Dynamic Coscheduling on Workstation Clusters

Patrick G. Sobalvarro[1], Scott Pakin[2], William E. Weihl[1], and Andrew A. Chien[2]

[1] Digital Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301 U.S.A.
{pgs, weihl}@pa.dec.com
http://www.research.digital.com/SRC/staff/{pgs, weihl}/bio.html

[2] Digital Computer Laboratory
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue
Urbana, IL 61801 U.S.A.
{pakin, achien}@cs.uiuc.edu
http://www-csag.cs.uiuc.edu/individual/{pakin, achien}

Abstract. Coscheduling has been shown to be a critical factor in achieving efficient parallel execution in timeshared environments [12, 19, 4]. However, the most common approach, gang scheduling, has limitations in scaling, can compromise good interactive response, and requires that communicating processes be identified in advance.
We explore a technique called *dynamic coscheduling* (DCS) which produces emergent coscheduling of the processes constituting a parallel job. Experiments are performed in a workstation environment with high performance networks and autonomous timesharing schedulers for each CPU. The results demonstrate that DCS can achieve effective, robust coscheduling for a range of workloads and background loads. Empirical comparisons to *implicit scheduling* and uncoordinated scheduling are presented. Under spin-block synchronization, DCS reduces job response times by up to 20% over implicit scheduling while maintaining fairness; and under spinning synchronization, DCS reduces job response times by up to two decimal orders of magnitude over uncoordinated scheduling. The results suggest that DCS is a promising avenue for achieving coordinated parallel scheduling in an environment that coexists with autonomous node schedulers.

1 Introduction

Coordinated scheduling of parallel jobs across the nodes of a multiprocessor is well-known to produce benefits in both system and individual job efficiency [12, 18, 4, 5, 17, 3]. Without coordinated scheduling, the processes constituting a parallel job suffer high communication latencies because of *processor thrashing* [12]. While multiprocessor systems typically address these problems with a mix of batch, gang, and timesharing scheduling (based on kernel scheduler changes), the problem is more difficult for shared workstation clusters in which stock operating systems kernels must be run.

With clusters connected by high performance networks that achieve latencies in the range of tens of microseconds [13, 20, 21, 9, 7], scheduling and context switching latency can increase communication latency by several orders of magnitude. For example, under Solaris, CPU quanta vary from 20 ms to 200 ms [10]; consequently uncoordinated scheduling can increase best-case latencies (~ 10 microseconds) by three to four orders of magnitude, nullifying many benefits of fast communication subsystems. Uncoordinated scheduling can also decrease the efficiency of resource utilization. Coordinated scheduling reduces communication latencies and increases system efficiency by reducing spin-waiting periods and context switches.

Coscheduling for clusters is a challenging problem because it must reconcile the demands of parallel and local computations, balancing parallel efficiency against local interactive response. Ideally a coscheduling system would provide the efficiency of a batch-scheduled system for parallel jobs and a private timesharing system for interactive users. In reality, the situation is much more complex, as we expect some parallel jobs to be interactive. Furthermore, in a cluster environment, there are advantages to be had in using existing commercial operating systems, so we restrict ourselves here to approaches that involve augmentation of existing operating system infrastructure.

The approach to coordinated scheduling that we use is a form of demand-based coscheduling called dynamic coscheduling (DCS) [17, 16], which achieves coordination by observing the communication between threads. This is a bottom-up, emergent scheduling approach that exploits the key observation that only those threads which are communicating need be coscheduled. This approach can achieve coscheduling without changes to the operating system scheduler or applications programs.

Our implementation of dynamic coscheduling is based on the Illinois Fast Messages communication layer which delivers low latency and high bandwidth user-space to user-space communication [13, 14]. We augmented this system with blocking communication primitives, and implemented dynamic coscheduling with changes to a device driver, network interface card firmware, and the communication library. The device driver influences the operating system scheduler's decisions through kernel interfaces, based on the communication traffic it observes on the network interface card.

Experiments using a variety of workloads (with different synchronization characteristics) and competing background loads are used to compare dynamic coscheduling against the unmodified Solaris 2.4 scheduler with spinning and spin-block synchronization. These results indicate that dynamic coscheduling, spin-block, and the combination of dynamic coscheduling with spin-block synchronization can effectively achieve coscheduling. The effectiveness of spin-block has been previously documented in [3] (where it was called *implicit scheduling*), and our measurements confirm their results. In addition, our work demonstrates that DCS achieves coscheduling with both spinning and spin-block synchronization, where implicit scheduling requires processes to block awaiting message arrivals for coscheduling to happen. Interestingly, coscheduling based on spin-block alone does not obtain a fair share of the CPU for parallel jobs, requiring a process to have blocked before communication can be treated as a demand for coscheduling. DCS can potentially treat all message arrivals as a demand for coscheduling, and obtains a fair share of CPU time.

The successful coscheduling approaches accrue benefits of higher system throughput and lower response time for jobs. However, one must be chary of drawing broad conclusions based on a modest set of experiments. The dynamics of schedulers and workloads are complex, and we have only begun to understand the benefits and limitations of demand-based coscheduling approaches. Important limitations of our study include use of only a single parallel job[1] and a modest sized cluster[2]. Both of these limitations are being remedied in future studies.

However, parallel jobs are only one possible application of DCS — we believe that DCS-like approaches can be used to implement coordinated resource management in a much broader range of cases, including:

- real-time and proportional-share processor scheduling
- multimedia and other quality-of-service-sensitive applications
- coordinated access to input/output devices
- coordinated memory management
- efficient parallel computing with demand-paged virtual memory

Most of these areas are still to be explored, and a discussion of specific approaches to them is beyond the scope of this paper. For the remainder of this paper we confine ourselves to a focus on achieving low latency, fairness, and efficiency for tightly-coupled parallel jobs. We found that DCS performs quite well in these cases. Our results show that under spin-block synchronization, DCS reduces job response times by up to 20% over implicit scheduling while maintaining fairness. Under spinning synchronization, DCS reduces job response times by up to two decimal orders of magnitude over uncoordinated scheduling with only a slight reduction in fairness.

The remainder of the paper is organized as follows. Section 2 summarizes the relevant related work. Section 3 describes the idea behind dynamic coscheduling briefly and our prototype implementation. Section 4 outlines our experiments and empirical results which provide evidence for the viability of dynamic coscheduling. A discussion of the results and their implications as well as limitations of our experiments are discussed in Section 5, along with some promising directions for future work. Finally, Section 6 briefly summarizes our results.

2 Related Work

There has been a wide variety of work on coscheduling, beginning with Ousterhout's seminal paper [12] which identified the need. The larger challenge can

[1] A limitation of our FM infrastructure.
[2] A limitation of our computing infrastructure.

be logically divided into three subproblems: detecting threads needing to be coscheduled, providing mechanisms for achieving coscheduling, and assessing the performance impact.

2.1 Performance Impact of Coscheduling

A variety of efforts on shared-memory machines have demonstrated and characterized the benefits of coscheduling for threads in a parallel job. Ousterhout provided a basic framework and presents a number of ways to achieve coscheduling while optimizing for system utilization [12]. Later work on the DASH multiprocessor [8, 2] demonstrated that coscheduling could be used to achieve efficiencies comparable to batch scheduling, while providing more flexible resource sharing. Specifically, coscheduling and process control (dynamic space-partitioning) performed similarly in the experiments described in [2].

2.2 Detecting Threads Requiring Coscheduling

Coscheduling implies coordinated scheduling of clusters of threads; identification of such clusters has been pursued through both explicit and implicit approaches. The shared memory workloads described in [8, 2] are parallel jobs which consist of thread collections, explicitly indicating which threads should be coscheduled. A variety of implicit schemes which do not require explicit programmer annotation have been explored. On distributed memory systems, the need for coscheduling has typically been associated with communication [17, 6, 3, 16]. Feitelson's Runtime Activity Working Set Identification (RAWSI) monitors the communication between processes or threads[3] to determine their rate of communication. Working sets of processes (which require coscheduling) are identified based on their rate of communication. RAWSI collects the information and uses a coordinated global mechanism to decide on a schedule. Both our dynamic coscheduling approach [17] and implicit scheduling [3] detect threads requiring coscheduling through their communication. However, neither system explicitly identifies the sets of processes to be coscheduled.

2.3 Mechanisms for Achieving Coscheduling

Once thread clusters have been identified, a mechanism for coscheduling must be used. In many systems, particularly those with shared memory, a gang scheduler which has the capability to achieve coordinated context switches across processors has been assumed [8, 12, 4, 6]. Such systems replace the basic process scheduler in the operating system, and schedule the related threads across the processing nodes. These schedulers can achieve high system efficiency on regular parallel applications, but have difficulty in selecting alternate jobs to run when processes block, require simultaneous multi-context switches across the nodes of the processor (which causes difficulty in scaling), and for good performance

[3] Feitelson's system actually addresses both distributed and shared memory systems.

require long scheduling quanta which can interfere with interactive response, making them a less attractive choice for use in a cluster of commodity workstations. It is largely these limitations which motivate the integrated approaches.

2.4 Integrated Scheduling Techniques

The requirement of centralized control and the poor timesharing response of other scheduling approaches have motivated new, integrated coscheduling approaches. Such approaches extend local timesharing schedulers, preserving their interactive response and autonomy. Further, such systems have typically not explicitly identified sets of processes to be coscheduled, but rather integrate the detection of a coscheduling requirement with actions to produce effective coscheduling. An earlier paper on *dynamic coscheduling* [17] detailed analysis and simulation of an integrated coscheduling technique. Subsequently, Dusseau's *implicit scheduling* was evaluated via simulation in [3], using synthetic single-program, multiple data applications. Because dynamic coscheduling is discussed extensively in the remainder of the paper, we only describe implicit scheduling here.

Implicit scheduling uses spin-block synchronization primitives and the priority boost given by the SVR4 scheduler to threads which block on input/output to produce coscheduling. Because threads awakened when the communication completes obtain a high priority, they are likely to run when their communication peer has just sent a message (and is therefore running). To further improve performance, implicit scheduling can also modify the spin times in spin-block synchronization, adapting to developed skew between threads in a coschedule. Implicit scheduling has been demonstrated in simulations to achieve good performance on a variety of "bulk-synchronous" applications (those which perform regular barriers, possibly with other communication taking place in between barriers), specifically a synthetic workload of SPMD programs. It remains an open question whether good performance can be achieved for more varied communication, computation, and synchronization structures, as well as in actual operating environments (with attendant daemons, scheduling idiosyncrasies, and other system activity). However, it is our understanding that efforts to explore these issues are currently underway.

In contrast, our dynamic coscheduling achieves coscheduling by explicitly treating all message arrivals (not just those directed to blocked processes) as a demand for coscheduling, and explicitly schedules the destination processes when it would be fair to do so through the explicit control of scheduler priorities. While this appears quite similar to implicit scheduling for the particular case of bulk synchronous jobs using spin-block synchronization, we believe dynamic coscheduling can be used to achieve coordinated scheduling in a broader range of cases.

3 Dynamic Coscheduling

3.1 Overview

Demand-based coscheduling [17,16] exploits communication between processes to deduce which processes should be coscheduled and to effect coscheduling. It is effective because of the key observation: the communicating (or synchronizing) processes are the ones that need be coscheduled. Thus, demand-based coscheduling produces emergent coscheduling without requiring explicit identification by programmer of the computations that need be coscheduled.

Dynamic coscheduling [17] is a type of demand-based coscheduling in which scheduling decisions are driven directly by message arrivals. If an arriving message is directed to a process that isn't running, a scheduling decision is made. This decision can be based on a wide variety of factors (e.g., system load, last time run, etc.), and is generally designed to maximize coscheduling performance while ensuring fairness of CPU allocation. Previously published modeling and simulation results [17] indicate that dynamic coscheduling produces robust coscheduling. Thus, the key elements of dynamic coscheduling are:

1. Monitoring communication/thread activity
2. Causing scheduling decisions
3. Making a decision whether to preempt

The latter two points are intimately tied to how the operating system operates. Causing scheduling decisions depends on the preemption capabilities and times context switches can occur. The decision procedure in particular can depend on fairness, and coscheduling stability concerns. The elements of DCS can be implemented in a host of different ways, and our experimental approach is described below. The details of any such implementation embody only a specific instance of a dynamic coscheduling.

3.2 Implementation Context

Fast Messages and Myrinet Our dynamic coscheduling prototype is implemented under Illinois Fast Messages (FM), a user-level messaging layer developed at the University of Illinois at Urbana-Champaign [15]. Fast Messages is a high-performance messaging layer that bypasses the operating system to provide direct access to an underlying Myricom Myrinet [1] and thereby achieve high performance. Details of FM can be found in [13,15]. We also employ an implementation of the Message Passing Interface (MPI) built atop FM, called MPI-FM [11], for some of the benchmarks.

Implementing Spin-block in Fast Messages Fast Messages was enhanced with a spin-block mechanism to support our experimentation. Adding a spin-block communication primitive required changes to the FM firmware, the network device driver, and the FM libraries. The firmware was modified to add an interrupt generation to wake a sleeping process upon message arrival. The device driver was modified to add a call that puts the caller to sleep, waiting for a message. Finally, the FM libraries were modified to integrate these changes.

We chose our maximum spin time of 1600 μsec based on the empirical evidence of experiments described in [16], in which the maximum delay we saw for response in the case where a context switch was required was approximately 1500 μsec. 1600 μsec is also slightly greater than twice the mean context-switch time plus the message round-trip time. It is noted in [12] that a two-context-switch maximum spin time is competitive, and in [3] it is argued 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.

Experimental Platform Our experimental platform consists of seven SPARC-station-2's connected by a Myrinet. The SPARCstation-2's have 40 MHz processors, 16 MB of main memory, and run the Solaris 2.4 operating system. The Myrinet provides high-bandwidth communication (up to 80 MB/sec) and is coupled to Lanai Version 2.3 interface boards (20 MHz Lanai's which matched the SBUS clock speed).

Despite the obsolete workstations, Illinois Fast Messages Version 2.0 achieves user-space to user-space latencies of 40 μsec with 128-byte messages, and bandwidths of 13 MB/sec.[4] FM maps the Myrinet interface board memory and control registers into both kernel and user space, allowing direct user access to the network for high performance. The mapping into kernel space enables convenient initialization and control of the device in response to system calls by the kernel.

Sunsoft Solaris Version 2.4 The dynamic coscheduling prototype runs under the Solaris 2.4 operating system. Two aspects of the implementation are specific to Solaris 2.4 (or this family of operating system): the mechanism for implementing scheduling decisions, and the fairness mechanism. Both of these are affected by the priority-decay algorithm of the scheduler.

Solaris 2.4's dispatcher for timeshared and interactive jobs is a table-driven Unix priority-decay scheduler. The scheduler uses 60 queues for user processes; these are numbered 0 through 59. The priority of a given queue is its number; the higher the number, the higher the priority. The scheduler schedules the job at the head of the highest-priority queue that is occupied. Timeslice expiration leads to demotion to a lower-priority queue; preemption without timeslice expiration causes the job to be placed at the end of the queue. Once per second, a routine called `ts_update` increases the priorities of processes that are on run queues, but not running. The routine also sets a flag (`dispwait`) on processes that are on sleep queues; if the flag is set when the process returns to a run queue, it

[4] Note that recent performance numbers for Illinois Fast Messages Version 2.01 with more modern Myricom hardware and more modern Sun workstations are $\approx$ 11 μsec and 56.3 megabytes/second.

experiences a substantial priority boost, to the value called `ts_slpret`, as shown in Table 1. Some of the effects of this priority boost will be described in Section 5.

	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 1. Default Solaris 2.4 dispatch table

3.3 DCS Prototype

We describe our prototype by relating each of the high level elements of dynamic coscheduling to its implementation. This not only provides a clear perspective on the rather myriad low-level details required for work of this type, but it also clearly illuminates the approximations and compromises in the prototype. A simple illustration of the DCS implemenation is shown in Figure 1. Our description of the implementation is necessarily brief; further detail can be found in [16].

Monitoring Communication/Thread Activity Communication and thread monitoring is performed on the network interface card. Myricom's network interface card provides a programmable processor, running the Fast Messages firmware. We modified the FM firmware to monitor the ongoing communication and thread activity. Monitoring communication is simple; the firmware is essentially a dispatch loop for each communication. The FM firmware monitors thread activity by periodically reading the host's kernel memory the address of the currently running lightweight process (LWP). Because this operation is achieved with DMA, and must cross an I/O bus bridge, is costs tens of microseconds. Thus, the firmware reads the value only once per millisecond, and hence the firmware has only an approximation of the currently running thread. However, because the scheduling quanta are 20 milliseconds or larger, this approximation is sufficient. The mechanisms used for dealing with cases where the information is inaccurate are described in [16].

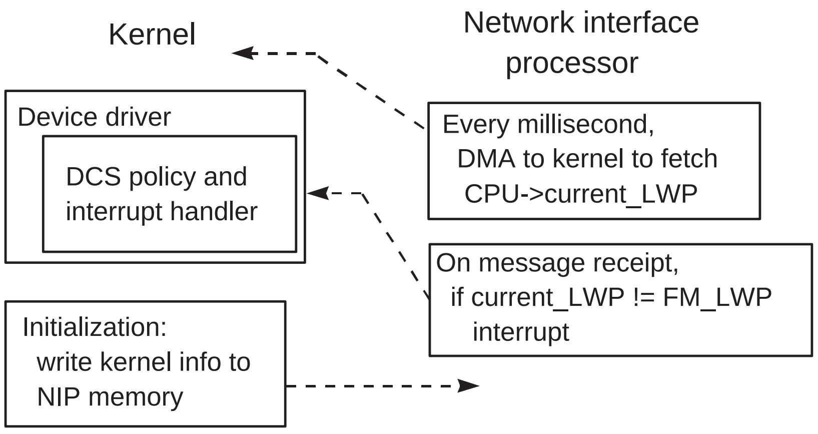
Kernel
Network interface processor
Device driver
DCS policy and interrupt handler
Every millisecond, DMA to kernel to fetch CPU->current_LWP
On message receipt, if current_LWP != FM_LWP interrupt
Initialization: write kernel info to NIP memory

Fig. 1. Simplified DCS implementation schematic (spin-block implementation not shown).

Causing Scheduling Decisions Scheduling decisions are effected by a device driver for the Myrinet network interface card (NIC). This driver is invoked by interrupts from the NIC, if the message received is not for the LWP currently running. FM messages are sent to LWP's, not to individual threads within an LWP. So, in the device driver, all of the threads within the receiving process have their priority boosted as shown in Figure 2. Note that all of our benchmarks have only a single thread within the process, so this approximation is not a concern in our tests.

```
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;
  }
}
```

Fig. 2. DCS scheduling decisions are affected in the device driver interrupt handler.

If the interrupt routine in the device driver finds that it would be fair to preempt the currently running process, each thread belonging to the LWP for the parallel process has its priority raised to the maximum allowable priority for user-mode timesharing processes[5] and is placed at the front of the dispatcher

[5] With the default Solaris 2.4 dispatcher table, this is 59.

queue. Flags are set to ensure that the Solaris 2.4 scheduler runs on exit from the interrupt routine, causing a scheduling decision based on the new priorities. This will cause the process receiving the message to be scheduled unless the process that was running had a higher priority than the maximum allowable priority for user mode.

Making a Decision Whether to Preempt In dynamic coscheduling, an incoming message's process is scheduled only if doing so would not cause unfair CPU allocation. The equalization mechanism described in [17] used detailed CPU time numbers. At the time we designed our implementation, we chose to implement fairness by limiting the frequency of priority boosts.[6] Our fairness criterion is the following inequality:

$$2^E(T_c - T_p) + C \geq T_q R \tag{1}$$

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

$$\begin{aligned}
E &= \text{Exponent} \\
T_c &= \text{Current time} \\
T_p &= \text{Time of previous priority boost and preemption attempt} \\
C &= \text{Constant, in milliseconds} \\
T_q &= \text{Length of minimum time quantum (20 msec)} \\
R &= \text{Number of jobs in the run queue}
\end{aligned}$$

Our approach limits the frequency of the preemptions for each cycle the scheduler makes through the run queue, assuming that all jobs on the run queue are running at the highest priority. E and C are chosen empirically for individual experiments. $R \times T_q$ is the "length of the run queue in time" if the jobs on the run queue each run for an entire minimum-length timeslice (T_q). $T_c - T_p$ is the time since the last preemption. For example, a value of -1 for E would enable a preemption (priority boost) only if the time since the last preemption were at least twice the length of the run queue in time.

4 Empirical Studies

In this section, we present the results of experiments with our FM-DCS system. These experiments include a range of parallel kernels and competitive workloads. First, we describe the performance metrics, workload, and scheduling variants used. Subsequently, we describe the experimental results and give analysis.

[6] Only later did we learn that a direct implementation based on CPU usage information was possible. This a subject of future work.

4.1 Experimental Parameters and Metrics

Performance Metrics We chose three performance metrics which capture the scheduler's effectiveness in providing both good response times and high system efficiency.

- **Job Response Time** is the wall-clock time from job initation to completion.
- **CPU Time** is the sum of system and user CPU time for the job. In all experiments, the system time was less than 5%, so this basically captures user CPU time.
- **Fairness** is the degree to which threads are allocated equal shares of processor time. We normalize CPU share to the ideal equal share and define a fairness fraction F. An ideal fairness fraction is unity.

$$F = N\frac{T_C}{T_E} \tag{2}$$

  Where T_C is the CPU time consumed by a process over its lifetime, T_E is its job response time, and N is the number of processes running on the machine.

In all graphs, the mean of several runs is shown along with 90% confidence intervals computed using Student's T-distribution. In many cases, the confidence intervals are too small to be seen on the graph.

Workload We used three distinct workloads in our experiments. These workloads consist of parallel kernels, run in competition with sequential jobs that compete for the CPU. The choice of a single parallel job is dictated by FM's current limitation to a single parallel job. The sequential competitors for the first two workloads are processes that execute a simple spin loop and run for the duration of the test; for the third workload they are real applications, as described below. The parallel kernels are designed to exercise different aspects of performance:

- **Latency** is a simple token-passing benchmark in which two nodes repeatedly exchange a 128-byte packet. Each node records the elapsed wall-clock time for each round trip, including the sending and receiving of the packet. System calls for timing add approximately three microseconds to each round trip.
- **Barrier** performs a sequence of barriers, interspersing local computation between the barriers. It provides a pattern of communication and synchronization different from the latency benchmark's. The root node initially broadcasts a "passed barrier" message to all nodes, then all nodes enter a spin loop (local computation). After the local computation is complete, each node sends an "at barrier" message to the root node. When the root node has received "at barrier" messages from all nodes, the loop begins anew.
- **Mixed Workload** consists of three programs in competition on the cluster. The parallel job is a SOR kernel, a two-dimensional Laplace's equation solver on a 128×128 element matrix (written in FORTRAN on an FM-based

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 2. Applications and arguments used for the Mixed Workload

implementation of MPI [11]). The sequential jobs are GNU tar, archiving and compressing a collection of 97 files, totalling 2.1 MB, and Ghostscript, a PostScript interpreter on a 1.7 MB, 103-page PostScript file. All files were read from a remote NFS filesystem.

Scheduling Variants The scheduling types used vary both the synchronization method (blocking or spinning) and whether demand-based coscheduling is included.

- ⟨**No DCS, spin only**⟩ The base Solaris 2.4 scheduler and FM using spin-based polling for incoming messages.
- ⟨**No DCS, spin-block**⟩ The base Solaris 2.4 scheduler and FM using spin-block synchronization, blocking after 1600 microseconds of spinning.
- ⟨**DCS, spin only**⟩ The Solaris 2.4 scheduler augmented with demand-based coscheduling, using spinning synchronization (without blocking). The relevant parameters are E, the exponent for the run queue length factor, and C, the offset constant.
- ⟨**DCS, spin-block**⟩ The Solaris 2.4 scheduler augmented with demand-based coscheduling, using spin-block synchronization, blocking after 1600 microseconds of spinning. The relevant parameters are E, the exponent for the run queue length factor, and C, the offset constant.

4.2 Experimental Results

To begin, we present results from our Latency and Barrier benchmarks. These kernels illuminate the behavior of DCS and the underlying Solaris 2.4 scheduler using different synchronization approaches — spinning and spin-block. Subsequently, we consider performance on the the Mixed Workload benchmark.

Latency and Barrier benchmarks

Job Response Time. The response times for the Latency and Barrier benchmarks for a range of coscheduling approaches are shown in Figures 3, 4, 5, and 6. Note that the vertical scale in the job response time graphs is logarithmic, to accomodate the extremely long times for uncoordinated scheduling. To see the differences on a linear scale for just ⟨DCS, spin-block⟩ and ⟨No DCS, spin-block⟩, see Figures 7 and 8.

For each graph, the number of sequential competitor jobs per node varies along the X-axis and the wall-clock time to complete the benchmark varies along the Y-axis. For each number of competitors, wall-clock times for four different scheduling approaches are presented. As mentioned above, the sequential competitors run for the duration of the test; the test ends when when the parallel job terminates.

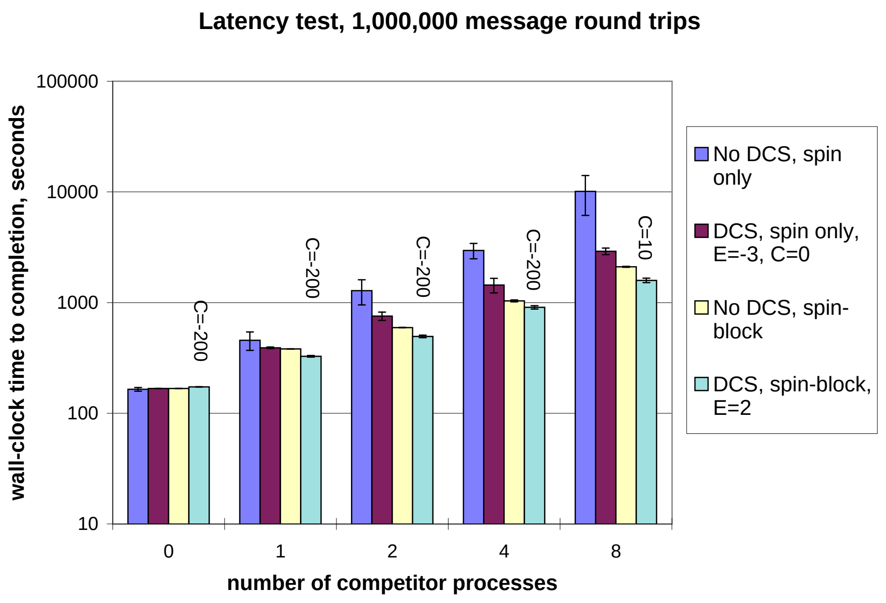

Fig. 3. Wall-clock time (job response time) in the latency benchmark for a variety of schedulers and numbers of competitors. Note the log scale; differences are larger than they appear — see Figure 7 for fine details.

In the presence of competition, the ⟨No DCS, spin only⟩ configuration performs poorly, and especially so for the Barrier benchmark. However, either changing to spin-block or adding DCS improved the effective coscheduling for both benchmarks. For the Latency benchmark, spin-block both with and without DCS are clearly more effective than ⟨DCS, spin only⟩, whereas for the Barrier benchmark, ⟨No DCS, spin-block⟩ and ⟨DCS, spin only⟩ have job response times that are quite close to each other, although ⟨DCS, spin only⟩ is slower. In all cases,

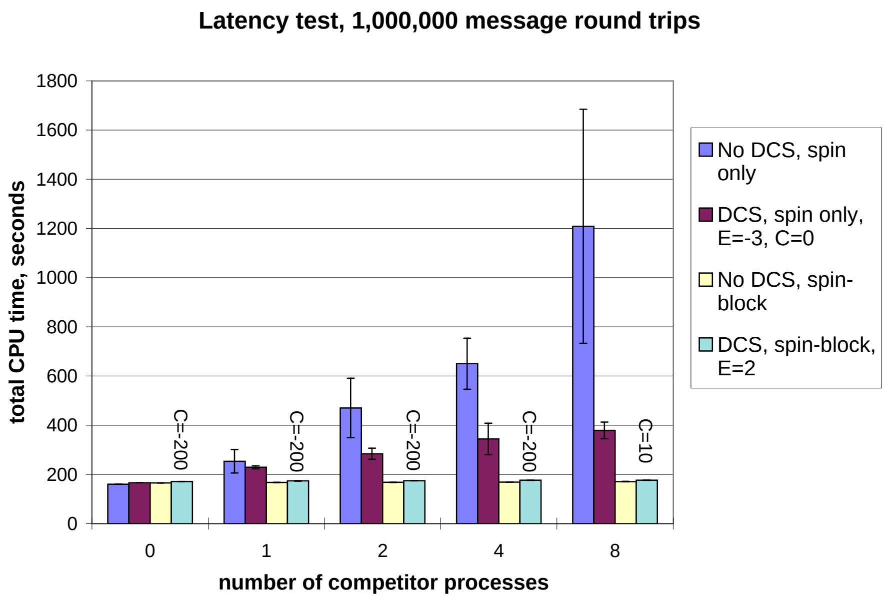

Fig. 4. CPU time in the latency benchmark for a variety of schedulers and numbers of competitors.

the policy ⟨DCS, spin-block⟩ produces the best coscheduling performance for these benchmarks.

Fairness. Figures 9 and 10 contain fairness data for the Latency and Barrier benchmarks. In each graph, the number of competitors varies along the X-axis, and the fairness metric of Equation 2 is given on the Y-axis. As before, for each number of competitors, data for several scheduling approaches are shown.

The fairness metric clearly distinguishes the different approaches to coscheduling. For the Latency benchmark, the base Solaris 2.4 scheduler delivers slightly more than a fair share of CPU for the parallel job. The ⟨No DCS, spin-block⟩ approach is a weak competitor, obtaining less and less than its fair share as the number of competitors is increased. However, in all cases the DCS approaches come closer to or even surpass a fair share of CPU. Thus we can see that DCS is a stronger competitor for CPU, whereas the ⟨No DCS, spin-block⟩ approach is a weaker competitor.

Mixed Workload

Job Response Time. The job response and CPU time results for the Mixed Workload benchmark are shown in Figures 11 and 12. The top four sets of bars

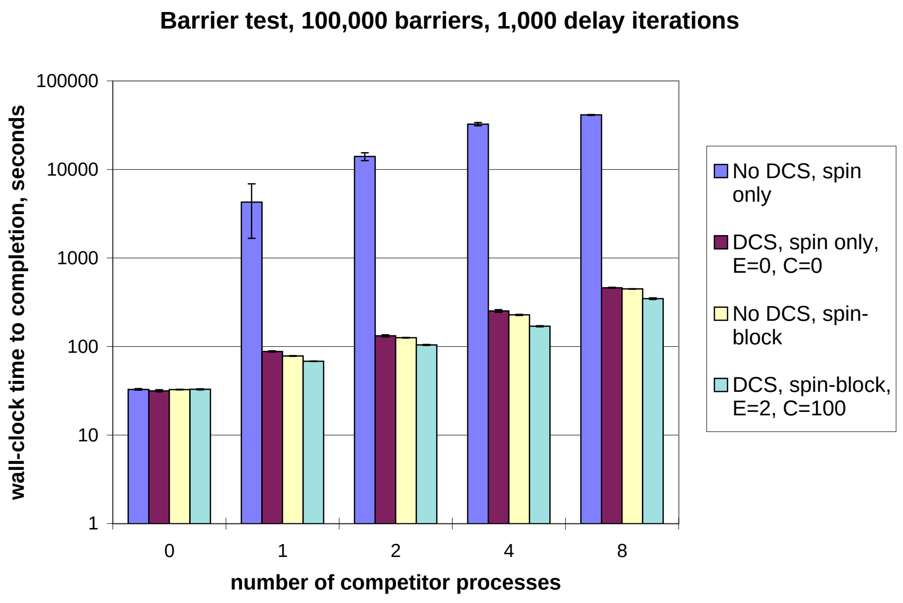

Fig. 5. Barrier Benchmark wall-clock times (job response times) under a variety of schedulers and numbers of competitors. Uniform 1,000 delay iterations (78 μsec) used on all nodes. Note the log scale used in the job response time graphs; differences are larger than they appear — see Figure 8 for fine details.

correspond to timesharing workloads in which all jobs are started simultaneously. The response time for each job is at its right end. Batch performance is shown as a baseline at the bottom.

For the mixed workload, all three of the schedulers employing either DCS or spin-block achieved system efficiencies and therefore job response times close to that of batch. The only scheduler which produced poor performance was the ⟨No DCS, spin only⟩ version, which fails to achieve coscheduling. Amongst the other schedulers, the combination of ⟨DCS, spin-block⟩ gives the best job response time, with ⟨No DCS, spin-block⟩ and ⟨DCS, spin only⟩ not far behind.

The CPU time data (in Figure 12) clearly capture the benefits of coscheduling on parallel job efficiency. While the CPU times for the sequential jobs are fairly consistent over the four schedulers, the CPU time for the parallel job, SOR, varies widely. In the absence of coscheduling, it is nearly four times larger than in the batch case. The ⟨DCS, spin only⟩ scheduler is able to reduce this CPU time significantly, by achieving coscheduling, however the spinning synchronization still increases CPU time by nearly one half. Adding spin-block effectively

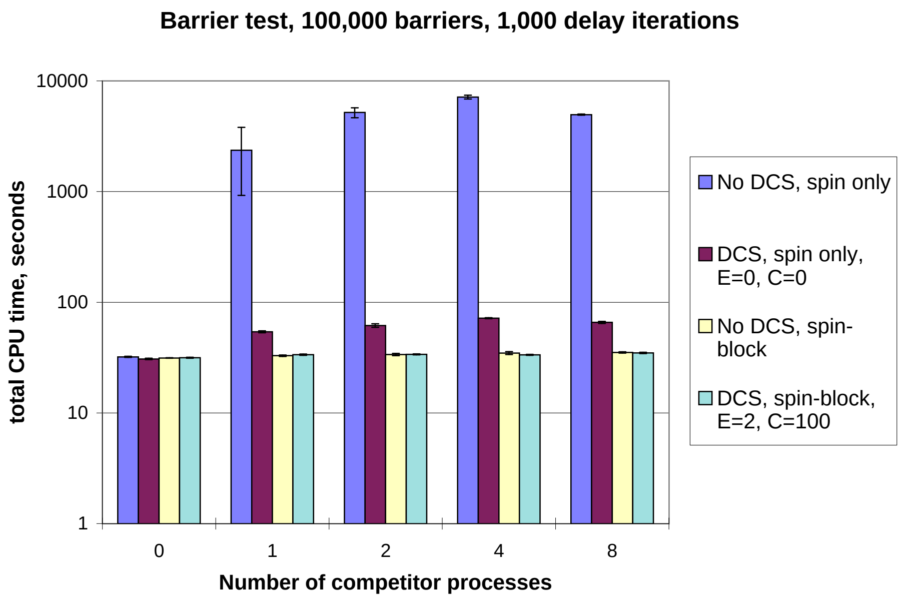

Fig. 6. Barrier Benchmark CPU times under a variety of schedulers and numbers of competitors.

eliminates the CPU time increase with the ⟨DCS, spin-block⟩ case giving best performance.

Fairness. Because jobs in the Mixed Workload test have different lifetimes, fairness is measured over the period until the first job to terminate does so; this is a shorter period than in the Barrier and Latency tests, where all processes run equally long and fairness is measured over the entire experiment. The results are shown in Figure 13. As with our Latency and Barrier benchmarks, several key characteristics are clear: the base Solaris scheduler (⟨No DCS, spin only⟩) is unfair, the ⟨DCS, spin only⟩ scheduler is least fair, the ⟨DCS, spin-block⟩ scheduler is able to use its fair CPU share, however, the ⟨No DCS, spin-block⟩ scheduler does not.

4.3 Analysis

We sought in our experiments to answer the following questions:

– Can dynamic coscheduling achieve coscheduling of processes in a parallel application?
– What are the effects of using spinning synchronization versus those of using spin-block synchronization?

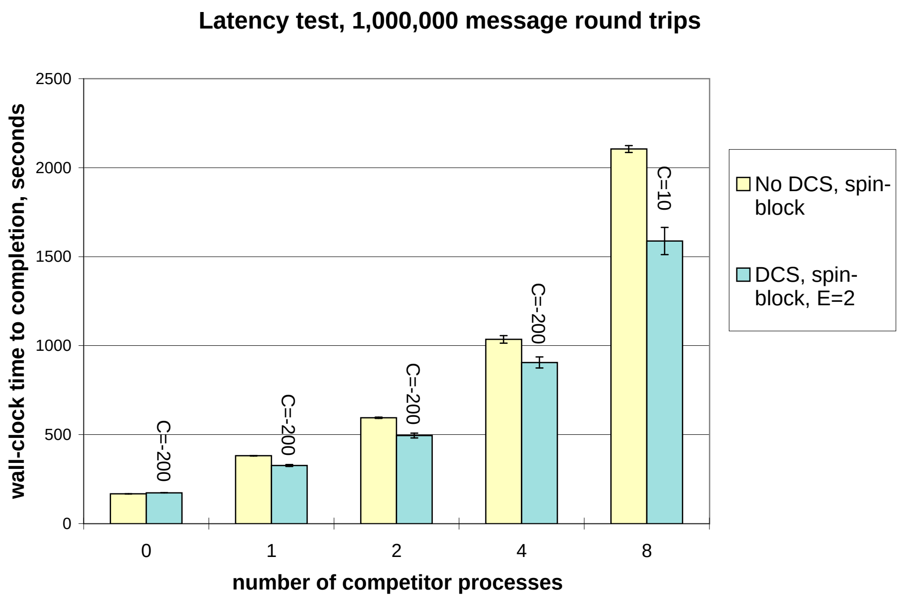

Fig. 7. Latency Benchmark wall-clock times (job response times) shown with a linear scale for the cases ⟨DCS, spin-block⟩ and ⟨No DCS, spin-block⟩. This is the same experiment as in Figure 3, with the linear scale to show detail.

– What are the effects on fairness, efficiency, and response time of using DCS?

In all three workloads, DCS clearly coscheduled the parallel applications. This can be seen from both the reduced job response times and the fact that fairness could be achieved even under spin-block synchronization. More detailed evidence, in the form of histograms of message round-trip times, is presented in [16]. The unmodified Solaris 2.4 scheduler also achieved some coscheduling under spin-block synchronization, as is discussed at greater length in Section 5. However, the unmodified Solaris 2.4 scheduler (⟨No DCS, spin only⟩) failed to achieve coscheduling under spinning synchronization, as indicated by increased CPU and response times.

Under spinning synchronization, DCS significantly reduced both CPU and job response time. However, to achieve better coscheduling, it was necessary to allow DCS to use up to 30% more than its fair share of CPU time. For the ⟨DCS, spin only⟩ scheduler, there is a clear tradeoff between fairness and coscheduling. It appears necessary to elevate the priority of parallel jobs to achieve coscheduling. Even with this priority elevation, efficiency declined with increasing load for ⟨DCS, spin only⟩, but not nearly as quickly as for the base ⟨No DCS, spin only⟩ scheduler.

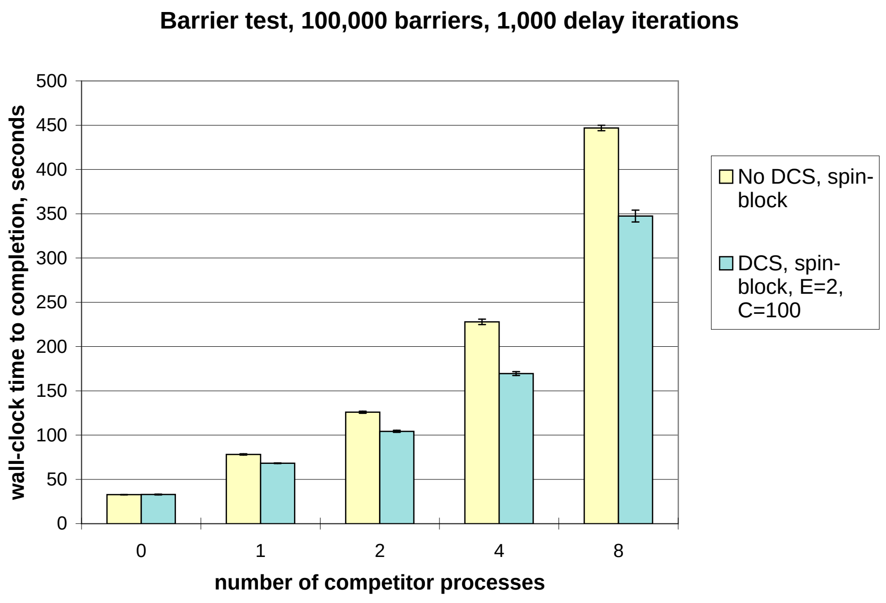

Fig. 8. Barrier Benchmark wall-clock times (job response times) shown with a linear scale for the cases ⟨DCS, spin-block⟩ and ⟨No DCS, spin-block⟩. This is the same experiment as in Figure 5, with the linear scale to show detail.

Under spin-block synchronization, coscheduling of a single parallel job could be achieved, and DCS allows fairness to be finely controlled to almost perfect values in most cases. Without DCS, parallel jobs obtained as much as 20% less than their fair share of CPU time. In effect, this meant that parallel jobs scheduled under ⟨DCS,spin-block⟩ in the presence of competition achieved better job response times than those scheduled under ⟨No DCS, spin only⟩.

CPU times were nearly identical for schedulers which achieved coscheduling. In the case of the latency test, DCS required slightly more CPU time than the ⟨No DCS, spin only⟩ scheduler. However, we have tracked this anomaly down an extra 5 microseconds latency being required for each message transmission, because of several extra instructions performed on message receipt under DCS in the Lanai control program's main loop [16] .[7] The effect is not apparent for the Barrier benchmark and Mixed Workload benchmark, because additional computation is performed which allows the overhead to be overlapped.

[7] This effect would be reduced with faster network controllers.

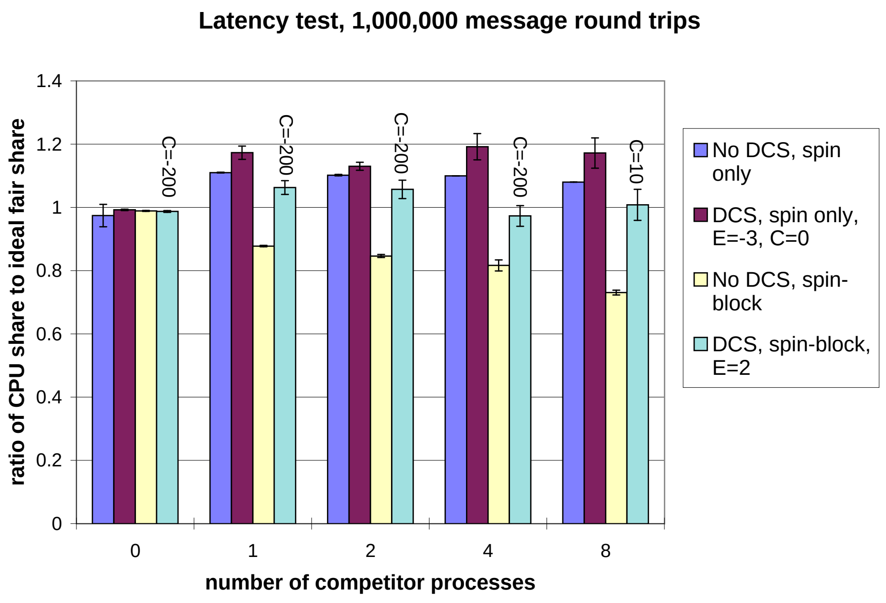

Fig. 9. Fairness metric for the Latency benchmark.

5 Discussion

In this section we examine the mechanisms responsible for our experimental results. We first discuss the workings of DCS and their implications; then we discuss the coscheduling effects of spin-block message receipt in schedulers that provide a priority boost on process wakeup; finally we describe directions for future work.

5.1 Mechanisms for Coscheduling with DCS

DCS achieves coscheduling for parallel processes because the arrival of a message for a process not currently running can cause it to be scheduled immediately. In programs with fine-grained communication, the sender and receiver are scheduled together and run until one of them blocks or is preempted. Larger collections of communicating processes are coscheduled by transitivity. Our experiments indicate that this basic, low-level mechanism effectively coschedules parallel processes under both spinning and spin-block synchronization.

DCS enables applications with spinning synchronization to execute more efficiently and thereby achieve lower job response times when compared to the unmodified Solaris 2.4 scheduler. In effect, the coscheduling allows the use of

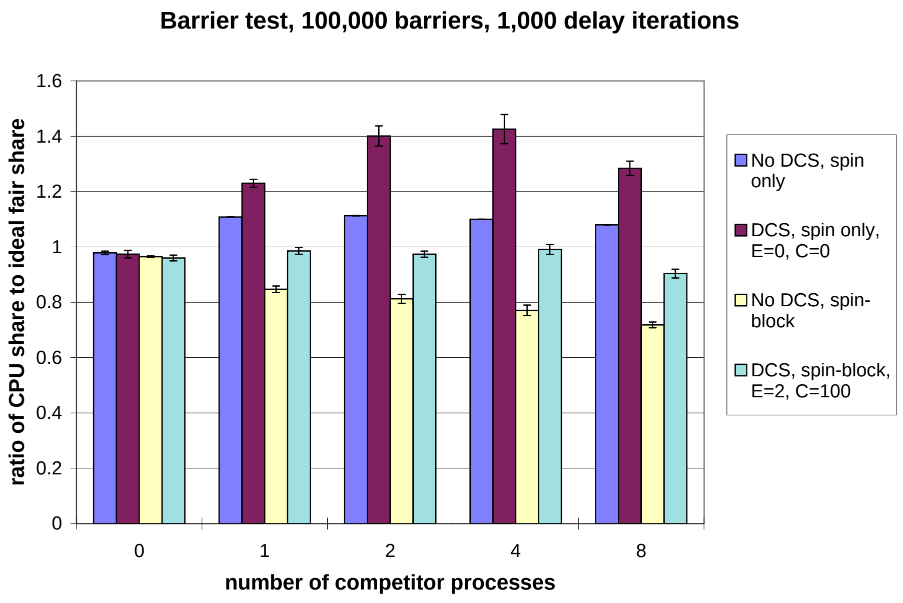

Fig. 10. Fairness metric for the Barrier benchmark.

synchronization strategies previously only viable on batch scheduled or dedicated machines. However, our experiments also show that spinning synchronization under DCS is less efficient than spin-block synchronization (further data are reported in [16]). We believe that this problem is exacerbated in Solaris 2.4 by the varying scheduling quanta used by the dispatcher (shown in Table 1), which can cause even processes that start their timeslices in synchrony to suffer long spinning phases when one ends its timeslice before others.

5.2 Coscheduling and Synchronization Mechanisms

For all three benchmarks, DCS with spin-block or spinning message receipt achieved coscheduling. The unmodified Solaris 2.4 schedule with spin-block synchronization (⟨No DCS, spin-block⟩) also achieved coscheduling, though with less success as the system load increases. Because spin-block is a weak competitor for CPU (each block yields), the parallel jobs use progressively less of their fair share of the processor with increasing load. This suggests that an explicit priority boosting mechanism for coscheduling may be appropriate for multitasking parallel system.

The ⟨DCS, spin-block⟩ and ⟨No DCS, spin-block⟩ schedulers were demonstrated to be most effective in achieving coscheduling efficiently (the ⟨DCS, spin-block⟩

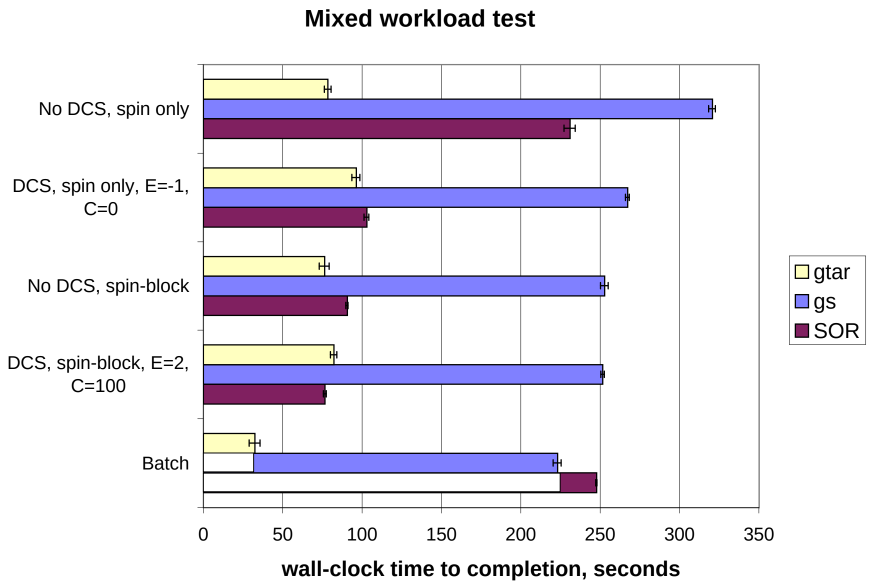

Fig. 11. Job response time in the mixed workload test. In the batch case, jobs were executed in sequence. (N.b.: axes and colors have different significance in these graphs than in the Latency and Barrier results.)

scheduler was slightly better). We explore the surprising reasons for this in the following discussion.

Spin-block synchronization in combination with the Solaris 2.4 scheduler was reported to achieve coscheduling [3] (for a synthetic bulk-synchronous workload). The posited mechanism for this *implicit scheduling* is the following: a process spins awaiting the arrival of a message and blocks if the message does not arrive in a short period; when the message arrives, because of the priority boost described in Section 3.2, the process will often be scheduled immediately, resulting in coscheduling. This priority boost is a characteristic of SVR4 derived schedulers and designed to enable input/output intensive jobs to get higher priority.[8]

Our experiments confirm that this priority boost is the mechanism producing coscheduling for implicit scheduling. We ran the Barrier benchmark (see

[8] Not all Unix priority-decay schedulers implement this boost in this way; for example, the OSF/1-derived Digital Unix gives a priority boost only to processes sleeping uninterruptibly in the kernel, which would exclude some ways of implementing spin-block message receipt. However, based on the results reported in [3] and our own work, it seems clear that the SVR4 approach is a very useful one for coscheduling.

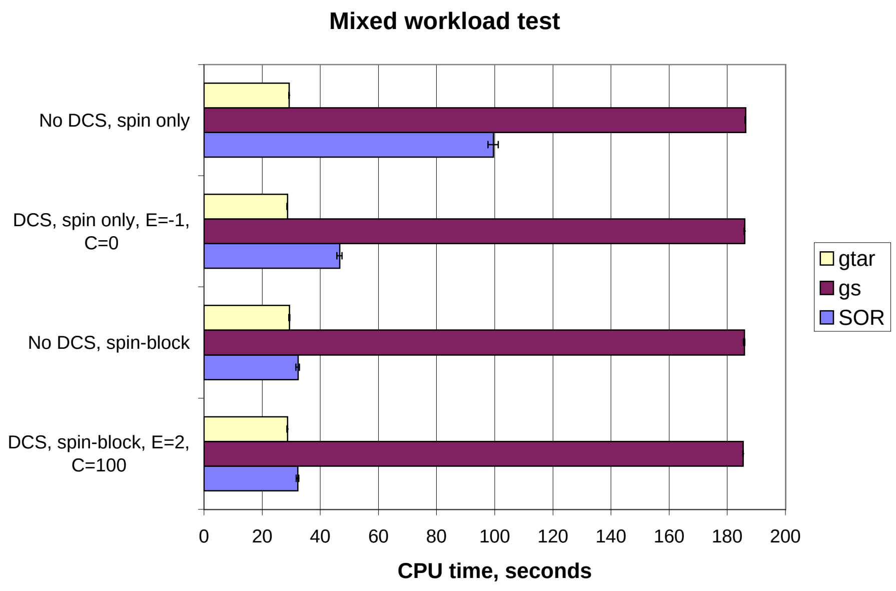

Fig. 12. CPU time in the mixed workload test.

Section 4.2) with an altered timesharing dispatcher table that cancels the priority boost by giving reawakened processes the same priority they had when they blocked. Specifically, in Table 1, we set the value of `ts_slpret` for each queue n to n. The results (see Figure 14) show that without the priority boost, no coscheduling is achieved. Job response times for spin-block are similar to those under spinning synchronization. This explains why the ⟨DCS, spin-block⟩ and ⟨No DCS, spin-block⟩ yielded such similar performance: the priority boosting performed by the Solaris 2.4 scheduler causes the processes receiving messages to be run immediately on message arrival — which is exactly how DCS works.

While the combination of the Solaris 2.4 scheduler and spin-block synchronization mimics DCS in this case, there are a wide variety of opportunities for DCS to coschedule programs with other characteristics. For example, DCS can schedule on message arrival programs that are not on sleep queues (due to polling, infrequent communication, asynchronous communication, or one-sided data movement). DCS can also be used to coschedule sets of threads, whereas the SVR4 and spin-block combination is only applicable for single thread coscheduling.

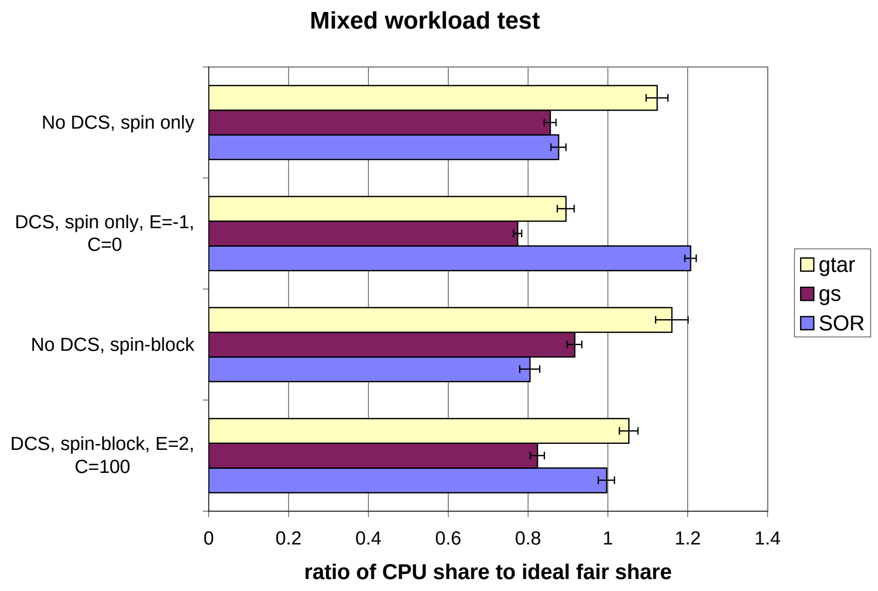

Fig. 13. Fairness in the mixed workload test. Note that fairness is shown here for competitor jobs as well.

5.3 Directions for Future Work

Experiments that vary the granularity of communication [16] indicate that spin-block message receipt paired with DCS or the unmodified Solaris 2.4 scheduler (with priority boosts on process wakeup) is less successful at coscheduling as the frequency of communication decreases. With relatively coarse-grained communication, DCS is more successful at coscheduling than the unmodified Solaris 2.4 scheduler, but no longer causes parallel processes to receive their full share of the CPU. Characterizing this sensitivity with a broader workload and exploring mechanisms to improve robustness of coscheduling (perhaps by artificially increasing communication frequency) are interesting topics for further research.

DCS with spinning synchronization is not as efficient as spin-block synchronization because processes which are not coscheduled simply spin until the end of their timeslice. Improving the efficiency of DCS with spinning synchronization is desirable, but must be achieved while maintaining fairness. We plan to explore a variety of approaches, including spin-yield synchronization, time slice size matching, or some form of shared priority for parallel jobs.

Our current prototype does not automatically achieve fairness — it isn't self calibrating. We originally thought a single setting of the fairness parameters would work for almost all cases, but in fact different parameter settings

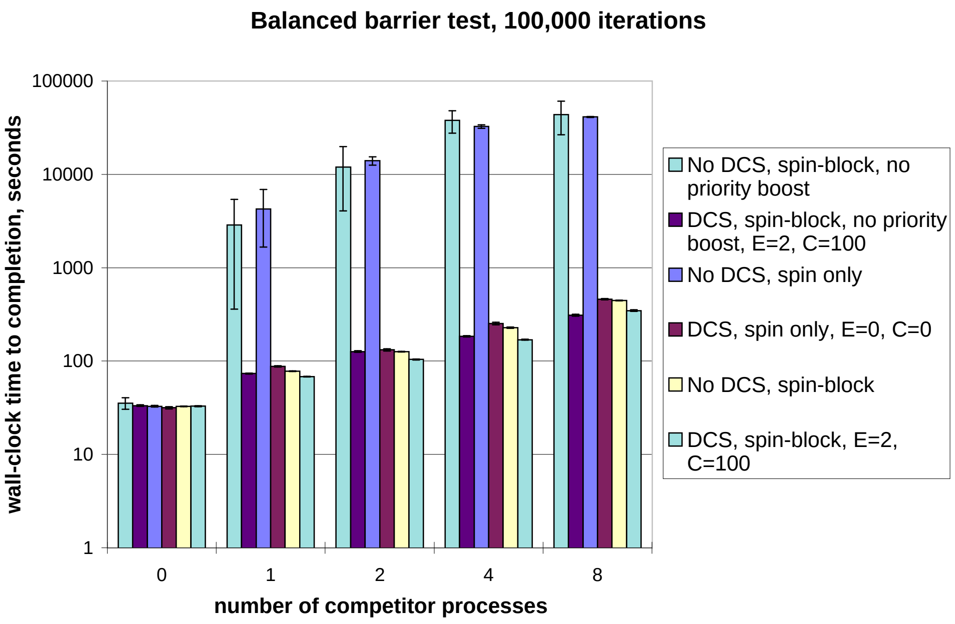

Fig. 14. Barrier test response times with the Solaris 2.4 priority boost for newly-awakened processes disabled. For comparison, times with the normal Solaris 2.4 dispatcher table are also shown.

were required for individual experiments. An automatic mechanism is clearly required for any robust coscheduling system. Obviously, the widespread use of priority-decay schedulers complicates the situation — stride schedulers [22] or other proportional share schedulers would provide a better platform for DCS, by separating the concepts of execution order and processor share. Our current efforts in the context of priority-decay schedulers focus on obtaining more accurate estimates of recent run time to implement an equalization criterion [17].

The experiments described in this paper have two key limitations: a single parallel job and a cluster of only seven nodes were used in all benchmarks. Future DCS experiments must include multiple parallel jobs, larger workloads, and larger configurations. Such broadening would enable evaluation of a wide range of issues, including the viability of the epoch number mechanism [17] for multitasking parallel jobs. Such efforts are already underway as part of the Illinois High Performance Virtual Machines Project.

Most of the workloads used to study coscheduling [3, 16] are quite regular in structure, communication, and computation. Such regularity often fails to exercise the behavioral richness of dynamic systems. Study of more irregular workloads and background loads are desirable.

6 Summary

We have presented an implementation of dynamic coscheduling on a cluster of workstations using a high-performance messaging layer. Experimental results found using our DCS implementation show that dynamic coscheduling can provide good performance for a 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: CPU times were nearly the same as for batch processing, and DCS reduced job response times by up to 20% over implicit scheduling while maintaining near-perfect fairness. Under spinning message receipt, DCS improved efficiency and reduced job response times by as much as two decimal orders of magnitude over the uncoordinated scheduling of the base Solaris 2.4 scheduler, with only a small penalty in fairness.

Further research remains to be done, with multiple parallel processes, a larger workstation cluster, and different sorts of test workloads, but our results to date show that DCS is a promising means of achieving coscheduling in workstation clusters.

Acknowledgments

Bert Halstead, Larry Rudolph, and Chandu Thekkath have all given us valuable suggestions and comments on the work described here. Mario Lauria helped us to use his implementation of MPI on FM. Roy Campbell and Dave Raila provided the workstation cluster at UIUC that we used in our experiments.

Sobalvarro and Weihl were 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.

Pakin and Chien are supported in part by in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grants MIP-92-23732, and NASA grant NAG 1-613. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, Microsoft, and Motorola is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1. 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`.

2. 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.
3. 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`.
4. Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, May 1990.
5. Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.
6. 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.
7. Richard B. Gillett. Memory Channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996. Available from `http://www.computer.org/pubs/micro/web/m1gil.pdf`.
8. 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`.
9. D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1), Feb. 1992.
10. Sun Microsystems Inc. `ts_dptbl(4)` manual page. *SunOS 5.4 Manual.* Section 4.
11. Mario Lauria and Andrew Chien. MPI-FM: High performance MPI on workstation clusters. Submitted to the Journal of Parallel and Distributed Computing. Available from `http://www-csag.cs.uiuc.edu/papers/mpi-fm.ps`.
12. John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, October 1982.
13. Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 1997.
14. Scott Pakin, Mario Lauria, Matt Buchanan, Kay Hane, Louis Giannini, Jane Prusakova, and Andrew Chien. *Fast Messages 2.0 User Documentation*, October 1996.
15. 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`.
16. Patrick G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors.* PhD thesis, Massachusetts Institute of Technology, 1997. MIT/LCS/TR-710.
17. 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.

18. 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`.
19. 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`.
20. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.
21. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995. Available from `http://www.cs.cornell.edu/Info/Projects/ATM/sosp.ps`.
22. Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, 1995. MIT/LCS/TR-667.