Implicit Coscheduling:
Coordinated Scheduling with Implicit Information in Distributed Systems

Andrea Carol Arpaci-Dusseau
Computer Sciences Department
University of Wisconsin – Madison

Abstract

In modern distributed systems, coordinated time-sharing is required for communicating processes to leverage the improved performance of switch-based networks and low overhead protocols. Coordinated time-sharing has traditionally been achieved with gang scheduling or explicit coscheduling, implementations of which often suffer from many deficiencies: multiple points of failure, high context-switch overheads, and poor interaction with client-server, interactive, and I/O-intensive workloads.

Implicit coscheduling dynamically coordinates communicating processes across distributed machines without these structural deficiencies. In implicit coscheduling, no communication is required across operating system schedulers; instead, cooperating processes achieve coordinated scheduling by reacting to *implicit information* carried by naturally-occurring communication within the parallel application. The implementation of this approach is simple and allows participating nodes to act autonomously.

We introduce two key mechanisms in implicit coscheduling. The first component is *conditional two-phase waiting*, a generalization of traditional two-phase waiting in which spin-time may be increased depending upon events that occur while the process spins. The second component is a fair and preemptive local operating system scheduler; we develop an extension to a stride scheduler with these properties.

To demonstrate that implicit coscheduling performs well, we show results from an extensive set of simulation and implementation experiments. To exercise the conditional two-phase waiting algorithm, we examine three distinct workloads: bulk-synchronous and continuous-communication synthetic applications as well as parallel application kernels written in the Split-C language. To exercise the local scheduler, we examine workloads containing competing jobs with different communication characteristics. Our experiments show that implicit coscheduling is effective and fair for a wide range of workloads; most workloads perform within 30% of an idealized model of gang scheduling. Finally, we demonstrate that our implementation scales well as a function of the number of jobs and the number of workstations and is robust to job placement.

1 Introduction

Clusters have emerged as an important platform for building cost-effective, incrementally-scalable, highly-available servers [2, 54]. Modern clusters can support a wide variety of job classes. First, due to the advent of low-latency, high-bandwidth, switch-based networks [8, 9] and low-overhead message protocols [38, 43, 49], clusters can efficiently execute fine-grain parallel applications [39]. Second, clusters can improve the client-server applications of traditional distributed systems (*e.g.*, naming, locking, and file services) by updating these services to leverage modern communication protocols [3]. Third, clusters constitute an excellent platform for applications with large I/O demands [7, 18]. Finally, autonomous workstations or PCs in the cluster can support the interactive jobs found in general-purpose workloads [4].

Scheduling and resource allocation are challenging problems in the presence of this broad range of job classes. Communicating processes, whether part of a parallel or a client-server application, are difficult to handle because they must be scheduled simultaneously on different workstations to achieve good performance. Therefore, the first step toward scheduling the complete range of job classes is to provide a scalable and reliable *mechanism* for dynamically coordinating communicating processes, while allowing each autonomous scheduler the flexibility to appropriately handle its interactive and I/O-bound jobs.

In this paper, we describe our mechanism for coordinating communicating processes: *implicit coscheduling*. Traditionally, coordinated time-sharing of parallel jobs has required explicit control; for example, in gang-scheduling [26], or explicit coscheduling [42], one or more master components determines a global schedule of processes over time and explicitly communicates this schedule to the participating remote schedulers. Due to the cost and complexity of explicitly contacting other components with control information, we propose that coscheduling should instead be built with *implicit control*, in which autonomous components infer the state of the rest of the system from existing local events. Implicit coscheduling is implemented by adding a small amount of intelligence to the existing communication layer in parallel applications; this modification enables each communicating process to determine the scheduling state of the parallel job across the cluster. Each process then independently decides if it is beneficial to run and shares this knowledge with its local operating system scheduler.

Processes learn the state of the system through two pieces of *implicit information*. We define implicit information as the characteristics of an event that exist outside of the defined interface of the event; examples of implicit information include the time between events, the amount of power consumed, or the number of cache misses generated, much like a *covert channel* [17]. Implicit information is useful when a component is then able to use this information to infer the state or actions of remote components in the system. An example of a system that leverages implicit information is the TCP congestion-control algorithm [33], which observes packet loss to infer network congestion.

Within implicit coscheduling, the first piece of implicit information is the round-trip time of request-response messages occurring within the parallel application; in general, receiving a fast response to a request message implies that the remote destination process is scheduled, whereas receiving a slow response implies that the remote process is not scheduled. The second piece of implicit information is the arrival rate of incoming messages; a message arrival implies that the remote sending process is scheduled.

We introduce a simple mechanism that allows processes to react to this information in the communication layer: *conditional two-phase waiting*, a generalization of traditional two-phase waiting [42]. In the first phase of two-phase waiting, the process spins for some baseline amount of time while waiting for the desired event (*e.g.*, for the message response to arrive); if the event does not occur, then, in the second phase, the process voluntarily relinquishes the processor and blocks. Unlike traditional two-phase waiting where the spin-time is determined before the process begins waiting, with conditional waiting the process may dynamically increase its waiting time, depending upon observed events (*e.g.*, the arrival of other messages). In this paper, we analyze in detail the required properties of conditional two-phase waiting; with a simple model of the relevant network and system parameters, we derive the amount of time that processes should spin-wait at communication events.

Each process then combines its knowledge of the scheduling state of jobs within the cluster with the *cost model* exported by the local scheduler; the cost model determines how frequently the process will be scheduled as a function of its previous behavior. We develop a local scheduler based on a *stride scheduler* [51] that exports a fair cost model; in our scheduler, a process receives the same amount of CPU averaged over a given interval regardless of how it utilizes its resources, within certain limitations. In this paper, we show that with a fair local scheduler, processes communicating at widely different rates receive approximately the same amount of resources over the entire cluster.

Due to the fact that no new modules, communication, or dependencies between nodes are added, the implicit coscheduling mechanism has many properties that are desirable of services in a distributed system.

First, the implicit coscheduling service survives the failure of any node in the system. Second, new machines can dynamically enter and exit the system without any registration with implicit coscheduling. Finally, each local scheduler is autonomous; thus it can adapt to heterogeneous computation or communication speed and is free to schedule interactive and I/O-bound processes along with its communicating jobs as it chooses. Due to these strengths, we believe that if implicit coscheduling can coordinate traditional parallel applications as well as gang scheduling, then implicit coscheduling is a worthwhile mechanism. In this paper, we demonstrate with an extensive combination of simulation and implementation experiments, that implicit coscheduling performs nearly as well as an ideal model of gang scheduling on those traditional parallel workloads for which gang scheduling was designed.

The rest of this paper is organized as follows. In Section 2, we cover previous research in scheduling communicating processes. We present our simulator and model of the cluster and its workload in Section 3 and give an overview of implicit coscheduling in Section 4. We focus on the requirements of the conditional two-phase waiting algorithm in Section 5 and on the operating system scheduler in Section 6. In Section 7 we describe our implementation on the U.C. Berkeley NOW cluster and present measurements of our system. Finally, we discuss the limitations of our work in Section **??** and summarize in Section 9.

2 Background

In this section, we describe how communicating processes have been scheduled in distributed systems in the past. Due to its complexity, parallel scheduling is usually decomposed into two interdependent steps: allocation and dispatching. The first step, *allocation*, determines the number of processors to allocate to a parallel job and the placements of processes on those processors. In the second step, *dispatching*, the allocated processes are context-switched over time.

The allocation step determines whether space-sharing or time-sharing is used: with pure *space-sharing*, only processes from the same parallel job can share a processor; with *time-sharing*, processes from competing parallel jobs can share a processor. A system that supports time-sharing has a number of advantages over pure space-sharing. First, with time-sharing, changes to the programming model are not required for good performance (*e.g.*, the popular Single-Program-Multiple-Data, SPMD, programming style with message-passing can be used). Second, adding time-sharing gives better response time than pure space-sharing, especially for jobs with large resource requirements [24, 53] or those performing I/O [44]. Third, time-sharing supports system services and daemon processes that must run on each node [4]. Finally, the appeal of time-sharing is reflected in the large number of implementations on MPP systems, shared-memory multiprocessors, and networks of workstations [23].

In this paper, we consider the problem of dispatching in a time-shared system after the parallel processes have been allocated and placed. We begin by considering three methods for time-sharing competing processes *local scheduling*, *gang scheduling*, and *dynamic coscheduling*; we discuss the limitations of each given the workloads in modern clusters.

2.1 Local Scheduling

With local scheduling, the operating system scheduler running on each workstation in the cluster simply dispatches the processes that have been allocated to it; each process is scheduled as if it were independent of the cooperating processes allocated to other machines. Given this lack of coordination across schedulers, when a message arrives, it is likely that the destination process is not currently scheduled; if the message requires a response, then the sending process must wait until the destination process is scheduled and returns the response before it can make forward progress. As a result, applications that communicate frequently and use small messages experience poor performance relative to that achieved in a dedicated or coordinated environment.

The precise performance of local scheduling depends on the action of a process waiting for a response from a remote node; the waiting process has three options. First, the process can *spin-wait*, anticipating a quick response from the destination process. Second, a process can *block immediately*, relinquishing the processor so that a competing process can be scheduled until the response arrives. Third, the process can use *two-phase waiting* [42]: in the first phase, the process spin-waits some amount of time for the desired response; if the response does not arrive, then the process blocks in the second phase.

Numerous studies have shown local scheduling with spin-waiting can lead to performance that is orders of magnitude worse than with coordinated scheduling [4, 13, 26, 37]. When scheduling is not coordinated, a waiting process often does not receive the desired response until it is scheduled again in its next time-slice. As a result, the waiting process pays the cost of spinning idly for the remainder of its time-slice without making forward progress. Thus, each communication operation incurs a spin-waiting overhead equal to the duration of a time-slice rather than an overhead equal to the round-trip time of the network.

The performance of local scheduling can be improved significantly when processes block immediately [13, 26]; for every communication operation, processes now pay the cost of a context-switch. On systems where the inherent waiting time of a communication operation is very high (*i.e.*, in coarse-grain applications with significant load-imbalance or on networks with high latency), immediate blocking can even achieve better performance than coordinated scheduling with spin-waiting [26, 27]; under these conditions, the waiting time can be overlapped with useful computation in a competing process. Nevertheless, because the cost of a context-switch is greater than the latency of modern networks, many fine-grain applications still perform much worse with immediate blocking than with coordinated scheduling.

Little attention has been paid to the use of two-phase waiting for communicating processes. In one study [27], Feitelson and Rudolph find that two-phase waiting with a spin-time equal to the local context-switch cost performs poorly. Because of their choice of a small spin-time and a simple round-robin local scheduler, communicating processes are usually uncoordinated and, thus, the waiting time at communication operations is longer than the time the process spin-waits. As a result, each communication operation incurs both the spin-waiting penalty and the penalty of a context-switch; for fine-grain applications, performance is nearly two times worse than with immediate-blocking. In this paper, we show that with the appropriate spin-time and a suitable local scheduler, two-phase waiting is a suitable mechanism for coordinating processes.

2.2 Gang Scheduling

Gang scheduling and coscheduling coordinate the scheduling of communicating processes across nodes; *gang scheduling* guarantees that communicating processes are scheduled simultaneously, whereas *coscheduling* attempts to do so [42]. Both approaches improve the performance of fine-grain applications relative to local scheduling, because when one process communicates with another, the destination process is scheduled and can promptly reply while the waiting process spin-waits. Despite the ability of gang scheduling and coscheduling to improve the performance of fine-grain parallel applications, both have a number of disadvantages. The first set of drawbacks occurs due to the structural limitations of the implementations; the second set relates to the workloads that can be supported. For simplicity in this paper, we focus our discussion on gang scheduling without loss of generality.

Many gang scheduling implementations are constructed such that a master process running on a single processor controls the entire system, but a centralized master has a number of problems. First, the master may limit the scalability of the system; since the master controls the timing of the global context-switch, the cost of a context-switch tends to to increase with the scale of the system [12, 28, 32]. Second, the master represents a single point-of-failure; while such a design may have been acceptable in a supercomputer where all components could be placed in a controlled environment, the operation of an entire cluster cannot depend on a single machine. Although implementations such as Distributed Hierarchical Control (DHC) [25] improve the scalability and reliability of gang scheduling, machines but must still act as slaves to higher-level controllers. Therefore, nodes cannot enter and exit the system without contacting the appropriate masters and potentially restructuring the hierarchy.

A more severe disadvantage of gang scheduling is its inability to handle general-purpose workloads. Due to their disparate requirements, interactive jobs do not mix well with parallel applications that are gang scheduled. If interactive processes are allocated to a single time-slice in the coscheduling matrix, then their response time suffers [36]. On the other hand, if an interactive jobs or paging activity preempt parallel applications, then the performance of fine-grain parallel applications suffers [4, 12]. Finally, client-server applications cannot be supported because gang scheduling requires that communicating processes be identified *a priori*. Although communicating jobs can be identified at run-time [27], a master process must still explicitly monitor communication between processes, construct a global schedule, and coordinate the simultaneous global context-switch.

2.3 Dynamic Coscheduling

Dynamic coscheduling has the goal of dynamically identifying and coordinating communicating processes. In dynamic coscheduling, an arriving message triggers the scheduling of the destination process, modulo workload-specific fairness criteria. Dynamic coscheduling has been modeled, simulated, and implemented as follows.

The first high-level models and simulations of dynamic coscheduling showed, for two competing jobs, that if messages are sent to random destinations at a sufficiently high rate, then the steady-state probability is that one of the two jobs is always coscheduled [46]. The major simplifications of their model are that jobs spontaneously context-switch independently across workstations (representing both the expiration of time slices and blocking due to waiting for communication) and that context-switches and communication occur instantaneously.

An implementation of dynamic coscheduling with Illinois Fast Messages (FM) [43] is described for WindowsNT [11] and for Solaris 2.4 [45]. These implementations appear to have required changes at multiple levels of the system: the device driver for the Myrinet interface card, the program on the LCP (LANai Control Program), and the FM messaging library. Due to the limitation that FM can only support one communicating job at a time, all measurements examine a single parallel application in competition with multiple sequential jobs. Their experiments show that raising the priority of the destination job on message arrival is a suitable mechanism for achieving good performance, assuming that the fairness criteria can be tuned for each workload.

3 System Model and Simulator

In this section, we discuss our assumptions about the types of systems in which implicit coscheduling can be used effectively. In describing our model of the system, we focus on the workstations and their interconnect, the operating system on each node, and the workload. This model is used in our analysis and as the basis of our an event-driven, process level simulator, `SIMplicity`. Our implementation environment is discussed in Section 7 in terms of this model as well.

3.1 Cluster Architecture and Communication Layer

For our distributed system environment, we consider *clusters*, consisting of a fixed number of machines, whether workstations or PCs. Our current analysis assumes a single processor per node (and, therefore, we use the terms *node*, *processor*, and *workstation* interchangeably) and that every machine in the cluster has identical performance.

The nodes are connected with a simple network that is characterized with the LogP model [16].

- L**:** the *latency*, or delay, incurred in communicating a message that contains a small, fixed number of words from the source node to the destination. We assume that messages are always transmitted reliably and received in-order.
- o**:** the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of a message; during this time, the processor cannot perform other operations.
- g**:** the *gap*, defined as the minimum time interval between consecutive message transmissions or receptions at a processor. The reciprocal of g is the available per-processor communication bandwidth.
- P**:** the number of processors.

We assume that the communication layer can handle multiple communicating processes simultaneously. Therefore, coordinated scheduling is needed only to improve performance, not for correctness.

The LogP parameters can be configured in `SIMplicity`. The simulations in this paper examine clusters containing $P = 32$ workstations. The network is configured such that $L = 10\mu s$, which closely matches the latency in our implementation. To simplify the simulator and improve its execution time, we often set processing overhead (o) and network gap (g) to zero. This simplification does not lead to significant discrepancies in performance from our implementation, especially given workloads with small messages and little destination contention.

3.2 Operating System Scheduler

We assume that every workstation runs its own copy of a commodity operating system and that its scheduler has been tuned for time-shared, general-purpose workloads. An important parameter of the operating system scheduler is the time, W, to context-switch to a new process. For most experiments in this paper, we examine a context-switch cost of $W = 50\mu s$, which roughly matches that of our implementation. We assume that the time to wake a process on a message arrival is equal to the basic context-switch cost.

We assume that a process must be scheduled to send, receive, or handle a message. If a message arrives for an unscheduled process, the message is buffered until the process is scheduled. If the process is sleeping, it is woken and placed on the ready queue; whether or not it is scheduled depends upon the decision of the local scheduler. In our model and simulations, we assume that a receiving process is informed of a message arrival through an asynchronous interrupt; in our implementation, the receiving process must poll the network to determine if a message has arrived.

The local scheduling component of `SIMplicity` is nearly identical to that of the default Solaris [29] scheduler in both functionality and structure; in fact, significant portions of our code are adapted directly from Solaris 2.4 source.[1] We also consider an extended stride scheduler described in Section 6.

To stress implicit coscheduling, we make several pessimistic assumptions about the timing of events across workstations. First, all timer events occur independently across processors (*e.g.*, the $10ms$ clock tick and the one second update timer in the Time-Sharing Class). Second, if multiple parallel jobs arrive in the system at the same time, then the processes are randomly ordered across the local scheduling queues.

To evaluate the ability of implicit coscheduling to obtain coordinated scheduling, we compare its performance to an idealized version of gang scheduling. First, for our model of gang scheduling, there is

[1] Note that the behavior of the simulated scheduler in this paper differs in two ways from that originally presented in [20]. First, in the previous simulations, a sleeping process was unrealistically given kernel priority (above 59) when a message arrived; thus, this process was almost always promptly scheduled. Second, the process immediately slept again if the arriving message was not the event for which the process was waiting.

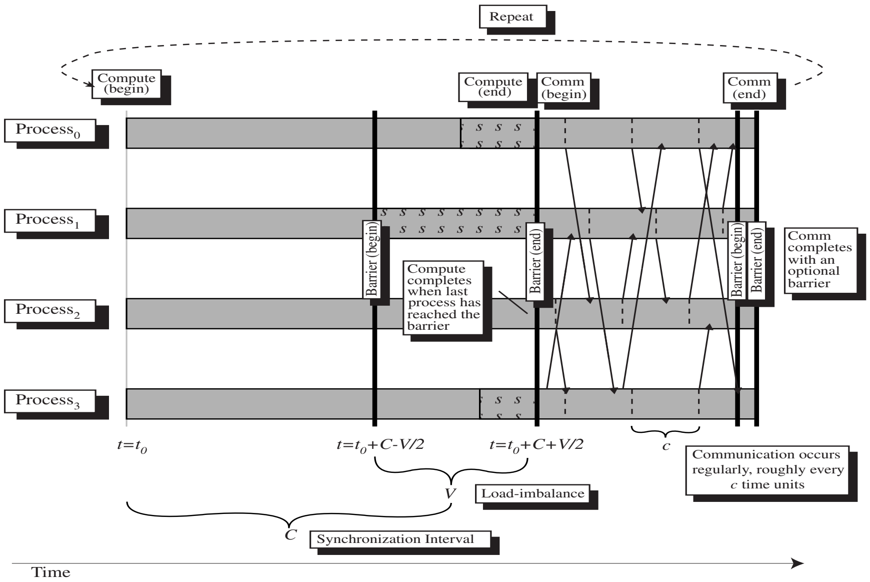

Figure 1: **Model of Bulk-Synchronous Synthetic Parallel Applications.** *Each process of a parallel job alternates between computation and communication phases. Processes compute for a mean time C before executing the opening barrier of the communication phase; the variation in computation across processes is uniformly distributed in the interval (0, V). Within the communication phase, each process computes for a small time, c, between reads; the communication phase ends with a barrier.*

no skew of time quanta across processors; that is, the global context-switch occurs simultaneously across all machines. Second, the cost of the global context-switch is identical to a local context-switch, W. Finally, to amortize the cost of the context-switch, we assume a long time-slice ($Q = 500ms$). When gang scheduled, processes always spin-wait for communication events to complete, as they would in a dedicated environment.

3.3 Application Workload

In this paper, we evaluate workloads that consist entirely of parallel applications. We believe that coordinating processes in this restricted environment is the first step in handling general-purpose workloads. Because our system leverages commodity schedulers tuned for interactive and I/O-bound workloads, we anticipate that implicit coscheduling will continue to perform well under more diverse circumstances.

We consider parallel applications written with the single-program-multiple-data (SPMD) model. We assume that the scheduler has no control over the communication or synchronization performed by the applications. We assume that cooperating processes communicate with low-level messages, such as those defined by the Active Message model [50]. The following parameters for characterizing a parallel job were chosen to match applications written in Split-C [15].

- **Request-response communication**: the requesting process must wait for a response from the destination process before it can proceed with additional computation. We consider a `read` of remote data the generic request-response operation.

- **All-to-all synchronization:** each participating process must wait until all other processes have reached the operation before any may continue. We model a `barrier` as prototypical of all synchronization operations.

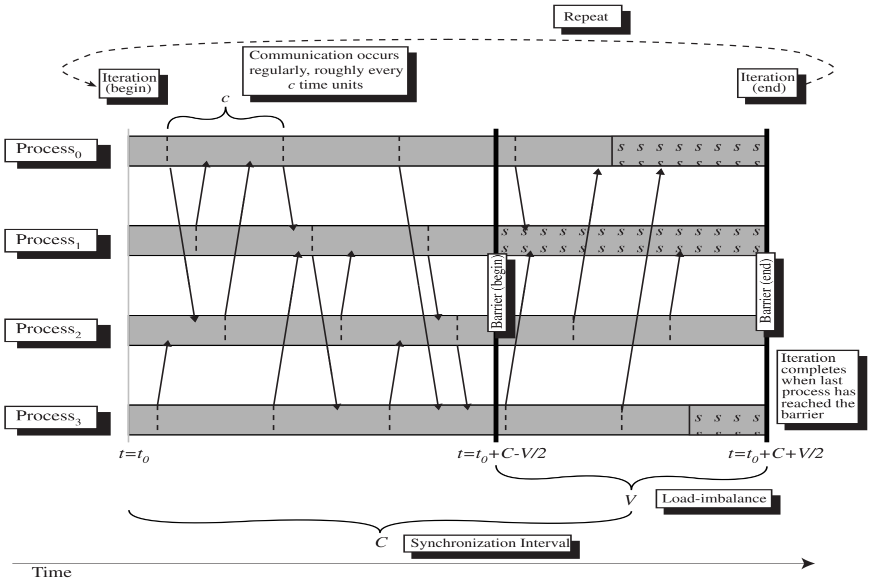

Figure 2: **Model of Continuous-Communication Synthetic Parallel Applications.** *The average time between communication requests for a given process is* c *time units. Each process periodically performs a barrier; the mean time between barriers is* C*, chosen uniformly from the interval (*$C - V/2$*,* $C + V/2$*).*

These operations have been found sufficient to model applications written in the MPI message-passing interface as well [55].

`SIMplicity` is driven by the arrival of a parallel job at a specified time; unless otherwise noted, we restrict the simulated workloads to contain three identical competing jobs that arrive at the same time. Each job is characterized with the following parameters.

- **Processes:** the number of processes, P, in the job. The processes are numbered 0 to $P - 1$. In our simulations, the number of processes in each job matches the number of workstations.
- **Communication Interval:** the average time, c, between reads on the sending process.
- **Communication Pattern:** the destination processes from which a process reads. We examine two patterns in this paper: in `NEWS`, each process reads from its four nearest neighbors in a grid; in `Random`, each process reads from a destination process chosen uniformly at random.
- **Synchronization Interval:** the average time, C, between barriers across all cooperating processes.
- **Load-Imbalance:** the difference in arrival time across processes at a barrier. v_i is the load-imbalance observed by process i and V is the worst load-imbalance observed across all processes. We choose each v_i independently from a uniform distribution.
- **Iterations:** the number of barriers performed by each process. In the simulations, the number of iterations is selected such that the execution of one application in a dedicated environment requires approximately 10 seconds of simulated time.

We simulate two styles of synthetic parallel jobs: bulk-synchronous and continuous-communication. A *bulk-synchronous* application alternates between phases of computation and communication, as shown in Figure 1; this style is common in many applications with bursty communication. In our simulations, the time between communication operations is fixed at $c = 8\mu s$. Because the bulk-synchronous applications

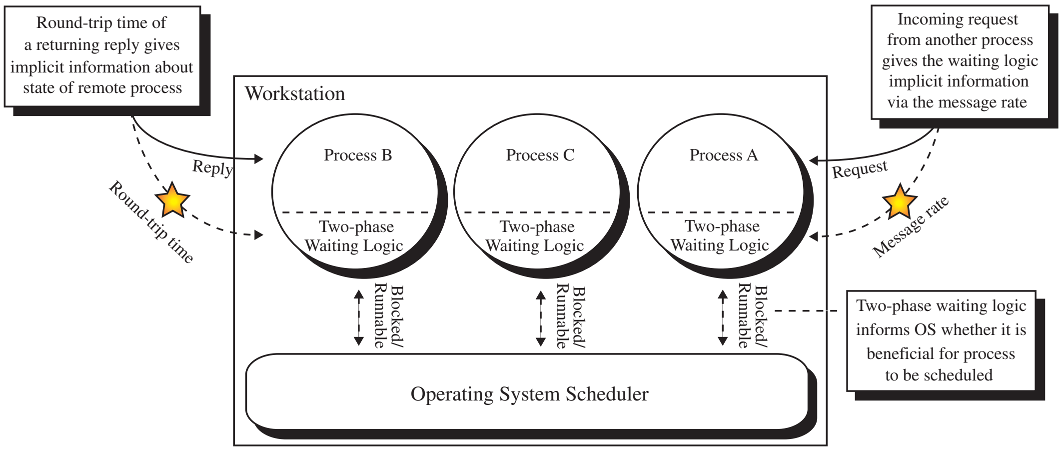

Figure 3: **Transfer of Information in Implicit Coscheduling.** *Processes that are part of a communicating job receive implicit control information from the processes they are currently communicating with. This information is used as part of the two-phase waiting logic within each communicating process. From the round-trip time of returning reply message and from the arrival of incoming messages, the local process can infer whether the remote process is likely to be currently scheduled. The local process informs the operating system scheduler whether it is beneficial to be scheduled by either remaining runnable or by blocking until a message arrives.*

are relatively easy to schedule, we also examine continuous-communication applications, as illustrated in Figure 2. In the *continuous-communication* benchmarks, there are no distinct phases of computation and communication; instead, each process reads from a cooperating process every c time units.

When we evaluate workloads containing jobs with identical characteristics, our performance metric is the *slowdown* of the last job in the workload to complete with implicit coscheduling versus with ideal gang scheduling. A slowdown greater than one indicates that implicit coscheduling incurs an overhead for lack of coordination through additional context-switches and idle time; a slowdown less than one indicates that implicit coscheduling achieves a benefit from its flexibility in scheduling. When we examine jobs with different communication characteristics, we examine the slowdown of a single job competing against continuously-running background jobs.

4 Overview of Implicit Coscheduling

Implicit coscheduling is an example of an implicitly-controlled service: coordinated scheduling is achieved for communicating processes without control messages between machines in the system. Instead, to infer the scheduling state of the system, each process observes implicit information that is associated with the communication inherent to the parallel application. In this section, we outline the functionality of the local operating system scheduler and the behavior of communicating processes that are required to achieve this coordination. Figure 3 summarizes the basic requirements.

4.1 Local Scheduler

To obtain coordination efficiently and fairly, implicit coscheduling requires that each operating system scheduler supports preemption based on events, exports a fair cost-model to competing processes, and amortizes context-switches over a sufficiently long time-slice. Fortunately, because this functionality is also required to efficiently and fairly support general-purpose workloads, these qualities are present to some extent in most commodity schedulers; we discuss two such schedulers in Section 6. We now describe these three requirements in more detail.

4.1.1 Selective Preemption

Coordinated scheduling may be dynamically initiated by preempting a scheduled process in favor of a competing process that has just received a message. Since the process that sent the message was scheduled in the recent past, scheduling the destination process promptly increases the likelihood of coordinating the two processes.

We note that to schedule a process on message arrival, it is not necessary to modify the scheduler. If a process is not scheduled when a message arrives, then either the process is blocked or the local scheduler has determined that it is more fair (or more appropriate by some other criteria) to schedule a competing process. In the case where the process is blocked on a communication event, the arrival of any message wakes the process and the scheduler may then preempt the running process (if it is fair to do so, or meets the other criteria set by the scheduler). In this way, the two-phase waiting algorithm is often sufficient for scheduling processes on message arrivals and triggering coordination.

In two cases, scheduling the destination process on message arrival is likely to coordinate it with other cooperating processes in addition to the sender. The first situation occurs when the arriving message signifies the completion of a `barrier` operation. In this case, all participating processes have reached the barrier and are being sent completion messages at approximately the same time. Thus, if each participating process is scheduled promptly when its notification arrives, they will all be coordinated with one another. The second situation occurs when multiple processes send request messages to a single destination process that is not scheduled. In this case, the request messages are temporarily buffered until the destination process is scheduled, at which point it sends back the responses at nearly the same time. Once again, if each of the waiting processes is scheduled when its response arrives, the processes will be coordinated. Thus, applications containing frequent barriers or communication patterns in which all processes are dependent upon the same set of processes are more easily coordinated.

4.1.2 Fair Cost-Model

User-level processes have the responsibility of notifying their local scheduler of when it is beneficial for them to run; in most cases, this corresponds to those times when their scheduling is coordinated with other cooperating processes. For processes to make this calculation correctly, each scheduler must export a well-defined cost-model so that each process can determine how much of the CPU it will receive as a function of its usage pattern. To ensure that parallel jobs are given the same amount of CPU regardless of how much communication they perform, the local schedulers must export a *fair* cost-model in which processes are given the same proportion of the CPU regardless of when and how they utilize the CPU (*e.g.*, whether they spin-wait, context-switch, or perform useful computation).

We assume that the goal of each communicating process is to complete its execution (and the execution of its parallel job) as quickly as possible. If each process is guaranteed the same amount of the CPU regardless of when it executes and has a constant amount of work independent of when it is scheduled, then it is beneficial for the process to run in two specific cases. First, the process should run when it has useful local computation to perform. Second, the process should run when it is waiting for a remote event to complete if the time the process wastes by spin-waiting is less than the time the entire parallel job wastes if the process blocks. A process notifies the local scheduler that it is beneficial to run by simply computing or spin-waiting; it notifies the scheduler that it is not beneficial to run by blocking.

Determining the amount of time wasted by the waiting process and the parallel job as a whole is the key challenge of implicit coscheduling. The manner in which each process calculates when it should spin-wait versus block is encoded in the conditional two-phase waiting algorithm and described in Section 5.

4.1.3 Amortized Context-Switches

Most general-purpose operating system schedulers are based on periodic time-slices. Since we assume that time-slices are not coordinated across machines, the scheduling coordination of communicating processes is lost when the time-slice expires on any of the machines and must be re-obtained when a new time-slice begins. Therefore, it is critical that the length of the time-slice is long relative to the cost of achieving coordination, which is a function of the local context-switch cost. We note that a long time-slice does not preclude handling interactive processes; the scheduler may immediately run an interactive process promptly when it becomes runnable by preempting the running process if it is fair to do so according to the cost-model.

4.2 Communicating Processes

When a process is waiting for a communication or synchronization event to complete, it must determine whether the benefit of spin-waiting and remaining scheduled meets or exceeds the cost of using the CPU. In general, the benefit of staying scheduled corresponds directly to the extent to which the process is coordinated with the other processes in the parallel job. Fortunately, a communicating process can determine the extent to which it is coordinated by observing two pieces of implicit information.

First, a waiting process can determine whether it is coordinated with the other processes participating in the current communication operation by observing how long the operation takes to complete (*e.g.*, two processes participate in a read, whereas all of the processes in the job participate in a barrier). If the operation completes in the "expected" time, then the process infers that the cooperating processes are running; if the operation requires more time, the process infers that the cooperating processes are not running.

Second, the waiting process can determine whether it is coordinated with the remaining processes in the job by observing the incoming message rate. If a message arrives from a sending process, the waiting process infers that the sender is also running; if a message does not arrive, the process cannot make any inference without additional information about the communication behavior of the application.

Conditional two-phase waiting, a generalization of two-phase waiting, allows the process to leverage implicit information to dynamically choose the appropriate waiting behavior and to notify the local scheduler. 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. Note that conditional two-phase waiting differs from *adaptive two-phase waiting* [34] 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.

In our implementation, there are two components of spin-time: *baseline* and *conditional*. The baseline amount, 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 of spin-time correspond naturally to maintaining coordination with the two sets of processes in the parallel job. First, the baseline amount allows the waiting process to maintain coordination with the processes participating in the current operation; this amount is set to the expected time of the operation. Second, the conditional amount enables the waiting process to maintain coordination with the other processes in the job; this amount is set such that cost of spin-waiting between message arrivals is less than the cost of blocking and losing coordination.

In most systems, for a given waiting time, either blocking immediately or spin-waiting until completion is optimal; two-phase waiting is used simply to bound worst-case performance [34]. However, in implicit coscheduling, conditional two-phase waiting may be superior to both immediate-blocking and spin-waiting.

When a process is waiting, the relative cost of spinning versus blocking depends upon the dynamic rate of arriving messages; thus, the best performance may be achieved when the process spin-waits while messages are arriving frequently, but then blocks before the operation completes.

5 Cost-Benefit Analysis of Waiting

Although the effects of changing the conditional two-phase waiting algorithm are not completely orthogonal to changes in the local scheduler, for simplicity we evaluate each component separately. In this section, we describe the waiting algorithm that each process performs when dependent on a communication or synchronization event. In these simulations, we use only the Solaris Time-Sharing scheduler because this scheduler is adequate for workloads containing jobs with the same communication granularities. In Section 6 we use our fully-developed conditional two-phase waiting algorithm to investigate different local schedulers.

There are three components to the conditional two-phase waiting algorithm that help processes build and maintain scheduling coordination, yet bound the penalty processes pay when not coordinated.

1. **Baseline Spin:** At communication and synchronization operations, the sending process waits long enough to remain coordinated with the other processes participating in the operation, if already in such a state. We refer to the participating processes as the *destinations*. This minimum waiting time is the *baseline spin*.

2. **Conditional Spin:** If a process waits the baseline amount and determines that the destination processes are not currently scheduled, it may still be beneficial to remain scheduled; if the local process is handling requests from other processes, then the cost of keeping this process scheduled may be less than the cost of relinquishing the processor. *Conditional spin* is the interval in which the waiting process must receive a message from a *sender* to justify keeping the process scheduled.

3. **Adjustments for Load-Imbalance:** To know when a synchronization operation is expected to complete given that all participating processes are coordinated, the process must estimate the amount of load-imbalance in the application. When the load-imbalance (and thus the potential waiting time) is small, the process should maintain coordination by increasing the baseline spin a corresponding amount; when the load-imbalance is large, the process should block and avoid paying for the CPU.

For each component, we analyze the desired spin time and present simulations that illustrate the important concepts.

5.1 Baseline Spin: Maintaining Coordination with Destinations

Once coordination has been obtained by scheduling a process previously blocked on a message arrival, it is the responsibility of the process to maintain this coordination. A communicating process remains coordinated with destinations by spin-waiting for the expected completion time of communication and synchronization operations given that the destinations are scheduled; this is the *baseline spin* amount. Baseline spin, denoted S_{Base}, differs for each type of operation: in our case, reads and barriers. For each operation, there are two circumstances in which cooperating processes are coordinated: in the first, the processes are coordinated before the request message arrives; in the second, the request message triggers the scheduling of the destination process. We define T_{Sched} and $T_{Trigger}$ as the time for these two cases, respectively. If the operation does not complete in the greater of these two times, the local process infers that the destinations are not coordinated and may block, thus bounding the amount of time it wastes with spinning.

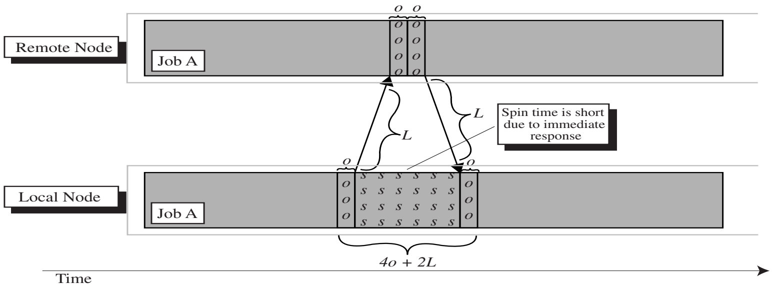

Figure 4: **Time for Request-Response Message with Remote Process Scheduled.** *The diagram illustrates T^R_{Sched}, the amount of time a local process must wait for a reply if the remote destination process is scheduled when the request message arrives.*

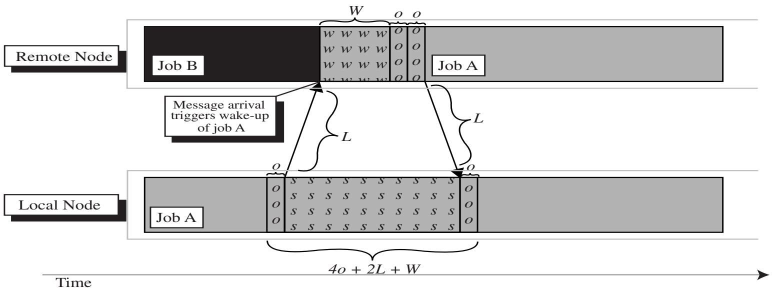

Figure 5: **Time for Request-Response Message with Triggering of Remote Process.** *The diagram illustrates $T^R_{Trigger}$, the amount of time a local process must wait for a reply if the request message triggers the scheduling of the remote destination process.*

5.1.1 Analysis

We begin by evaluating the baseline spin amount for reads. S^R_{Base} is the maximum of T^R_{Sched}, the time required for a read when the destination process is scheduled before the request arrives, and $T^R_{Trigger}$, the time required when the arriving request triggers the scheduling of the destination process.

First, we examine T^R_{Sched}, the minimum time for a read to complete if the destination process is scheduled and attentive to the network. T^R_{Sched} is essentially the round-trip time of the network for a given message size. Figure 4 shows the round-trip time with the LogP model, assuming that the read data fits in a single packet. First, the local process spends time o in overhead injecting the request message into the network. The request travels through the network for L time units, at which point it arrives at the destination node. Assuming that there is no contention with other messages, the request is handled after another o time units. The destination process spends time o returning the response, which arrives L units later at the local node and is handled in o units.

$$T^R_{Sched} \quad = \quad 2L + 4o$$

In some cases, although the destination process is not scheduled before the request arrives, the arriving request triggers it to be scheduled. Waiting for $T^R_{Trigger}$ ensures that the system does not thrash between competing sets of communicating processes. This case occurs when the destination process is blocked waiting on a response from another process; since any arriving message will wake the process, the destination process may be scheduled if the local scheduler considers it fair to do so (as described in the next section).

Figure 5 shows that the expected completion time of $T^R_{Trigger}$ is longer than T^R_{Sched} by the amount of a context-switch, W, on the remote node.

$$T^R_{Trigger} \quad = \quad 2L + 4o + W$$

In summary, S^R_{Base} is the time that the local process should spin at a read to ensure that it remains coordinated with the destination process. To be precise, S^R_{Base} does not include the overhead, $2o$, paid on the local processor to send the request and to handle the response:

$$\begin{aligned} S^R_{Base} &= \max(T^R_{Sched}, T^R_{Trigger}) - 2o \\ &= 2L + 2o + W. \end{aligned}$$

We continue by evaluating the baseline spin amount for a process p at a barrier, S^B_{Base}. The completion requirements of a barrier are stricter than those of a read: not only must all participating processes be scheduled, but each process must also have completed the work that preceded the barrier. To simplify the discussion in this section, we assume that there is no load-imbalance within the application; we remove this restriction in Section 5.3.

The time for a barrier with no load-imbalance depends on how the barrier is constructed and the number of processes. In our model, each of the processes participating in the barrier sends a message to a root process. When the root has received P such messages, it broadcasts a barrier-completion message back to the waiting processes. All processes communicate directly with the root rather than in a hierarchical tree structure, so that one unscheduled process does not affect the notification of other processes. Once again, we consider the completion time when all processes have been scheduled since the last barrier, T^B_{Sched}, and when the arrival of a late process triggers the scheduling of blocked processes, $T^B_{Trigger}$.

The time for T^B_{Sched} is illustrated in Figure 6. Assuming each of the P processes arrives at the barrier simultaneously, then each spends time o injecting a message to the root process; these messages arrive at the root process L time units later. Due to contention, the root requires time $(P-1)\max(o, g) + o$ to handle the messages. After the root has determined that all P processes have reached the barrier, it notifies each in turn. For a given process, the time for the barrier to complete depends upon its rank in the order in which processes are notified. Process p must wait time $p\max(o, g) + o$ for its message to be injected into the network, L units for the message to travel through the network, and o units to handle the notification.[2]

$$T^B_{Sched}(p) \quad = \quad 2L + 4o + (p + P - 1) \cdot \max(o, g)$$

Next, we consider the case where scheduling has not been precisely coordinated since the previous barrier and some processes block while waiting for the barrier to complete, $T^B_{Trigger}$. Since the completion of this barrier may trigger the scheduling of blocked processes, the last processes to reach the barrier should remain scheduled by spin-waiting. In the worst-case, $P - 1$ processes are late and the root process has already blocked, requiring a context-switch.

$$T^B_{Trigger}(p) \quad \leq \quad 2L + 4o + W + (p + P - 2) \cdot \max(o, g)$$

In summary, $S^B_{Base}(p)$ is the time that process p should spin-wait at a barrier to ensure that it remains coordinated with the other processes in the parallel job. As with reads, the waiting process does not spin for

[2]The equation could be trivially modified to account for the fact that the root process may not need to send messages to itself.

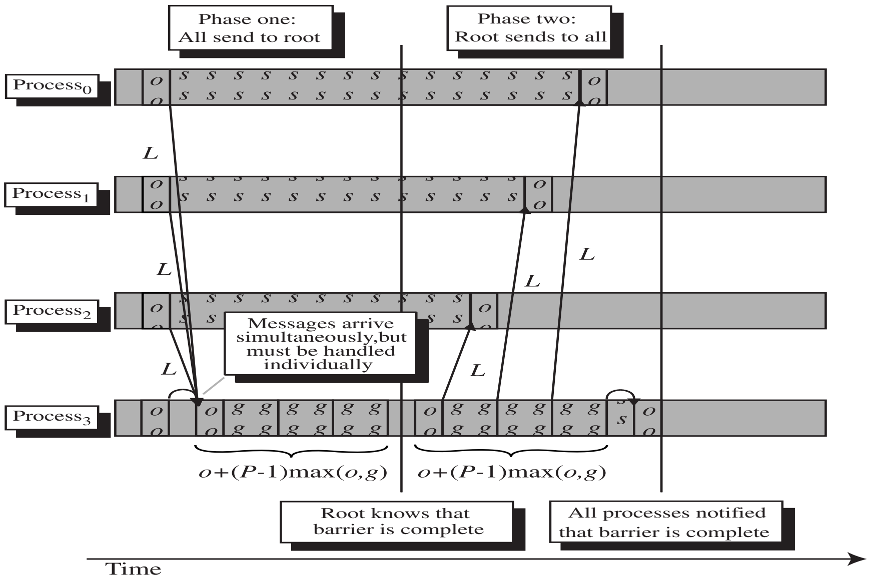

Figure 6: **Time for All-to-all Synchronization with Processes Scheduled.** *The diagram shows T^B_{Sched}, the time for a barrier with no load-imbalance to complete if all participating processes are scheduled in a coordinated fashion.*

the overhead of sending the request and receiving the response. The implementation may be simplified by having each of the processes wait for $S^B_{Base}(P-1)$ as if it were the last process to be notified.

$$\begin{aligned} S^B_{Base}(P-1) &= \max(T^B_{Sched}(P-1), T^B_{Trigger}(P-1)) - 2o \\ &= 2L + 2o + 2(P-1)\cdot\max(o,g) + W \end{aligned}$$

5.1.2 Simulations

The simulations in this section illustrate the benefit of spinning the baseline amount at communication and synchronization operations. We will show that while this simple waiting algorithm is sufficient to keep bulk-synchronous applications coordinated, it is not adequate for continuously-communicating applications.

In these simulations, we measure the slowdown of three competing jobs with implicit coscheduling versus gang scheduling, while varying the baseline spin in the two-phase waiting algorithm of implicit coscheduling. We set the system parameters to $W = 50\mu s$, $L = 10\mu s$, and $o = g = 0$; therefore, $T_{Sched} = 20\mu s$ and $T_{Trigger} = 70\mu s$ for both reads and barriers.

We begin by evaluating two bulk-synchronous workloads with no load-imbalance: processes perform a barrier either every $C = 100\mu s$ or $C = 1ms$ and then communicate in a NEWS pattern with $c = 8\mu s$. The first graph in Figure 7 examines the impact of changing the read baseline spin, while setting the barrier baseline spin to its optimal value. The performance for a given spin amount can be explained by the percentage of operations that are *successful*; that is, those operations that complete while the waiting process is still spinning. The percentage of successful operations corresponds to how well the communicating processes are coordinated.

When a process waiting for a read blocks immediately or spins for less than $T^R_{Sched} = 20\mu s$, reads cannot complete successfully; thus, on every operation, a context-switch cost of W is incurred by the local

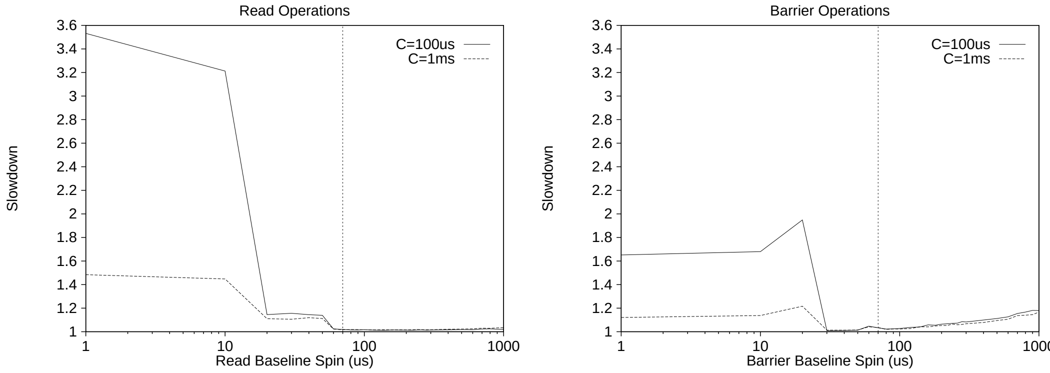

Figure 7: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs.** *Two workloads are examined with different synchronization granularity; the time between barriers, C, is set to either $100\mu s$ or $1ms$. The vertical line in each graph designates the derived baseline spin amount, S_{Base}. The metric along the y-axis is the slowdown of the workload with implicit coscheduling versus gang scheduling. Along the x-axis, the baseline spin for either reads or barriers is varied between $1\mu s$ and $1000\mu s$, while the baseline spin for the other operation is fixed at its derived amount.*

and destination processes. As a result, performance is poor relative to gang scheduling, which only pays the cost of spin-waiting for the round-trip time of the network.

As the baseline spin for reads is increased, performance improves dramatically at both T^R_{Sched} and $T^R_{Trigger}$. When processes spin T^R_{Sched}, 97% of reads complete successfully. Since the participating processes only incur the cost of a context-switch on unsuccessful operations, performance improves to 1.2 times slower than gang scheduling. When processes spin the additional amount $T^R_{Trigger}$, the read success rate increases to 99.9%; thus, context-switches are paid for very few operations and performance improves to 1.02 slower than ideal gang scheduling. Spinning for longer than $T^R_{Trigger}$ does not further increase the read success rate and increases the spin-cost of the unsuccessful operations; therefore, it degrades performance slightly.

The second graph in Figure 7 illustrates the performance of the same workload as the barrier baseline spin is varied. Once again, two-phase waiting of T^B_{Sched} improves performance significantly compared to blocking immediately. However, we note that waiting for $T^B_{Trigger}$ does not result in any benefit beyond waiting T^B_{Sched}: when processes spin T^B_{Sched}, 98% of the barriers complete successfully; as processes spin longer, the percentage of successful operations does not increase further, due to the more strict completion requirements of the barrier. As processes spin longer than $T^B_{Trigger}$, performance degrades more rapidly than occurred for reads, due to the lower success rate of barriers.

When we have examined bulk-synchronous applications with different communication patterns and systems with other context-switch costs and network latencies, the trends have been identical: performance is always similar to that of gang scheduling when processes spin the baseline amount at communication and synchronization operations. However, continuously-communicating applications behave much differently, as shown in Figure 8. In these continuous-communication workloads, the time between barriers is set to ($C = 100ms$), there is no load-imbalance, and the time between reads from random destinations is varied between $c = 10\mu s$ and $c = 2ms$. We change the baseline time of reads and barriers simultaneously since, given the small number of barriers, the baseline spin at barriers has negligible impact on performance.

The graph shows that, regardless of the amount of baseline spin, the completion time of fine-grain continuous-communication workloads remains nearly two times slower than with gang scheduling. The fact that performance improves when processes spin $T_{Trigger}$, but not T_{Sched}, indicates that processes are not coordinated until a message arrival triggers the scheduling of a process. However, since there are few

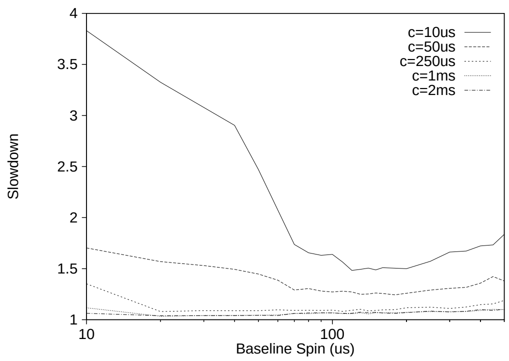

Figure 8: **Sensitivity to Baseline Spin for Continuous-Communication Programs.** *Each line in the graphs designates a different communication granularity, c. The metric along the y-axis is the slowdown of the workload with implicit coscheduling versus gang scheduling. Along the x-axis, the baseline spin amounts for both reads and barriers are changed simultaneously.*

barriers in the applications to initiate global coordination, simply maintaining coordination with destinations is not sufficient. An additional technique to build coordination through pairwise communication is required.

5.2 Conditional Spin: Building Coordination with Senders

If a process is unable to make forward progress because it is waiting on an unscheduled destination process, then it should usually be descheduled; that is, it should block after spin-waiting the baseline amount. However, if the process is handling many incoming messages, then it should continue to spin-wait; since an incoming message indicates that the sending process is scheduled, keeping the destination process scheduled can be globally beneficial. By maintaining coordination with senders in addition to destinations, an application with few coordinating barriers can build global coordination through pairwise interactions.

5.2.1 Analysis

Coordinating a destination process with sending processes requires scheduling the destination on message arrival. With two-phase waiting and a preemptive scheduler, a blocked process may be scheduled on the message arrival, as desired; however, even when the scheduler dispatches the destination process, the cost of blocking and waking may incur a larger cost than simply remaining scheduled, depending on the incoming message rate.

In this section, we examine the interval, T_{Cond}, in which a process must receive a message to warrant extra spinning. To determine when conditional spinning should be activated, we compare the cost when the waiting process blocks versus when it spin-waits for a single incoming message. To be precise, the *local* process has already waited the baseline amount in its communication with a destination process that is not scheduled; the *remote* process is reading from the local process.

To simplify the discussion, we consider only the costs incurred by the pair of processes involved in a read. Ideally, we would examine the costs propagated across the entire parallel job; for example, if the remote process blocks because the local process is blocked, then cooperating processes that communicate with the remote process also incur a higher cost. Since this global cost depends upon communication patterns and the current coordination of the job, determining this cost is an open problem. However, our results show that in most cases, it is sufficient to examine only the pair of involved processes.

We begin with the case where the local process spin-waits on its own communication operation and remains scheduled, as at the top of Figure 9. While the local process spins, a request arrives every T_{Cond}

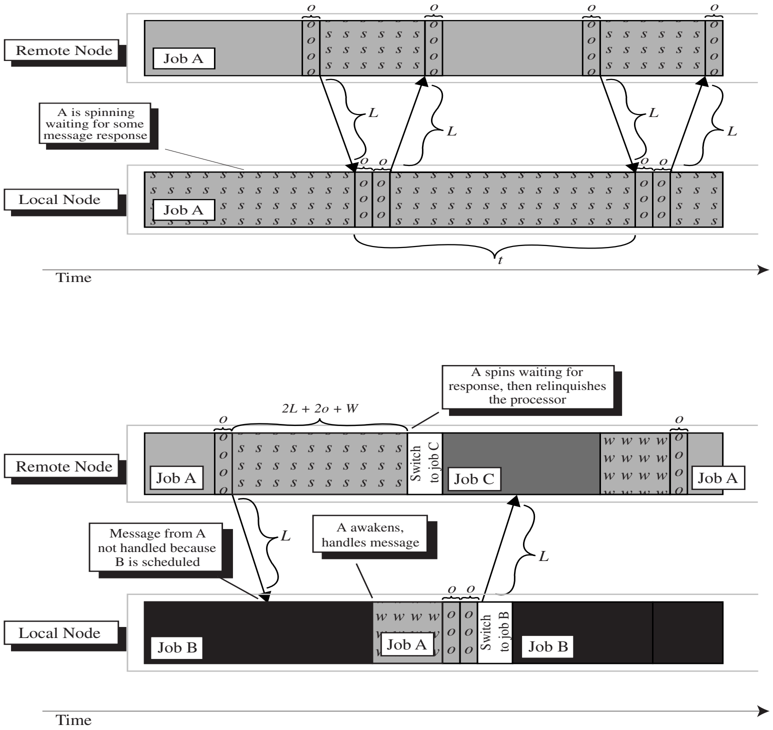

Figure 9: **Cost of an Incoming Message.** *To help calculate T_{Cond}, this figure illustrates the cost to the local (receiving) process and the remote (sending) process depending upon whether the local process spin-waits or blocks between message arrivals.*

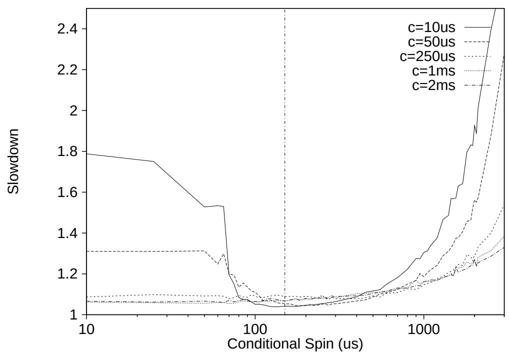

Figure 10: **Sensitivity to Conditional Spin for Continuous-Communication Programs.** *We evaluate the same workloads as shown in Figure 8. The metric along the y-axis is the slowdown of the workload with implicit coscheduling versus gang scheduling. Along the x-axis, the baseline spin amounts for both reads and barriers are changed simultaneously. The vertical line marks $3W$, our derived value of S_{Cond}.*

units; thus, the local process spins idly for time T_{Cond} to handle every 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 when the local process spins is thus $2L + 4o + T_{Cond}$.

In the second case, shown at the bottom of Figure 9, the local process blocks before the request arrives. The remote process spends o sending, and then unsuccessfully spins $S^R_{Base} = 2L + 2o + W$ 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. Finally, the original sender pays another W to be scheduled and o to handle the reply. Thus, the total cost to the system when the local process blocks is $2L + 6o + 3W$.

Assuming that message arrivals are evenly spaced in time, a local process should continue spinning rather than block when the interval between messages is less than or equal to T_{Cond}. [3]

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

This analysis assumes that the local process can predict whether a message will arrive in an interval T_{Cond} to determine if it should spin or block. Because this is not possible in practice, future arrival rates are instead predicted from behavior in the recent past. Since message arrival rates may be bursty (or even decline as more processes reach a barrier and no longer communicate), the arrival rate over a relatively short interval is used. For example, 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.

5.2.2 Simulations

The simulations in this section illustrate the benefit of conditional spinning for continuously-communicating parallel jobs. Figure 10 shows the performance of the same continuously-communicating workload as shown in Figure 8, but with conditional spinning. In these simulations, processes always wait S_{Base} at reads and

[3]This comparison assumes that paying a cost in one process is equivalent to paying that cost in another. For this to be true, neither process must form the critical path of the job. For example, if the sending process determines the performance of the job, then the local process should always spin to minimize the waiting time on the sender. However, if the local process is the critical component, then only its costs should be examined. Due to the difficulty of determining which process forms the critical path and because we believe this inaccuracy is small for our workloads, we ignore this adjustment. However, for client-server applications where the performance of the server is likely to be the critical factor, these adjustments need to be taken into account.

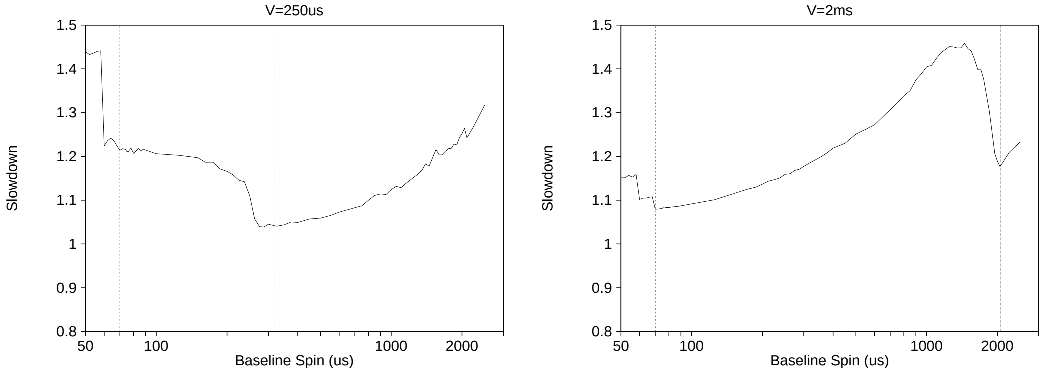

Figure 11: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance.** *The metric along the y-axis is the slowdown of the workload with implicit coscheduling versus gang scheduling. Along the x-axis, the baseline spin amounts for reads and barriers are changed simultaneously. In the graph on the left, load-imbalance is set to* $V = 250\mu s$*; in the graph on the right, it is set to* $V = 2ms$*. The vertical lines in each graph mark* S^B_{Base} *and* $S^B_{Base} + V$*, which correspond to two local minima.*

barriers and conditionally spin as long as the interval between incoming messages is less than the amount on the x-axis. Since the performance of bulk-synchronous workloads is nearly ideal with only baseline spinning, performance is not noticeably affected by conditional spinning and is not shown.

In the figure, as conditional spin time increases along the x-axis, messages can arrive less frequently and the waiting process will continue to spin. The graph illustrates that there is a region of conditional spin-time around our derived value of $S_{Cond} = 3W + 2o$ where performance is relatively flat. With conditional spinning set appropriately, the performance of this difficult workload is only 1.1 times slower than with gang scheduling. When the conditional spin time is set either much too large or much too small, performance degrades. On one extreme, when the conditional spin-time is very small (*e.g.*, less than S_{Base}), messages do not arrive at a fast enough rate to activate conditional spinning; in this region, performance is nearly identical to that with only baseline spinning. On the other extreme, when the conditional spin-time is very large (*e.g.*, greater than $6W$), processes continue spinning even when receiving messages infrequently; therefore, performance is again poor because processes pay the cost of spinning even when coordinated with few senders.

Across all workloads and system parameters we have examined, the slowdown with S_{Cond} relative to ideal gang scheduling is always below 1.35. In summary, conditional spinning greatly improves the performance of implicit coscheduling for continuously-communicating programs, without harming that for bulk-synchronous applications.

5.3 Adjustments for Load-Imbalance

The preceding analysis assumed that the load across all processes within a parallel application is perfectly balanced: that is, if all processes are coordinated, then the time for a barrier to complete is simply S^B_{Base}. However, to stay coordinated at a barrier with load-imbalance v_p, process p must spin $S^B_{Base} + v_p$. This raises two new issues. First, given the amount of load-imbalance and the communication behavior of the application, the process must determine if staying coordinated is actually beneficial. Second, the process must predict the amount of load-imbalance. Both of these issues are currently unresolved and more difficult than indicated by previous work [20].

As noted previously, although spinning for the completion time of the barrier keeps the processes coordinated, the penalty of spin-waiting for a barrier with a large amount of load-imbalance may be higher

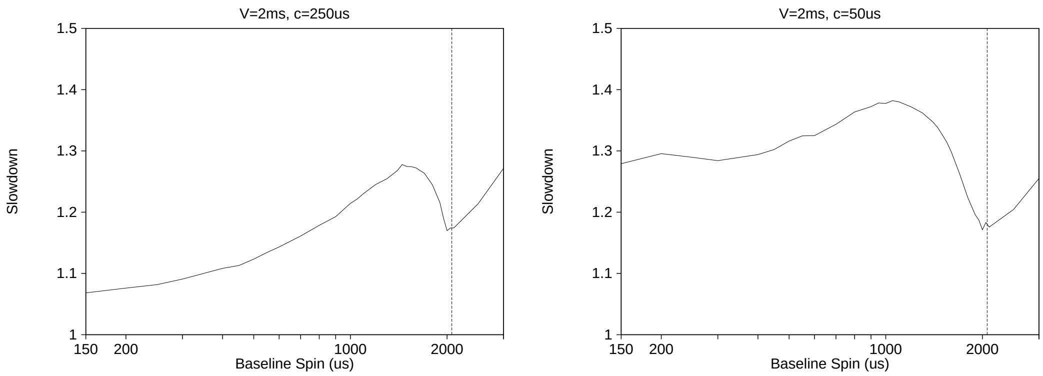

Figure 12: **Sensitivity to Baseline Spin for Continuous-Communication Programs with Load-Imbalance.** *The metric along the y-axis is the slowdown of the workload with implicit versus gang scheduling. Along the x-axis, the baseline spin amounts for reads and barriers are changed simultaneously. In both graphs, load-imbalance is set to $V = 2ms$. In the graph on the left, processes communicate relatively infrequently ($c = 250\mu s$); on the right, processes communicate more frequently ($c = 50\mu s$). The vertical line in each graph marks $S^B_{Base} + V$, which corresponds to a local minimum.*

than the penalty of blocking and losing coordination; at some point, the useful work of a competing process should be overlapped with the waiting time. For example, Figure 11 shows the performance of two bulk-synchronous workloads with different amounts of load-imbalance: $V = 250\mu s$ and $V = 2ms$. Along the x-axis, the amount of baseline spin is increased. On each graph, a vertical line indicates two critical spin times: the minimum baseline spin time, S^B_{Base}, and the baseline spin time accounting for load-imbalance, $S^B_{Base} + V$. Each of those two points corresponds to a local minimum; which point is the global minimum depends upon the amount of load-imbalance. In applications with a small amount of load-imbalance (*e.g.*, $V = 250\mu s$), processes obtain better performance when they spin-wait $S^B_{Base} + V$ and remain coordinated; in applications with a large amount (*e.g.*, $V = 2ms$), processes obtain better performance when they only spin-wait S^B_{Base} and then allow the system to schedule competing processes.

The previous work suggested that the amount of load-imbalance for which processes should block rather than spin-wait was simply a function of the context-switch cost, W. This analysis focused only on bulk-synchronous applications, assuming that the penalty of losing coordination was limited to the cost of the barrier itself. However, in continuous-communication programs with a significant amount of load-imbalance, other communication operations still benefit from coordination. Figure 12 shows that for two workloads with the same amount of load-imbalance ($V = 2ms$), the workload with communication every $c = 50\mu s$ benefits from coordinated scheduling, while that with communication every $c = 250\mu s$ does not. In fact, for fine-grain continuous-communication benchmarks, processes should remain coordinated regardless of the amount of load-imbalance. In summary, to determine whether to spin-wait for the load-imbalance, processes must take into account communication that is currently occurring and communication that occurs after the barrier.

The next issue is that each process must predict the amount of load-imbalance at a given barrier. The previous work noted that when processes are coordinated, load-imbalance can be directly observed by the root process of the barrier, which then distributes its observations in the completion message to the participating processes. The crux of the problem is that processes must be coordinated for the observed times to reflect the inherent load-imbalance; however, to become coordinated, processes must know the inherent load-imbalance to spin-wait appropriately. To bootstrap the system, we originally proposed that the root process should initialize its prediction to the maximum load-imbalance for which it is beneficial to spin-wait; this amount could be bounded by assuming that the maximum beneficial load-imbalance is a function of only the context-switch cost, and not of on-going communication. Thus, when the application begins, processes spin-wait long enough to remain coordinated and the root can discover the inherent load-imbalance. However, this proposal is not adequate when the amount of load-imbalance for which it is beneficial to spin-wait is a function of the communication pattern and therefore unbounded.

Due to these two complications, processes cannot always predict the inherent load-imbalance of the application, and even when they can, they may not know if it is beneficial to spin-wait for this amount. As a result, our current implementation simply spin-waits S_{Base} at barriers. While this may not result in the best possible performance, this spin-time does correspond to a local minimum for slowdown; in the previous simulations, spinning S_{Base} resulted in slowdowns that are still within 30% of gang scheduling. Our implementation measurements in Section 7 confirm that we achieve respectable performance for most real workloads.

6 Local Scheduler

Given that we have a conditional two-phase waiting algorithm that maintains scheduling coordination, we now evaluate the other component of implicit coscheduling: the operating system scheduler. We focus on the ability of two types of operating system schedulers to support fairness in implicit coscheduling. The first type of scheduler, based on *multilevel feedback queues*, is often used in practice. The second type, a *proportional-share scheduler*, is more popular in the research community than in practice; it is for such a scheduler that we describe and implement simple extensions that meet the needs of implicit coscheduling.

6.1 Solaris Time-Sharing Scheduler

The Time-Sharing (TS) scheduler in Solaris [22], based on Unix System V Release 4 [29], is an example of a *multilevel feedback queue scheduler*. Its objective is to fairly and efficiently schedule a mix of processes with a variety of execution characteristics. By controlling how a process moves between priorities, the scheduler can treat processes with different characteristics appropriately.

Processes waiting at higher priorities are always scheduled over those at lower priorities; processes at the same priority are scheduled in a round-robin fashion. To separate processes based on their characteristics, the priority of a process is lowered after it consumes its time-slice; its priority is raised if it is sleeping or has not consumed its time-slice when a *starvation interval* expires. Thus, compute-bound jobs filter down to the lower priorities, and interactive jobs propagate to the higher priorities. The durations of the time-slices, the changes in priorities, and the starvation interval are specified in a tunable dispatch table.

6.1.1 Selective Preemption

The Solaris scheduler preempts a running process if a higher-priority process becomes runnable. Thus, if a higher priority process wakes due to the arrival of a message, it will be promptly scheduled. This property helps trigger coordinated scheduling, as desired.

6.1.2 Fair Cost-Model

In general, multi-level feedback queue schedulers do not export a precisely fair cost-model to user-level processes [30, 35, 41]. Thus, a process may receive more or less of the CPU over a time interval depending upon how frequently the process blocks. The Solaris time-sharing scheduler approximates fair allocations by decreasing the priority of a job the more that it is scheduled. However, due to the specific configuration of the default dispatch table (*i.e.*, the starvation interval is set to zero), the priority of every process is raised once a second, regardless of its computation characteristics. Because the allocation history of each process

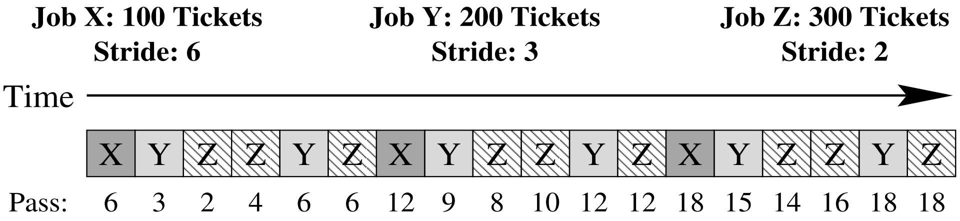

Figure 13: **Basic Stride Scheduling.** *Three jobs (X, Y, and Z) with different ticket allocations are running on a single workstation. At each time-slice, the job with the minimum* `pass` *is scheduled and its* `pass` *is incremented by its* `stride`.

is erased every second, compute-bound processes tend to acquire more than their fair share of the resources. The implication for implicit coscheduling is that processes that communicate frequently (and thus block frequently) are given less of the CPU than processes that rarely communicate.

6.1.3 Amortized Context-Switches

In the default dispatch table, the length of a time-slice varies from $20ms$ at the high priorities up to $200ms$ at the low priorities. Therefore, the more time a process spends at high priorities, the shorter its time-slice and the more time it wastes achieving coordination. However, due to the starvation interval of zero, with more jobs in the system, processes execute at high priorities for a greater fraction of their execution time and receive shorter time-slices, potentially harming the performance of fine-grain parallel applications.

6.2 Stride Scheduler with System Credit

The idea behind a *proportional-share scheduler* is to allocate resources to processes in proportion to their relative number of shares; we focus on *stride scheduling* [51]. We describe the basic system as introduced by Waldspurger and as present our *system credit* extension to provide a fair cost model to processes.

Stride scheduling is a deterministic algorithm that allocates resources to competing processes in proportion to the number of *tickets* they hold. For example, if a process has t tickets in a system with T total tickets, the process receives t/T of the resources. To perform these allocations, each process has a time interval, or `stride`, inversely proportional to its ticket allocation that determines how frequently it is scheduled. For example, a process with twice the tickets of another, has half the `stride` and is allocated twice as frequently. As shown in Figure 13, a `pass` associated with each process is incremented by its `stride` each time it is scheduled for a fixed time-slice; the process with the minimum `pass` is selected each time-slice. A `global stride` and `global pass` for the workstation are also tracked; when a process enters the system (either after being first started or after waking from an event), its `pass` is set equal to the `global pass`.

6.2.1 Selective Preemption

Promptly scheduling a process when it wakes from an event is simple within the stride scheduling framework. The new process is scheduled if its `pass` is lower than the `pass` of the currently scheduled process. Care must be taken to ensure that the preempted process is only charged for the amount of the time-slice that it was able to consume.

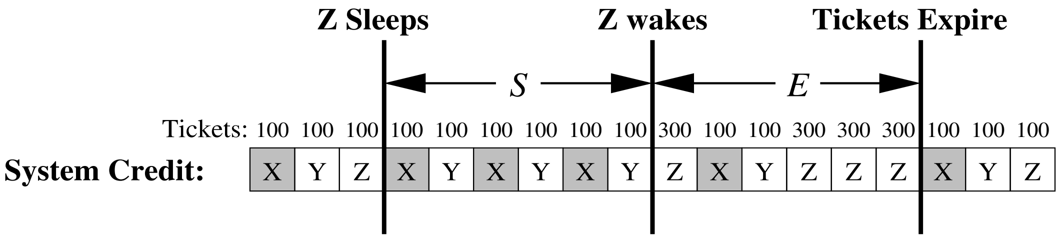

Figure 14: **Stride-Scheduling with System Credit Extension.** *Three jobs with equal ticket allocations are competing for resources. Jobs X and Y are compute-intensive and do not relinquish the processor; job Z is an interactive job. Job Z temporarily exits the system, and sleeps for an interval S; in the time-interval E, job Z catches up for the time it missed. All jobs receive their proportional-share of 4 allocations over the* $S + E$ *interval of 12 time-units*

6.2.2 Fair Cost-Model

While proportional-share schedulers provide a precise cost-model, they do not provide a fair model guaranteeing that each process receives its share regardless of its computational characteristics. The definition of proportional-share scheduling states that only clients *actively* competing for resources receive resources in proportion to their shares. When a process sleeps on an event, it no longer competes for resources and receives no portion of the processor. When the process wakes, allocations are performed as if the process never went to sleep and the process is given no additional resources in compensation for the time that it was sleeping. Thus, with a proportional-share scheduler, processes are given no incentive to relinquish the processor.

We describe an extension to ensure that jobs that voluntarily relinquish the processor receive their proportional-share of resources over an extended time interval. The *system credit* policy (SSC) gives *exhaustible tickets* [52] (*i.e.*, tickets with an expiration time) to compensate a process for the time it voluntarily did not receive any resources. The number of exhaustible tickets and the expiration time are selected by the scheduler such that the process has received its proportional-share of resources by the time the exhaustible tickets expire.

Figure 14 illustrates this behavior for a workload with three competing jobs, X, Y, and Z, each with an equal number of tickets, $t = 100$. Initially, each job is active and therefore receives 1/3 of the CPU. When job Z sleeps for an interval S, jobs X and Y are temporarily given $t/(T-t) = 1/2$ of the resources instead of $t/T = 1/3$. When job Z awakens, the scheduler picks some number of exhaustible tickets, e, and an expiration interval, E, such that over the interval $S + E$, job Z receives its desired proportion of resources, t/T. Obviously, it is also required that jobs X and Y receive their proportion of resources over this same $S + E$ interval.

The implementation is as follows. When a process sleeps, the current time is recorded and the total number of tickets in the system is updated. When the process awakens, the time that it was asleep, S, is calculated. For the job to receive $t/T \cdot (S+E)$ units of service over the interval E, the job must have $(t+e)$ of the $(T+e)$ tickets in the system, where the number of exhaustible tickets e is as follows.

$$\frac{(S+E)\cdot\frac{t}{T}}{E} = \frac{t+e}{T+e}$$

$$e = \frac{tTS}{ET-(S+E)t}, \text{ given } E > \frac{St}{T-t}$$

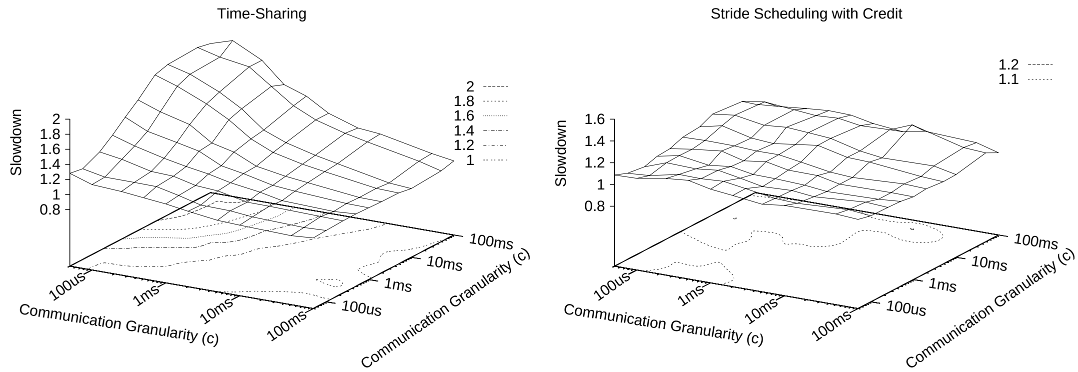

Figure 15: **Fairness.** *Along the x-axis, the time between reads for the job under evaluation is varied between 50μs and 100ms. Along the y-axis, the time between reads is varied for two continuously running competing jobs. The metric along the z-axis is the slowdown of the evaluated job with implicit coscheduling versus gang scheduling.*

The scheduler must also choose a expiration interval, E, over which to allocate the e exhaustible tickets. The current system selects E so as to simplify the calculation for the number of exhaustible tickets. There are three possible cases, depending upon the relative number of client tickets, t, and system tickets T. In the simplest case, when $t < \frac{1}{2}T$, the system chooses $E = S$ and the number of exhaustible tickets as follows.

$$e = \frac{tT}{T - 2t}, \text{ if } E = S$$

When $T > t \geq \frac{1}{2}T$, the system selects E and e as follows.

$$\begin{aligned} E &= \frac{S(T+t)}{T-t}, \; t \neq T \\ e &= t \end{aligned}$$

Finally, when $t = T$, the sleeping process is the only process in the system, in which case it cannot receive its full share of resources and is not given any exhaustible tickets.

Two additional complications exist in the implementation. First, if a process relinquishes the CPU while it still has exhaustible tickets, the number of exhaustible tickets and their remaining lifetime must be recorded. When the process later receives more exhaustible tickets, the two sets must be appropriately combined. Second, when a process wakes, the scheduler calculates its exhaustible tickets independently of any exhaustible tickets given later to competing processes; since the system does not account for this possible future increase in T, a process may not be fully compensated. Thus, the system credit extension performs most accurately when few processes are simultaneously leaving and joining the system. See [5] for a description of a more involved extension that removes this limitation.

6.2.3 Amortized Context-Switches

Due to the constant-length time-slices in stride-scheduling, a time-slice duration, Q, can be picked to amortize the costs of obtaining coordination and context-switching. In our simulations and implementation, we use a time-slice of $Q = 100ms$, matching the average time-slice of many systems. Once again, due to selective preemption, interactive processes do not have to wait the length of a time-slice before they can be scheduled.

6.3 Simulations

In the next experiments, we show the impact of the cost-model of the local scheduler on jobs that communicate at different granularities. Our previous experiments measured the completion time of a fixed workload; however, this metric does not illustrate the relative completion times of the jobs in the workload. Therefore, in the experiments that follow, we examine the completion time of a single job executing in competition with two continuously-running background jobs and independently vary the communication granularity of the examined job and the background jobs.

We calculate the slowdown of the single job relative to that experienced with ideal gang scheduling. We call the slowdown that a job experiences against jobs with identical communication characteristics its *base slowdown*. If the scheduler tends to favor the measured job within a mixed workload, its slowdown in that workload is less than its base slowdown; if the scheduler is biased against this measured job, its slowdown is greater than its base slowdown. The local scheduler provides *fair* performance if it exhibits slowdown near to the base slowdown; that is, slowdown is flat as the characteristics of the competing applications are changed.

To stress the system, we consider applications that continuously communicate. Each process synchronizes every $g = 100ms$, is perfectly load-balanced, and communicates in a `Random` pattern. Across the x-axis of each graph, the communication interval, c, of the evaluated job is increased from $50\mu s$ to $100ms$, while across the y-axis the communication interval of the background jobs is increased. The points along the diagonal correspond to workloads where all jobs have identical communication characteristics. In these experiments, we examine a high context-switch cost of $W = 200\mu s$; therefore, the slowdowns are higher than those observed in our implementation.

The first graph in Figure 15 shows that the fairness of the TS scheduler degrades slowly as a function of the communication granularity of the competing jobs. For example, a fine-grain job communicating at an interval of $c = 50us$ exhibits a slowdown of only 1.3 when competing against background jobs with similar characteristics; however, the same job exhibits a slowdown of more than 2 when competing against more coarse-grain background jobs (*e.g.*, jobs that communicate every $c = 100ms$).

The second graph in Figure 15 shows the performance of the stride scheduler with system credit. We make two observations from the graph. First, the extended scheduler improves the base slowdown of the workloads; for example, with SSC, most base slowdowns are near 1.1, while with TS, they are nearer 1.3. The improvement with the stride scheduler is primarily due to its longer time-slice, on average. Second, SSC greatly improves fairness across jobs with different communication granularities: no job exhibits a slowdown greater than 1.2, even when competing against much more coarse-grain jobs. Thus, the stride scheduler with system credit provides an accurate and fair cost-model to communicating processes.

In summary, the Solaris Time-Sharing scheduler is sufficient for handling jobs that communicate at the same rates, due to its selective preemption policy. However, with this scheduler, fine-grain jobs competing against more coarse-grain jobs do not receive their fair share of resources, since fine-grain jobs relinquish the CPU more frequently. It is therefore the responsibility of the local scheduler to adjust for these differences in computational demands. We have shown that our stride scheduler with system credit adequately compensates jobs for their different usage patterns and results in fair allocations across the cluster.

7 Implementation

We have implemented implicit coscheduling as part of the U.C. Berkeley NOW cluster. In this section, we give a brief overview of the cluster, the requirements of the communication layer, and the changes to the parallel runtime layer to support conditional two-phase waiting. We also demonstrate that our implementation performs near to that of an ideal model of gang scheduling for a wide range of synthetic workloads and Split-C kernels; furthermore, our implementation can fairly handle jobs that communicate at different rates and is robust to increasing the number of jobs and workstations and to changing layout of processes. Our overall experience was that implementing implicit coscheduling was a straight-forward task, in part due to our confidence in our simulation results.

7.1 Cluster Architecture and Communication Layer

The U.C. Berkeley NOW cluster [14] contains 105 Ultra 1 Model 170 workstations. Each machine contains a single 167 MHz UltraSPARC processor and 128 MB of main memory. The measurements in this paper examine performance on 16 workstations. The workstations are connected with Myrinet, a switch-based, high-speed, local-area network [9].

Processes communicate with AM-II [38], an extension of the Active Message paradigm [50] that handles multiple communicating processes, client-server applications, and system services. The Active Message model is essentially a simplified remote procedure call. A fundamental difference in our implementation compared to our model is the manner in which processes are notified of a message arrival. In our simulations, a process receives an asynchronous interrupt whenever a message arrives; the message is then immediately handled if the process is scheduled. In our implementation, the process is notified of a message arrival only when it touches the network: either by sending a message or by polling with `AM_Poll`. Thus, to a process waiting for a response, a destination process that is ignoring the network appears as if it is not scheduled. While this behavior does not match our previous assumptions, we have found that it does not adversely affect the performance of implicit coscheduling.

To support conditional two-phase waiting, the message layer must meet three requirements.

1. A waiting process must be able to voluntarily relinquish the processor until a message arrives. AM-II provides this functionality with the `AM_SetAndWait` interface.

2. A waiting process must be able to determine the number of arriving messages in a given interval. The initial version of AM-II did not contain such a mechanism; AM-II was subsequently modified to return the number of messages handled in each call to `AM_Poll`.

3. The message layer must never spin-wait on a remote event. The initial version of AM-II spin-waits when sending a message if there are insufficient flow-control credits or if the next slot in an outgoing FIFO queue is full. AM-II now provides a non-blocking interface to return control to the calling process in these circumstances. However, this modification was not made early enough to be incorporated into our implementation of implicit coscheduling; instead, as described below, our parallel language run-time layer circumvents these situations.

7.2 Operating System Scheduler

Each workstation runs a copy of Solaris 2.6 [22], a modern, multi-threaded operating system based on SVR4 [29]. GLUnix is used to start the processes across the nodes of the cluster [28]. Unless otherwise stated, we use the Solaris Time-Sharing scheduler for our measurements of implicit coscheduling.

Rather than implement a version of gang scheduling to serve as a comparison point, we model the ideal gang scheduling performance of a workload by simply adding together the execution time of each application when run in a dedicated environment with spin-waiting. This approach is ideal for gang scheduling because it does not suffer from any cache effects or additional overhead for global context-switching.

We implemented the stride scheduler with system credit (SSC) as a new scheduling class in Solaris; the drawback of this approach is that the required functionality of the SSC class does not precisely match the scheduling interface within Solaris. To ensure that the Solaris dispatcher selects the process with the lowest `pass` value, the SSC module sets the priority of this process lower than the priority of any other process. The natural location to perform this adjustment is in the clock-tick routine that executes every $10ms$; however, a lock must be acquired before modifying a priority and locks for processes other than the currently scheduled one cannot be acquired in the clock-tick routine. Therefore, our implementation adjusts priorities when a new $100ms$ timer expires; this timer runs at a lower system-level priority and can freely acquire and release locks. In the timer routine, the SSC module sets the priority of every process in reverse rank according to its `pass` value. The implication is that the dispatcher does not necessarily pick the process with the lowest `pass` value at every instance. Instead, within this $100ms$ interval, the highest-priority (*i.e.*, lowest-pass) process is always scheduled when it is runnable. If a process voluntarily relinquishes the processor, the process with the next highest priority is automatically dispatched. When the highest-priority process reawakens, it is given exhaustible tickets by the system, and automatically preempts the lower-priority process. We have not found this change to adversely impact the fairness of our workloads.

7.3 Application Workload

For our parallel language, we use Split-C [15], a parallel extension to C with operations for accessing remote memory built on AM-II. By design, Split-C closely matches our model: `read` and `write` operations which access remote memory are built on request-response messages while `barrier` operations perform all-to-all synchronization. In addition to these basic operations, Split-C contains split-phase assignments (*i.e.*, the local process does not wait for the response until a later synchronization statement, `sync`), one-way messages (*i.e.*, `store` operations that do not require a response), synchronization operations for broadcasts, reductions, and scans, and bulk transfer operations for each communication style; the conditional two-phase waiting in implicit coscheduling supports all of these operations.

Previous implementations of the run-time library in Split-C assumed that parallel jobs run in either a dedicated environment or under gang scheduling, and, therefore, spin-wait at communication and synchronization events. Consequently, if Split-C users run multiple applications simultaneously with local scheduling, each user sees performance orders of magnitude worse than in a dedicated environment [4]. Implementing implicit coscheduling within the Split-C run-time library required three simple modifications: ensuring that AM-II does not spin-wait, altering the operations that wait for replies (*i.e.*, `syncs`, `store_syncs` and `barriers`) to use conditional two-phase waiting, and providing this waiting algorithm to user applications.

First, to ensure that AM-II does not spin-wait, the Split-C run-time layer does not send messages if the next slot in the outgoing AM-II FIFO queue may not be available. To implement this, Split-C tracks the number of outstanding messages, and, after sending the number that fit in the underlying queue (16), waits for all messages to be acknowledged before sending more. This modification to the run-time layer is transparent to the application. Unfortunately, this slows down a few applications by artificially limiting the number of outstanding messages. As mentioned above, the correct approach would be to use the new non-blocking interfaces of AM-II.

Second, the conditional two-phase waiting algorithm was added to `sync`, `store_sync`, and `barrier` operations; with the exception of the amount of baseline spin, the implementations are identical. The Split-C library begins by polling the AM-II layer a fixed number of times as designated by S_{Base}, checking if the desired condition is satisfied and recording the number of incoming messages. If the condition is satisfied, the Split-C library returns control to the user application. Otherwise, if a sufficient number of requests arrive, Split-C continues to spin for intervals of length S_{Cond} until no messages are received in an interval. At this point, the process calls `AM_SetAndWait` to put the waiting process to sleep until a message targeted for that process arrives. When the process is woken and scheduled again, 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 spinning.

To more accurately configure the baseline spin-time at `syncs` and at `barriers`, we run a small set of

Variable	Description	Value
L	network latency	$9.8\mu s$
o	overhead	$3.6\mu s$
g	gap	$13.7\mu s$
P	processors	16
W	context-switch	$70\mu s$
T^R_{Sched}	request-response time (remote process scheduled)	$60\mu s$
$T^R_{Trigger}$	request-response time (remote process triggered)	$130\mu s$
S^R_{Base}	read baseline spin	$130\mu s$
T^B_{Sched}	barrier time (remote processes scheduled)	$450\mu s$
$T^B_{Trigger}$	barrier time (remote processes triggered)	$300\mu s$
S^B_{Base}	barrier baseline spin	$450\mu s$
S^R_{Cond}	request-response conditional spin	$210\mu s$

Table 1: **Parameters of Implementation.** *The table summarizes the measured LogP and local scheduler parameters for our system as well as the derived spin times.*

Program	Description	Problem Size (per P)
`mm`	Matrix multiply	256 x 256
`radix`	Radix sort	2M keys
`radix:small`	Radix sort	128K keys
`fft`	Fast Fourier transform	512K points
`fft:small`	Fast Fourier transform	256K points
`em3d`	Electro-magnetic wave propagation	5000 nodes, 20 steps degree: 20, 40% remote distribution 3
`em3d:small`	Electro-magnetic wave propagation	5000 nodes, 10 steps degree: 20, 40% remote distribution 3

Table 2: **Communication Characteristics of Benchmark Applications.** *Problem sizes are chosen so that the applications require between 10 and 60 seconds when run alone and three copies fit in 128 MB of memory.*

benchmarks. These microbenchmarks allow us to include the computation in the Active Message handlers and to observe the full distribution of completion times. The results from running each benchmark in a dedicated environment are then fed directly into the Split-C library. The conditional spin amounts are still determined by the models presented in Section 5.2. The system parameters and spin-times are summarized in Table 1.

Finally, user-level code that must wait for a remote event is supported with a new Split-C function that performs conditional two-phase waiting. Finding these spin statements in the user-level code is simple because of the polling model used in existing Split-C implementations: tight loops containing polls are simply replaced with the new interface.

To evaluate our implementation, we examine both synthetic and kernel applications. The synthetic applications closely match the benchmarks employed in the simulations; this enables us not only to evaluate our implementation in a controlled fashion, but also to compare our results to those predicted by the simulations. The primary difference from the simulation benchmarks is that we run on 16 workstations instead of 32 and increase the number of iterations within the application so that each requires between 20 and 30 seconds in a dedicated environment. We repeat each experiment between five and ten times and report the average slowdown.

To evaluate a more varied set of applications, we also measure the performance of seven Split-C programs that exhibit a variety of communication characteristics. The application kernels include matrix multiplication, `mm`, two versions of radix sort, `radix` [1, 21], two fast Fourier transforms, `fft` [1, 16], and two versions of a model of electro-magnetic waves propagated in three dimensions, `em3d` [15]. These applications contain the full range of Split-C operations: not only reads and barriers, but also writes, one-way stores, bulk messages, broadcasts, and reductions. When two versions of an application exist, one copy has

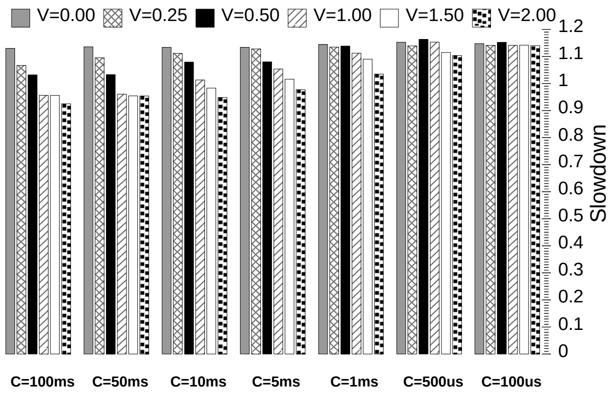

Figure 16: **Performance on Bulk-Synchronous Workloads.** *The graph compares implicit coscheduling to ideal gang scheduling with spin-waiting. The workload consists of three bulk-synchronous jobs on 16 workstations. The time between barriers, C, is varied between 100μs and 100ms across the sets of bars. Within each set of bars, the load-imbalance across processes is varied from $V = 0$ to $V = 2 \cdot C$.*

been optimized to communicate with large messages, while the other uses short messages. The problem sizes, shown in Table 2, were chosen such that three copies of each application fit into 128MB of memory without paging and such that each application executes between 10 and 60 seconds in a dedicated environment. The only modification made to any of the programs was to replace spin-waiting in `radix` sort with calls to the Split-C functions for conditional two-phase waiting.

7.4 Performance with Waiting Algorithm

We now demonstrate that the performance of our implementation of implicit coscheduling is near that of an ideal model of gang scheduling. To show that implicit coscheduling can handle basic workloads, we again consider workloads containing three copies of identical applications, from synthetic bulk-synchronous and continuous-communication jobs to Split-C kernels.

Our first set of experiments show that a wide range of bulk-synchronous applications perform well with implicit coscheduling. The measurements in Figure 16 simultaneously vary the synchronization interval and the load-imbalance of the three competing applications. The results indicate that for all workloads, the slowdown with implicit coscheduling is no greater than 15% worse than ideal gang scheduling. Applications with a large amount of load-imbalance (*e.g.*, $V > 20ms$) exhibit speedups relative to gang scheduling, due to overlapping waiting time and computation. As discussed in Section 5.3, our implementation does not predict the load-imbalance at a barrier; instead, processes only spin-wait the baseline amount, and then lose coordination if unbalanced. However, the excellent performance of these workloads indicates that little additional benefit can be achieved by approximating load-imbalance for bulk-synchronous applications.

Our second set of experiments shows that while most continuous-communication benchmarks perform well, those with both high load-imbalance and high communication rates can be problematic. In Figure 17 we investigate workloads with a wide variety of synchronization and communication granularities, for two different levels of load-imbalance.

For workloads with no load-imbalance, performance is within 20% of gang scheduling. The only workloads that do not perform well are those where both load-imbalance and communication rates are very high. For example, when $V = 150ms$ and processes communicate every $c = 10\mu s$, performance is almost 60% worse than gang scheduling; however, we believe that such workloads will not occur often in practice. To

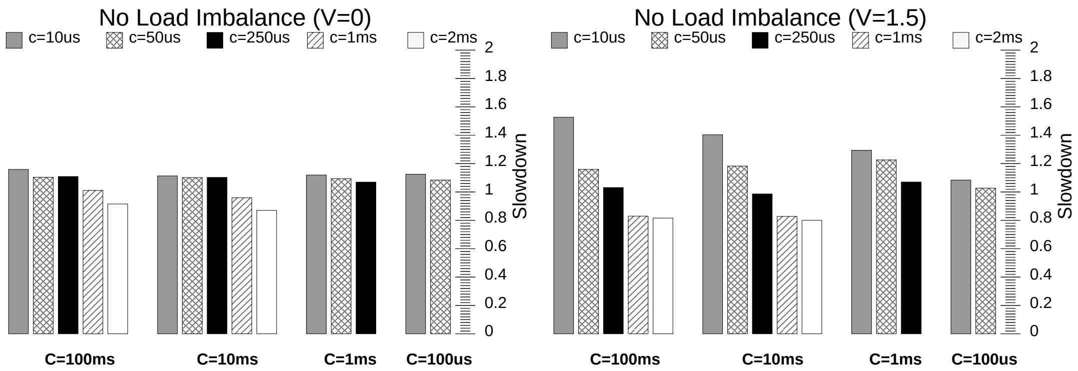

Figure 17: **Performance of Continuous-Communication Programs.** *The time between barriers is varied between $C = 100\mu s$ and $C = 100ms$ and the time between reads is also varied between $c = 10\mu s$ and $c < C$. The applications in the first set of graphs have no load-imbalance ($V = 0$); in the second set of graphs, the load-imbalance is $V = 1.5C$.*

support these extreme applications, processes must predict the amount of load-imbalance and adjust the baseline spin time to match the expected completion time of the barrier. With this approach, processes remain coordinated at the barrier, which allows waiting processes to handle incoming message requests.

Implicit coscheduling is able to achieve speedups relative to gang scheduling in two cases. First, as expected, jobs with significant load-imbalance and infrequent communication (*e.g.*, $c > 1ms$) exhibit speedups; in this regime, waiting time is high enough that more useful work is performed if the process blocks before the barrier completes. Second, applications with no load-imbalance can observe speedups if they have infrequent barriers and infrequent communication. This effect occurs because a process is notified of a message arrival only when it either sends a message or polls the network. As a result, even in a dedicated environment, a sending process may spin-wait a significant amount of time while the destination is computing. Thus, processes can achieve better performance by spinning only the baseline amount and relinquishing the processor if the response does not return promptly.

Our third set of experiments illustrates that our kernel applications perform well with implicit coscheduling, matching the more well-behaved synthetic benchmarks. The performance of the seven Split-C applications with baseline and conditional spinning is shown in Figure 18. For all applications, performance with conditional spinning is within 30% of ideal gang scheduling. A few of the applications exhibit significant speedups relative to gang scheduling due to load-imbalance and bulk messages: `mm` and `fft`. Applications with bulk messages can exhibit speedups for the same reason as applications with load-imbalance: a local process dependent on a bulk message must wait a significant time for it to be transmitted through the network; for large messages, better performance is achieved when the system overlaps useful work with communication rather than when it maintains coordination. Experiments reported elsewhere [6] verify this benefit for synthetic applications containing bulk request-response messages.

The two applications that perform the worst with implicit coscheduling, `radix:small` and `emd3d:small`, rarely synchronize yet communicate frequently, which matches the characteristics of the worst-case continuous-communication synthetic applications. However, these two applications are not representative of real workloads in that they were specifically written to stress communication behavior. As expected for continuous-communication applications, conditional spinning significantly improves performance relative to baseline spinning; with baseline spinning, `em3d:small` runs nearly 80% slower than gang scheduling; with conditional spinning it improves to within 30%. Finally, the performance of `em3d:small` with implicit coscheduling is artificially reduced even in a dedicated environment, due to the restriction of the sending messages in

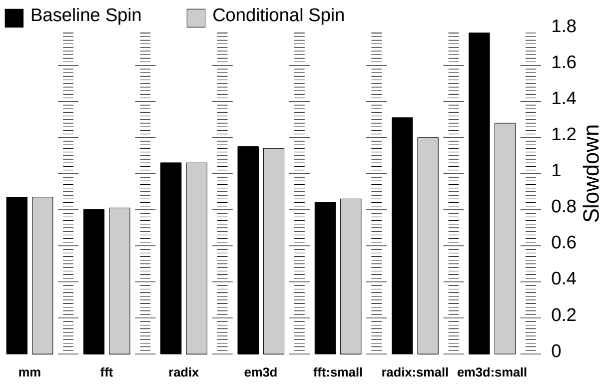

Figure 18: **Performance of Split-C Application Kernels.** *Three copies of each Split-C application are run on 16 workstations. The first four applications use bulk messages, while the last three use small messages.* `mm` *is matrix multiply;* `fft` *is a fast Fourier transform;* `radix` *is a radix sort;* `em3d` *models the propagation of electro-magnetic waves in three dimensions.*

bursts of no more than 16 messages; leveraging the new non-blocking interfaces to AM-II layer is expected to improve this workload.

7.5 Performance with Local Schedulers

To evaluate the fairness of the Solaris Time-Sharing scheduler (TS) and the stride scheduler with system credit (SSC), we reproduce a subset of our previous simulations. Rather than repeat the full combination of workloads, we focus only on a single continuous-communication fine-grain job in competition with two background applications whose communication rates are varied from fine to coarse-grain.

The performance shown in Figure 19 illustrates that the SSC scheduler remains more fair than the TS scheduler for all measured workloads. However, the TS scheduler performs better than predictions in Figure 15; for example, with the TS scheduler, we predicted slowdowns nearly 2 times worse than ideal gang scheduling; however, in our implementation, the worst-case slowdowns are only 1.5. We believe the difference in these results is due to the higher context-switch cost of $W = 200\mu s$ in those simulations. Overall, we believe that the baseline performance and fairness of implicit coscheduling is quite acceptable for a wide range of parallel workloads.

7.6 Sensitivity Performance

In this section, we stress the scalability of implicit coscheduling, increasing both the number of competing jobs and the number of workstations, and test its robustness to different layouts of processes across workstations. For these experiments we evaluate workloads containing jobs with the same communication characteristics and thus return to the Solaris Time-Sharing scheduler.

Our first experiments investigate the performance of implicit coscheduling as the number of competing jobs in the system is increased from one to seven. We stop at seven jobs because this is the number of communicating processes the AM-II layer can simultaneously support without paging communication endpoints to and from the network interface card. Figure 20 shows the performance of implicit coscheduling on four different synthetic workloads. We examine the bulk-synchronous and the continuous-communication workloads in turn.

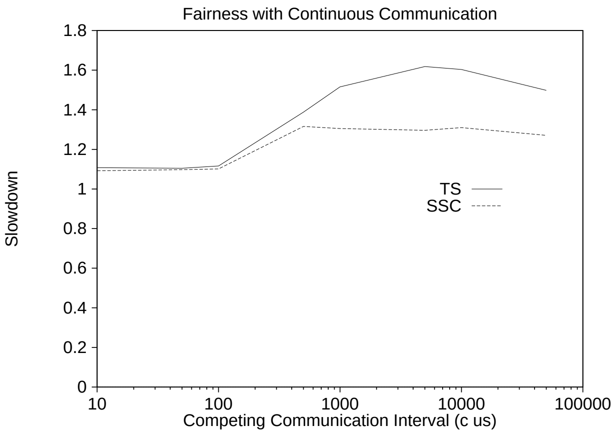

Figure 19: **Fairness with Continuous-Communication Programs.** *One measured job and two competing background jobs are allocated to 16 workstations. Only the slowdown of the one job is measured; the background jobs run continuously during the experiment. We evaluate continuous-communication workloads where barriers are rarely performed. In each workload, $C = 100ms$, there is no load-imbalance, and processes read from remote processes in a* `Random` *pattern. The examined job has the interval between reads set at $c = 50\mu s$, while c is varied for the two background jobs. The top line uses the default Solaris Time-Sharing scheduler on each workstation; the bottom line uses our implementation of the Stride Scheduler with System Credit.*

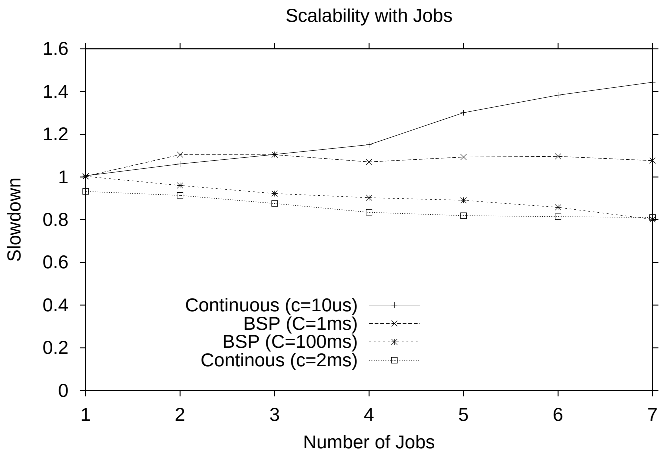

Figure 20: **Job Scalability.** *The number of jobs in the system is increased from one to seven, each running on 16 workstations. The first bulk-synchronous workload represents medium-grain applications ($C = 1ms$) with no load-imbalance ($V = 0$); the second bulk-synchronous workload represents coarse-grain applications ($C = 100ms$) with a large amount of load-imbalance ($V = 200ms$). In the continuous-communication workloads, barriers are rarely performed (*i.e.*, $C = 100ms$), there is no load-imbalance, and processes read from remote processes in the* `Random` *pattern; we examine fine-grain jobs with $c = 50\mu s$ and medium-grain with $c = 2ms$.*

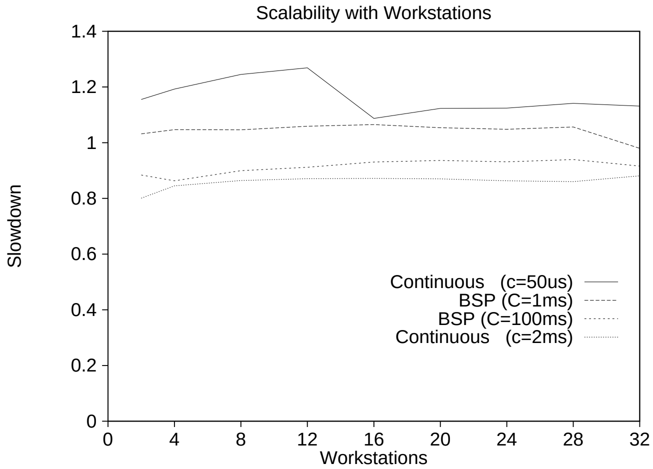

Figure 21: **Workstation Scalability.** *Three competing jobs are allocated to an increasing number of workstations, shown along the x-axis.*

As shown, the medium-grain bulk-synchronous workload ($C = 1ms$) has very stable performance as the number of competing jobs is increased. The performance of the coarse-grain bulk-synchronous workload with high load-imbalance ($C = 100ms$ and $V = 200ms$) actually improves relative to gang scheduling as the number of jobs is increased. For workloads with high load-imbalance, the best performance is achieved when processes relinquish the processor when waiting at barriers and allow a competing process to perform useful work. With only a few processes in the system, a significant amount of idle time exists on each workstation, indicating that the full benefit of relinquishing the processor is not realized; with more competing jobs, the likelihood increases that another process has useful work to perform.

The performance of the fine-grain continuous-communication workload ($c = 50\mu s$) is sensitive to the number of competing jobs. As the number of jobs is increased from three to seven, performance degrades from 10% to 45% worse than gang scheduling. This degradation occurs because with more jobs, the Solaris TS scheduler keeps the jobs at higher priorities which have shorter time-slices (*e.g.*, $Q = 20ms$ versus $Q = 200ms$).[4] We have found that setting the length of all time-slices to $Q = 100ms$ improves the performance of implicit coscheduling to within 30% of gang scheduling for up to seven jobs [6].

The graph also shows that the performance of the medium-grain continuous-communication workload relative to gang scheduling actually improves with more competing jobs. As noted for Figure 17, in workloads containing infrequent barriers and infrequent communication, a sending processes may have to wait a long time for a scheduled destination process to touch the network and thus handle the request; with high waiting times, processes achieve better performance by relinquishing the processor after the baseline spin amount. With three competing jobs, the improvement was small relative to gang scheduling; as the number of jobs is increased, this effect increases in magnitude.

In our next experiments, we show that the performance of implicit coscheduling is relatively constant as the number of workstations is increased from two to 32. We examine the same four workloads as in the previous experiments. Figure 21 shows the performance of these four workloads as a function of the number of workstations in the cluster. The graphs show that in no case does performance degrade as the number of workstations is increased; the slowdown is always within 30% of ideal gang scheduling. Performance is slightly worse for cluster sizes less than 16 workstations for the one fine-grain continuous-communication workload because the shorter execution time of barriers implies that coordinated scheduling is relatively

[4]This occurs because the Solaris TS scheduler raises the priority of every process once a second and lowers its priority only when a process completes a time-slice.

more beneficial. All in all, these results are very promising for larger configurations.

Finally, we show that implicit coscheduling performs well when the load across machines is unbalanced and that performance may even improve relative to gang scheduling. The three workloads that we examine (coarse, medium, and fine) consist entirely of bulk-synchronous applications. Figure 22 shows four parallel job layouts and the corresponding slowdowns of the workloads. The layouts are not necessarily ideal for job throughput, but, instead represent placements that may occur as jobs dynamically enter and exit the system; for example, the jobs in layout b 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 a, b and c verify that implicit coscheduling can handle uneven loads across workstations. As desired for these three 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 fewer jobs are idle with implicit coscheduling for roughly the same time as a gang scheduling implementation. Thus, medium and fine-grain jobs perform similarly to gang scheduling, and the coarse-grain jobs achieve a speedup, as expected.

Layout d shows 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, implicit coscheduling achieves an additional speedup relative to gang scheduling because processes can run at any time, not only in the predefined slots of the scheduling matrix. Thus, the coarse-grain workload requires only 55% of the execution time with implicit coscheduling as with gang scheduling.

8 Discussion

The simulations and implementation measurements presented in this paper have shown that most parallel applications written in Split-C perform similarly with implicit coscheduling as with gang scheduling. In this section, we discuss how implicit coscheduling might behave given different programming models, workloads, and environments.

8.1 Sensitivity to Programming Model

Applications written with a message-passing library, such as MPI [48] or PVM [47], represent a substantial segment of parallel programs and therefore need to be handled well by any scheduling service. While a preliminary study of implicit coscheduling on MPI applications showed promising results, more in-depth evaluation is needed [55]. In some ways, with implicit coscheduling, message-passing applications are easier to support than Split-C applications, and in some ways are more difficult.

Most message-passing applications are optimized to transfer data in large messages, thus amortizing the relatively high overhead of communication. Therefore, most message-passing programs communicate less frequently and with larger messages than the Split-C applications we measured. Our data and analysis indicates that these characteristics make applications easier to schedule, since they do not require as much coordination.

However, with message-passing semantics, communication and synchronization are combined into a single send-receive operation. If the send must wait for a corresponding receive to be posted on the destination, then the time for the send includes not only the time to transfer the data, but also the time spent waiting for the destination process to post the receive. Both of these components must be incorporated into the baseline spin time, complicating the analysis. Predicting this synchronization time is similar to predicting the amount of load-imbalance at barrier operations; adapting our prediction algorithms to send-receive operations would be an interesting and non-trivial area to investigate.

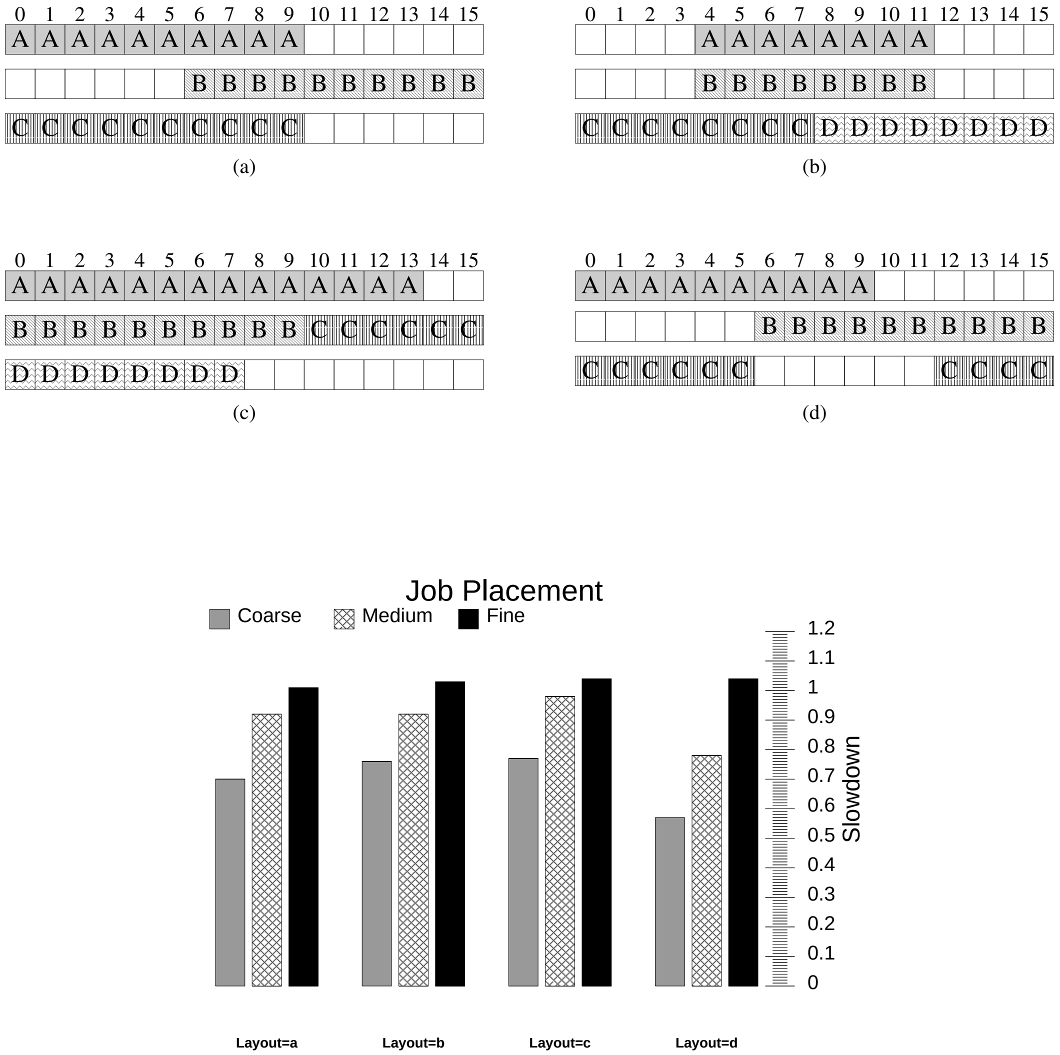

Figure 22: **Sensitivity to Job Placement.** *Three or four jobs, A, B, C, and, optionally, D are placed across 16 workstations. The figures designate how the processes would be scheduled over time with gang scheduling. For each of the four layouts, we investigate three workloads: coarse, medium, and fine. The coarse-grain workload consists of applications synchronizing every $C = 100ms$ with $V = 200ms$; the medium-grain workload contains applications synchronizing every $C = 10ms$ with $V = 5ms$; finally, the fine-grain workload contains applications synchronizing every $C = 100\mu s$ with $V = 0$. All workloads read in a* `NEWS` *pattern with $c = 8\mu s$ of intervening computation.*

8.2 Sensitivity to Workloads

While we have shown that implicit coscheduling handles workloads consisting entirely of parallel applications, we have not yet evaluated more diverse, general-purpose workloads. However, we believe that given its design, implicit coscheduling is better suited to client-server, I/O-intensive, and interactive jobs than is gang scheduling.

Client-server applications are not supported by gang scheduling because it is not known *a priori* which processes communicate with each other. Implicit coscheduling should adapt to this workload without difficulty; however, we do expect two new issues with client-server applications. First, the two-phase waiting algorithm must be biased to minimize the scheduling costs on the server versus that of the clients (which are expected to be less loaded than the server). Second, the requests sent by the clients are expected to be more general than simple read and write operations; therefore, in addition to predicting the network transmission time, the client should predict the time for the operation to be computed on the server.

Intensive I/O workloads perform poorly with gang scheduling because processes must remain scheduled for their time-slice even when waiting for I/O to complete; unless the waiting process is actively handling messages from other processes, no useful work is performed in this interval and significant resources may be wasted [36, 44]. The natural behavior of implicit coscheduling is to perform conditional two-phase waiting for I/O. With this approach, processes remain scheduled during long I/O activity when receiving messages from other processes, but relinquish the CPU when not performing useful work. We anticipate the primary challenge to be implementing the two-phase waiting algorithm such that the process can wake either if a message arrives or if the I/O operation completes. Our expectation is that implicit coscheduling will work well with I/O-intensive parallel and sequential applications.

Workloads containing interactive jobs are not handled well with gang scheduling [4]. Since sharing the cluster dynamically between parallel and interactive applications is likely to give the best utilization [19, 40], it is important to solve this problem. Whether or not implicit coscheduling can handle this workload is an open question: the current fairness mechanisms in our local scheduler may allow interactive processes that have accumulated little CPU time to preempt the scheduled (communicating) job too readily, disrupting its coordination. We think there are two interesting changes that are worth investigating. First, the scheduling of interactive jobs could be postponed by an amount small enough to not noticeably impact user response time; that is, if the interactive job wakes at an inconvenient time for the parallel job, then the preemption could be briefly postponed until after the communication phase. Second, interactive jobs could be allowed to preempt a parallel job immediately only if computing for a very short time (on the order of a few context-switches) and the baseline spin-time could be increased by this amount. In this way, other processes communicating with this machine would not lose coordination when the interactive job briefly interrupts.

8.3 Sensitivity to Environment

The analysis and experiments in this paper have examined only tightly-coupled, distributed-memory clusters of homogeneous, single-processor machines. We now briefly discuss the impact of removing each of these assumptions.

Simple analysis and simulations show that in loosely-coupled or wide-area environments where network latency exceeds the cost of a context-switch, coordinated scheduling (whether achieved from gang scheduling or implicit coscheduling) is not beneficial [6]. With these parameters, processes should block immediately, rather than spin-wait for any communication operations to complete. We note that given an implementation of implicit coscheduling, it is trivial to set the baseline spin time to zero in order to be equivalent to blocking immediately.

In a cluster with hardware support in the network interface for shared-memory [31], processes can read from and write to remote memory even when the cooperating, destination process is not scheduled. However, coordinated scheduling is still needed to efficiently handle synchronization with the parallel application. Thus, implicit coscheduling is still appropriate in such environments, but should be simpler to implement.

Finally, although gang scheduling can not achieve good performance with heterogenous nodes, we expect implicit coscheduling to work well in this setting. Given that nodes can act autonomously with implicit coscheduling, processes running on faster processors will simply complete their computation phases more rapidly and wait longer at synchronization events. The main challenge in supporting clusters of heterogeneous machines (or of multiprocessor nodes) with implicit coscheduling is accounting for the variation in communication costs across different nodes.

9 Summary

Implicit coscheduling coordinates the scheduling of communicating processes in a distributed system, without requiring explicit messages between independent schedulers. Instead, each process leverages naturally-occurring communication to infer when it is beneficial to be scheduled and shares this knowledge with the local operating system scheduler.

The two primary components of implicit coscheduling are the conditional two-phase waiting algorithm with the correct baseline and conditional spin-times, and a preemptive, fair local scheduler. Conditional two-phase waiting maintains scheduling coordination, while bounding the penalty that processes pay when not coordinated. Picking the baseline and conditional spin amounts is critical to achieving the best performance. Communicating processes remain coordinated across distributed machines by matching the baseline spin-time to the expected worst-case completion time of the operation when the arriving message triggers the scheduling of the remote processes. Processes use the arrival rate of messages to determine if they should conditionally spin longer. If messages arrive frequently enough that the cost of blocking and reawakening exceeds the cost of idly spinning, then the waiting process continues to spin.

By choosing whether to spin-wait or block at a communication operation, each process informs its local operating system scheduler whether it is beneficial for this process to be scheduled. Each scheduler is free to respond to this information as it desires: it can schedule any of the runnable processes. However, fair implicit coscheduling of jobs communicating at different rates is only achieved when the local schedulers provide a fair cost-model to user processes. Schedulers based on multilevel feedback queues, such as the Solaris Time-Sharing scheduler, are not precisely fair; that is, they do not adequately compensate processes for the time that they do not compete for the processor. Proportional-share schedulers, such as a stride scheduler, are fair only to processes that are actively competing for the processor. We have described an extension to a stride scheduler that gives compensation tickets to processes that relinquish the processor.

We have both simulated and implemented implicit coscheduling. The two environments were constructed to be quite similar to one another in workload and architectural assumptions. In this paper, we have examined workloads containing both bulk-synchronous and continuously-communicating jobs and have varied three important application parameters: the rate of communication, the rate of synchronization, and the amount of load-imbalance internal to the application. We have exercised implicit coscheduling further by examining scalability in terms of both jobs and workstations, and sensitivity to process placement. All of our measurements have shown that, given the appropriate spin amount, baseline spinning is sufficient to achieve respectable performance for bulk-synchronous applications and applications that synchronize frequently. Applications that rarely synchronize and yet communicate frequently are difficult to schedule effectively. Such applications require that processes remain scheduled when only partial coordination exists; this is achieved with conditional spinning. The baseline and conditional spin amounts that gave the best implicit coscheduling performance match the spin times derived in our analysis.

It is promising to note that the types of applications that perform the best with implicit coscheduling are the same that achieve good speedups in a dedicated environment (*e.g.*, coarse-grain applications [1] and applications with frequent barriers [10, 21]), whereas those that were most difficult to coordinate were specifically written to stress communication performance (*e.g.*, applications communicating continuously and very frequently [1, 15]). Therefore, we believe that most parallel workloads will behave well with implicit coscheduling, even when not specifically targeted for this environment. In summary, with implicit coscheduling in our environment, well-behaved applications can expect speedups or slowdowns within 15% of ideal gang scheduling, and even almost all difficult applications perform within 30% of gang scheduling.

Acknowledgments

The author would like to thank the many people who have helped with the development, implementation, and presentation of implicit coscheduling over the years. This list begins with my advisor at U.C. Berkeley, David Culler, for innumerable discussions and insights. I also thank Doug Ghormley and Alan Mainwaring for providing the GLUnix and AM-II software, respectively, within the NOW project. Mendel Rosenblum deserves much appreciation for giving me the opportunity to prepare this document while at Stanford. Tom Anderson, Seth Goldstein, Steve Lumetta, Girija Narlikar, are among those who have helped with the presentation of implicit coscheduling. Finally, my list ends with husband and colleague, Remzi Arpaci-Dusseau, for his help developing the initial concepts, his untiring assistance drawing figures and fixing graphs, and his continual feedback on writing and ideas.

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] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[4] 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.

[5] 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.

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

[7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, pages 243–254, 1997.

[8] E. Biagioni, E. Cooper, and R. Sansom. Designing a Practical ATM LAN. *IEEE Network*, 7(2):32–39, March 1993.

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

[10] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium*, pages 858–867, Cancun, Mexico, Apr. 1993.

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

[12] 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.

[13] 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.

[14] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *Ninth Joint Symposium on Parallel Processing*, Kobe, Japan, May 1997.

[15] D. E. Culler, A. Dusseau, S. C. 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.

[16] 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.

[17] D. E. Denning and P. J. Denning. Data Security. *Computing Surveys*, 11(2):227–249, Sept. 1979.

[18] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High-Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[19] L. Dowdy. On the Partitioning of Multiprocessor Systems. Technical Report 88-06, Department of Computer Science, Vanderbilt University, July 1988.

[20] 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.

[21] 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.

[22] J. R. Eykholt, S. R. Kleiman, S. Barton, J. Voll, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, M. Weeks, and D. Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. In *Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition*, pages 11–18, San Antontio, TX, USA, June 1992.

[23] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995. Second Revision, August 1997.

[24] D. G. Feitelson and M. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] J. L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[31] 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.

[32] 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.

[33] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88 Symposium: Communications Architectures and Protocols*, pages 314–29, Aug. 1988.

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

[35] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[36] 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.

[37] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 226–36, May 1990.

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

[39] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver, CO, June 1997.

[40] R. Nelson and D. Towsley. A Performance Evaluation of Several Priority Policies for Parallel Processing Systems. *Journal of the ACM*, 40(3):714–740, July 1993.

[41] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 Unix Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Nov. 1993.

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

[43] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 Supercomputing Conference*, San Diego, California, 1995.

[44] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. The Impact of I/O on Program Behavior and Parallel Scheduling. In *SIGMETRICS '98/PERFORMANCE'98*, June 1998.

[45] 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.

[46] 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.

[47] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[48] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, Nov. 1993.

[49] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP*, pages 40–53, Copper Mountain, CO, December 1995.

[50] 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.

[51] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995. Also appears as Technical Report MIT/LCS/TR-667.

[52] C. A. Waldspurger and W. E. Weihl. An Object-Oriented Framework for Modular Resource Management. In *5th Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, October 1996.

[53] F. Wang, M. Papaefthymiou, and M. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[54] 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.

[55] F. C. Wong, A. C. Arpaci-Dusseau, and D. E. Culler. Building MPI for Multi-Programming Systems using Implicit Information. In *Proceedings of The 6th European PVM/MPI User's Group Meeting*, Barcelona, Spain, Sept. 1999.