Scheduling with Implicit Information in Distributed Systems

Andrea C. Arpaci-Dusseau, David E. Culler, Alan M. Mainwaring

Computer Science Division
University of California, Berkeley
{dusseau, culler, alanm}@cs.berkeley.edu

Abstract

Implicit coscheduling is a distributed algorithm for time-sharing communicating processes in a cluster of workstations. By observing and reacting to implicit information, local schedulers in the system make independent decisions that dynamically coordinate the scheduling of communicating processes. The principal mechanism involved is *two-phase spin-blocking*: a process waiting for a message response spins for some amount of time, and then relinquishes the processor if the response does not arrive.

In this paper, we describe our experience implementing implicit coscheduling on a cluster of 16 UltraSPARC I workstations; this has led to contributions in three main areas. First, we more rigorously analyze the two-phase spin-block algorithm and show that spin time should be increased when a process is receiving messages. Second, we present performance measurements for a wide range of synthetic benchmarks and for seven Split-C parallel applications. Finally, we show how implicit coscheduling behaves under different job layouts and scaling, and discuss preliminary results for achieving fairness.

1 Introduction

Scheduling parallel applications in a distributed environment, such as a cluster of workstations [2], remains an important and unsolved problem. For general-purpose and developmental workloads, time-sharing approaches are attractive because they provide good response time without migration or execution-time predictions. However, time-sharing has the drawback that communicating processes must be scheduled simultaneously for good performance. Over the years, researchers have developed time-sharing approaches more suitable for general-purpose workloads in a cluster of workstations.

Local scheduling, where each workstation independently schedules its processes, is an attractive time-sharing option for its ease of construction, scalability, fault-tolerance, workload adaptivity, and autonomy [8, 15, 19, 30]. However, with local scheduling, the performance of fine-grain communicating jobs is orders of magnitude worse than with explicit coscheduling because the scheduling is not coordinated across workstations.

Explicit coscheduling [24] ensures that the scheduling of communicating jobs is coordinated by constructing a static global list of the order in which jobs should be scheduled; a simultaneous global context-switch is then required across processors. Unfortunately, straight-forward implementations are neither scalable nor reliable; hierarchical constructions can remove single points-of-failure and scalability bottlenecks, but only with increased implementation complexity [14]. Explicit coscheduling also requires that the schedule of communicating processes be precomputed, which complicates the coscheduling of client-server applications and requires pessimistic assumptions about which processes communicate with one another. Simulations have shown that communicating processes can be identified at run-time, but local schedulers still must agree on a common schedule and context-switch time [16]. Finally, explicit coscheduling of parallel programs interacts poorly with interactive jobs and with jobs performing I/O [3, 5, 13, 22].

Alternatively, with *dynamic* or *implicit coscheduling*, independent schedulers on each workstation coordinate parallel jobs through local events that occur naturally within the communicating applications [11, 25]. The work stemming from Sobalvarro et al's initial simulations achieves dynamic coscheduling by scheduling processes on message arrival [7, 27, 26]. The simulations in [11] have instead used *two-phase spin-blocking* as the primary mechanism for implicit coscheduling. In this paper, we continue to use two-phase spin-blocking to achieve coordination, extending the previous results in several ways.

First, the previous work has consisted of either only simulations on synthetic workloads [11, 27] or implementations supporting *one* parallel job in competition with multiple sequential jobs [7, 26]. In this paper, we describe the first implementation of implicit coscheduling that is effective for multiple parallel programs with different communication characteristics. Our measurements on a cluster of 16 UltraSPARC 1 workstations connected with a high-speed network show that most applications are implicitly coscheduled with performance similar to an ideal model of explicit coscheduling. We examine a more extensive range of synthetic benchmarks than previous work, as well as seven Split-C parallel applications. We also show that implicit coscheduling is robust to the layout of jobs across processors and to changes in the local priority-based scheduler.

Experience with our implementation and applications other than the bulk-synchronous benchmarks analyzed in our previ-

To appear in the
SIGMETRICS'98/PERFORMANCE'98 Joint Conference on the Measurement and Modeling of Computer Systems, June 1998, Madison, Wisconsin.

Information	Type	Local Observation:	Remote Implication	Local Action	Mechanism
Response Time	Inherent	Fast:	Remote scheduled	Stay scheduled	Two-phase spin-block
		Slow:	Remote not scheduled	Relinquish processor	Two-phase spin-block
Message Arrival	Inherent	Request:	Sender scheduled	Increase Spin-Time	Two-phase spin-block
		Response:	Receiver scheduled	Wake original sender	Two-phase spin-block
Scheduling Progress	Derived	Starving locally:	Starving globally	Schedule job more	Raise priority
		Acceptable locally	Acceptable globally	(No action)	(None)

Figure 1: **Implicit Information.** *The table summarizes the information that is locally available to each scheduler, the remote implication of each event, the local action that leads to effective implicit coscheduling, and the mechanism we use to achieve this behavior.*

ous simulations has helped us to refine the calculations for spin time. We show that the spin time before a process relinquishes the processor at each communication event consists of three components. First, a process should spin for the *baseline* time for the communication operation to complete; this component keeps coordinated jobs in synchrony. Second, the process should increase the spin time according to a *local* cost-benefit analysis of spinning versus blocking. Third, the process should spin longer when receiving messages from other processes, thus considering the impact of this process on others in the parallel job.

The rest of this paper is organized as follows. We begin in Section 2 by discussing the goals and philosophy of using implicit information for coscheduling. In Section 3 we show how to empirically derive spin-time as a function of several system parameters. In Section 4 we describe our initial implementation. We present measurements on both synthetic and real applications in Section 5. In Section 6, we vary the number of jobs, the layout of jobs across processors, and the communication characteristics of the jobs in competition with one another. Finally, we conclude and describe future work in Section 7.

2 Implicit Coscheduling

2.1 Implicit Services

Building high-performance, highly-available services for any system with general-purpose workloads is a challenging task. Implementing these services in a *distributed* environment introduces even more challenges. For example, not only must the single-node performance be acceptable, but performance must scale as the system grows. The system must be robust to node failures, as well as allow new nodes to be added. Finally, the mechanisms for the global environment must not interfere with the functionality of the previously well-tuned local system. These three goals of scalability, reliability, and autonomy, are more difficult to obtain due to communication between the participating components.

The simplest distributed service to construct contains *no additional communication* between components in the system; instead, each local service makes intelligent, independent decisions. We define an *implicit system* as a system where cooperating clients infer remote state from local events, and thus make decisions that lead to a global goal. An implicit service performs no additional communication beyond that inherent within the applications it supports. Note that this classification is much stricter than defining a system as *dynamic*, where a system may adapt to underlying behavior, but may require additional communication between components in the system.

We classify implicit information into two categories: *inherent* and *derived*. Inherent information is propagated across nodes regardless of the behavior of the local components. Derived information requires knowledge of how the local components react to the inherent events.

2.2 Local Information: Events and Implications

In this section, we present the implicit information available for implicit coscheduling. We describe two inherent events, *response time* and *message arrival*, as well as a third derived event, *scheduling progress*. For each event, we describe the implied remote scheduling state, the desired local action, and an efficient mechanism for implementing this action. The three pieces of implicit information are summarized in Figure 1.

The first inherent piece of information, *response time*, is the time for the response to a message request to return to the sending process. Assuming the destination process must be scheduled for a response to be returned, a fast response indicates to the sending node that the corresponding destination process is probably currently scheduled. Therefore, the desired action for dynamic coscheduling is to keep the sender scheduled. Conversely, if the response is not received in a timely fashion, the sending node can infer that the destination is probably *not* scheduled; thus, it is not beneficial to keep the sender scheduled.

The mechanism that achieves these desired actions is *two-phase spin-blocking*. With two-phase spin-blocking, a process spins for some amount of time, and if the response arrives before the time expires, it continues executing. If the response is not received within the threshold, the process voluntarily relinquishes the processor so a competing process can be scheduled. In this paper, we determine the appropriate spin time as a function of system and communication parameters.

The second inherent event, *message arrival*, is the receipt of a message from a remote node. When a message arrives, the implication is that the corresponding remote process was recently scheduled. Therefore, it may be beneficial to schedule, or keep scheduled, the receiving process.

Our implementation of implicit coscheduling uses incoming messages to either wake-up processes that are sleeping while waiting for a message response, or to increase the spin-wait time of currently scheduled processes. Therefore, we require no additional mechanism beyond two-phase spin-block to react to arriving messages; this is in contrast with Sobalvarro et al.'s implementation [25], where a message arrival may increase the priority of the intended process and requires new interactions with the local scheduler.

The third event, *scheduling progress*, is the rate at which a local process makes forward progress and is derived from knowledge of component behavior. While the previous two events were required for implicit coscheduling to converge efficiently, this third event is used to ensure that competing processes are scheduled *fairly*.

Given that processes relinquish the processor when waiting for message responses, each of the local schedulers can determine if a job is making forward progress by observing how much it has been scheduled recently. If the job performs fine-grain communication, the local scheduler can infer that the corresponding processes on other nodes have also made progress, since the forward progress of each process depends on others being scheduled. If the job performs little communication, no such inference can be made, but, in that case, coordinated scheduling across nodes is not required. For fair scheduling, each local scheduler should bias future decisions towards jobs which have received less CPU time.

3 Determining Spin-Time

The most crucial component of implicit coscheduling is that processes perform a two-phase spin-block when waiting for any result from a remote process. The goal of two-phase spin-blocking is to keep coordinated processes coscheduled, and to let uncoordinated processes relinquish the processor.

In this section, we begin by describing our model of parallel applications and then derive the appropriate spin-time in three steps. First, processes should spin the *baseline* amount which keeps processes coscheduled when they are already so; this requires that processes spin for the time of a communication operation when all involved processes are scheduled. The second and third steps adjust the baseline spin time by comparing the *cost* to the *benefit* of spinning longer. The second component accounts for the cost and benefit from the perspective of the local process, while the third component examines pairs of communicating processes.

3.1 Background: Process Model

In the terminology in this paper, a *job* refers to a set of communicating processes. This job may be a dynamic, changing collection of processes, such a server with multiple clients, or a more traditional static, predefined collection, as in a parallel application. A *process* is the unit scheduled by each local operating system scheduler.

We consider two basic communication operations: *reads*, request-response messages between pairs of processes, and *barriers*, messages synchronizing all processes. In our model, the important parameters for characterizing an application are the time between reads, the time between barriers, and the load-imbalance across processes (or the difference in time between the arrival of the first and the last process at a barrier).

3.2 Baseline Spin: Keeping Processes Coordinated

The baseline component of spin-time, S_{Base}, ensures that processes stay coordinated if already in such a state. Thus, all processes must spin at least the amount of time for the communication operation to complete when all involved processes are scheduled. We now evaluate the expected completion time for reads and barriers in our current implementation and describe how these values match our previous simulations [11].

In general, a request-response message pair is expected to complete in the round-trip time of the network, plus potentially the time, W, to wake the destination process when the request arrives. Using the LogP model [10], where L is the latency of the network and o is the processing overhead of sending and receiving messages, round-trip time is simply $2L + 4o$. Therefore, the read baseline time is $S_{RBase} = 2L + 4o + W$, a value easily determined with simple microbenchmarks. In our implementation, $S_{RBase} = 120\ \mu s$.[1] Our previous simulations did not model communication overhead, o, and therefore used a baseline spin time of $S_{RBase} = 2L + W$, and assumed W was identical to the context-switch cost.

In a parallel application running in a dedicated environment, the time for a barrier operation to complete is the sum of the minimum time for a barrier plus the load-imbalance, v, across the processes. In our system, the minimum time for a barrier across 16 processes was measured with a microbenchmark as $S_{BBase} = 378\ \mu s$.[2] In our simulations, the minimum time for the barrier was identical to a read: $S_{BBase} = 2L+W$. In both environments, the process must spin $S_{BBase}+v$ for the barrier to complete. In the next section, we analyze whether or not it is cost-effective to spin for this entire amount.

3.3 Local Cost-Benefit: Single Process

Spinning for the baseline amount keeps processes coordinated when they are already coscheduled. Additionally, we may need to modify spin time according to a local cost-benefit analysis. For example, if the destination process will be scheduled soon in the future, it may be beneficial to spin longer and avoid the cost of losing coordination and being rescheduled later. On the other hand, when a large load-imbalance exists across processes in the parallel job, it may be wasteful to spin for the entire load-imbalance even when all the processes are coscheduled. In this section, we analyze these two factors from the perspective of a single spinning process to determine the *local* spin component, S_{Local}.

We begin by considering the additional time a process should spin when waiting for a communication operation with no load-imbalance to complete. To do so, we must know the cost of blocking for the current local process. For both reads and barriers, the local process pays an additional penalty to be rescheduled in the future when the desired response arrives: the time to wake-up and schedule a process on a message-arrival, W.

It is well understood that given a penalty, such as W, and a fixed distribution of waiting times, then spinning for a time W is competitive with a ratio of two; that is, the performance of this on-line algorithm is at worst twice that of the optimal off-line algorithm [21]. The problem with directly applying this analysis to spin-time in implicit coscheduling is that the waiting times for each process in the system may change drastically depending upon the spin-time chosen by any other process. This feedback between spin-time and the resulting distribution of wait times is not as significant of a factor in other environments where competitive spin-times have been analyzed, such as acquiring locks in a shared-memory environment [20].

We have empirically determined that spinning for an additional time equal to the penalty W works well for both reads and barriers in the simulations and the current implementation,

[1] We found that a large variation in round-trip and wakeup times exists in practice; therefore, the knee-of-the curve should be used rather than the mean. In our implementation, the knee usually occurred at the time at which 95-97% of the operations had completed.

[2] Our barrier is currently implemented with all processes sending directly to process 0. In this way, in contrast to a tree implementation, the scheduling of nodes higher in the tree does not adversely affect nodes closer to the leaves. We have not yet evaluated the impact of a tree-barrier.

Variable	Description	Value (μs)
W	message wakeup cost	65
$2o + 4L$	round-trip time	55
S_{RBase}	baseline read spin	$2o + 4L + W$ = 120
S_{RLocal}	local read spin	$S_{RBase} + W$ = 185
B	barrier latency	378
S_{BBase}	baseline barrier spin	B = 378
S_{BLocal}	local barrier spin	$S_{BBase} + W$ = 438
V_{Local}	block imbalance	$4W + 2B$ = 1000
T_{Pair}	pairwise message interarrival	$5W$ = 325
S_{Pair}	conditional pairwise spin	T_{Pair} = 325

Figure 2: **Components of Spin-Time.** *The table shows, for our implementation, the time a process should spin at reads and barriers before blocking.*

provided that the base spin time is always used. Spinning for the S_{Base} softens the impact of the spin time chosen by one process on the wait times of other processes in the system.

This analysis corroborates the simulation results that spinning $S_{Local} = S_{Base} + W \approx 2W$ offers the best fixed spin-time performance.[3] In our implementation, all read operations spin for at least $S_{RLocal} = S_{RBase} + W$ and all barriers spin for at least $S_{BLocal} = S_{BBase}+W$. Likewise, when a process is woken on a message arrival that does not complete its current two-phase spin-block, the process spins for an additional W before blocking again.

The second question is how long a single process should spin-wait when load-imbalance exists across processes. With high load-imbalance spinning for the entire completion time of barriers keeps applications coordinated, but has two disadvantages. First, as load-imbalance increases, so does the time processes must spin wastefully before concluding that the scheduling is uncoordinated. Second, at some point of load-imbalance, the penalty for spinning while waiting for the barrier to complete is higher than the penalty for losing coordination.

The term V_{Local} is the load-imbalance at which it is better to spin the minimum amount, S_{BLocal}, and then block, rather than spin for the entire load-imbalance. Thus, V_{Local} is the point at which the expected benefit of relinquishing the processor exceeds the cost of being scheduled again. Analyzing the impact on an individual process, each of which can expect to wait $\frac{V}{2}$ at the barrier, shows:

$$\begin{array}{rrcr} & \text{Blocking Benefit} & > & \text{Blocking Cost} \\ & \frac{V}{2} - S_{BLocal} & > & W \\ & \frac{V}{2} - (B + W) & > & W \\ \Longrightarrow & V_{Local} & > & 4W + 2B \end{array}$$

[3]The simulations made the approximation $2L + 2W \approx 2W$, because the context switch time was large ($W = 200\mu s$) relative to network latency ($L = 10\mu s$).

3.4 Pairwise Cost-Benefit: Incoming Messages

Performing cost-benefit analysis not only for a single process, but also between pairs of communicating processes, improves the performance of implicit coscheduling. Intuitively, a process handling requests is benefiting the job as a whole and should spin longer than one that is not receiving messages. While the previous two components of spin time were constants, pairwise spin-time, S_{Pair} only occurs when other processes are sending to the currently spinning process, and is therefore conditional.

The following analysis considers a pair of processes: the *receiver* who is performing a two-phase spin-block while waiting for a communication operation to complete, and a *sender* who is sending a request to the receiver. We can determine the interval T_{Pair} in which the receiver must receive a message for spinning to be beneficial, by comparing the costs for the sender and the receiver when the receiver spins to the costs when the receiver sleeps.

We begin with the case where the receiver is spinning when the request arrives. The sender waits only time $2L + 4o$ for the response to be returned, while the receiver spins idly for time T_{Pair} to handle this one message. The total cost to the sender and receiver when the receiver spins is thus $2L + 4o + T_{Pair}$.

In the second case, the receiver blocks before the request arrives. The sender unsuccessfully spins for time $S_{RLocal} = 2L + 4o + 2W$ and then blocks; later, when the receiver is rescheduled and replies to this request, the sender is woken at a cost of W before continuing. Meanwhile, the receiver pays a cost of W when it is woken to handle the message and another W spinning. The total cost to the processes is $2L + 4o + 5W$.

Assuming that message arrivals are evenly spaced in time, a receiver should block rather than spin when the interval, T_{Pair}, between message arrivals is as follows:

$$\begin{array}{rrcr} & \text{Spinning Cost} & > & \text{Blocking Cost} \\ & 2L + 4o + T & > & (2L + 4o + 5W) \\ \Longrightarrow & T_{Pair} & > & 5W \end{array}$$

This analysis assumes that a receiver can predict whether a message will arrive in an interval T_{Pair} in order to determine if it should spin or block. In practice, future arrival rates can be predicted from behavior in the recent past, suggesting the following implementation.

When waiting for a remote operation, the process spins for the base and local amount, S_{Local}, while recording the number of incoming messages. If the average interval between requests is sufficiently small (*i.e.*, less than $T_{Pair} = 5W$), the process assumes that it will remain beneficial in the future to be scheduled and continues to spins for an additional time of $S_{Pair} = 5W$. The process continues conditionally spinning for intervals of $5W$ until no messages are received in an interval. Since incoming communication is expected to decline as processes reach the barrier, communication from only the most recent interval should be considered, and not averaged over a longer period. The next time this process performs a two-phase spin-block, the process begins anew by spinning only S_{Local} and reevaluating the benefits of spinning longer.

The simulations [11] did not find pairwise spinning necessary due to the simplicity of the bulk-synchronous benchmarks examined. Since no communication was performed in the phases that varied granularity or load-imbalance, few messages arrived as processes waited at barriers. As we verify in Section 5, spinning for the pairwise amount in bulk-synchronous applications does not improve performance beyond local spinning.

3.5 Summary

In this section, we have described three components to determining spin-time within the two-phase spin-block algorithm used in implicit coscheduling. The first component, spinning for the baseline amount, ensures that processes stay coordinated. The second component, an additional spin-time equal to the cost of waking, is derived from a competitive argument for the penalty seen by a single process. The final spin-time component accounts for communication between pairs of processes, and conditionally keeps processes scheduled when receiving messages. Each component, its relationship with various system parameters, and its value in our implementation, is summarized in Figure 2.

In the ideal case, we would also analyze the benefit of one process spinning for the job as a whole. However, a sleeping process can have a cascading effect on the scheduling of multiple processes only indirectly dependent upon this one. This analysis remains an open problem.

4 Implementation

In this section we describe our implementation of implicit coscheduling for multiple parallel jobs. After describing our basic environment, we explain why the parallel run-time layer is the appropriate layer for implementing the two-phase spin-block algorithm and, why *no* level in the system may perform unbounded spinning when waiting on a remotely produced result.

4.1 Background

The measurements in this paper use a cluster of 16 UltraSPARC I workstations running Solaris 2.6 connected with 8-port Myrinet switches [6]. GLUnix is used to start the processes across the nodes of the cluster [17].

For our parallel language, we use Split-C [9], a parallel extension to C with operations for accessing remote memory built on Active Messages. We use Split-C because many of its applications are communication intensive, and, therefore, sensitive to scheduling perturbations. In addition, it closely matches the model assumed in the original simulations. Simple `read` and `write` operations which access remote memory are built on a request-response model, requiring the requesting process to wait for the response. With *split-phase* assignments and `stores`, the initiating process does not wait for the response until a later synchronization statement, `sync`. Bulk transfer is also provided for each of these communication styles. `Barriers` perform synchronization across processes.

Previous implementations of Split-C assumed that parallel jobs are run either in a dedicated environment or under explicit coscheduling, and therefore spin while waiting for message responses to arrive. Consequently, if Split-C users run multiple applications simultaneously in our cluster with local scheduling, each user sees performance orders of magnitude worse than in a dedicated environment.

Our communication layer is AM-II [23], which extends the Active Message paradigm [28] to a general-purpose cluster environment. The Active Message model is essentially a simplified remote procedure call that can be implemented efficiently on a wide range of hardware. AM-II handles multiple communicating processes by virtualizing the network; it also supports client-server applications and system services.

4.2 Appropriate Semantic Layer

The parallel language run-time library is the appropriate layer to implement the two-phase spin-block algorithm for two reasons. First, the performance of the run-time library and the layers below it determine the time for communication operations to complete, which forms the baseline spin time for processes to stay coordinated. Second, the run-time layer contains semantic information about the relationship between requests and responses within higher-level primitives, such as reads and barrier. The run-time level knows when an incoming message satisfies the condition on which the process is waiting and can also calculate load-imbalance within barriers.

Modifying the Split-C run-time library required changing only a few operations that wait for replies (*i.e.*, `reads`, `writes`, `syncs`, and `barriers`). To implement two-phase spin-blocking, the Split-C library begins by polling the AM-II layer a fixed number of times as designated by S_{Local}, checking if the desired condition is satisfied (*e.g.*, the message response has arrived) and recording the number of incoming requests. If the condition is satisfied, the spin-block is complete and the process continues its computation. Otherwise, if a sufficient number of requests arrive, Split-C continues to spin for intervals of length S_{Pair}, until no messages are received in an interval. At this point, the process calls an AM-II function that puts the calling process to sleep until a message targeted for that process arrives. When the process is woken, the Split-C layer polls the network to handle the new message. If this message satisfies the condition for which the process is waiting, the routine returns; otherwise, the process spins again, once more recording message arrivals to enable conditional pairwise spinning.

4.3 Non-Blocking Layers

Successful implicit coscheduling requires that all levels of an application apply two-phase spin-blocking whenever waiting for a result generated from a remote process. If different layers implement different waiting policies, the performance of the entire application will suffer. Therefore, all the layers from the user-level application down to the low-level messaging code must act in unison.

User-level code that spins while waiting for a message to arrive is easily supported by providing a Split-C function that performs the necessary spin-block algorithm. Finding these spin statements in the application is simple because of the polling model used in existing Split-C implementations: tight loops containing polls are simply replaced with our interface.

More serious issues arise when the underlying message layer spin-waits. For example, spinning occurs in the current AM-II implementation when waiting for flow-control credits or queue space. Before we identified this problem, programs sending many small messages to random destinations without waiting for replies exhibited erratic performance, with some slowdowns five times worse than ideal coscheduling. Inspection of the AM-II implementation revealed that outgoing messages sometimes had to spin-wait for the next slot in a fixed-length queue to become available.

Ideally, the message layer should provide an interface that immediately returns control to the run-time layer rather than spin-wait. However, our temporary solution is to make the Split-C library *message-layer-aware*, by avoiding those cases that cause spinning in the message layer. Thus, Split-C tracks the number of outstanding messages, and, after sending the number that fit in the underlying queue (16), waits for all mes-

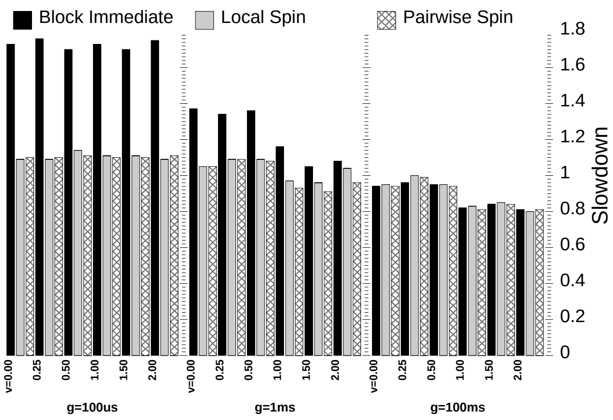


Figure 3: **Bulk-Synchronous NEWS.** *Three copies of the same synthetic benchmark are run on 16 workstations with immediate blocking, two-phase spin-block with the local spin time, and two-phase spin-block with conditional pairwise spin. Three synchronization granularities, g, and six load-imbalances, v, are shown. The slowdown reported is the average of 5 runs, and is relative to ideal explicit coscheduling.*

sages to be acknowledged, using the appropriate two-phase spin-block time, before sending more. Unfortunately, as we will see in the measurements in Section 5.3, this slows down a few applications by artificially limiting the number of outstanding messages.

5 Spin-Time Experimental Results

In this section, we evaluate the benefit of variations of the two-phase spin-block algorithm on both synthetic and real applications. To evaluate the algorithms in a controlled environment, we implemented a range of synthetic benchmarks. The first synthetic benchmarks are *bulk-synchronous* and match those analyzed in the original simulations [11]; however, because their communication traffic is bursty, these applications are relatively easy to schedule. To better approximate the worst-case behavior of applications and to stress the spin-block algorithm, we implemented a new set of *continuous* synthetic benchmarks.

Throughout these experiments, we always run exactly three competing jobs. Our performance metric is the slowdown of the last job to complete in our workload relative to an idealized, perfect model of explicit coscheduling (*i.e.*, three times the dedicated time).

5.1 Benefit of Two-Phase Spin-Block

To verify the initial simulation results, we begin with a synthetic benchmark, *bulk-synchronous* NEWS, that matches the model in the simulations precisely: for a computation granularity, g, and load-imbalance, v, each of the sixteen processes in the parallel job computes for a time uniformly distributed in the interval $(g - v/2, g + v/2)$, performs a barrier, reads a single word from each of its four nearest neighbors, and then performs another barrier. This loop is repeated such that the program executes for approximately 20 seconds in a dedicated environment.

The slowdown on this benchmark is shown in Figure 3 for three blocking algorithms: immediate blocking, two-phase spin-block with local spin, and two-phase spin-block with conditional pairwise spin. Only a few of the computation granularities and load-imbalances we have measured are shown.

On coarse-grain jobs ($g = 100\ ms$), all three blocking algorithms perform as well as ideal explicit coscheduling. Additionally, because processes with less work can relinquish the processor for use by another process, jobs with large load-imbalances exhibit a speedup relative to explicit coscheduling [15]. However, on fine-grain jobs ($g = 100\ \mu s$), blocking immediately performs approximately 80% worse than ideal coscheduling; while this is an order of magnitude improvement over strict spin-waiting, it is still not acceptable.[4]

Using two-phase spin-block with local spinning brings performance within 10% of ideal for all granularities. Incorporating conditional pairwise spinning does not provide additional benefit for this application, due to the simplicity of the communication structure: since all communication is contained between two barriers with no load-imbalance, there is little incoming communication to conditionally increase spin time.

In the simulations, an adaptive barrier that calculated the amount of load-imbalance, v, in the job and informed the processes to spin either S_{Local} or $S_{Local} + v$, was beneficial for a range of load-imbalances [11]. However, we found that in practice, the adaptive barrier had no benefit (not shown). We believe that this is in part due to the cost of calculating load-imbalance in the adaptive barrier, but primarily to a difference in potential gain: in the simulations, the difference between the minimum local spin time and the maximum spin time with load-imbalance was substantial (400 μs versus 2 ms) , while in the implementation, the difference is much smaller (443 μs versus 1 ms).

5.2 Benefit of Conditional Pairwise Spin

The deficiency of the previous benchmark is that its bursty communication behavior is not representative of more complex applications; since it is relatively easy to schedule, it does not illustrate the benefits of pairwise spin time. Therefore, we now analyze a set of *continuous* synthetic benchmarks, in which processes communicate every c time units in the interval $(g - v/2, g + v/2)$; thus messages continuously arrive, even as processes wait at barriers. This benchmark is especially difficult to schedule when the interval, g, between barriers is large because the globally coordinating effect of the barrier is lost after a time-slice expires (every 20-200 ms in the default Solaris 2.6 time-sharing class).

Figure 4 shows the performance of continuous all-to-all reads to random destinations for several values of g, v, and c, with local and pairwise spinning. With immediate blocking (not shown), this application exhibits a slowdown *four* times worse than ideal. In all cases, spinning for the local amount improves performance beyond immediate blocking: all slowdowns are now within two of ideal. When communication is frequent and the time between barriers is large, spinning for the pairwise amount improves performance further. When the interval between messages, c, is greater than the pairwise interval, $5W = 325\ \mu s$, the two spin-block algorithms are equivalent.

[4] Our measured performance with immediate blocking is much better than that seen in the simulations of [11], due to the smaller scheduling cost versus round-trip time ratio for our implementation (*i.e.*, $W : 2L + 4o$ is 65 μs : 55 μs in our implementation versus 200 μs : 20 μs in the simulations).

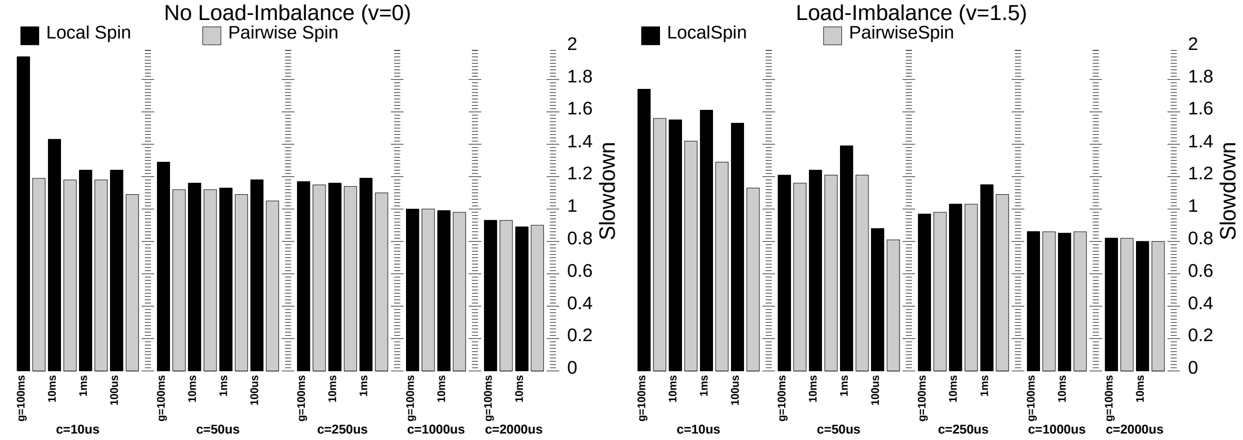


Figure 4: **Continuous All-to-All Random Reads.** *Three copies of the same application are run on 16 workstations with local and conditional pairwise two-phase spin-blocking. The time between barriers (g) and the time between reads (c) is varied for two load-imbalances. Pairwise spinning improves performance when the interval between messages is small ($c < T_{Pair} = 325\ \mu s$).*

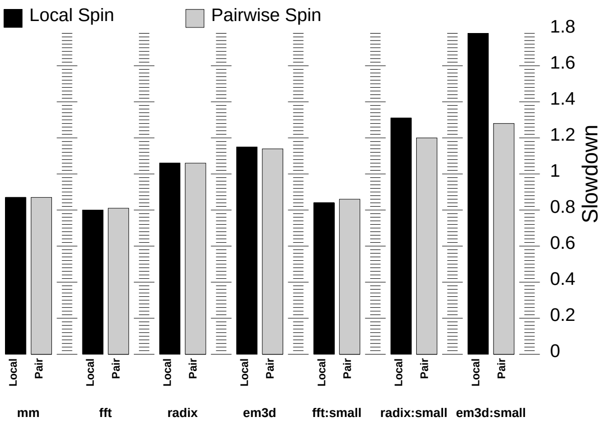


Figure 5: **Performance on Split-C Applications.** *Three copies of each Split-C application are run on 16 workstations. The first four Split-C applications use bulk messages, while the last three use small messages. We compare two-phase spin-block with local versus pairwise spin time.*

In addition to reading one word from random destinations, we have also measured reading bulk messages of 16 KB. These measurements (not shown) reveal a small *speedup* relative to coscheduling with both local and pairwise spinning, regardless of the other parameters in the application (g, v, c). This speedup occurs because the time required to send or receive a 16 KB packet (approximately 1 ms) is significantly larger than the time to wake on a message-arrival; therefore, processes should always sleep when waiting.

5.3 Experience with Applications

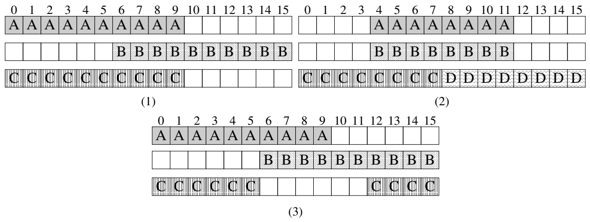
Analyzing synthetic applications exposed some of the strengths and weaknesses of our implementation in a controlled environment. We now evaluate seven Split-C programs: matrix multiplication, `mm`, two copies of radix sort, `radix` [1, 12], two fast Fourier transforms, `fft` [1, 10], and two versions of a model of electro-magnetic waves propagated in three dimensions, `em3d` [9]. When two versions of an application exist, one copy has been optimized to communicate with large messages, while the other uses short messages. These applications exhibit a variety of communication characteristics. For example, in the bulk version of `em3d` there is 60 μs of computation between most messages and 900 ms between barriers, while in `em3d:small` there exists only 10 μs between messages and nearly 3 seconds between barriers.

Figure 5 shows the performance of these applications with local and pairwise spinning. The two applications that perform the worst with implicit coscheduling, `radix:small` and `emd3d:small`, both rarely synchronize yet communicate frequently with random destinations. As expected with this type of communication pattern, pairwise spinning significantly improves performance relative to local spinning. For example, with local spinning, `em3d:small` runs nearly 80% slower than ideal; with pairwise spinning it improves to within 30% of ideal.

Both `radix:small` and `em3d:small` perform not only `read` operations, but also Split-C `stores` (*i.e.*, one-way requests without replies). Measurements of a synthetic benchmark performing continuous stores to random destinations has confirmed that stores are currently problematic, due to an artificial scheduling dependency created by the interaction between Split-C and AM-II. As described in Section 4, to avoid spinning in AM-II, the Split-C run-time layer sends messages in bursts of no more than sixteen before waiting for all to return. Thus, store operations which do not require a round-trip response when the application runs in a dedicated environment, must sometimes wait on an acknowledgment when implicitly coscheduled. As a result, running just a single copy of the applications with this constraint results in slowdowns of 10%. Providing an AM-II layer with a non-blocking interface would solve this performance problem.

6 Sensitivity to Scheduling Environment

Now that we have seen that implicit coscheduling succeeds for a variety of applications in a controlled setting, we investigate several more realistic environments. In this section, we present implicit coscheduling measurements for different layouts of



	Slowdown		
Layout	**Coarse**	**Medium**	**Fine**
1	0.70	0.92	1.01
2	0.76	0.92	1.03
3	0.57	0.78	1.04

Figure 6: **Sensitivity to Job Placement.** *The pictures represent the layout of three or four jobs (A, B, C, and D) across 16 workstations. If the jobs were explicitly coscheduled, each workload would require three coscheduling rows. The slowdowns in the table are calculated relative to explicit coscheduling for three bulk-synchronous workloads: coarse ($g = 1\ sec$, $v = 2$) medium ($g = 10\ ms$, $v = 1$) and fine-grain ($g = 100\ \mu s$, $v = 0$).*

jobs across machines, for an increasing number of jobs, and for competing jobs that communicate at different rates.

6.1 Job Placement

Our previous experiments considered only workloads where each workstation ran the same number jobs; hence, when load-imbalance existed, it was due only to load-imbalance within the applications. In this section, we show that implicit coscheduling performs well when the load across machines is unbalanced, and that performance may even improve relative to explicit coscheduling.

Figure 6 shows three of the parallel job layouts that we have examined as well as the slowdowns we have measured for a variety of bulk-synchronous NEWS applications. The chosen layouts are not necessarily ideal for job throughput, but, instead represent placements that may occur as jobs enter and leave the system (*e.g.*, the jobs in layout 2 would be more efficiently placed with job A on workstations 0 through 7 and job B on workstations 8 through 15, thus requiring only two coscheduling rows).

Layouts 1 and 2 were constructed to verify that implicit coscheduling can handle uneven loads across workstations. As desired for these two layouts, the applications perform as if they were running on machines evenly loaded with three jobs. Because the applications run at the rate of the most loaded workstation, workstations with less jobs are idle with implicit coscheduling for the same time as an explicit coscheduling implementation. Thus, medium and fine-grain jobs perform nearly identically to coscheduling, and the coarse-grain jobs with load-imbalance achieve a speedup, in agreement with the results in Figure 3,

Layout 3 was designed to show that implicit coscheduling automatically adjusts the execution of jobs to fill available time on each workstation. In this layout, three coscheduling rows are required for each of the parallel jobs to be scheduled simultaneously, yet the load on each machine is at most two. For jobs with coarse-grain communication dependencies, implicit coscheduling achieves a significant speedup relative to explicit coscheduling because processes can run at any time, and not only predefined slots of the scheduling matrix. Obtaining this benefit with *explicit* coscheduling is only possible if the appropriate alternate job is carefully chosen to run when a machine is idle.

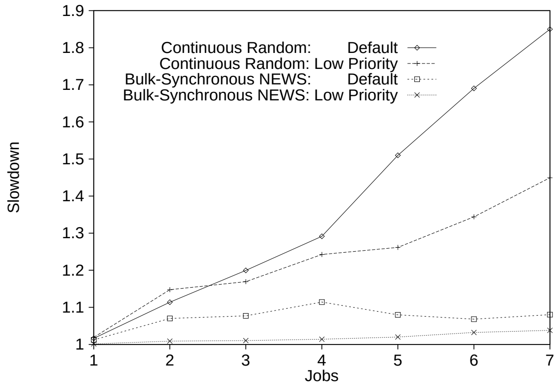


Figure 7: **Scalability with Jobs.** *The number of competing jobs is increased from one to seven, for two synthetic applications: bulk-synchronous NEWS ($g = 100\ \mu s$ and $v = 0$) and continuous random ($g = 100\ ms$, $v = 0$, and $c = 10\ \mu s$). Each is scheduled with both the default Solaris 2.6 time-sharing scheduler, where time-slices vary between 20 and 200 ms, and a version where the priority of the job is limited, such that the time-slice is always 200 ms.*

6.2 Job Scalability

To show the scalability of implicit coscheduling with more competing jobs we continue to evaluate synthetic applications, so as not to be limited by memory constraints. We examine both a relatively easy and a relatively difficult synthetic application to schedule: a fine-grain bulk-synchronous NEWS pattern and a continuous random pattern with frequent reads and infrequent barriers. Figure 7 shows the performance of implicit coscheduling with conditional pairwise spinning on both benchmarks as the number of competing jobs is increased from one to seven.

The *default* lines for the two benchmarks show the slowdown with the default Solaris 2.6 time-sharing scheduler. While bulk-synchronous NEWS scales well to seven jobs, with a relatively stable slowdown of 11% or less, the continuous random pattern exhibits an ever-increasing slowdown up to 85% with seven jobs. This poor performance can be attributed to the interaction of time-slice length with job priorities in the Solaris time-sharing scheduler.

Scheduling in Solaris is realized with a dynamic priority allocation scheme [18]. A job's priority is lowered after it consumes its time-slice and raised when it has not completed a quantum before a *starvation interval* expires. The new priorities, the length of the time-slices, and the starvation interval are specified in a user-tunable dispatch table. In the default dispatch table, the starvation interval is set to zero, with the result that the priority of every process is simultaneously raised once a second, regardless of whether or not the job is actually starving. Therefore, as the number of jobs is increased, each job spends more of its execution time at the high priorities. Since time-slices vary from 20 ms at the highest priority down to 200 ms at the lowest priorities, with more jobs in the system, each is given smaller time-slices.

The *low priority* lines for each benchmark show that the scalability of implicit coscheduling can be improved if parallel jobs are limited to priority 0, where the time quantum is 200 ms. With a 200 ms time-slice to amortize coscheduling skew, performance improves to within 45% of ideal for continuous random reads and 4% for bulk-synchronous NEWS.

The most interesting implication of this performance improvement is that implicit coscheduling does not need priorities to achieve coordination as originally proposed in the previous simulations [11].[5] As a result, the priorities of the underlying scheduler can be used to enforce fairness across competing jobs.

6.3 Fairness

The workloads we have measured so far in this paper, like those the previous simulations [11], have consisted of multiple copies of the same application. In each experiment, we found that the three jobs in the workload finished at roughly the same time, and, therefore, used the completion time of the last job as our only metric.

However, Figure 8 shows that with the default Solaris time-sharing scheduler, an infrequently communicating job is given more of the processor than a competing medium and fine-grain job. While the last job in the workload finishes in the expected time (*i.e.*, three times its dedicated time), the most coarse-grain job finishes significantly earlier. Thus, if another coarse-grain job entered the system after the first exited, the medium and fine grain jobs would not finish at the desired time. This scheduling bias occurs because fine-grain jobs are more sensitive to the scheduling on remote nodes, and frequently sleep when waiting for communication to complete. If a fine-grain job competes against similar jobs, it soon re-acquires the processor when the other jobs sleep; however, the competing coarse-grain job rarely relinquishes the processor.

While the Solaris time-sharing scheduler is not the ideal building block if fairness is the primary criteria, we discuss a preliminary change that improves fairness across jobs. As described in the previous section, with the starvation interval in the default dispatch table set to zero, the priority of each job in the system is raised once a second, *regardless of the past allocation of each job*. Therefore, the only mechanism that guides fairness is that the priority of a process drops after each executed time-slice; while the priority of a coarse-grain jobs drops at a faster rate, the priority boost once a second is the dominant effect.

Providing a fair allocation of the CPU requires boosting the priority of a process as determined by its past allocation; that is, since fine-grain jobs are sleeping more frequently, they should be given a higher priority when runnable. The interval after which a job is considered starving should balance two opposing forces: if the starvation interval is too short, the priorities of all jobs are kept too high; if the interval is too long, the priorities of jobs are rarely raised. In either case, there is no *priority differentiation* between jobs, and coarse-grain jobs dominate based upon spin-block behavior. In the ideal situation, the starve interval, S, slightly exceeds the time-slice, Q, multiplied by the number of jobs, J. Experiments for a range of S, J, and Q have verified this relationship works well for a variety of workloads.

[5] We believe that the performance irregularities seen in the simulations for fine-grain jobs and the round-robin scheduler occurred due to a defect in the two-phase spin-block algorithm: when a process woke from a message other than the desired reply, it slept again immediately; in our implementation, we found spinning for the cost of the wakeup, W, greatly improved stability.

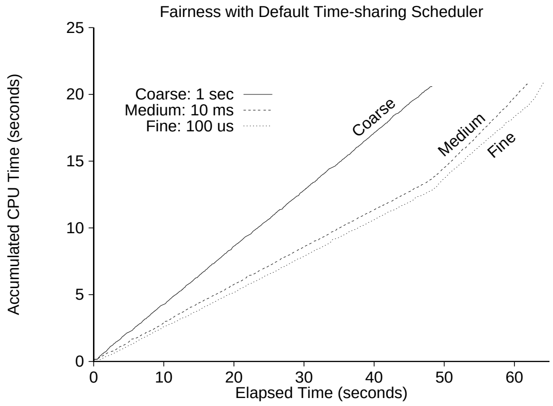


Figure 8: **Different Granularities with the Default Scheduler.** *Three competing synthetic parallel jobs with different communication characteristics (synchronization granularities of 1 sec, 10 ms, and 100 μs), are implicitly coscheduled on 16 workstations. The jobs were configured to require 20 seconds in a dedicated environment. While the workload finishes in the expected time (i.e., 60 seconds), the most coarse-grain job finishes significantly earlier than the other jobs.*

To achieve fairness for the entire parallel job, we leverage the derived third piece of implicit information available to each local scheduler. Since each process in a fine-grain communicating job can only make forward progress if the other processes are also making progress, the processes are likely to have their priorities raised at roughly the same time across workstations. If the program contains few synchronization dependencies, the local schedulers may not simultaneously raise the job's priority, but in this case, no global scheduling coordination is required for the job.

As each process of the parallel job has its priority raised and is scheduled, it runs as long as it communicates only with processes whose priorities have already been raised. Once the job must wait for a reply (most likely from a low-priority process), it sleeps; however, when a new request arrives, the process wakes and is scheduled immediately. Therefore, when the priority of the final process in this job is raised, the entire parallel job becomes implicitly coscheduled. Each process then travels down the priority levels at the same rate, until reaching priority zero or until another job has its priority raised. While a skew equal to the timer granularity may pass before the entire parallel job is scheduled, this time is not wasted by the system; since the high-priority job relinquishes the processor when unable to make progress, other runnable lower-priority processes are usefully scheduled.

To evaluate the level of control this mechanism has for improving fairness, we have implemented a simple prototype. Because the granularity of the starvation interval in Solaris 2.6 is in units of seconds, we use a rather long time quantum of $Q =$ 500 ms to be able to set $S > J \cdot Q = (3 \cdot 500\, ms) \approx 2\, sec$. In a implementation handling general-purpose workloads, the timer should be set to expire more frequently (*e.g.*, every 100 ms), so Q could be set lower for interactive workloads.

Figure 9 shows the completion time of each of three jobs in a workload containing a mix of granularities; each job performs bulk-synchronous NEWS and is configured to require 30 secs in a dedicated environment. As desired, regardless of the mix of jobs or the behavior of the local scheduler, the last job in each workload experiences little slowdown. How-

	Default Solaris Scheduler			Prototype Fair Scheduler		
	Finish Time (secs)			Finish Time (secs)		
Jobs	Job 1	Job 2	Job 3	Job 1	Job 2	Job 3
C-C-M	80.6	81.2	89.1	79.8	83.4	88.6
C-C-F	80.7	81.2	91.4	82.6	83.4	89.9
C-M-M	68.3	90.7	90.7	76.5	88.9	90.7
C-M-F	68.1	88.8	91.9	73.5	89.9	88.4
C-F-F	68.1	91.5	92.5	77.4	88.9	92.2
M-M-F	87.5	88.1	92.7	90.4	90.6	80.5
M-F-F	82.2	93.6	93.6	89.5	89.4	89.6

Figure 9: **Fairness across Competing Jobs.** *The completion time of each of three jobs is shown with the he default Solaris 2.6 time-sharing scheduler, and a prototype scheduler. The workload consists of three bulk-synchronous NEWS programs where the granularity is either 1 sec (coarse-grain, "C"), 10 ms (medium-grain, "M"), or 100 μs (fine-grain, "F"). Each job was configured to execute in approximately 30 secs in a dedicated environment. With the modified scheduler, the coarse-grain jobs finish later in the workload than with the default scheduler, as desired.*

ever, with the default scheduler, coarse-grain jobs can finish up to 24 seconds earlier than expected, which is a significant bias. The table shows that with our modified parameters, jobs that rarely communicate receive less of the CPU than with the default scheduler, now finishing no more than 17 seconds early. However, while our modification has improved the relative execution rates between coarse and fine-grain jobs, the medium-grain job now suffers the most.

Our preliminary results indicate that while priorities can modify the relative execution rates of competing jobs without hurting the throughput of the system, more work is needed to bias scheduling decisions fairly. Since time-sharing schedulers are designed to provide a compromise between interactive response time and fairness, it is our opinion, that a different local scheduler should be used as a building block. In future work, we will investigate leveraging a proportional-share scheduler for allocating resources to parallel jobs [4, 29].

7 Conclusions

We believe that leveraging *implicit information* simplifies the construction of highly-available, scalable services in a distributed system. Rather than performing communication to query remote nodes, extra intelligence is added to existing local services to infer remote state. This allows cooperating local services to retain their autonomy, while making local decisions that lead to a common global goal.

In this paper we have presented an implementation and empirical evaluation of *implicit coscheduling*, a method for dynamically coordinating the scheduling of communicating processes [11]. We have shown that global coordination can be achieved for multiple parallel jobs without a global component and without additional communication if each independent scheduler observes and reacts to naturally-occurring local information. The implicit information available for scheduling consists of the round-trip time of request-response messages, the arrival of incoming messages, and the ability of a process to make forward progress.

Communicating processes can infer whether or not they are currently coscheduled from the round-trip time of request-responses, and subsequently continue running when it is beneficial. By using *two-phase spin-blocking*, processes stay scheduled when responses arrive quickly, and otherwise voluntarily relinquish the processor. The spin-time should be chosen as a function of several system and communication-layer parameters, which can be automatically determined with a few microbenchmarks and simple computations. In this paper, we have shown that processes should increase their spin time when receiving messages.

The measurements of our system of 16 UltraSPARC I workstations connected with a high-speed network have shown multiple competing parallel jobs can be coscheduled implicitly with good performance. Our results hold for both synthetic programs that communicate either continuously or in bulk-synchronous style, as well as real applications with a mix of communication characteristics. We have also shown that jobs can be placed such that the load across workstations is not balanced. Furthermore, performance scales moderately well as the number of competing jobs increases if longer time-slices are used for parallel jobs. Finally, implicit coscheduling does not require the use of priorities in the underlying local scheduler; thus priorities can be used to bias fairness.

Implicit coscheduling is easier to implement than *explicit coscheduling* [24], while being naturally fault-tolerant to node failures and inherently scalable. Since implicit coscheduling does not require communicating processes to be identified statically, it also readily supports client-server applications. Implicit coscheduling is useful even when competing parallel jobs are strictly space-shared, if the jobs leverage distributed services that communicate, such as a high-performance distributed file system,

Numerous areas for future work remain within implicit coscheduling. For example, the performance of parallel jobs with interactive and I/O-bound jobs as well as with client-server workloads needs to be evaluated. Further, we would like to analyze more parallel programming models than the SPMD, global-address space model of Split-C; we have an initial implementation of MPI with two-phase spin-blocking and are currently in the process of evaluating its performance. Finally, allocating a fair-share of resources to competing parallel jobs is an unsolved problem; leveraging proportional-share schedulers as the local building block instead of priority-based time-sharing schedulers may resolve this problem.

Acknowledgments

We would like to thank Tom Anderson, Remzi Arpaci-Dusseau, Steve Lumetta, and Girija Narlikar for their many helpful comments and discussions on this work. This research has been supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C0014), the National Science Foundation (CDA 9401156), Sun Microsystems, and California MICRO.

References

[1] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures (SPAA'95)*, July 1995.

[2] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[3] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[4] A. Arpaci-Dusseau and D. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, June 1997.

[5] M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and Algorithms for Co-scheduling Compute-Intensive Tasks on a Network of Workstations. *Journal of Parallel and Distrbuted Computing*, 16:319–327, 1992.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet—A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[7] M. Buchanan and A. Chien. Coordinated Thread Scheduling for Workstation Clusters Under Windows NT. In *Proceedings of USENIX Windows NT Workshop*, Aug. 1997.

[8] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. Technical Report 385, University of Rochester, Computer Science Department, February 1991.

[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[10] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–273, May 1993.

[11] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1996.

[12] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.

[13] K. Efe and M. A. Schaar. Performance of Co-Scheduling on a Network of Workstations. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 525–531, 1993.

[14] D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.

[15] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[16] D. G. Feitelson and L. Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[17] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: A Global Layer Unix for a Network of Workstations. In *Software Practice and Experience*, 1989.

[18] B. Goodheart and J. Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[19] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–32, May 1991.

[20] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[21] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *1st Annual ACM Symposium on Discrete Algorithms*, pages 301–309, Jan. 1990.

[22] S. T. Leutenegger and X.-H. Sun. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Proceedings of Supercomputing '93*, pages 143–152, 1993.

[23] A. M. Mainwaring. Active Message Application Programming Interface and Communication Subsystem Organization. Master's thesis, University of California, Berkeley, 1995.

[24] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[25] P. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, January 1997.

[26] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the IPPS '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[27] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, April 1995.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[29] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11. USENIX Association, 1994.

[30] J. Zahorjan and E. D. Lazowska. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proceedings of the IFIP International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.