A Cooperative Approach to Two-Phase Waiting

Andrea C. Arpaci-Dusseau
dusseau@cs.stanford.edu
Stanford University

David E. Culler
culler@cs.berkeley.edu
University of California, Berkeley

Abstract

Competitive analysis is traditionally used to bound the worst-case performance of two-phase waiting, a common algorithm in multiprocessor environments. Despite the popularity of competitive algorithms, we observe that their assumptions are not valid in closed systems in which processes can influence their future waiting times. By evaluating *implicit coscheduling* on time-shared, communicating processes in a distributed system, we show that spin-time should instead be calculated with *cooperative* methods. Rather than reacting to a fixed stream of adversarial waiting times, our cooperative algorithm minimizes future waiting times by keeping processes that belong to the same parallel job coordinated across machines.

Another common assumption in two-phase waiting is that the penalty of blocking is a fixed amount, equal to the cost of a context-switch performed by the operating system scheduler. However, in implicit coscheduling, the penalty of blocking is dynamic, related to the number of arriving messages. To handle this complication, we leverage *conditional two-phase waiting*, a generalization in which spin-time may be altered, depending upon events that occur while the process is waiting. Through simulations, we show that our algorithm performs within 35% of ideal on a range of workloads, and significantly better than two-phase waiting with a fixed spin-time.

1 Introduction

A recurring problem in multiprocessor systems is determining the waiting algorithm that processes should employ when dependent on events created by other processes (*e.g.*, the release of a lock, the signaling of a semaphore, or the completion of a barrier). Due to its simplicity and elegance, *competitive analysis* is often used in these environments to bound the worst-case performance of the waiting algorithm. However, we have found that the assumptions underlying competitive analysis do not always hold in practice.

Traditionally, a waiting process has three options. First, the process can *spin-wait* until the event occurs. Second, a process can *block immediately*, relinquishing the processor (so that a competing process can be scheduled) until the event occurs. Third, the process can use a hybrid approach called *two-phase waiting* [27]: in the first phase, the process spin-waits some predetermined amount of time for the desired event; if the event does not occur, then the process blocks in the second phase.

To determine the optimal waiting algorithm, the system is typically modeled as follows. While the process spin-waits, it pays a cost in wasted processor cycles equal to the time that it spin-waits. If the process blocks, it pays the fixed cost of a context-switch to be scheduled again when the event completes. Under this model, the optimal behavior is for the process to spin-wait if the completion time of the operation is less than the time for a context-switch; otherwise, the process should block.

Since the waiting time for an event is not generally known until after the operation has completed, the optimal waiting algorithm cannot be determined. Instead, competitive analysis is used to bound the worst-case performance of a given operation. It is well known that a two-phase waiting algorithm that spins for an amount of time equal to the blocking penalty, B, performs within a factor of two of the optimal off-line algorithm [21]. The reasoning for this is simple. If the completion time, C, is less than B, then the cost of the on-line strategy and the optimal off-line strategy are equivalent: each idly spins for C, wasting processor resources. If the completion time is greater than B, then the on-line algorithm pays a cost of $2B$ while the off-line algorithm blocks immediately and pays only B. Therefore, the algorithm is *competitive* with a factor of two. We note that an important assumption is that neither algorithm can affect the completion times with their actions.

However, we observe that this model of the system is too simplistic for many environments in practice; as a result, spinning for the cost of a context-switch before blocking is not guaranteed to yield performance within a factor of two of the optimal algorithm. In this paper, we focus on one such important example: scheduling cooperating processes in a distributed, time-shared environment in which processes must wait whenever they communicate or synchronize with processes running on remote machines. In particular, we examine *implicit coscheduling*, a recent algorithm that dynamically coordinates sets of communicating processes with autonomous operating system schedulers and no explicit control messages [4, 12].

The implicit coscheduling environment does not match the assumptions of the system model in three ways. First, while the process spin-waits, it may not be wasting cycles; while waiting, the process may also handle messages sent by cooperating remote processes. Second, the penalty of blocking is not a fixed cost; the blocked process may incur a context-switch for every message that arrives from a remote process. Third, and most importantly, the operations for which a process must wait are not independent; processes can influence their future waiting times (and those of the cooperating processes) with their behavior at the current operation.

The implication of the first two differences is that, to apply competitive analysis, the cost of spinning versus blocking must be calculated as a function of the incoming message rate. Since the message rate may change while the process is waiting, the relative costs of spinning versus blocking may also change. As a result, the optimal waiting algorithm may be neither spin-waiting to completion nor blocking immediately, but instead two-phase waiting: the process should spin while receiving messages and then block when the message rate falls below a threshold. In order to adapt to these dynamic changes in cost, implicit coscheduling leverages a novel waiting algorithm: *conditional two-phase waiting*, a generalization of two-phase waiting in which processes adapt spin-time depending upon events that occur while the process is waiting.

The implication of the last difference is that competitive analysis is not solving the desired problem in its complete context. Competitive analysis is usually applied to *open systems*, in which the inputs to the on-line algorithm are not affected by the algorithm itself; for example, task lengths and arrivals in scheduling and load-balancing [6, 7, 26], memory requests in page replacement algorithms [28], or arriving packets for bandwidth allocation [5]. On the other hand, cooperating processes and the completion time of their communication operations in implicit coscheduling is a *closed system*; feedback exists between the waiting algorithm and the observed completion times. In this case, it is incorrect to view completion times as adversarial.

If different algorithms create and react to different waiting times, one cannot compare the performance of the algorithms and comment on their competitive ratios. This suggests that entities in a system with feedback should use a *cooperative* approach, and thereby minimize the completion time of the operations for which they are waiting, rather than simply react the best they can to large completion times. It is well known that when processes are scheduled at the same time, communication completes much faster than when they are uncoordinated [14, 15]. Therefore, processes should choose waiting algorithms that keep their scheduling coordinated.

In this paper, we develop a cooperative approach for two-phase waiting in the context of implicit coscheduling. In Section 2, we describe the application of competitive analysis to traditional two-phase waiting problems in more detail. In Section 3, we summarize our cooperative heuristics for determining spin-time with conditional two-phase waiting. We describe the simulator for evaluating implicit coscheduling in Section 4 and present our performance results in Section 5. We discuss our conclusions in Section 6.

2 Motivation

Determining how a process should wait for an event to complete is an example of an on-line algorithm. Competitive analysis is an attractive technique for bounding the worst-case performance because it requires no knowledge of the input distribution. Researchers have examined competitive algorithms for waiting algorithms in a number of contexts: processes competing for shared locks in a multiprocessor [17, 20, 25], processes synchronizing with barriers in a space-shared environment [22], and processes communicating in a distributed time-shared setting [15].

In each of these cases, competitive analysis is not appropriate because the input stream of waiting times is part of the system and not adversarial: the waiting process often can impact future waiting times in some manner. Since two different algorithms may produce different streams of input values, the performance of the two algorithms cannot be compared competitively. Specifically, a waiting algorithm that spin-waits for the penalty of blocking before relinquishing the processor is not guaranteed to perform within a factor of two of the optimal algorithm.[1] However, because experimental results have not clearly contradicted the theory, one may be misled into thinking that the adversarial assumptions hold. In this section, we examine two closed feedback systems where competitive algorithms have been applied: processes waiting for locks to be released and processes waiting for communication to complete in a time-shared environment.

2.1 Shared Locks

A common application of two-phase waiting is when processes compete for shared locks in a multiprocessor; however, even this mundane scenario violates the adversarial assumptions of competitive analysis. Since different behavior by the waiting process can cause processes to be scheduled in different orders, the relative timing of critical sections and thus waiting time for future locks may be altered.

Measurements performed by Karlin et. al. [20] verify that different waiting algorithms produce different lock-waiting times; this has two implications. First, their analytic results on a fixed distribution of waiting times do not match their experimental results in which the waiting algorithm can alter the future inputs.

[1] In fact, in systems with interdependent events, determining the optimal algorithm or enumerating all actions is more difficult.

For example, in their work, given a fixed input distribution, the `Hanoi` application performs better with a spin-time of half the context-switch cost than with a spin-time equal to the context-switch cost; in the real environment with feedback, the performance is reversed. Second, spinning for the context-switch cost does not always perform within the factor of two guaranteed by competitive analysis. Our interpretation of their experimental data indicates that with the optimal algorithm, only 10 seconds are spent in synchronization, but with a spin-time equal to the context-switch cost, 52.7 seconds are spent.

One of their hypotheses for these discrepancies matches ours: the waiting strategy changes the relative timing of events across threads and subsequently the lock-waiting times. We note that, despite these violations, it is easy to ignore the feedback in this system, since it is unlikely that the waiting process can systematically bias or even predict the impact of its behavior on future waiting times.

2.2 Coordinated Time-Sharing

In this paper, we focus on one of the important problems in Networks of Workstations, or clusters [1, 8, 18, 30]: the scheduling of general-purpose workloads containing sets of communicating processes. Given this workload, time-sharing is an attractive strategy because it provides good response time without migration or changes in programming models. However, in a time-shared environment, communicating processes must wait whenever they require responses from processes running on remote machines.

Given the current costs of communication and context-switching, most fine-grain parallel jobs exhibit better performance when their scheduling is coordinated across workstations [14, 15].[2] When scheduling is coordinated, the completion times for communication and synchronization tend to be small, tracking the inherent performance of the underlying network hardware and message-layer software. Therefore, the best scheduling performance is obtained not when processes choose the optimal waiting algorithm for each communication operation, but when processes choose the waiting algorithm that keeps their scheduling coordinated.

The traditional approach for ensuring that scheduling is coordinated is *explicit coscheduling*, in which a global schedule of processes is constructed for the cluster and context-switches are performed simultaneously across all processors by a master [27]. In this environment, processes spin-wait during communication and thus remain scheduled for their entire time-slice. However, explicit coscheduling is not an appropriate solution for clusters because it cannot handle workloads containing client-server, I/O-bound, or interactive processes [2, 9, 15, 23] and many implementations are neither scalable nor reliable [13, 19].

With the alternative, *local scheduling*, the operating system scheduler running on each workstation independently schedules the processes that have been allocated to it. The problem with local scheduling is that sets of communicating processes are not coordinated across machines. When a message arrives at a workstation, the message cannot be handled until the destination process is scheduled. Since the sending process may have to wait a long time for the destination to be scheduled, local scheduling can perform orders of magnitude worse than explicit coscheduling [2, 10, 14, 24, 31, 32].

In the presence of long completion times, the optimal behavior of local scheduling is to block-immediately. Simulations have shown that two-phase waiting with a spin-time equal to the local context-switch cost performs two times worse than blocking immediately, since very few of the communication operations complete while the waiting process is spinning [15]. Thus, with two-phase waiting, for each operation, the sending process first spins for a time equal to the context-switch and then relinquishes the processor, paying another context-switch.

Since the performance of two-phase waiting is within a factor of two of the "optimal" behavior of blocking-immediately, one could mistakenly conclude that the assumptions of competitive analysis hold for this closed system. One could easily ignore the fact that blocking-immediately is not the optimal algorithm when processes can influence their waiting times. However, the fact that blocking-immediately performs

[2]Parallel jobs with little communication and a large amount of load-imbalance across processes may perform better with uncoordinated scheduling. For simplicity, the analysis in this paper assumes that coordinated scheduling is always beneficial.

nearly five times worse than coordinated scheduling with spin-waiting indicates the potential for a better waiting algorithm: one that minimizes future completion times.

Implicit coscheduling, like local scheduling, relies only on the autonomous operating system schedulers running on each workstation and requires no global control information. Unlike local scheduling, it achieves performance near that of explicit coscheduling by dynamically coordinating sets of cooperating processes. The key to implicit coscheduling is that each process adapts its waiting algorithm to *cooperate* with the current state of the system. In this system, unlike processes waiting to acquire locks, processes can systematically bias future completion times with their behavior. Through cooperation, communicating processes can converge on a globally coordinated schedule across the distributed system. In this coordinated schedule, sets of cooperating processes are scheduled one after another for a duration correlated to the time-slices of the underlying operating system scheduler.

3 A Cooperative Approach

In this section, we describe our cooperative approach for influencing the state of the system. In the closed system of implicit coscheduling, a waiting process can positively influence its scheduling coordination and future communication and synchronization completion times by whether it spin-waits or blocks at the current operation. Processes can detect when the system is coordinated and can encourage the system to remain in this beneficial state; processes can also detect when the system is not coordinated and limit the cost of this undesirable condition.

Our approach relies on two insights. First, the waiting algorithm can determine when scheduling is coordinated by observing the completion time of communication operations and the arrival rate of incoming messages: if an operation takes longer than expected, and no messages are arriving, the process can infer that scheduling is not coordinated. Second, the penalty for blocking depends on the dynamic frequency of arriving messages: if the waiting process blocks, not only may it then incur a context-switch for every arriving message, but the sending processes must also wait longer for the response.

In the remainder of this section, we present cost/benefit heuristics for spin-time in implicit coscheduling. We begin by summarizing our model of the system and by describing conditional two-phase waiting. We then analyze the two components of our waiting algorithm: the baseline and conditional spin-times. Note that this work corrects errors that appear in previous analysis of implicit coscheduling in [4] and [12].

3.1 System Model

In our system, processes communicate over a substrate that adheres to the LogP model [11]. LogP captures the relevant aspects of communication performance: latency (L), overhead (o), and gap (g). We assume that messages are composed of *requests* and *responses*, such as those in Active Messages [29]; when a message arrives at a destination processor, the appropriate process must be scheduled to handle the message (and, in the case of an arriving request, to return a response). Higher-level communication and synchronization operations, such as `reads` and `barriers`, are then built on top of the Active Message layer. We model the time to wake and schedule a process blocked on a message arrival as W; this is essentially the cost of a context-switch in the operating system scheduler.

3.2 Conditional Two-Phase Waiting

Conditional two-phase waiting, a generalization of two-phase waiting, fulfills the requirements of implicit coscheduling for adjusting spin-time to variations in the cost of blocking. Unlike traditional two-phase waiting where the spin-time, S, is determined before the process begins waiting, with conditional waiting the process may increase its spin-time based on events that occur while the process is waiting. In our implementation, there are two components of spin-time: *baseline* and *conditional*. The baseline mount,

S_{Base}, is the minimum time the process spins and is determined before the operation begins. The conditional amount S_{Cond}, is the additional spin-time based on spontaneous events.

These two components correspond naturally to the two requirements of implicit coscheduling. First, the baseline amount allows processes to spin-wait long enough to maintain scheduling coordination; this amount is set to the expected time of the operation when all cooperating processes are scheduled. Second, the conditional amount accounts for the additional cost of blocking when receiving more messages; thus, the waiting process continues spinning when the arrival rate of messages justifies the cost.

Note that conditional two-phase waiting differs from *adaptive two-phase waiting* [21], in which the spin-time varies from operation to operation, but is predetermined before the current operation begins. With conditional waiting, the process gathers information while it is spinning that helps it to evaluate the state of the system and to more accurately choose how long to spin for the current operation. Thus, given an oracle of this dynamic information, one could replace conditional waiting with adaptive waiting.

We now model the expected completion time of request-response operations (*i.e.*, reads) and analyze the arrival rate that justifies keeping processes scheduled. Due to space constraints, we do not analyze other communication or synchronization operations; the model for barrier operations follows naturally and can be found in [3].

3.3 Baseline Spin

There are two circumstances in which cooperating processes are coordinated; to ensure that the system remains in this beneficial state, a waiting process must spin-wait for the greater of the two times. In the first case, the sending and receiving processes are coordinated before the request message arrives. In the second case, the request message triggers the scheduling of the remote process. Figure 1 illustrates with LogP the time for a read operation under these two circumstances.

To begin, we examine T_{Sched}, the amount of time between the initiating of a request and the reception of the response, given that the receiver is scheduled throughout the operation. The local sending process first spends time o in overhead injecting the request into the network. The request then travels through the network for time L. Assuming that there is no contention with other messages, the request is handled by the receiver after another o time units. The receiver spends o returning the response, which arrives L units later at the initial sending node; the sender handles the response in time o. Thus, T_{Sched} is the minimum time for a short request-response message to complete.

$$T_{Sched} = 2L + 4o$$

However, it is often the case that the remote process is not scheduled before the request arrives, but that the request message triggers it to be scheduled. For example, if the remote process is blocked waiting on a response from another process, then this request will wake the remote process and place it on the ready queue; given a preemptive local scheduler, the remote process will then be scheduled if it is fair to do so (or, if the processor is idle). Therefore, if the request triggers the scheduling of the remote process, then the expected round-trip time, $T_{Trigger}$, is longer than if the process were already running by the amount of a context-switch, W, on the remote node.

$$T_{Trigger} = 2L + 4o + W$$

S_{Base} is the time that the local process should spin-wait in the two-phase waiting algorithm to ensure that it remains scheduled if the remote process is also scheduled. To be precise, the spin-time does not include the overhead, $2o$, on the local processor to send the request and to handle the response.

$$\begin{aligned} S_{Base} &= \max(T_{Sched}, T_{Trigger}) - 2o \\ &= 2L + 2o + W \end{aligned}$$

3.4 Conditional Spin

One of the difficulties of implicit coscheduling is in determining the penalty of blocking: it is not simply the cost of the context-switch paid by the local process to be rescheduled after its event completes. Not only must the waiting process pay a context-switch for every incoming message, but cooperating processes incur a higher penalty as well: every process that communicates with a blocked process waits longer for the response. The conditional spin-time in our waiting algorithm accounts for this variable cost of blocking. After the process has waited the baseline amount, S_{Base}, and the operation has not completed, it may still be beneficial to remain scheduled if handling requests from other processes. In this section, we examine the interval, T_{Cond}, in which a waiting process must receive a message for the benefits of staying scheduled to outweigh the costs.

In our analysis, we consider a pair of processes. First, there is the local process that is waiting for a response from an uncoordinated process; this process is performing two-phase waiting while its communication operation completes, but has exceeded the baseline spin amount. Second, there is a remote process that is sending a request to the local process. To evaluate T_{Cond} we compare the cost to both processes when the local process spin-waits for its own operation versus when the local process blocks, as shown in Figure 2.

We begin with the case where the local process spin-waits and remains scheduled. After the process begins spinning, a request arrives t units later; thus, the local process spins idly for time t to handle this one message. The remote process spends o injecting the request, waits $2L + 2o$ for the response to be returned, and spends another o handling the response. The combined cost to the pair of processes is thus $2L + 4o + t$.

In the second case, the local process blocks before the request arrives. The remote process spends o sending, and then unsuccessfully spins $S_{Base} = 2L + 2o + W$ before blocking. Later, the local process pays a cost of W to be woken and scheduled on the message arrival, o to handle the request, and o to send the response.[3] Finally, the original sender pays another W to be scheduled and o to handle the reply. Thus, the total cost to the system is $2L + 6o + 3W$.

Assuming that message arrivals are evenly spaced in time, a local process should continue spinning rather than block as long as the interval between messages is less than or equal to T_{Cond}. [4]

$$\begin{array}{rrcl} & \text{Cost of Spinning} & = & \text{Cost of Blocking} \\ & 2L + 4o + t & = & 2L + 6o + 3W \\ & t & = & 3W + 2o \\ \Longrightarrow & T_{Cond} & = & 3W + 2o \end{array}$$

The implication of this analysis is that the optimal waiting algorithm for a process may be to first spin, while receiving messages, and then to block before the operation completes. Under the traditional assumptions, this would never be true: either spin-waiting until the operation completes or blocking immediately is optimal and two-phase waiting is simply a way of bounding the penalty of the wrong decision.

The preceding discussion assumes that a local process can predict whether a message will arrive in an interval T_{Cond}. Because this is not possible in practice, future arrival rates must be predicted from behavior in the recent past. Since message arrivals are expected to be bursty (or steadily decline as more processes reach barriers and complete their communication phase), the average rate over a relatively short interval should be used. In our algorithm, if a message has arrived in the last T_{Cond} interval, the process spins for another interval $S_{Cond} = T_{Cond}$, continuing until no messages are received in an interval. At the next

[3] We ignore the fact that the cost W may be amortized over several messages that are handled in the same scheduling interval.

[4] This comparison assumes that paying a cost on one workstation is equivalent to paying that cost on another workstation. For this to be true, neither process must form the critical path for the system. For example, if the sending process on a remote machine determines the performance of the job as a whole, then the local receiving process should always spin in order to minimize the waiting time on the remote sender. However, if the local process is the critical component, then only its costs should be examined when determining whether to spin or block. Due to the difficulty of determining which process is along the critical path and because we believe this inaccuracy is small for our workloads, we ignore this adjustment in our analysis.

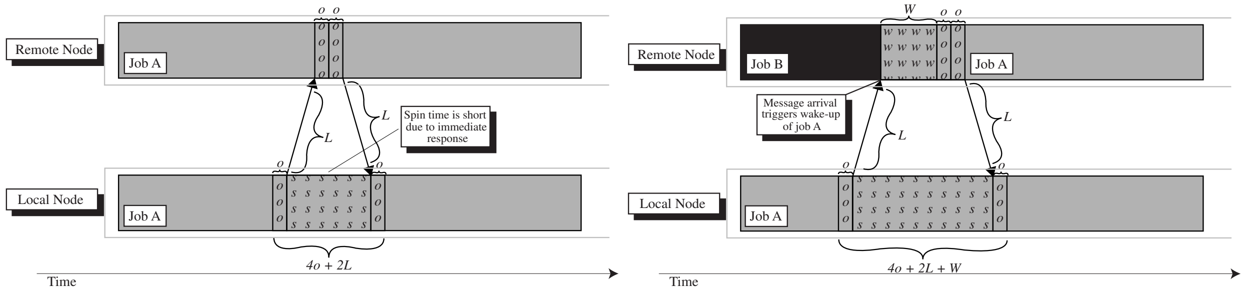

Figure 1: **Time for Request-Response Message.** *The diagram on the left illustrates T_{Sched}, the amount of time a local process must wait for a reply if the remote destination process is scheduled when the request message arrives. If the remote destination process is already scheduled, then it immediately handles the request and promptly returns the reply to the waiting process. The diagram on the right illustrates $T_{Trigger}$, the amount of time a local process must wait for a reply if the request message triggers the scheduling of the remote destination process. In this case, the local process must spin-wait the additional time, W, while waiting for the remote process to be scheduled by its local operating system scheduler.*

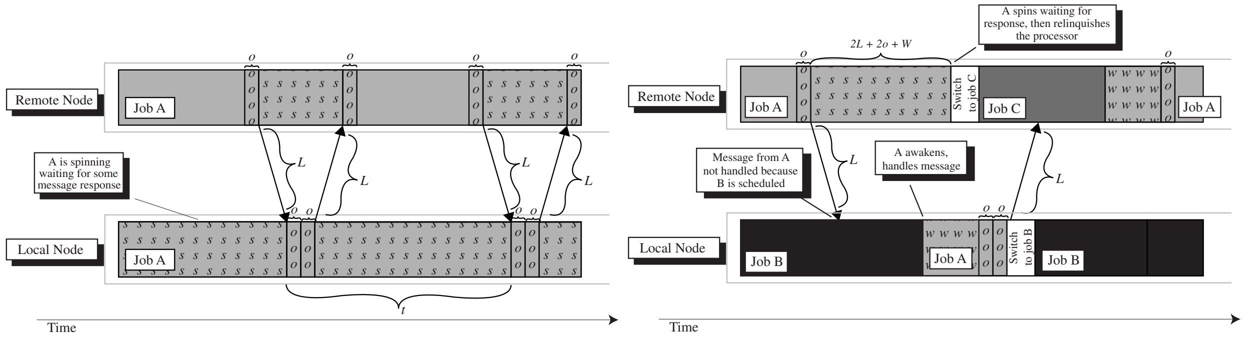

Figure 2: **Comparison of Costs for Incoming Request-Response Messages.** *To help calculate T_{Cond}, this figure compares the cost to the local (receiving) process and the remote (sending) process when the local process spin-waits or sleeps between message arrivals. The diagram on the left shows the case where the local process spin-waits; the diagram on the right shows the case where the local process blocks. We assume a constant time t between message arrivals.*

two-phase wait, the process reevaluates the benefits of conditionally spinning based on the current arrival rate.

4 Evaluation Methodology

To evaluate the performance of the conditional two-phase waiting algorithm, we extend the simulator, `SIMplicity`, originally used in [12]. `SIMplicity` can vary a range of parameters in the application workload, the operating system scheduler, and the communication layer. We discuss each of these components in turn.

At the highest level, `SIMplicity` is driven by the arrival of a parallel job at a specified time and requiring a certain number of processes. In these experiments, we restrict our measurements to workloads containing three identical jobs, each of 32 processes, that arrive simultaneously on a cluster of 32 workstations. We simulate two styles of synthetic parallel jobs: bulk-synchronous and continuously-communicating. Each style is described in the next section.

We constructed the local scheduling component of `SIMplicity` to closely match that of the Solaris Time-Sharing [16] scheduler in both functionality and structure. An important parameter of the operating system scheduler is the time required to context-switch to a new process. We assume that waking a process on a message arrival, W, is equal to the basic context-switch cost, incurring no additional cost at other layers of the system. Since the value of W has a large impact on scheduling performance, we evaluate two values: a moderate cost of $50\mu s$ that roughly matches that of current operating systems and a higher cost of $200\mu s$ to stress the performance of the two-phase waiting algorithm.

The most important parameter that can be varied in the communication layer is network latency (L); for the experiments in this paper, L is set to 10 μs, matching current network performance. We assume that a message arrives in exactly L time units (even when a process sends a message to itself) and no congestion exists in the network or at endpoints. We do not model processing overhead (o) on either the sender or the receiver, due to the number of additional events that would be generated for every simulated message; given the small values of o in current systems, we expect this simplification to have negligible impact.

Within `SIMplicity`, a process is notified of a message arrival through an interrupt. If the process is already running when the message arrives, then the message is handled immediately with no cost to the system. If the process is on the ready queue, then the message is buffered until the process is scheduled. If the process is sleeping, it is woken and placed on the ready queue; whether the process is scheduled depends upon the characteristics of the local operating system scheduler.[5]

To evaluate implicit coscheduling, we examine the time to complete a workload with a fixed number of jobs (other measurements have verified that jobs are scheduled fairly within each workload). Given our model of parallel jobs and the relative values of network latency and context-switch cost, explicit coscheduling with a long time-slice ($500ms$) relative to the context-switch cost is nearly optimal. Throughout this paper, our performance metric is *slowdown*, the ratio of the workload completion time with implicit coscheduling and two-phase waiting to ideal explicit coscheduling with spin-waiting.

5 Performance Results

In this section we demonstrate that our cooperative approach leads to performance that is within 35% of ideal for a range of parallel jobs. We start by verifying that our baseline spin-times begin to maintain scheduling coordination among cooperating processes. We then show that baseline spinning is not sufficient

[5] Note that this behavior differs in two primary areas from that originally presented in [12]. First, in the previous simulations, a sleeping process was given kernel priority (above 59) when a message arrived; thus, this process was almost always promptly scheduled. The current version better matches the behavior of the Solaris operating system. Second, in the previous simulations the process immediately slept again if the arriving message was not the event the process was waiting for. In the current version, the process spin-waits for an interval of W, which was found to improve the robustness of the algorithm.

for applications with only pairwise interactions across processes; in fact, no fixed spin-time achieves performance within a factor of five of optimal. Our final experiments illustrate that conditional spinning significantly improves performance; with the additional information carried by arriving messages, processes can better maintain coordination and estimate the penalty of blocking.

5.1 Baseline Spinning

In our first experiments, we verify that spin-waiting for S_{Base}, the required time for operations when scheduling is coordinated, achieves better performance than any other fixed spin-time. In fact, performance improves at two distinct points: when the baseline spin matches the time when the destination process is scheduled before the request arrives and when it matches the time when the request triggers the destination process.

To demonstrate this, we evaluate workloads containing synthetic *bulk-synchronous* jobs. The bulk-synchronous style, which alternates between phases of computation and communication, is common in many parallel applications. Because communication is bursty and involves all of the processes in the job, all processes must be coordinated for any one to make forward progress in a communication phase. This strong dependency causes all processes of the job to be scheduled as a group whenever one is scheduled.

In these experiments, each process computes locally for C time units and then performs a barrier; after the barrier, each process reads from its four nearest-neighbors with $8\mu s$ of intervening computation; each then performs another barrier. These computation and communication steps are repeated until the job executes for a total of 20 seconds in a dedicated environment. Since jobs that communicate frequently are more sensitive to how they are scheduled, we concentrate on such workloads, examining applications where a barrier is performed every $C = 100\mu s$ or $C = 1ms$. In these workloads, there is no load-imbalance across processes.

The simulation results, shown in Figure 3, vary the baseline spin on systems with two different context-switch costs ($W = 50\mu s$ and $W = 200\mu s$); no conditional spinning is performed. As predicted, performance improves at two distinct points: when $S_{Base} = T_{Sched}$ and when $S_{Base} = T_{Trigger}$. In each case, the performance improvement corresponds to a jump in the percentage of operations that complete *successfully*, that is, while the waiting process is still spinning.

The data shows that if processes block-immediately or do not spin at least $T_{Sched} = 2L + 4o = 20\mu s$, then processes never remain coordinated and pay a context-switch for every communication operation. When processes wait at least $20\mu s$, the initiating process remains coordinated if the remote process was scheduled before the request message arrived; in this region, 97% of reads complete successfully, requiring no context-switch. Thus, maintaining coordination greatly improves performance; for example, given an application synchronizing every $C = 100\mu s$ on a machine with a context-switch cost of $W = 200\mu s$, performance improves from 12 times worse than explicit coscheduling to only 60% worse.

Spin-waiting for $T_{Trigger}$ keeps processes coordinated when the arrival of the request triggers the destination process, and thus further improves performance. If processes spin this additional amount, then 99.9% of communication operations complete successfully. Due to this increase in successful operations, performance improves to within 10% of ideal explicit coscheduling.

Spinning for longer than $T_{Trigger}$ does not increase the read success rate and, therefore, is not beneficial. In general, spinning longer hurts the overall execution time of the program, since it wastes more processing time. However, in this case, longer spin times do not noticeably degrade performance because the spin penalty is paid for very few (0.1%) of the read operations.

5.2 Motivation for Conditional Spinning

Applications in which communication and synchronization occurs only between *pairs* of processes are more difficult to coordinate. We now show that baseline spinning is not sufficient for continuously-communicating

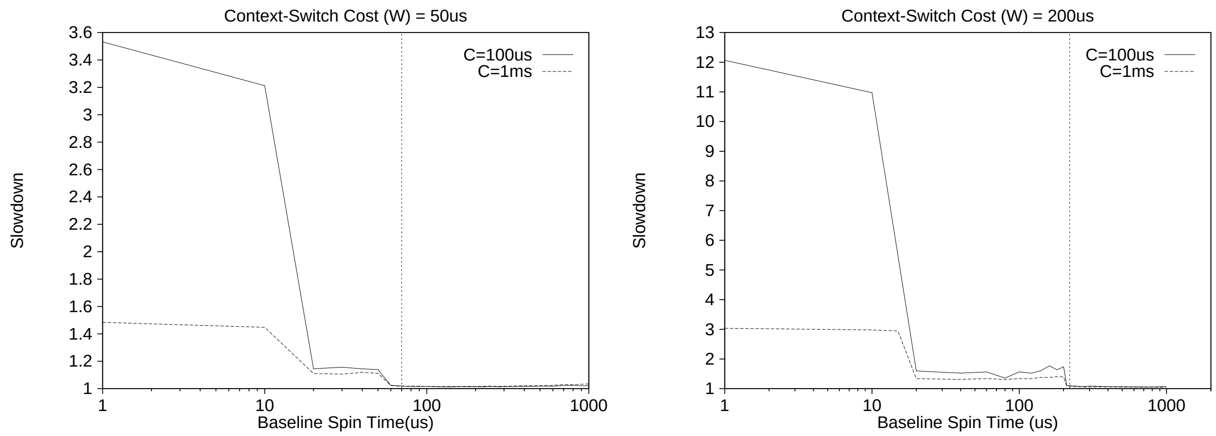

Figure 3: **Sensitivity to Read Baseline Spin for Bulk-Synchronous Programs.** *In each graph, the time between barriers, C, is set to either 100μs or 1ms. The vertical line in each graph designates the derived read baseline spin, $S_{Base} = 2L + W$. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the baseline spin for reads is varied between 1μs and 1000μs. In the first graph, the context-switch cost is set to $W = 50\mu s$; in the second, $W = 200\mu s$. Note the change in scale along the y-axis between the two graphs.*

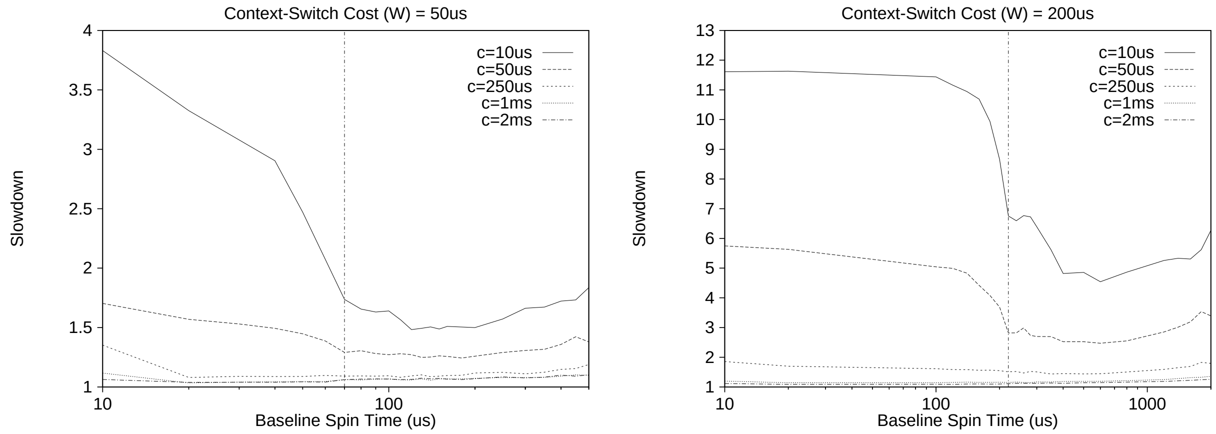

Figure 4: **Sensitivity to Baseline Spin for Continuous-Communication Programs.** *The time between reads, c, is varied in each graph. The vertical line in each graph designates the derived read baseline spin, $S_{Base} = 2L + W$. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the baseline spin for reads is varied. In the first graph, the context-switch cost is set to $W = 50\mu s$; in the second, $W = 200\mu s$. Note the change in scale along the y-axis between the two graphs.*

applications with few barrier operations; for some programs, all fixed spin-times perform at least 4.5 times worse than explicit coscheduling.

In the *continuously-communicating* benchmarks, there are no distinct phases of computation and communication. Since processes are always communicating, these benchmarks are more sensitive to their scheduling condition. As in the bulk-synchronous applications, each process performs a barrier every C time units; however, in the interval between barriers, each process is also communicating with a random process every c time units. Since jobs with infrequent barriers are more difficult to coordinate, we only examine workloads where $C = 100\,ms$. The time between communication events is varied between $c = 10\mu s$ and $c = 2ms$.

Figure 4 shows that, unlike the bulk-synchronous workloads, the performance of continuously communicating workloads does not significantly improve until processes spin at least $T_{Trigger}$=$2L + 4o + W$. The sudden improvement at this later point implies that processes are usually not coordinated until a message arrival triggers the destination process. Without periodic barriers in the application, only a subset of the cooperating processes may be coordinated at a given time and their coordination is more fragile. Since processes must build coordination with only pairwise message interactions, processes must stay scheduled given evidence that they are coordinated with any other process, even if unable to make forward progress themselves. The graphs also show that the optimal baseline spin amount increases with the frequency of communication in the application, indicating that the penalty for blocking is higher when more messages arrive.

5.3 Conditional Spinning

The previous experiments indicated that applications need to remain scheduled when receiving messages, both to maintain fragile coordination and to account for the higher penalty of blocking. In our final experiments, we show that conditional spinning improves the performance of continuously-communicating applications significantly beyond that of any fixed spin-time; with our derived value for conditional spin-time, performance is within 35% of ideal explicit coscheduling. Since the performance of bulk-synchronous workloads is nearly ideal with only baseline spinning, performance is not noticeably affected by conditional spinning and, therefore, is not shown.

Figure 5 shows the slowdown of the continuous-communication workload as S_{Cond} is increased along the x-axis. In these experiments, processes always wait at least $S_{Base} = 2L + 2o + W$ at reads and barriers, and continue spinning as long as the interval between incoming messages is less than S_{Cond}. To explain the behavior of conditional spinning, we discuss the three performance regions exhibited as the conditional spin amount is increased.

First, when the conditional spin time is very small, messages do not arrive at a fast enough rate to increase spin time. Therefore, processes usually spin only the baseline amount. Subsequently, performance when the conditional spin amount is $10\mu s$ matches the performance in Figure 4 when the baseline spin is $S_{Base} = 2L + 2o + W$.

As the conditional spin time increases, messages can arrive less frequently and the waiting process will continue to spin. For each application, there is a region between approximately $2L + 2o + W$ and $6W$ where performance is relatively flat; within this flat region, falls our derived value of $S_{Cond} = 3W + 2o$. Across all workloads and context-switch costs, the performance with $3W + 2o$ is within 35% of ideal explicit coscheduling and within 10% of the best performance achieved for all measured conditional spin-times. In some cases, the conditional spin amount with the best measured performance is slightly lower than $3W + 2o$; in other cases, it is slightly higher.

In the final region, conditional spin time is increased well past its optimal point; thus, processes continue spinning even when receiving messages relatively infrequently. When messages arrive less frequently than once every $6W$, the cost of spinning exceeds the benefit of maintaining partial coordination. Note that performance may degrade more noticeably for a given amount of conditional spin than baseline spin, because multiple consecutive conditional spins may be performed for a given communication operation.

In summary, our simulations demonstrate three results. First, no fixed spin amount with two-phase waiting achieves acceptable performance for all workloads within implicit coscheduling. Second, processes must be willing to spin longer under dynamically changing conditions; the penalty of blocking is higher when the frequency of incoming messages is higher. Third, conditional spinning is a practical and effective way of adapting to this penalty at run-time; with conditional spinning, a wide range of applications can be scheduled with almost ideal performance.

6 Conclusions

With implicit coscheduling, we have explored an interesting on-line algorithm: determining the interval which time-shared processes should spin when waiting for responses from remote processes. While competitive analysis has been used for similar waiting problems, such as acquiring shared locks, its fundamental assumptions are often violated; the waiting times in the stream of inputs are neither independent nor adversarial, but instead belong to a closed system. In our environment, sets of processes are capable of influencing their future waiting times based upon their current behavior.

In this paper, we have presented the intuition for a cooperative algorithm. The basic idea is that processes can detect when the system is coordinated and can encourage the system to remain in this beneficial state; alternatively, processes can detect when the system is not coordinated and limit the cost of being in this undesirable condition. Coordinated scheduling is beneficial because it minimizes the waiting times of communication and synchronization events.

An additional challenge in our environment is that the penalty of blocking is not constant and is not known before the process begins waiting; instead, the penalty extends to other processes and depends upon the number of incoming messages. Our solution is to apply conditional two-phase waiting, in which the process adapts its spin-time according to the rate that messages are arriving. Our simple model of the system allows us to analyze the minimum time a process should spin-wait, as well as the frequency at which messages must arrive to warrant spinning.

Our simulation results corroborate our analysis. Due to the cooperative nature of the system and the variable penalty of blocking, spinning for the cost of a context-switch (or any fixed amount) performs much worse than a factor of two from optimal. However, with our conditional two-phase waiting algorithm, all the workloads we have analyzed perform within 35% of ideal.

While our simulation results are extremely promising for a range of workloads, we cannot yet bound the worst-case performance of implicit coscheduling. Rigorous analysis that accounts for cooperative feedback in distributed system is clearly still needed. Our unabashed hope is that the parallel theory community will find this problem interesting and apply their expertise.

References

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

[2] 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.

[3] A. C. Arpaci-Dusseau. *Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*. PhD thesis, University of California at Berkeley, December 1998.

[4] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1998.

[5] A. Bar-Noy, Y. Mansour, and B. Schieber. Competitive Dynamic Bandwidth Allocation. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39, June 1998.

[6] H. Bast. Dynamic Scheduling with Incomplete Information. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 182–191, June 1998.

[7] P. Berenbrink. Parallel Continuous Randomized Load Balancing. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 192–201, June 1998.

[8] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, April 1994.

[9] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Supercomputing'94*, Nov. 1994.

[10] 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.

[11] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. E. von. 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–18, November 1993.

[19] A. Hori, H. Tezuka, and Y. Ishikawa. Global State Detection using Network Preemption. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

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

[21] A. R. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms For Nonuniform Problems. *Algorithmica*, 11(6):542–71, June 1994.

[22] L. I. Kontothanassis and R. W. Wisniewski. Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[23] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[24] S. T. Leutenegger. *Issues in Multiprogrammed Multiprocessor Scheduling*. PhD thesis, Univerisity of Wisconsin-Madison, 1990.

[25] B.-H. Lim and A. Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, Aug. 1993.

[26] S. Muthukrishnan and R. Rajaraman. An Adversarial Model for Distributed Dynamic Load Balancing. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 47–54, June 1998.

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

[28] D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202–208, 1985.

[29] 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.

[30] M. S. Warren, D. J. Becker, M. P. Goda, J. K. Salmon, and T. Sterling. Parallel Supercomputing with Commodity Components. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1372–1381, Las Vegas, Nevada, June 1997.

[31] 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.

[32] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, Apr. 1991.

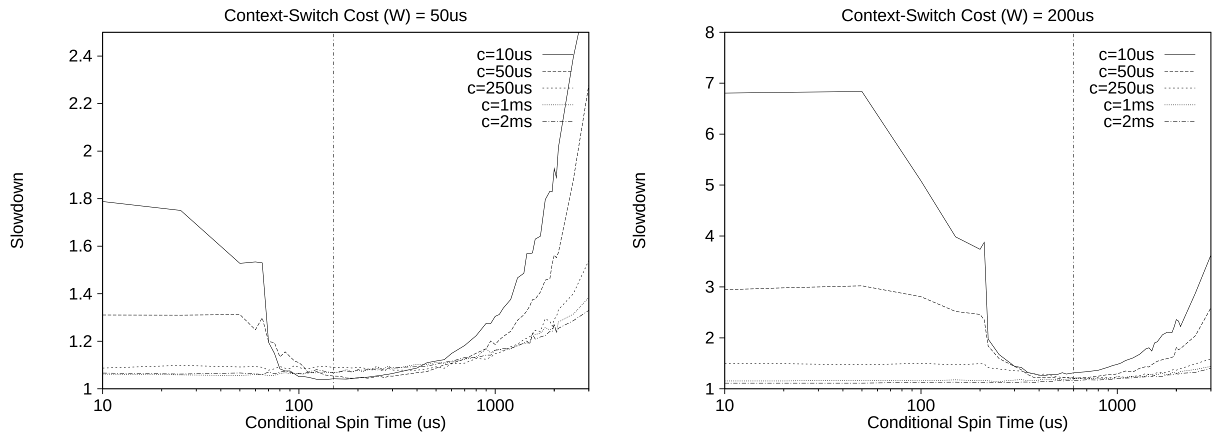

Figure 5: **Sensitivity to Conditional Spin for Continuous-Communication Programs.** *Each line in the graphs designates a different communication granularity, c. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the pairwise spin time for both reads and barriers are changed simultaneously. The vertical line marks* $3W$*, our chosen value of* S_{Cond}*.*