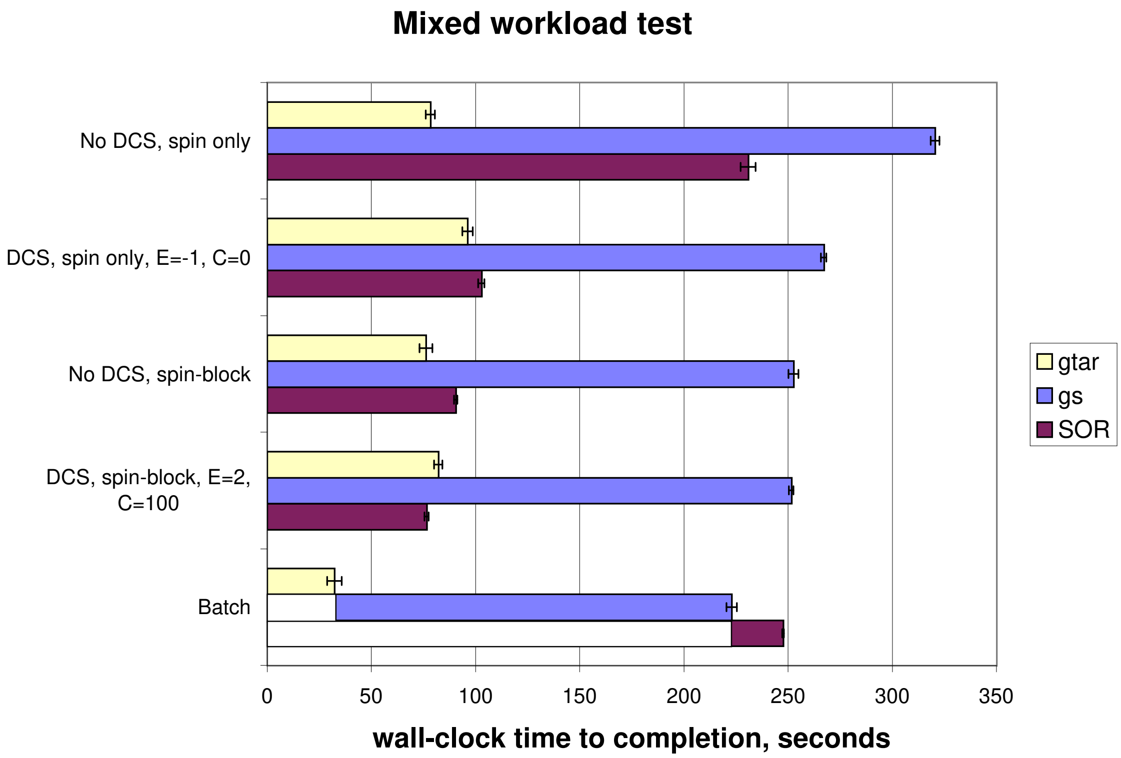

Figure 5-15: Wall-clock times to completion in the mixed workload test. In the batch case, jobs were executed in sequence; bars show time from the beginning of the entire test to the completion of the job.

significantly more CPU time than the 0-competitor case. Also as with the latency results, DCS achieves better wall-clock times to completion by using its full share of the machine, or close to its full share of the machine.

5.3.3 Mixed workload test results

Figure 5-15 shows the wall-clock time to completion of each job in the mixed workload test. "Batch" represents the time needed to run each job in turn; the bars in this case show the time from the beginning of the experiment, when all three jobs were submitted to the batch queue, to the completion of the job in question. In the cases of spin-block and spinning message receipt with and without DCS, the jobs actually ran under timesharing, in competition with each other. In all cases, the length of the longest bar indicates the time to completion of the entire workload.

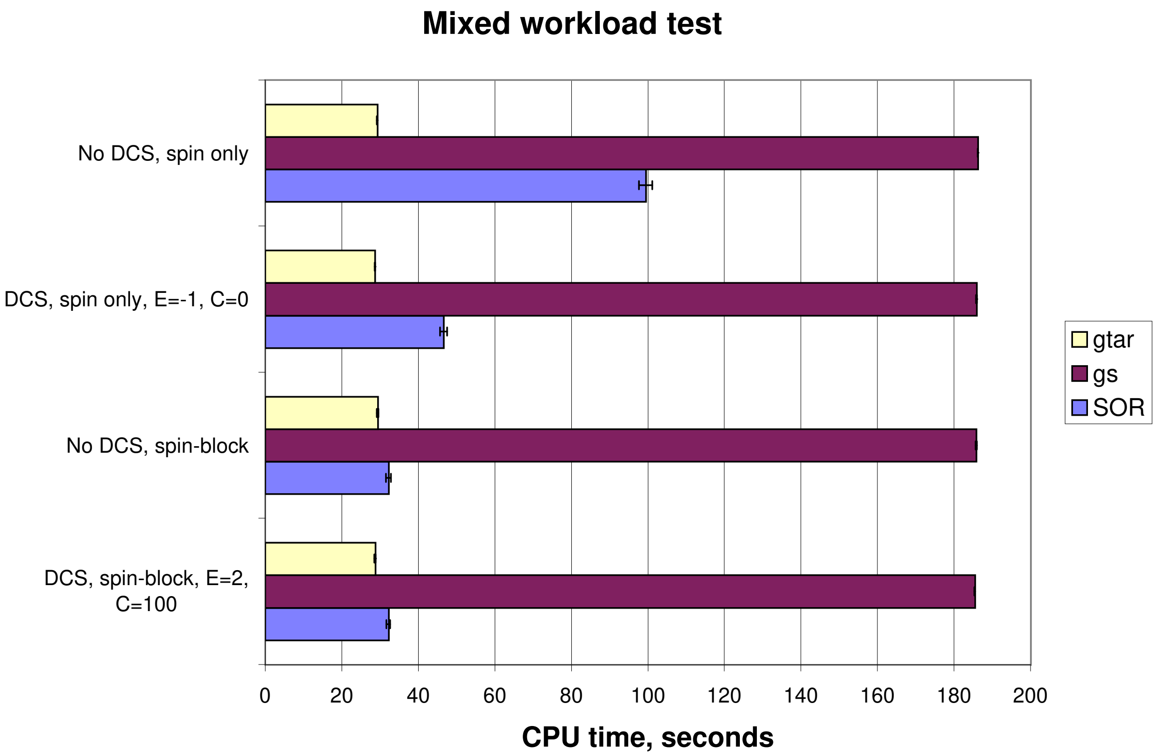

Figure 5-16: CPU times in the mixed workload test.

It can be seen that the major difference in overall elapsed times is between the case of spinning message receipt and the default Solaris 2.4 scheduler, and all the other cases. This is reflected in the CPU times consumed by the jobs in the workload, shown in Figure 5-16. Without DCS or spin-block message receipt, the parallel job took about twice as many CPU-seconds to complete as with DCS.

The combination of DCS and spin-block message receipt took very slightly less CPU time than spin-block message receipt alone; also it required very slightly less overall time to complete, both of these effects presumably being the result of fewer context switches and reduced cache contention due to better coscheduling. The most significant advantage of using DCS with spin-block message receipt in this case, as in the other tests we have seen, is that it reduces latency for the parallel job over the use of spin-block message receipt alone.

The fairness results, shown in Figure 5-17, also echo the results of previous experiments. It can be seen that it is possible with DCS and spin-block message receipt to give the parallel job its fair share of the CPU, whereas spin-block message receipt

Figure 5-17: Fairness in the mixed workload test. The value shown is the ratio of the fraction of the CPU used by the parallel process to its ideal fair share of the CPU.

alone causes the parallel job to be a weak competitor for the CPU. It can also be seen that it was necessary to give the spin-only job about 20% more than its fair share of the CPU to allow it to coschedule efficiently.

5.4 Further analysis

Our initial experimental results raised a number of questions, which we sought to answer in further experiments. The most salient of these questions was raised by the unexpectedly good performance of parallel programs using spin-block message receipt under Solaris 2.4; we sought to find the reasons for this in a series of experiments. We were also interested in the effects of varying the granularity of communication, and of using the Unix `nice()` command to boost base priorities of parallel programs under the unmodified Solaris 2.4 scheduler.

Each of these issues is examined below.

5.4.1 Spin-block message receipt under Solaris 2.4

Prior work

Other researchers [10, 23] have reported improved response times and efficiency with spin-block synchronization with no active coscheduling mechanism, and the reasons for this are clear: with spin-block synchronization, CPU time that would otherwise be wasted in spinning can be used by other processes present on the node. However, as we mentioned in Section 2.5, Dusseau *et al.* reported in [6] that SPMD programs that used fixed spin times and spin-block message receipt when running on a simulated workstation cluster under the Solaris 2.4 scheduler had performance that was within a factor of two of that found under an idealized gang scheduler. Dusseau *et al.* attributed this relatively good performance to the priority-boosting behavior of the Solaris 2.4 scheduler in [6].

Their attribution of this performance describes coordinated scheduling arising as follows. First, a process awaiting a message from a descheduled process on another node will block when its maximum spin time has elapsed. Then, when the descheduled process is scheduled and sends its message, on the receiving node, the interrupt routine for the network interface will unblock the receiving process, and the operating system will move it from a sleep queue to a run queue. In [6], the next step is described as one in which the newly-awakened process receives a significant priority boost from the Solaris 2.4 scheduler; although, as we shall see below, this is not invariably true. Finally, the dispatcher schedules the highest-priority job in the system; because of the priority boost, this is probably the newly-awakened process. Now the newly-awakened process begins a timeslice in near-synchrony with the process that sent it a message.

Note that this is a description of achieving coscheduling by running a process immediately when a message arrives; that is, what we have called dynamic coscheduling. It is not a description of all of dynamic coscheduling: if the process has been preempted while spinning and is not on a sleep queue, this mechanism will not cause it to be scheduled when the message arrives. Because this happens more often with coarse-grain programs than with fine-grain programs, the performance of priority boosting on process wakeup is also worse than that of a full implementation of DCS on more coarse-grained programs, as will be seen in Section 5.4.3. Also if the process is one that receives messages only by polling, whether periodically during computation or in a spin loop, then this mechanism will not be invoked, whereas DCS will still coschedule such processes. However, for the case of spin-block message receipt with fine-grain message-passing and short spin times, where it is quite unlikely that the process has been preempted while runnable but spinning, this mechanism as described does indeed implement the most significant part of dynamic coscheduling.[2]

[2]Although our earlier paper [22] on dynamic coscheduling is briefly cited in [6], Dusseau *et al.* apparently failed to notice that achieving coscheduling by scheduling a process when a message

	Prio.	Quantum (ms)	ts_slpret
lowest priority	0– 9	200	50
	10–19	160	51
	20–29	120	52
	30–34	80	53
	35–39	80	54
	40–44	40	55
	45–49	40	56
	50–54	40	57
	55–58	40	58
highest priority	59	20	59

Table 5.2: Default Solaris 2.4 dispatch table

As we mentioned briefly above, a newly-awakened process does not invariably receive a priority boost under Solaris 2.4.[3] The actual behavior is slightly more complex. Once per second, the scheduler routine `ts_update` runs. This routine increases the priority of processes on run queues, but which are not running. It does so by incrementing and examining a per-process counter called `dispwait` that is zeroed whenever a process begins a new timeslice; if `dispwait` exceeds a value specified in the dispatcher table, the priority of the process is boosted by a value also specified in the dispatch table. However, `dispwait` serves a dual purpose. The counter is also incremented for processes that are blocked and therefore on sleep queues. When a process is returned to a run queue, if the counter is nonzero, the priority of the process is boosted (typically quite substantially) to the value `ts_slpret` specified in the dispatcher table shown in Table 5.2.

Thus we see that the priority boost will probably not happen for processes that have been blocked for only a short while when the message for which they are waiting arrives. This is because, when the message arrives, an interrupt occurs and the interrupt routine immediately removes the job from the sleep queue and places it on a run queue; the awakened job only receives the priority boost if `ts_update` has run while it was asleep.

Nonetheless, as we saw in Section 5.3.1, even with this only occasional priority

---

destined to it arrives, which is the phenomenon underlying the relatively good performance of parallel processes using spin-block message receipt under Solaris 2.4, is first proposed and analyzed in our work.

[3]We had also believed that this was the behavior of the Solaris 2.4 scheduler, because, in additions to the description in [6], the Solaris 2.4 time-sharing dispatcher parameter table manual page (`ts_dptbl`) [13] describes it this way; but a series of experiments we undertook to show that such behavior would allow a user-mode program to receive an unfairly large proportion of CPU time failed to demonstrate that the priority boosting happened invariably, and a reading of the sources showed how the mechanism actually works.

boosting on process wakeup, the performance of parallel processes using spin-block message receipt under Solaris 2.4 is relatively good. It is indeed remarkable that the priority-boosting mechanism that has been present in Unix at least since 4.3BSD and which was originally intended to enhance responsiveness for serial interactive processes (as described in [17]) has the effect of coscheduling communicating processes on separate workstations.

Passive coscheduling

We conjectured that a particular behavior, which we called *passive coscheduling*, might arise in systems where spin-block message receipt was used without any active coscheduling mechanism. The scenario we envisioned was as follows: if the two communicating processes did not begin their timeslices at nearly the same time, then after the spin time the one that started first would block, waiting for the other to be rescheduled and respond. Because the response would not awaken the first process immediately, the second would block, and so on until they started within a spin period of each other, when they would run together for a timeslice. If scheduling quanta were very long, one would expect the passive coscheduling effect to be enhanced, as the overhead of the multiple very short timeslices in which the two processes sent only a single message, spun, and blocked would be amortized over the occasional very long timeslices in which the processes started in near-synchrony. Passive coscheduling would also work better if all the processes running on the multiprocessor were parallel processes with fine-grain communication, because then none of them would run unless coscheduled. However, if there was no active coscheduling mechanism, then outside of these special circumstances we expected to see the effect decline quickly as the number of competing processes increased, because then it would take longer for the case in which the processes started in near-synchrony to arise.

We believe it is possible that passive coscheduling may account for part of the relatively good performance of parallel programs using spin-block message receipt under Solaris 2.4 in our experiments. This would explain why only occasional priority boosts serve to coordinate scheduling as well as they do. Because our competitor loads are typically balanced and do not block, the effect would be enhanced. Further investigation, using different sorts of competitors, will be necessary to determine whether this behavior is in fact arising.

Reasons for improved performance under spin-block message receipt

We sought to confirm experimentally the claim in [6] that the priority boosting performed by the scheduler was responsible for the relatively good performance we observed in programs using spin-block message receipt. To do so, we ran a simple experiment. We ran our barrier test, described in Section 5.2.2 above, with an altered timesharing dispatcher table that gave reawakened processes exactly the priority they had had when they went to sleep; that is, in Table 5.2, we set the value of `ts_slpret` for each queue n to n. As described in Section 5.3.1, we used a fixed maximum spin time of 1600 μsec, and ran the barrier test with $1,000$ delay iterations for $100,000$

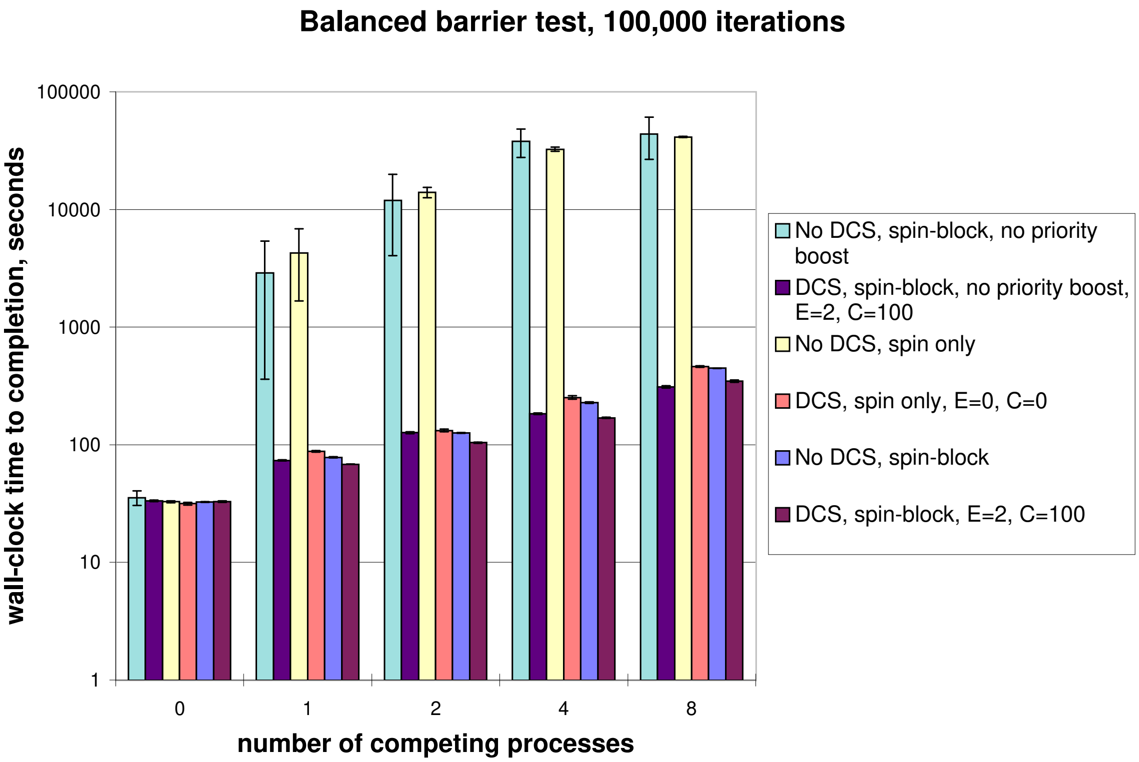

Figure 5-18: Barrier test wall-clock times to completion under spin-block message receipt with no priority boost, with and without DCS. For comparison, times with the normal Solaris 2.4 dispatcher table are also shown.

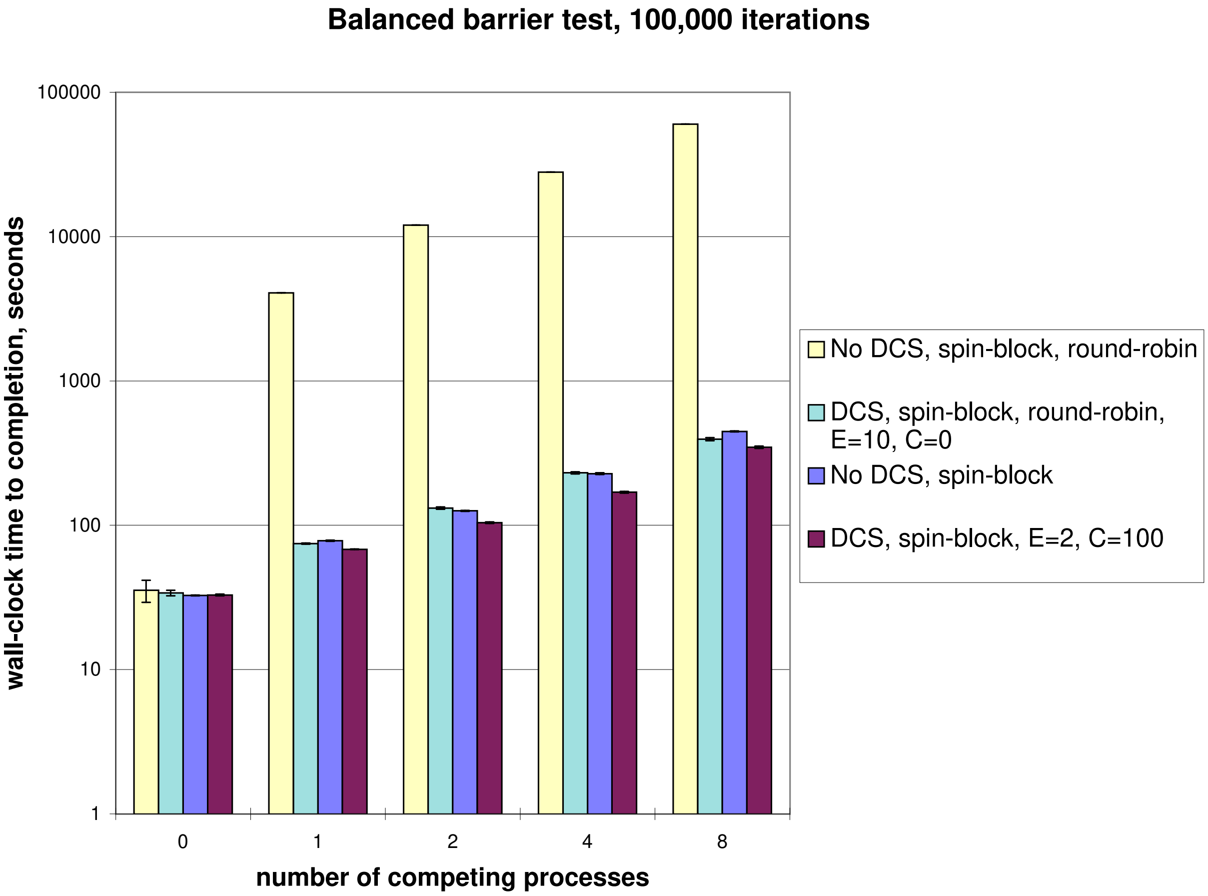

Figure 5-19: Barrier test wall-clock times to completion under spin-block message receipt with a single run queue, with and without DCS. For comparison, times with spin-block message receipt with the normal Solaris 2.4 dispatcher table are also shown.

barriers in competition with a varying number of serial jobs. The serial competitor jobs were balanced; the same number were present on each of the nodes, including the root.

The results are shown in Figure 5-18. In each data series, the figure shows the mean of between 3 and 10 runs. Because each of the longer runs took more than 10 hours, some of these cases have larger confidence intervals, as time did not permit more data to be gathered. The results unequivocally confirm that the priority boost is responsible for the relatively good performance of spin-block message receipt under Solaris 2.4: without the priority boost, job completion times under simple spin-block synchronization are on the same order as under spinning synchronization, which is to say, very much greater than under spin-block synchronization with priority boosts, or under DCS with spinning synchronization alone, or under DCS with spin-block synchronization, which achieves the best performance of all.

It was hypothesized in [6] that the reason others who had investigated the performance of parallel programs on timeshared multiprocessors using uncoordinated scheduling with spin-block synchronization had found poor performance was that they had used a round-robin scheduler, rather than a priority scheduler. Dusseau *et al.* ran an experiment in which they used a single run queue, rather than the sixty run queues normally used by Solaris 2.4 for timesharing and interactive jobs, and saw that spin-block message receipt performed very poorly. Because the Solaris 2.4 timesharing dispatcher maintains separate sleep queues, and waking jobs are placed at the back of their run queue, even without DCS, the single-run-queue scheduler is not a strict round-robin scheduler; execution order varies when jobs wake and sleep.

We also ran such an experiment, and found similar results for spin-block message receipt with the standard Solaris 2.4 scheduler; but DCS performed relatively well even with only a single run queue, although fairness parameter values had to be set to very aggressive values to achieve good performance. We ran our barrier test experiment with a round-robin scheduler, which we achieved by using a modified dispatcher transition table in which all transitions were to a single run queue, so that within one timeslice all processes were on this single queue. As mentioned above, the Solaris 2.4 scheduler services individual run queues in round-robin order, except in the case of transitions from sleep queues, where the newly awakened process is placed at the end of the queue. In our case, DCS used its usual message-driven scheduling algorithm, which allows it to vary execution order by placing jobs for which messages have arrived at the front of their run queues. The results are shown in Figure 5-19. Once again, we see that running the process when the message arrives results in good performance; good performance can be achieved regardless of whether a priority scheduler is used.

Estimating the number of priority boosts

It would have been interesting to instrument the Solaris 2.4 kernel to count the number of priority boosts performed with spin-block message receipt, and evaluate whether the boosts would cause the process to run immediately. It would have been possible to do so without building a new kernel, because our device driver was invoked both when a process blocked awaiting a message and when it was awakened when the message arrived (see Section 4.2.1). Thus we could have examined the value of `dispwait` at this point to determine whether the process would experience a priority boost on being returned to a run queue. However, constraints of time did not permit this instrumentation to be installed.

We can bound the number of priority boosts above by the number of seconds for which the program ran. This is because a priority boost happens only if `ts_update` runs while the process is on a sleep queue, and `ts_update` runs once per second. For the barrier test with spin-block message receipt, these values varied between about 33 seconds with 0 competitors to about 447 seconds with 8 competitors. For comparison, under DCS, on the root node, we saw about 75 priority modifications in the base case of 0 competitors, where the test also ran in 33 seconds, and about 1450 with 8 competitors, where DCS ran the test to completion in about 347 seconds.

As a consistency check, we can derive another estimate of the number of priority boosts due to blocking in the case of spin-block message receipt by examining his-

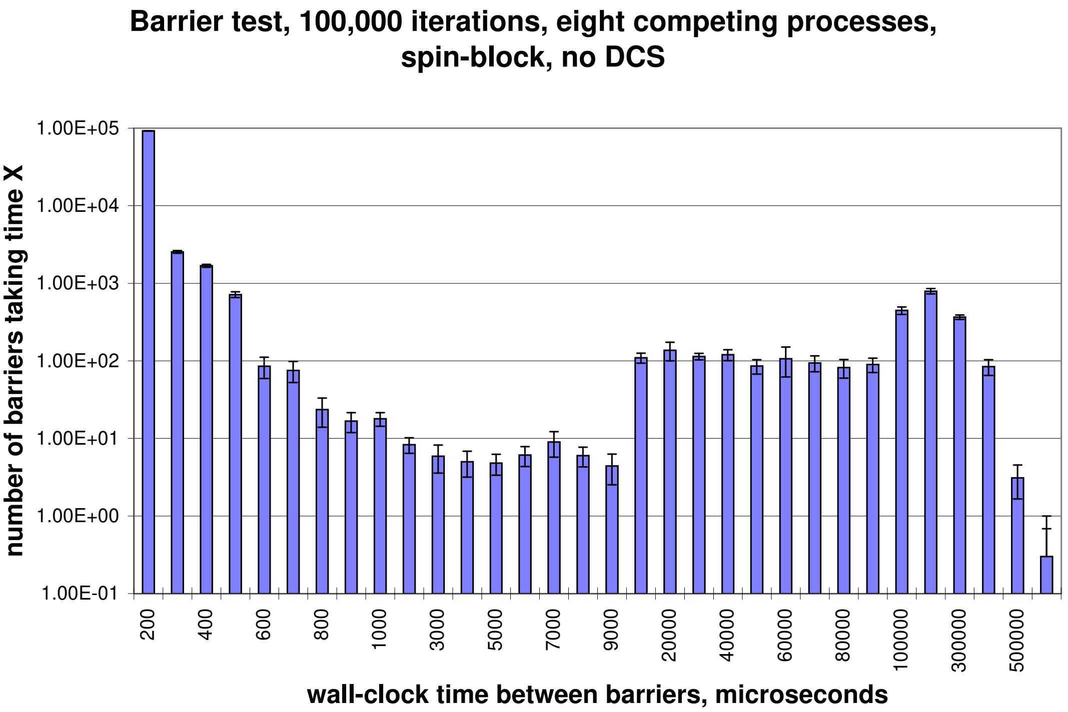

Figure 5-20: Barrier test inter-barrier times for the case of spin-block message receipt with 8 competing processes.

tograms of the time between the sending of a "pass barrier" message sent by the root and the receipt of the last "at barrier" message from a leaf node. We show such a histogram in Figure 5-20, for the case of 8 competing processes. Note once again that some of the features of the graph are artifacts of the data collection method — see the text in Section 5.3.1 describing the histogram in Figure 5-7 for details.

We can estimate the number of priority boosts from blocking by assuming random incidence of the beginnings of blocked periods into the 1-second intervals between runs of `ts_update`. Recall that if `ts_update` runs while a process is on a blocked queue, then that process will experience a priority boost upon returning to a run queue. We will ignore cases in which the end of a timeslice for the barrier test process came about because of preemption at the end of a quantum. Then if we assume random incidence of the beginnings of blocked periods into the 1-second intervals between runs of `ts_update`, the probability that a blocked period spans a run of `ts_update` is simply the length of the blocked period divided by one second.

Knowing that the fixed spin time is 1600 microseconds, we can easily compute the expected number of priority boosts from the histogram. If we perform this calculation using the histogram in Figure 5-20, we get 388 as our estimated number of priority boosts due to blocking — a number only a little lower than the 447 runs of `ts_update`. Of course, if the process is not blocked, but not running, it will receive a priority boost from `ts_update` anyway, in the routine's other capacity as an anti-starvation device.

It is interesting that spin-block synchronization can achieve performance only 22 percent worse than DCS with such a small number of priority modifications. In fact, in the case of the latency program, which is as fine-grain a program as we can write with FM, it is possible to improve the coscheduling at some expense to fairness by additionally using the Unix `nice()` call to boost its base priority (see Section 5.4.2).

We might conjecture that another phenomenon involved here is that processes being placed on blocked queues will have higher mean priorities than processes selected at random with uniform probability from the set of all processes on run queues. This is so because a process that blocks has not reached the end of its quantum, and so is not demoted in priority as are processes that reach the ends of their quanta; and it was necessarily the highest-priority process in the system when it blocked, because it was running when it blocked. Thus even if a process on a blocked queue does not receive a priority boost while blocked, it may well be the highest-priority process in the system when it returns from blocking. However, we know that this effect cannot be responsible for "most" of the performance of spin-block, because our experiment of removing the priority boost would have left this effect intact, and yet performance declined to approximately the level of spin-only.

Further investigation into sources of the performance of spin-block message receipt is warranted here; but we would also point to the fact that the competing loads here are balanced and consist of simple spin loops, which may allow some passive coscheduling behavior to persist in periods between successive runs of `ts_update`: due to symmetry across the nodes in the cluster, after becoming coordinated once through priority boosts, they run through several quanta in step. Other sorts of competing loads — more realistic ones — might shed further light on the way in which this mechanism achieves what it does.

In conclusion, simple spin-block message receipt under Solaris 2.4 performs unexpectedly well due to priority boosting on process wakeup by the Solaris 2.4 scheduler; this priority boosting effectively provides a partial implementation of DCS. A full implementation of DCS can perform better because it can schedule on message arrival programs that are not on sleep queues, as well as those that are on sleep queues. This matters tremendously for programs that poll for message arrivals; they achieve no coscheduling with the unmodified Solaris 2.4 scheduler. It also matters for programs that use spin-block message receipt, if these have relatively coarse-grain communication, because under simple spin-block synchronization in Solaris, preempted processes (those whose timeslices have expired without their having blocked on I/O) are not normally scheduled immediately by the dispatcher on message arrival.

Possibly another effect at work in the better performance of a full implementation of DCS is that newly reawakened processes are less likely to run immediately under the default Solaris 2.4 dispatcher than under our full DCS implementation. Under our implementation the newly reawakened process is placed at the front of the highest-priority run queue, if doing so does not starve other processes, while in the default Solaris 2.4 scheduler, the process is always placed on the back of some queue.

5.4.2 Effects of boosting the base process priority with `nice()`

As we saw in Section 5.3.1, under spin-block without DCS, the latency test fails to use its fair share of the machine. Interestingly, this is a program that spends nearly all of its time sending and receiving messages (that is, waiting for the LANAI network interface to send and receive messages), so it is quite likely that nearly half the time when one of the latency processes is not running, it is on a sleep queue — that is, the processes have the opportunity to be rescheduled together about as often as one might deem possible with the priority boost on wakeup in Solaris 2.4.

We would like to understand the reason for this reduced performance. Unfortunately, we have not had the time to answer the question fully; we record here some conjectures and the results of an experiment. We begin by noting that the program might have failed to use its fair share of the processor because its timeslices were too short, or because it received too few timeslices, or both.

The timeslices could be too short because of insufficient coscheduling: that is, one process could have been scheduled on one node, sent a message to its peer on the other node, and blocked while waiting for the response; then the peer could be scheduled after the first process had blocked, could respond to its message, and block while waiting for the response, and this scenario could be repeated often, so that the two processes each ran for the duration of one message transmission. This is the processor-thrashing behavior that Ousterhout originally described in [18], and which we have reproduced by disabling priority boosts on process wakeup in our experiment of Section 5.4.1, the results of which are shown in Figure 5-18. This behavior could still occasionally occur even with the priority boost in place, because the priority boost is not as strong an imperative for scheduling on message arrival as it might be: the priority is only raised somewhat; the process goes to the back of the run queue on which it is placed.

The second possibility is that the coscheduling mechanism is sufficient so that very few timeslices are very short, but that timeslices do not start often enough. If the process often blocks and on return from blocking is placed at the end of its queue, then if it has not received a priority boost, it may not be scheduled immediately. It is conceivable that this might happen often enough so that fewer timeslices start for the parallel process than for the competitor processes.

Finally, of course, both phenomena could be occurring. Metering would tell us the answer, but time constraints did not allow us to meter the kernel in this way. However, we performed a related experiment. If either of the cases we described above obtained, better performance would be achieved by raising the base priority of the process using the Unix `nice()` command. In the first case, the process would already be running at a higher priority on average before blocking, so when returning from blocking it would have a higher priority on average whether or not the priority boost occurred; thus it would be scheduled on return from blocking more often than without the use of `nice()`; better coscheduling would result. In the second case, maintaining a

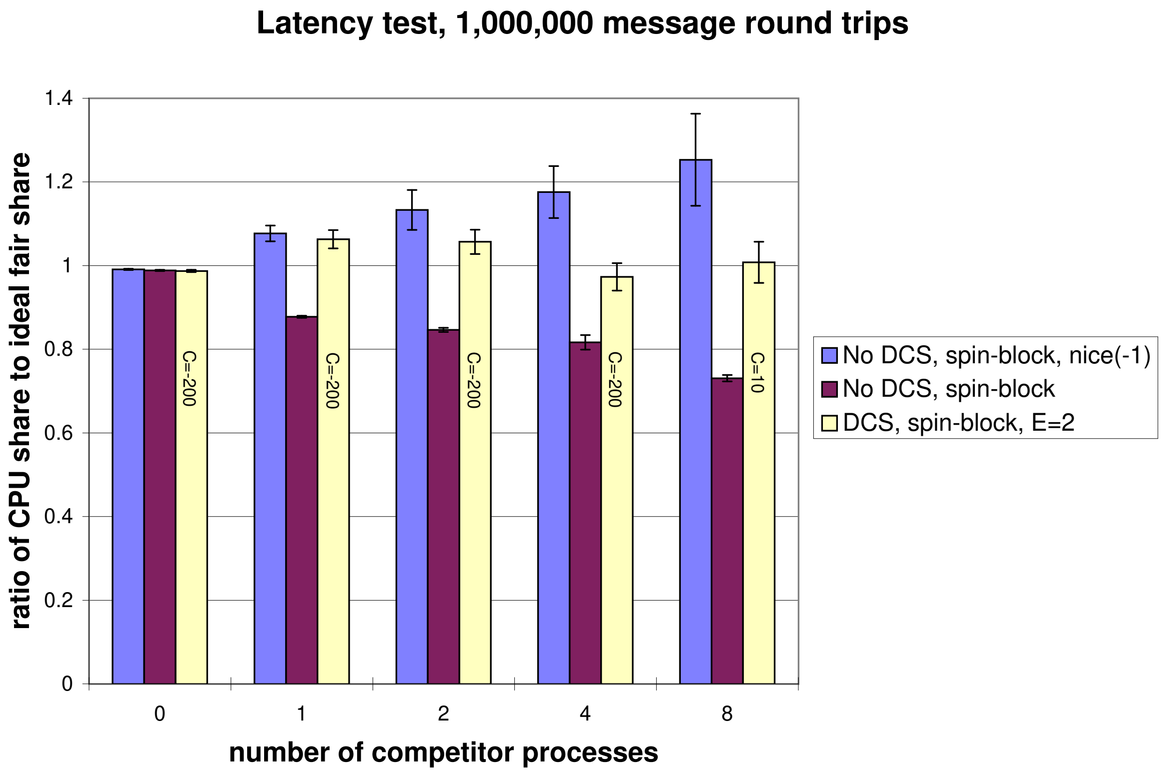

Figure 5-21: Fairness in the latency test under spin-block synchronization without DCS, but with priority boosts on process wakeup and an additional priority boost from `nice(-1)`. The value shown is the ratio of the fraction of the CPU used by the parallel process to its ideal fair share of the CPU.

higher mean priority would mean more timeslice initiations for the process. Thus the experimental result would not differentiate between the two phenomena, but may nonetheless be related to both.

We tried this experiment, using the lowest possible priority boost of -1.[4] The fairness results are shown in Figure 5-21. It can be seen that, with this increased base priority, the latency program using spin-block message receipt with priority boosting on process wakeup achieves quite good performance, except that it seizes too large a share of the CPU in many cases, where DCS can be controlled more finely by

[4]Arguments to the Unix `nice()` library call increase the priority of the job if they are negative and decrease it if they are positive.

manipulating its fairness parameters.[5] Because -1 is the smallest possible priority increase available with `nice()`, the latency program cannot be made more fair in the case without DCS.

Leaving aside the underlying mechanisms for now, we note that from the perspective of finding an engineering solution to the problem of scheduling parallel programs, for this very fine-grained program, if spin-block synchronization with priority boosting is used, the tools available in Solaris 2.4 can result in sufficient coscheduling to produce quite a good result. Without the use of `nice()`, the elapsed time to completion is at worst 25% worse than with DCS; with the use of `nice()`, the fairness is at worst about 25% worse than with DCS. We conjecture that a program with coarser communication granularity would prove more difficult to coschedule fairly with `nice()` and priority boosts on process wakeup, because the priority-boosting mechanism is invoked less often for programs with coarser communication granularity. Constraints of time have not allowed us to experiment further along these lines, but they are an interesting area for future research. We do in any case believe that a full implementation of DCS with an adaptive mechanism for setting the fairness parameters would be a better engineering solution, because it would allow finer control of fairness and would also allow coscheduling of programs that do not block for message receipt. However, the combination of spin-block message receipt, priority boosting on process wakeup, and judicious use of `nice()` may be sufficient in many practical cases.

5.4.3 Coarse-grain computation: varying the granularity of the barrier test

We sought to evaluate the effects of increasing the granularity of communication in our experiments. In particular, we wished to determine whether the result we found with the simple model and simulation of Chapter 3, where we saw that increasing granularity led to decreased coscheduling under DCS, also appeared in our experimental system.

In our initial experiments, with only spinning message receipt, we used a variety of communication resolutions on an eight-node system. We ran experiments with granularities as large as $100,000$ delay iterations, which is to say, 7.8 milliseconds.

The results were intriguing, but each run took a very long time to complete and the variances of the results at high granularities were very large. Constraints of time do not allow us to run these experiments in sufficient number to achieve reasonable confidence intervals — we estimate that to do so would require a dedicated month or more of cluster time.

Thus we will simply summarize the observations we made. As the granularity of the computation increased, performance of the barrier test running under the unmodified Solaris 2.4 scheduler decreased rapidly, for 1, 2, and 4 competitors, so

[5]Presumably an automatic method for manipulating the fairness parameters adaptively would allow DCS to achieve fairness even closer to perfect than it does; because experiments took a long time, and we were only interested in demonstrating the practicality of the scheme, we simply did some binary searching in the parameter space and chose the best values we found after a few trials.

that the user CPU time (not elapsed time!) for the one-competitor case and 100,000 delay iterations took as much as nine times as long as for the 0-competitor case. Performance also declined for DCS in this case, but only to a factor of about three times the base case.

The granularity of communication in this case bears some examination. 100,000 delay iterations is approximately 7.8 milliseconds. Because parallel processes running under DCS are typically running at the maximum priority, they usually use 20-millisecond timeslices, and thus we have only 2.6 messages being sent on average per timeslice in this case. Some exploration of the space revealed that performance began to deteriorate severely at 30,000 delay iterations, or about 9 messages per timeslice, with user CPU time around twice that of batch in this case.

This behavior, in which the effectiveness of dynamic coscheduling decreases as the granularity increases, was predicted by our models, described in Chapter 3. In the case of spinning synchronization with DCS, if messaging is relatively inexpensive, some relief can be afforded by artificially decreasing the granularity of the computation, by periodically "prefetching" peer processes by sending them empty messages; further research is required to determine how this might be done most efficiently.

In the case of spin-block synchronization, we were able to conduct one of these experiments again with the coarser granularity of 10,000 delay iterations, or about 0.78 milliseconds of delay between barriers. We see in Figures 5-22 and 5-23 the results. For both spin-block synchronization alone and spin-block synchronization with DCS, fairness declines as load increases, but fairness declines more quickly in the case where DCS is not used. However, even with its fairness mechanism completely disabled (which is what use of the parameters $E = 20, C = 0$ will do), DCS is unable to maintain perfect fairness. Possibly artificially increasing the base priority with `nice()` would help; we did not pursue this line of inquiry. As with the spinning message-receipt results, we suggest that artificially decreasing granularity may be one solution here, and research is required to determine just how this would best be done.

In conclusion, DCS is not as successful in coscheduling applications with coarse-grain communication as in coscheduling those with fine-grain communication. In the case of spinning message receipt, it is much more successful than the completely uncoordinated scheduling of the default Solaris 2.4 scheduler, but does not approach the performance of batch processing. Our models, in Chapter 3, had predicted this result.

In the case of spin-block message receipt, both DCS and the unmodified Solaris 2.4 scheduler (with priority boosts on process wakeup) are less successful at coscheduling as the granularity of the computation increases. DCS does better than the unmodified Solaris 2.4 scheduler, but does not cause parallel processes to receive their full share of the CPU. This does not create an efficiency problem — CPU time is not substantially different from the 0-competitor CPU time in either case — but it does create a latency problem.

In the case of DCS, in both spinning and spin-block message receipt, the degree

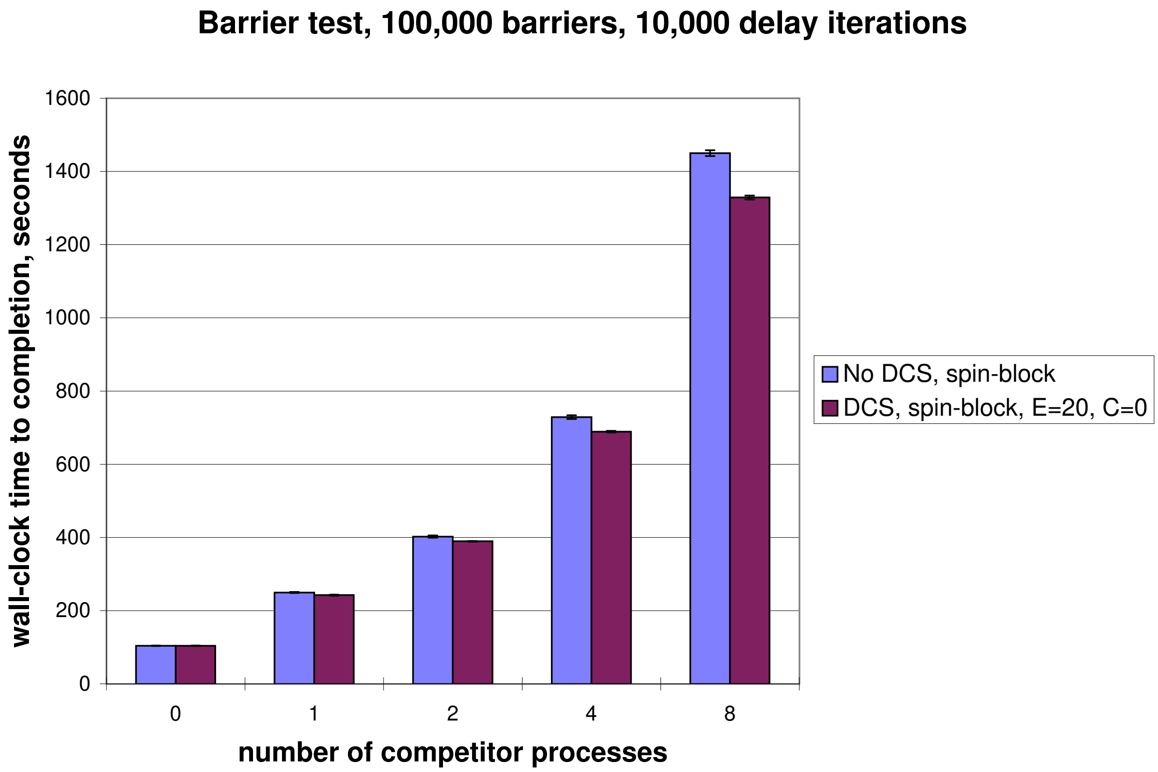

Figure 5-22: Barrier test wall-clock times to completion with 10,000 delay iterations, or 0.78 milliseconds of delay between barriers. Loads were balanced.

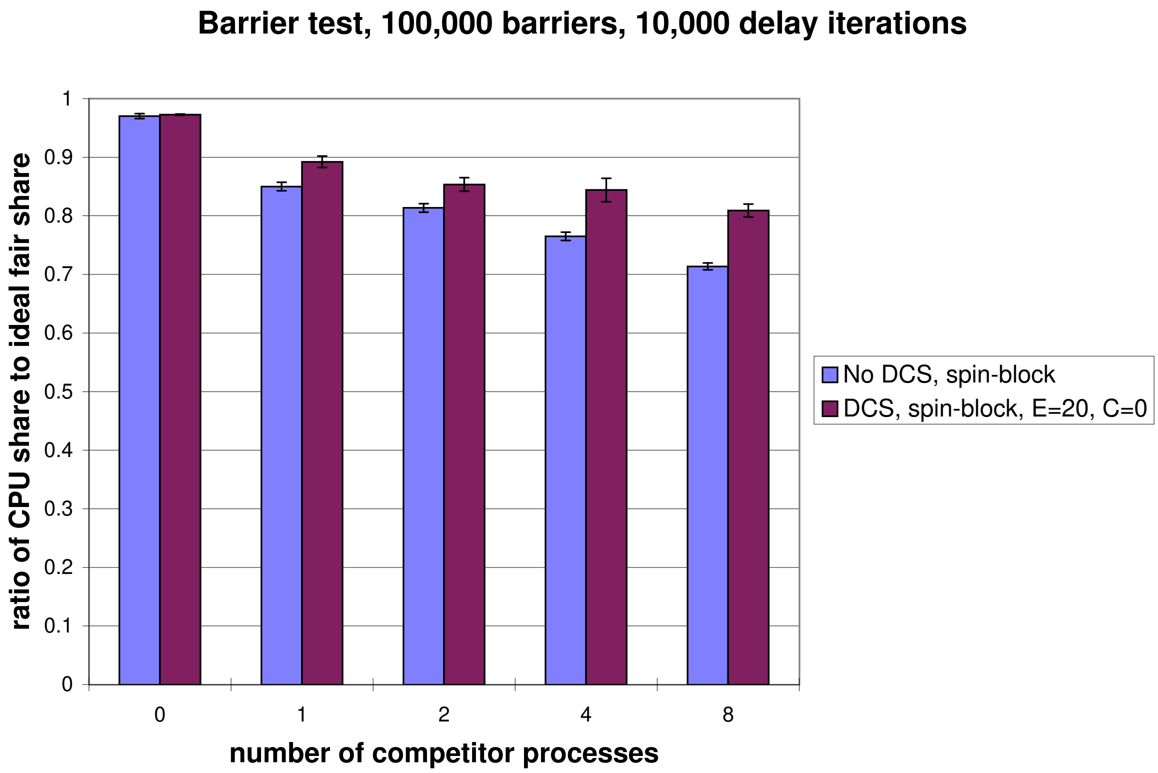

Figure 5-23: Fairness in the barrier test experiment of Figure 5-22, where 10,000 delay iterations were used. The value shown is the ratio of the fraction of the CPU used by the parallel process to its ideal fair share of the CPU.

of coscheduling can be increased by decreasing the granularity of communication by sending empty "prefetching" messages. Further research is required to determine how this might be done most efficiently.

5.5 Summary of experimental results

We have seen that our dynamic coscheduling implementation does cause coordinated scheduling of the parallel programs we tested.

In the case of spin-block message receipt, performance on fine-grained programs under DCS was close to ideal — CPU time was close to that in the 0-competitor case, and the fairness parameters could be tuned to bring the fairness ratio to 1, so that elapsed time was as small as it could be without hurting fairness.

In the case of spinning message receipt, performance was much better with DCS than without, but efficiency was not close to perfect when processes were restricted to using less than 1.5 times their fair share of the processor. Efficiency could be brought closer to perfect by decreasing fairness; small decreases in fairness led to large increases in efficiency. We believe that this problem is exacerbated in Solaris 2.4 by the varying scheduling quanta used by the dispatcher (shown in Table 5.2), which can cause even processes that start their timeslices in synchrony to suffer long spinning phases when one ends its timeslice before others.

Performance for spin-block message receipt without DCS was surprisingly good, although not as good as that with DCS. We experimentally verified the claim made in [6] that the reason for this relatively good performance was priority boosting on process wakeup by the Solaris 2.4 scheduler. By causing descheduled processes to sometimes be scheduled on message arrival, this behavior has the effect of a partial implementation of DCS.

For the latency test with spin-block message receipt, we were able to increase the coscheduling realized by the unmodified Solaris 2.4 scheduler by using the Unix `nice()` call to boost the base priority of the parallel process, but fairness suffered. Still, this raised the possibility that for fine-grained programs using spin-block message receipt, the tools available in the unmodified Solaris 2.4 implementation may be sufficient to achieve good performance. We conjecture that it would be more difficult to achieve good performance for more coarse-grained programs.

Performance for spinning message receipt without DCS was terrible, with elapsed times sometimes as much as a hundred times greater than those under DCS.

We found that when we increased granularity of communication, both the unmodified Solaris 2.4 scheduler and DCS suffered decreased performance, although the performance of DCS was still better than that of the unmodified scheduler. Under spinning synchronization, the decreased performance was manifested as decreased efficiency. Under spin-block synchronization the decreased performance shows up as increased response time.

DCS improves performance sufficiently so that a full implementation with an automatic mechanism for modifying fairness parameters will be worth implementing. It can coschedule both parallel programs using spin-block message receipt, and those using spinning message receipt, with good control over fairness if the programs have relatively fine-grained communication. However, the performance of DCS in the cases of spinning message receipt and of spin-block message receipt with coarse-grained computation is less than ideal. Further research is required to determine how to improve this performance.

Chapter 6

Conclusions and Future Work

We have presented demand-based coscheduling, a new approach to scheduling parallel computations on multiprogrammed multiprocessors that achieves good performance by coscheduling those processes that communicate with each other. Demand-based coscheduling was designed to be:

- Non-intrusive — the programmer is not required to write parallel programs in a particular style. *E.g.*, multithreading is not required; if full-fledged processes are a better abstraction, they can be used instead. Process placement or migration algorithms are not imposed by demand-based coscheduling.

- Flexible — If a job composed of a large number of processes is run on a multiprocessor with a small number of nodes, demand-based scheduling can take advantage of local communication patterns that may provide better performance.

- Dynamic — Newly-initiated communication between processes is detected automatically as a demand for synchronization by demand-based coscheduling. Thus it is well-suited to newer programming paradigms (*e.g.*, OLE) that may result in fine-grain communication between processes the programmer could not have anticipated would communicate.

- Decentralized — scheduling decisions are made locally in demand-based coscheduling. Unlike in traditional coscheduling, there is no "alternate coscheduling problem," because there is no centrally imposed notion of a single currently scheduled job.

6.1 Conclusions

6.1.1 Conclusions drawn from our model and simulations of dynamic coscheduling

We presented analytical and simulation results that show that the number of messages sent per timeslice is a key factor in achieving good coscheduling behavior under dynamic coscheduling, and that with a mean communication rate of more than approximately 300 messages per timeslice in our simulations, strong coscheduling behavior was achieved. Our simulations also showed that even under very pessimistic assumptions, dynamic coscheduling could achieve strong coscheduling behavior while maintaining fairness in scheduling, if the granularity of communication was small enough.

6.1.2 Conclusions drawn from our implementation of dynamic coscheduling

We implemented dynamic coscheduling on a network of workstations running Solaris 2.4. The implementation suffered from two limitations. The first was that multiple parallel jobs could not be run, because the current version of the messaging layer we used, Illinois Fast Messages, did not allow multiplexing of the network interface. The second was that our implementation of equalization (enforcement of fairness) required manual tuning, although we believe that a straightforward feedback control implementation would allow the tuning to be done automatically.

Experimental results found using our DCS implementation showed that dynamic coscheduling could provide good performance for a single parallel process running on a cluster of workstations, in competition with serial processes. Performance was close to ideal for the case of fine-grained processes using spin-block message receipt. Efficiency suffered for processes using spinning message receipt, although there was a tradeoff here: substantial increases in efficiency could be attained through small decreases in fairness; and efficiency under DCS was still far better than under the unmodified Solaris 2.4 scheduler, by a factor of as much as a hundred in some cases.

Increased granularity of communication caused performance to decline, as had been seen in the models and simulations we performed. The decreased performance was still better than that of the unmodified Solaris 2.4 scheduler. The performance decrease was manifested as increased response time in the case of spin-block synchronization and decreased efficiency in the case of spinning synchronization. We conjectured that a "prefetching" scheme in which processes artificially increased their granularity of communication by sending empty messages to processes with which they would soon communicate would improve performance in this case, but we did not experiment with such a scheme.

We found that parallel jobs using spin-block message receipt under the Solaris 2.4 scheduler performed unexpectedly well due to priority boosting on process wakeup. This behavior results in some of the effects of DCS, by scheduling processes when messages destined for them arrive at their node. However, parallel processes using spin-block message receipt under Solaris 2.4 still did not perform as well as those running under DCS. We were able to improve the performance of a very fine-grained process by using the Unix `nice()` call to boost its priority, but fairness suffered; and we conjectured that it would be more difficult to achieve the same effect with more coarse-grained programs.

We concluded that dynamic coscheduling performed well enough with both spin-block and spinning message receipt so that we can recommend that a version with an automatic fairness mechanism be added to operating systems or messaging layers meant to be used on workstation clusters. However, the performance on coarse-grained programs and under spinning message receipt is sufficiently far from ideal so that we also feel that further work in this area is needed in order to improve performance in these cases.

6.2 Future Work

A number of areas might profitably be explored in future research on dynamic coscheduling.

6.2.1 Multiple processes

Perhaps the most important open question about dynamic coscheduling is how it will perform with multiple processes. While dynamic coscheduling would still be useful if it allowed only a single parallel job to be coscheduled on a network of workstations, clearly the ability to timeshare the cluster among multiple parallel jobs would be very useful. Our simulation and modeling showed that a thrashing behavior can emerge when a fairness policy is implemented in simulation, and that the epoch mechanism ameliorates the problem; thus we expect that in practice epochs will be important. An implementation allowing multiple parallel jobs is required to allow the mechanism to be tested.

6.2.2 Spinning synchronization

As mentioned in previous sections, the differing timeslice lengths used in Solaris 2.4 hurt coscheduling with DCS under spinning synchronization because the first job to awaken after a period of sleeping in a group of communicating processes may have a different, and longer, scheduling quantum than the processes it awakens by sending messages. The processes it awakens will have short timeslices, because they are running at the highest priority; therefore their timeslices will probably end before those of the first process, which will then spin to the end of its timeslice, resulting in reduced efficiency. It would be useful to attempt an implementation of DCS with a dispatcher that provided uniform timeslice lengths to processes — we conjecture that with scheduling quantum length matching, the efficiency of DCS with spinning synchronization would improve dramatically.

6.2.3 Coarse-grained processes

As described above, we found in our experiments that the performance of our implementation decreased as the granularity of communication by the parallel job became coarser. We conjectured that a predictive mechanism that attempted to eagerly preschedule other processes with which a process could be expected to communicate soon might improve performance. This is an important area for future work.

6.2.4 Fairness and scheduling

In order to achieve good performance, our fairness mechanism required tuning for individual loads. An automatic mechanism for performing this tuning dynamically should be straightforward to implement.

We might have mitigated the narrow applicability of a single setting of our fairness parameters somewhat by decreasing the priority of a process that is part of a parallel job after its scheduling quantum expired. This could be done without modification of the scheduler as follows: if an interrupt had recently been raised due to a message arrival, then when the transition from the "FM job believed running" state of the LCP to the "FM job believed not running" state took place, the LCP could set a flag and signal an interrupt. The flag would be used to signal the meaning of the interrupt. When set to one value, an interrupt would be treated by the device driver as signifying that a message had arrived; but when set to another value, the interrupt would mean that the FM job's timeslice had recently expired. The device driver could then manipulate the run queues appropriately to decrease the priority of the FM job. As a result, the priority boost would not have the lasting effect of increasing the job's processor share beyond the timeslice that had just expired.

It would be best, however, to implement a fairness mechanism in a scheduler in which precise processor shares could be allotted to processes. Priority-decay schedulers confuse execution order (which is important in coscheduling) with processor share (which should be separately modifiable).

Dusseau *et al.* stated in [6] that fairness was not yet a solved problem in implicit scheduling; they observed that fine-grained processes suffered decreased processor shares when sharing a processor with coarse-grained processes in their simulation. They conjectured that this was because the fine-grained processes blocked often, yielding the processor. This might upon first examination appear to contradict our finding that dynamic coscheduling coschedules fine-grained programs better than coarse-grained programs, because priority boosting on process wakeup effectively implements part of DCS. However, note that under the Solaris 2.4 scheduler, the frequency in time of priority boosts that a process using spin-block synchronization can receive as a result of message arrivals is bounded above by a constant — 1 per second. Thus finer-grained communication under the unmodified Solaris 2.4 scheduler does not result in finer-grained information being provided to nodes about demands for coscheduling, and we believe that this is the probable cause of the discrepancy between their findings and ours. However, it would be best to inquire further into this issue with an implementation that allowed multiple parallel jobs to be run.

6.2.5 Other application types

We have speculated that dynamic coscheduling might be useful in a variety of application domains, including implementations of shared memory through message-passing and distributed client-server applications with fine-grain synchronization requirements. Due to limitations of the experimental platform and of time, however, we were unable to test such applications. In the case of client-server applications, we would like to explore the performance benefits of coscheduling; some benefits might include increased locality of reference due to smaller buffer sizes when the applications are coscheduled, or simply improved performance in the case of fine-grain synchronization requirements.

The case of implementing shared memory by message-passing raises another question: that of treating different types of messages differently with respect to scheduling. In the case of a shared-memory implementation, for example, if the platform allows reads to be effected regardless of which process is currently scheduled on the node (this is possible in, *e.g.,* the FLASH multiprocessor [15]), it might not be necessary or desirable to treat every memory read as a demand for coscheduling; but cache line invalidations might need to be treated as demanding coscheduling. An implementation would allow experimentation with different schemes.

Similarly, it may be the case that some messages in scientific applications should be treated differently with respect to coscheduling than others.

6.2.6 Predictive coscheduling

We describe in Appendix A a scheme for predictive coscheduling on a shared-memory multiprocessor. Further work would be required to evaluate the utility of this scheme.

However, the techniques of predictive coscheduling need not be limited to bus-based shared-memory multiprocessors, and one could envision using them together with a dynamic coscheduler. That is, the application could "preschedule" (by analogy with prefetching) other processes with which it would soon communicate by sending them "wake-up" messages, or the scheduler could do the same at the beginning of a process's timeslice by watching outgoing message traffic and sending messages to recent correspondents. Such techniques might serve to maximize time spent coscheduled.

6.2.7 Other issues

We found we achieved good performance with a fixed spin time and after initial experimentation settled on one. However, in [6], Dusseau found that adaptive variation of spin times enhanced the performance of SPMD programs using spin-block message receipt under Solaris 2.4. We believe that some of the conclusions reached in [6] were peculiar to the communication patterns of the limited set of applications examined in that work; however, it is possible that adaptive variation of spin times would also improve the performance of a broader class of applications, and that further work in this area might prove fruitful.

We considered a scheme in which outgoing messages would contain the residual lifetime of the sender's current timeslice (assuming it went to completion), so that recipients would be scheduled for only that period, to achieve greater synchrony of scheduling, but we did not implement it. In further experiments, it would be useful to characterize the synchrony of scheduling (the degree to which timeslices actually overlap) with a variety of applications and consider whether it is too low. If so, the technique of sending residual lifetimes of timeslices in messages might improve performance.

Appendix A

Predictive Coscheduling

We considered a scheme for implementing demand-based coscheduling on a simple bus-based shared-memory multiprocessor, called *predictive coscheduling*. We did not implement or simulate this scheme, and so we present it here in an appendix as an indication of a possible direction for future work.

If we wish to implement coscheduling on a bus-based shared memory multiprocessor, with hardware-only cache-coherence protocols, the detection of communication becomes more complicated than on message-passing architectures. If the program uses library routines for heavyweight remote procedure calls or for semaphores, the invocation of the kernel to deliver messages, perform blocking tests of semaphores, or set semaphores will allow the scheduler to be aware of communication between processes, and dynamic coscheduling can be used.

But if instead processes communicate only through shared memory pages in user mode, the kernel is not invoked, and cannot detect communication when it happens. We might consider using memory protection on shared memory pages to signal the kernel the first time during a process's lifetime that it requests access to a shared memory page, but this could be quite expensive if a large number of shared memory pages are touched and memory protection traps are slow.

Another possibility is to recognize that coscheduling is a performance optimization, and is not required for correctness, so that it is feasible to use a mechanism that simply provides hints as to which processes are likely to communicate with each other. If such a mechanism is correct with high probability, it will be sufficient to allow good performance.

We proceed by describing predictive coscheduling in the next section, and then proceed to describe an inexpensive mechanism for detecting communication using virtual memory hardware.

A.1 Correspondents

Under *predictive coscheduling*, processes that have recently communicated with each other are called *correspondents*. As in LRU demand paging, past behavior is treated as a predictor of future behavior, and so predictive coscheduling works by coscheduling runnable correspondents. In particular, when a process is scheduled on a node, an attempt is made to simultaneously schedule on other nodes its runnable correspondents. On a message-passing multiprocessor, this could be done by sending messages to the nodes on which the correspondents resided. On a bus-based shared-memory processor, other processes would be selected for preemption, interrupts would be signalled on their nodes, and the correspondents would be scheduled.

We have not yet tested this strategy, although it appears promising. Clearly a runtime equalization mechanism would be necessary to ensure fairness; possibly a mechanism like epochs would be desirable to reduce thrashing. The selection of processes for preemption is another open question. It might be desirable to select for preemption the processes with the fewest correspondents, because such processes will be runnable in the future under a wider variety of circumstances.

It is also worth noting that if communication between processes is entirely memoryless — so that one pair of processes that have recently communicated is no more likely to communicate in the future than is any other pair of processes — predictive coscheduling will not perform well. This is because predictive coscheduling attempts to predict future behavior on the basis of past behavior, a strategy that will work no better than random selection with uniform probability for memoryless processes. Of course, this scenario is unlikely to arise in most parallel jobs, where the constituent processes will communicate with each other repeatedly.

A.2 Detecting Communication through Shared Memory on Bus-Based Shared-Memory Multiprocessors

It remains to describe a means of identifying correspondents on bus-based shared-memory multiprocessors — that is, detecting communication through shared memory. Because, as we noted above, we do not think that this information need be perfect, we propose using the information in translation lookaside buffers (TLBs) on processors where these are readable. Processor-readable TLBs are becoming more common, because of the attractiveness of handling TLB misses in software on RISC processors. Among the processors that have readable TLBs are MIPS processors, DEC Alpha processors, and HP PA-RISC processors. Intel 486-series processors also provide a means of reading the TLB through the test instruction interface; reading the TLB does not require disabling virtual memory.

The algorithm for finding correspondents is simple. In the process control block (PCB) for each user process that shares memory, we maintain a field that contains a list of correspondents. We maintain in kernel memory a data structure, keyed by virtual page address, that contains an entry for each of the shared memory pages mapped by any user process. The structure is called the Shared Page Recent Accessors Table (SPRAT). Each entry in the SPRAT contains a list of processes that have recently accessed the page. At certain points in the execution of a process, we iterate over the TLB, finding entries for shared data pages in this process's address space.

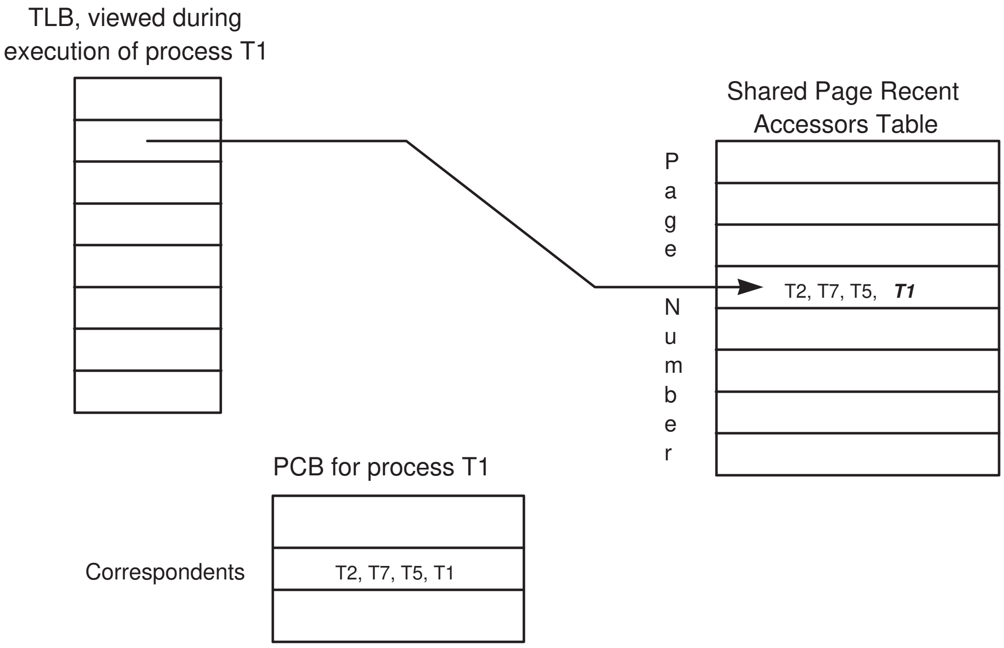

Figure A-1: Data structures used in proposed algorithm for predictive coscheduling.

For each such entry, we find the corresponding page entry in the SPRAT and add the current process to the list there. Then we add the processes in the SPRAT entry to the correspondents list in this process's PCB. The data structures are depicted in Figure A-1.

Because the TLB is typically small (256 or fewer entries), it can be searched quickly. Because the replacement policy is typically approximate LRU within a set (or simply LRU on fully-associative TLBs), the TLB contains information about which pages have been read or written recently. We may choose to search it just before descheduling a process, or perhaps also at other convenient times, such as system calls and exceptions.

Entries will also need to be cleared from the SPRAT and the correspondents list in the PCB. One inexpensive possibility is to maintain the lists as FIFOs of fixed and limited size — for example, the number of nodes on the machine would be a good limit. Another possibility is simply to clear the information at pseudorandom intervals — this can also be implemented inexpensively.

There are likely to be other means of detecting communication through shared memory on bus-based shared-memory multiprocessors using virtual memory information — as has been found in the field of lifetime-based garbage collection, the information maintained by a virtual memory system is very rich.

Bibliography

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

[2] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[3] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 224–234, Santa Clara, California, 1991.

[4] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 12–24, San Jose, California, 1994.

[5] 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 Processing*, pages 590–597, 1991.

[6] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, 1996. Available from `http://www.cs.berkeley.edu/~dusseau/Papers/sigmetrics96.ps`.

[7] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

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

[9] Dror G. Feitelson and Larry Rudolph. Coscheduling based on run-time identification of activity working sets. *International Journal of Parallel Programming*, 23(2):135–160, April 1995.

[10] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, May 1991. Available from `http://xenon.stanford.edu/~tucker/papers/sigmetrics.ps`.

[11] D. P. Helmbold and C. E. McDowell. Modeling speedup(n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):250–256, April 1990.

[12] IEEE Computer Society TCCA and ACM SIGARCH. Chicago, Illinois, April 18–21, 1994. *Computer Architecture News,* 22(2), April 1994.

[13] Sun Microsystems Inc. `ts_dptbl(4)` manual page. *SunOS 5.4 Manual*. Section 4.

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

[15] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [12], pages 302–313. *Computer Architecture News,* 22(2), April 1994.

[16] Mario Lauria. Mpi-fm: A high performance cluster implementation the message passing interface on fast messages. Master's thesis, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois., March 1996.

[17] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.

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

[19] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing*, December 1995. Available from `http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps`.

[20] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [12], pages 325–336. *Computer Architecture News,* 22(2), April 1994.

[21] Patrick G. Sobalvarro. Adaptive gang-scheduling for distributed-memory multiprocessors. Technical Report MIT-LCS-TR-622, MIT Laboratory for Computer Science, July 1994. Available from `http://www.nchpc.lcs.mit.edu/Workshop94/Papers/s4_p1.ps`.

[22] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the Parallel Job Scheduling Workshop at IPPS '95*, 1995. Available from `http://www.psg.lcs.mit.edu/~pgs/papers/jsw-for-springer.ps`. Also appears in Springer-Verlag Lecture Notes in Computer Science, Vol. 949.

[23] Andrew Tucker. Efficient scheduling on multiprogrammed shared-memory multiprocessors. Technical Report CSL-TR-94-601, Stanford University Department of Computer Science, November 1993. Available from `http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs/CSL-TR-94-601`.

[24] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM SIGOPS Symposium on Operating Systems Principles*, pages 159–186, 1989. Available from `http://xenon.stanford.edu/~tucker/papers/sosp.ps`.