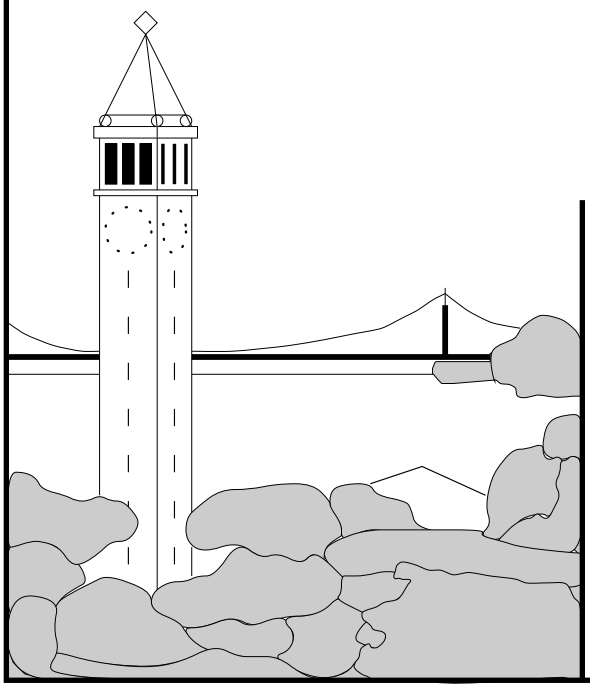
Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems

Andrea Carol Apraci-Dusseau



Report No. UCB/CSD-99-1052

December 1998

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems

by

Andrea Carol Arpaci-Dusseau

B.S. (Carnegie Mellon University) 1991
M.S. (University of California, Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David Culler, Chair
Professor Katherine Yelick
Professor David Freedman

1998

The dissertation of Andrea Carol Arpaci-Dusseau is approved:

Chair Date

Date

Date

University of California at Berkeley

1998

Implicit Coscheduling:
Coordinated Scheduling with Implicit Information in Distributed Systems

Copyright 1998
by
Andrea Carol Arpaci-Dusseau

Abstract

Implicit Coscheduling:
Coordinated Scheduling with Implicit Information in Distributed Systems

by

Andrea Carol Arpaci-Dusseau
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David Culler, Chair

In this thesis, we formalize the concept of an *implicitly-controlled system*, also referred to as an *implicit system*. In an implicit system, cooperating components do not explicitly contact other components for control or state information; instead, components infer remote state by observing naturally-occurring local events and their corresponding *implicit information*, *i.e.*, information available outside of a defined interface. Many systems, particularly in distributed and networked environments, have leveraged implicit control to simplify the implementation of services with autonomous components.

To concretely demonstrate the advantages of implicit control, we propose and implement *implicit coscheduling*, an algorithm for dynamically coordinating the time-sharing of communicating processes across distributed machines. Coordinated scheduling, required for communicating processes to leverage the recent performance improvements of switch-based networks and low overhead protocols, has traditionally been achieved with explicit coscheduling. However, implementations of explicit coscheduling often suffer from multiple failure points, high context-switch overheads, and poor interaction with client-server, interactive, and I/O-intensive jobs.

Implicit coscheduling supports not only traditional parallel applications on Networks of Workstations, but also general-purpose workloads. By observing and reacting to implicit information (*e.g.*, the round-trip time of request-response messages), processes across the system make independent decisions that coordinate their scheduling in a fair and efficient manner. The principle component of implicit coscheduling is *conditional two-phase waiting*, a generalization of traditional two-phase waiting in which spin-time is only partially determined before the process begins waiting and may be conditionally increased depending upon events that occur while the process spins. A second important component is a fair and preemptive local operating system scheduler.

With simple models and analysis, we derive the appropriate baseline and conditional spin amounts for the waiting algorithm as a function of system parameters. We show through simulation and an implementation on a cluster of 32 workstations that implicit coscheduling efficiently and fairly handles competing applications with a wide range of communication characteristics. We predict that most well-behaved parallel applications will perform within 15% of ideal explicit coscheduling.

Professor David Culler
Dissertation Committee Chair

To a nine-year-old girl

Contents

List of Figures viii

List of Tables xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Scheduling Requirements . . . 2
  - 1.2.1 Problem Statement . . . 3
- 1.3 Previous Approaches . . . 4
- 1.4 Implicit Coscheduling . . . 4
  - 1.4.1 Local Scheduler . . . 5
  - 1.4.2 Conditional Two-Phase Waiting . . . 6
- 1.5 Organization . . . 7

2 Implicit Systems 8
- 2.1 Constructing Traditional System Services . . . 8
  - 2.1.1 Control Structure . . . 9
  - 2.1.2 Obtaining Information . . . 9
- 2.2 Implicitly-Controlled Systems . . . 10
  - 2.2.1 Traditional Sources of Information . . . 10
  - 2.2.2 Implicit Information . . . 11
- 2.3 Examples . . . 12
  - 2.3.1 Single-Processor Systems . . . 13
  - 2.3.2 Multiprocessor Systems . . . 14
  - 2.3.3 Networked Systems . . . 15
- 2.4 Summary . . . 17

3 Scheduling Background 19
- 3.1 Evolution of Clusters . . . 19
- 3.2 Requirements . . . 20
- 3.3 Cluster Scheduling . . . 22
  - 3.3.1 Local Scheduling . . . 22
  - 3.3.2 Explicit Coscheduling . . . 28
  - 3.3.3 Dynamic Coscheduling . . . 34

3.3.4 Fair Allocation across a Shared Cluster . . . 35
3.4 Summary . . . 35

4 Implicit Coscheduling Framework **37**
4.1 Components of System . . . 37
4.1.1 Machine Architecture . . . 37
4.1.2 Message-Layer . . . 38
4.1.3 User Processes . . . 38
4.1.4 Application Workload . . . 39
4.1.5 Operating System Scheduler . . . 40
4.2 Components of Implicit Coscheduling . . . 40
4.2.1 Interaction between Processes and the Scheduler . . . 42
4.2.2 Interaction between Communicating Processes . . . 43

5 Local Scheduler **46**
5.1 Requirements . . . 46
5.1.1 Preemption . . . 47
5.1.2 Amortized Context-Switches . . . 47
5.1.3 Fair Cost-Model . . . 48
5.2 Multilevel Feedback Queue Schedulers . . . 48
5.2.1 Overview of Solaris Time-Sharing Scheduler . . . 48
5.2.2 Preemption . . . 49
5.2.3 Amortized Context-Switches . . . 49
5.2.4 Fair Cost-Model . . . 49
5.3 Proportional-Share Schedulers . . . 52
5.3.1 Overview of Stride Schedulers . . . 52
5.3.2 Preemption . . . 53
5.3.3 Amortized Context-Switches . . . 53
5.3.4 Fair Cost-Model . . . 53
5.3.5 Scheduling in the Cluster . . . 57
5.4 Summary . . . 61

6 Cost-Benefit Analysis of Waiting **63**
6.1 Conditional Two-Phase Waiting . . . 63
6.2 Maintaining Coordination with Destinations . . . 65
6.2.1 Request-Response . . . 65
6.2.2 One-Way Requests . . . 67
6.2.3 All-to-all Synchronization . . . 67
6.2.4 Discussion . . . 71
6.3 Maintaining Coordination with Senders . . . 72
6.3.1 Incoming Request-Response . . . 72
6.3.2 Incoming One-way Requests . . . 74
6.3.3 Incoming Synchronization Messages . . . 76
6.3.4 Discussion . . . 76
6.4 Optimizations for Long Waiting Times . . . 77

6.4.1 Network Latency . . . 77
6.4.2 Load-Imbalance . . . 79
6.5 Summary . . . 83

7 Simulation Environment **86**
7.1 Machine Architecture . . . 86
7.2 Message Layer . . . 86
7.3 User Processes . . . 88
7.3.1 Communication Primitives . . . 88
7.3.2 Waiting Algorithm . . . 90
7.4 Application Workload . . . 90
7.5 Operating System Scheduler . . . 96
7.5.1 Class Independent Functionality . . . 96
7.5.2 Scheduling Classes . . . 97

8 Simulation Results **99**
8.1 Scheduling Coordination . . . 99
8.1.1 Sensitivity to Network Latency . . . 99
8.1.2 Sensitivity to Workload Parameters . . . 103
8.1.3 Discussion . . . 106
8.2 Baseline Spin . . . 106
8.2.1 Bulk-Synchronous Communication . . . 107
8.2.2 Continuous-Communication Workloads . . . 110
8.2.3 Discussion . . . 112
8.3 Conditional Spin . . . 112
8.3.1 Bulk-Synchronous Workloads . . . 112
8.3.2 Continuous-Communication Workloads . . . 113
8.3.3 Discussion . . . 118
8.4 Load-Imbalance . . . 118
8.4.1 Bulk-Synchronous Workloads . . . 119
8.4.2 Continuous-Communication Workloads . . . 122
8.4.3 Approximating Load-Imbalance . . . 123
8.4.4 Discussion . . . 127
8.5 Local Scheduler . . . 127
8.5.1 Bulk-Synchronous Workloads . . . 127
8.5.2 Continuous-Communication Workloads . . . 132
8.5.3 Discussion . . . 135
8.6 Summary . . . 137

9 Prototype Implementation **138**
9.1 System Architecture . . . 138
9.2 Message Layer . . . 140
9.3 User Processes . . . 141
9.3.1 Communication Primitives . . . 142
9.3.2 Waiting Algorithm . . . 142

9.4 Application Workload 147
9.4.1 Synthetic Applications 147
9.4.2 Real Split-C Applications 152
9.5 Operating System Scheduler 152
9.5.1 Local Schedulers 152
9.5.2 Explicit Coscheduling 154
9.5.3 Job Placement 154

10 Implementation Study **155**
10.1 Verification of Simulations 155
10.1.1 Baseline Spin 156
10.1.2 Conditional Spin 160
10.1.3 Local Scheduler 164
10.2 Range of Workloads 168
10.2.1 Additional Communication Primitives 168
10.2.2 Real Applications 172
10.2.3 Job Scalability 173
10.2.4 Workstation Scalability 177
10.2.5 Job Placement 179
10.3 Summary 181
10.3.1 Implementation Performance 181
10.3.2 Comparison to Simulation Predictions 182
10.3.3 Advice for Programmers 183

11 Conclusions **184**
11.1 Summary 184
11.1.1 Conditional Two-Phase Waiting 184
11.1.2 Local Operating System Scheduler 185
11.1.3 Performance 185
11.2 Future Work 187
11.2.1 Implementation Issues 187
11.2.2 Programming Model 188
11.2.3 Workloads 189
11.2.4 Job Allocation and Placement 190
11.2.5 Theoretical Analysis 191
11.2.6 Implicit Systems 192

Bibliography **193**

List of Figures

2.1 Model of an Implicitly-Controlled System. 11

3.1 Local Scheduling. 23
3.2 Simulated Performance of Local Scheduling versus Coscheduling. 25
3.3 Explicit Coscheduling Matrix. 29
3.4 Impact of Interactive Processes on Parallel Jobs. 33

4.1 Transfer of Information in Implicit Coscheduling. 42

5.1 Measured Fairness with Solaris Time-Sharing Scheduler. 51
5.2 Basic Stride Scheduling. 52
5.3 Stride-Scheduling with System Credit Extension. 54
5.4 Stride-Scheduling with Loan & Borrow Extension. 56
5.5 Fairness in Cluster with Independent Stride Schedulers. 57
5.6 Fairness in Cluster with Cooperating Stride Schedulers. 58
5.7 Measured Fairness with Ticket Server and Stride Scheduler. 60

6.1 Time for Request-Response Message with Remote Process Scheduled. . . . 66
6.2 Time for Request-Response Message with Triggering of Remote Process. . . 66
6.3 Time for All-to-all Synchronization with Processes Scheduled. 69
6.4 Time for All-to-all Synchronization with Triggering of Root Process. 70
6.5 Cost of Incoming Request-Response Messages if Local Process Spin-Waits. . 73
6.6 Cost of Incoming Request-Response Messages if Local Process Blocks. . . . 73
6.7 Cost of Incoming One-Way Requests if Local Process Spin-Waits. 75
6.8 Cost of One-Way Requests if Local Process Blocks. 75
6.9 Cost with Network Latency when Processes are Uncoordinated and Block. . 78
6.10 Cost with Network Latency when Processes are Coordinated and Spin-Wait. 79

7.1 Model of Bulk-Synchronous Synthetic Parallel Applications. 92
7.2 Characteristics of Bulk-Synchronous Benchmarks. 93
7.3 Model of Continuous-Communication Synthetic Parallel Applications. . . . 94
7.4 Characteristics of Continuous-Communication Benchmarks. 95

8.1 Impact of Latency and Context-Switch Time on Continuous-Communication Workloads. 101

8.2 Impact of Latency and Context-Switch Time on Bulk-Synchronous Workloads.102
8.3 Performance of Immediate Blocking for Bulk-Synchronous Programs. 104
8.4 Performance of Immediate Blocking for Continuous-Communication Programs.105
8.5 Sensitivity to Read Baseline Spin for Bulk-Synchronous Programs. 108
8.6 Sensitivity to Barrier Baseline Spin for Bulk-Synchronous Programs. 109
8.7 Sensitivity to Baseline Spin for Continuous-Communication Programs. . . . 111
8.8 Sensitivity to Conditional Spin for Continuous-Communication Programs. . 114
8.9 Closeups of Sensitivity to Conditional Spin Amount ($W = 50\mu s$). 115
8.10 Closeups of Sensitivity to Conditional Spin Amount ($W = 200\mu s$). 116
8.11 Sensitivity to Conditional Spin for Continuous-Communication Programs with Frequent Barriers. 117
8.12 Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance ($W = 50\mu s$). 120
8.13 Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance ($W = 200\mu s$). 121
8.14 Sensitivity to Baseline Spin for Continuous-Communication Programs with Load-Imbalance ($W = 50\mu s$). 124
8.15 Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance ($W = 200\mu s$). 125
8.16 Performance with Global Approximation of Load-Imbalance for Bulk-Synchronous Programs. 126
8.17 Fairness for Bulk-Synchronous Programs with Identical Placement (3 Jobs). 129
8.18 Fairness for Bulk-Synchronous Programs with Identical Placement. 131
8.19 Fairness for Bulk-Synchronous Programs with Random Placement. 133
8.20 Fairness for Continuous-Communication Programs ($g = 100ms$). 134
8.21 Fairness for Continuous-Communication Programs ($g = 1s$). 136

9.1 Internals of Ultra 1 Workstation. 139
9.2 Network Topology for Cluster of 32 Workstations. 139
9.3 Microbenchmark Results for Read Baseline Spin. 145
9.4 Microbenchmark Results for Linear Barrier Baseline Spin. 146
9.5 Communication Characteristics of Radix. 149
9.6 Communication Characteristics of EM3D. 150
9.7 Communication Characteristics of Radix:Small. 151
9.8 Communication Characteristics of EM3D:Small. 151

10.1 Sensitivity to Baseline Spin for Bulk-Synchronous Programs. 157
10.2 Performance on Bulk-Synchronous Workloads. 159
10.3 Sensitivity to Waiting Algorithm for Continuous-Communication Programs. 161
10.4 Performance of Continuous-Communication Programs with Request-Response Messages. 162
10.5 Fairness with Bulk-Synchronous Programs. 165
10.6 Fairness with Continuous-Communication Programs. 167
10.7 Performance of Continuous-Communication Programs with One-Way Requests.169
10.8 Performance of Continuous-Communication with Bulk Messages. 171

10.9 Performance of Real Split-C Applications. 173
10.10Job Scalability with Bulk-Synchronous Programs. 174
10.11Job Scalability with Continuous-Communication Programs. 176
10.12Workstation Scalability. 178
10.13Sensitivity to Job Placement. 180

List of Tables

4.1 System Parameters. 41
4.2 Workload Parameters. 41

5.1 Solaris 2.6 Time-Sharing Default Dispatch Table. 50

6.1 Conditional Two-Phase Waiting Parameters for Implicit Coscheduling. . . . 84

7.1 Measured Network Latency. 87
7.2 System Parameters in Simulations. 87
7.3 Conditional Two-Phase Waiting Parameters for Simulation. 89
7.4 Measured Context-Switch Cost. 97

9.1 System Parameters in Implementation. 140
9.2 Conditional Two-Phase Waiting Parameters for Implementation. 144
9.3 Communication Characteristics of Benchmark Applications. 148

Acknowledgements

I would like to begin by thanking my advisor, Professor David Culler, whom I greatly admire and respect. Receiving a complement from David is a reward in itself. David is an inspiring person to work with – always enthusiastic and ready to look at another set of graphs and hypothesize about what is happening. I appreciate that whenever I ask him an ambiguous question, he assumes I am asking the deeper question (whether or not that is the case).

Despite all of my previous complaints and gripes, I am grateful for the experience of being a part of the Network of Workstations project. In addition to my advisor, I thank Professors Tom Anderson and David Patterson for leading our large group. I could not have implemented any of this work without the GLUnix and AM-II software provided by Doug Ghormley and Alan Mainwaring, respectively. I thank them and all of the students for the infrastructure they provided, as well as the general shared experience. I find it surprising how much I already miss the meetings and the retreats.

I'd also like to thank my committee members for their feedback on this dissertation. Professors Kathy Yelick, Tom Anderson, and David Freedman gave me valuable feedback and great perspectives on this document, as well as previous related papers. Thanks to Professor Joe Hellerstein for general advice and encouragement.

I thank the older graduate students in David's group, Thorsten von Eicken, Klaus Schauser, and Seth Goldstein for being inspiring in their accomplishments, if not a bit intimidating. But, most of all, I thank them for acquiring the corner window office back in Evans Hall. Thanks as well to Steve Lumetta, Alan Mainwaring, and Girija Narlikar, for sharing office space, spare change, technical discussions, and occasional worries.

The students and faculty who participated in David Culler's System of System graduate class in the Fall of 1997 gave me interesting perspectives on implicit systems and helped me to think about the general issues. In particular, I remember useful comments made by Armando Fox, Steve Gribble, David Wagner, and Professor Joseph Hellerstein.

I still think very fondly back to my undergraduate days at Carnegie Mellon University. In those days, I spent a great deal of time and energy questioning whether or not I wanted to remain in the Computer Engineering department. Bob Barker and Professors Bill Birmingham, Daniel Siewiorek, John Shen, and Jim Hoburg all gave me encouragement and/or research opportunities at key decision points. I doubt they know how much their actions influenced the direction of my life.

Thanks to Vanessa Karubian for giving me the opportunity to volunteer at the Computer Club at Hawthorne Elementary School. She has a gift for both making things happen and sharing the credit with others. Thanks to the fourth and fifth graders for reminding me of the importance of attitude and of how to find significant meaning in work.

I am grateful to my parents for the environment they raised me in. Ever since my dad took me to his work and let me play Adventure while drinking hot chocolate, the path has been set. I thank my parents for managing to simultaneously convey that they will always love me no matter what I do and that they are so proud of my accomplishments. I will always remember how happy my mom sounded when I told her I had been accepted at Berkeley and then seeing her seven years later wiping tears from her eyes at my graduation.

Finally, there are no sufficient words for me to express my indebtedness to my husband, Remzi. How many years will it take me to understand? So slowly I realize how profoundly lucky I am to be with you and to work with you. To borrow from Goethe, you are my true measure of all that is good: support, faith, and, most of all, love.

Chapter 1

Introduction

Building system services in a distributed environment is a difficult task; the primary challenge is that the components implementing the service must know the current state of the entire system in order to act in a cooperative manner. Since the distributed components do not inherently have this global knowledge, they usually explicitly query a set of remote components to gather this information. Unfortunately, adding more communication to the distributed system can cause several problems: the transfer of control information may be costly or consume critical resources, the information may be out-of-date by the time it is received, the desired interface may not be available, or errors may occur while the information is being transmitted.

The thesis of this work is that *implicit control* simplifies the construction of distributed system services; with implicit control, components neither query remote components for information nor explicitly control remote components in their actions. Instead, autonomous components infer remote state by observing naturally-occurring local events and their corresponding *implicit information*, *i.e.*, information available outside of a defined interface. An example of a naturally-occurring event is the arrival of a data request from a remote node; the corresponding implicit information is the arrival rate of the incoming requests. Thus, an *implicit system* contains no additional communication beyond that which is inherent in constructing the service.

This chapter presents a high-level synopsis of this thesis. The first section describes the requirements for services in a distributed system, in particular a scheduler for communicating processes in a Network of Workstations. This is followed by a brief summary of related work. The third section gives a description of the components and contributions of implicit coscheduling. The chapter concludes with an overview of the organization of the remainder of this thesis.

1.1 Motivation

The importance and the prevalence of distributed systems is continuing to grow. One emerging platform of particular importance are Networks of Workstations (NOWs) or clusters, which are tightly-coupled collections of commodity workstations (or personal computers) connected with a switch-based, high-bandwidth network. NOWs represent a

cost-effective way of building high-performance, scalable servers [4, 19, 37, 61, 75, 143, 154, 175].

Unlike previous distributed systems which supported only long-running, sequential applications [104, 178], modern clusters can efficiently support a variety of job classes. First, due to the advent of low-latency, high-bandwidth, switch-based networks [12, 21] and low-overhead message protocols [11, 110, 136, 165, 166], NOWs can execute fine-grain parallel applications [36, 116]. Second, clusters can improve the client-server applications of traditional distributed systems (*e.g.*, naming, locking, and file services) by updating these services to modern communication protocols [5]. Third, acting as a shared server across many competing users, NOWs can time-share the interactive, developmental jobs found in general-purpose workloads [7]. Finally, clusters constitute an excellent platform for applications with large I/O demands [9, 43].

1.2 Scheduling Requirements

Due to the presence of this wide range of job classes, scheduling and resource allocation is a much more difficult problem. In addition to achieving high throughput for long-running applications, the operating system scheduler in the cluster must provide the following criteria.

- **Coordinated Scheduling:** Communicating processes from the same parallel job must be scheduled simultaneously across a set of workstations to achieve efficient performance. Only if processes are coordinated can communication proceed at the rate of the underlying hardware without incurring a context-switch.
- **Dynamic Identification:** Communicating processes must be identified at run-time to support client-server applications and processes that communicate with a wide set of processes over their life-time.
- **Fast Response Time:** Interactive jobs must receive quick response time, while negligibly harming the throughput of long-running parallel jobs.
- **Efficient Resource Usage:** Jobs that are waiting for events (*e.g.*, disk or user I/O) should relinquish the processor so that they do not waste resources.
- **Fair Allocation:** Since the clustered environment is shared across many competing users each running jobs with different characteristics, the amount of resources allocated to each user should be independent of the number of jobs each user runs and the communication behavior of those jobs.

Further, all system services should have the following structural qualities.

- **Autonomous:** Each node should maintain control over its own actions, participating in the system because it is in its best interest to do so, not by imposition; participating in the cluster should not degrade the absolute performance of the single node.

- **Reconfigurable:** Workstations should be able to join and leave the cluster dynamically without restarting the system service.
- **Reliable:** The service should tolerate the failure of any node of the system.

1.2.1 Problem Statement

In this thesis, we investigate the design and implementation of coordinated scheduling as an implicitly-controlled distributed service. Traditionally, coordinated time-sharing of cooperating processes across the nodes of a cluster has required explicit control; for example, in explicit coscheduling [134], or gang-scheduling, a set of master components determines a global schedule of processes over time, and explicitly communicates this schedule to the participating local schedulers. With *implicit coscheduling*, cooperating processes achieve coordinated scheduling without explicit control or additional communication.

The primary goal of implicit coscheduling and of this thesis is obtaining fair, coordinated scheduling for communicating processes in traditional parallel applications. We believe that this is the first and most difficult step in constructing a system that can support more complete workloads. Therefore, while implicit coscheduling has been designed to support the additional needs of a general-purpose workload with client-server, interactive, and I/O-bound applications, we have not yet evaluated such workloads.

We refer to a set of communicating processes as a *job*. This job may be a dynamic collection of communicating processes (*e.g.*, a server and multiple clients) or a predefined, static collection (*e.g.*, a traditional parallel application). The set of processes within a job are *cooperating* processes, while the others in the system are *competing* processes.

The scheduling of parallel jobs in a multiprogrammed environment has long been an active area of research. The problem is typically decomposed into two steps: *allocation*, where individual processes are placed on processors, and *dispatching*, where those processes are scheduled over time. The allocation step for parallel jobs has been investigated in detail [31, 33, 66, 70, 101, 118, 119, 120, 126, 139, 149]. Given the popular single-program-multiple-data (SPMD) parallel programming model, it has generally been found that the response time and throughput of the workload are best when competing processes from different jobs can share the same processor [35, 53, 82, 105, 144, 148, 173].

In this work, we focus on the second step of dispatching the communicating processes over time. We consider the problem after the parallel jobs have been divided into a fixed number of communicating processes and after those processes have been placed on a set of processors. We evaluate allocations where only competing processes, and not cooperating processes, share the same processor. Further, we assume that the allocation step has accounted for all memory considerations. Whether or not processes migrate between processors over time is orthogonal to our work.

For our programming model, we assume that cooperating SPMD processes communicate with low-level messages, such as those defined by the Active Message model [165]. We assume that the receiving process must be scheduled to handle a message and to return a response. We consider applications written with request-response and one-way communication operations and with barrier synchronization operations. Finally, we assume that we have no control over the communication or synchronization performed by the applications.

1.3 Previous Approaches

Previous work in this area of time-sharing parallel workloads has examined two distinct approaches: local scheduling and explicit coscheduling. Each has significant weaknesses that render it unfit for the NOW environment and workload.

The most straight-forward approach for time-sharing jobs is for the operating system scheduler on each workstation to schedule its allocated processes independently. While this simple approach of *local scheduling* has the desired structural qualities of a distributed service (*i.e.*, autonomy, reconfigurability, and reliability), it fails the most important criteria: performance. Because communicating processes are not scheduled in a coordinated fashion, each machine spends a significant portion of its time context-switching between processes; the resulting performance for fine-grain parallel applications is not acceptable. Over the years, numerous researchers have found that the slowdown of local scheduling may be orders of magnitude worse than ideal for frequently communicating processes [7, 35, 55, 56, 70, 99].

To resolve the performance inefficiencies of local scheduling, *coscheduling* [134] or *gang scheduling*, explicitly schedules the cooperating processes from a single job simultaneously across processors. A strict round-robin global schedule of processes is constructed, and a global context-switch is performed across all schedulers at the same time. With coscheduling, each job is given the impression that it is running on a dedicated machine. We refer to this traditional form of coscheduling as *explicit coscheduling*.

However, explicit coscheduling also fails to meet many of the requirements of our distributed system service. Straight-forward implementations are neither scalable nor reliable; hierarchical constructions can remove single points-of-failure and scalability bottlenecks, but only with increased implementation complexity [54]. In all cases, components must act as slaves, rather than in an autonomous manner. Difficulties also occur when using explicit coscheduling with a mixed, general-purpose workload: explicit coscheduling requires that the schedule of communicating processes be precomputed and the strict round-robin schedule performs poorly with interactive jobs and with jobs initiating I/O [7, 10, 49, 100].

1.4 Implicit Coscheduling

To meet the requirements of a time-shared scheduling approach for general-purpose workloads on networks of workstations, we have developed *implicit coscheduling*. Implicit coscheduling requires no additional communication between components, either to determine remote state or to direct other components in their actions. Instead, each process determines for itself when it is beneficial to be scheduled with implicit information and shares this decision with the local operating system scheduler.

With implicit coscheduling, it is advantageous for a process to remain scheduled when it is coordinated with remote, cooperating processes. A process determines whether scheduling is coordinated with two pieces of locally-visible implicit information. The first piece of implicit information is the observed round-trip time of request-response messages naturally occurring within the 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.

After a process has determined whether scheduling is coordinated and whether it should remain scheduled, it informs the local operating system scheduler. If it is beneficial to the job as a whole for the process to be scheduled, then the process simply remains runnable. If it is not beneficial, then the process voluntarily relinquishes the processor and sleeps until it is so. Thus, no changes are required to the interface of the local scheduler. The local scheduler can thus remain tuned for sequential workloads containing interactive and I/O-bound jobs.

By construction, implicit coscheduling has the three structural properties desired for any distributed system service: autonomy, reconfigurability, and reliability. Because the components running on each workstation in the cluster act autonomously and rely on only implicit information, nodes can enter and exit the system at any time (whether due to choice or to failure), without impacting the service on other nodes.

Achieving distributed coordination with implicit coscheduling requires two key components. First, the operating system scheduler running on each machine in the cluster must provide a fair cost-model to user processes; this cost-model allows each process to calculate whether it should compete for the CPU at the current time. Second, each process must use *conditional two-phase waiting* when dependent on an event from a remote process (*e.g.*, when the process must wait until it receives a response to a previous request before it can continue computing). We now describe this two components in more detail.

1.4.1 Local Scheduler

The first key to implicit coscheduling is the behavior of the local operating system scheduler. To obtain good workload throughput, fair allocation across jobs communicating at different rates, and fair allocation across users running different numbers of jobs, the operating system scheduler must have the following three properties:

- **Preemption.** Due to the interdependencies of communicating processes across machines, the scheduler must be willing to preempt a process to prevent deadlock; a simple round-robin scheduler is sufficient. Performance may be also improved if the scheduler immediately dispatches processes when they become runnable due to message arrivals.

- **Amortized Context-Switch.** To amortize the cost of obtaining coordination across machines, the duration of a time-slice should be long relative to a context-switch.

- **Fair Cost-Model.** The local scheduler should export a well-defined cost-model so that processes use the CPU only when it is beneficial for them to do so. This also allows higher-level policies to provide fair allocations to competing users across the cluster.

In this thesis, we describe and implement a local scheduler that achieves these three properties. Our approach is based on extensions to a *stride scheduler*, a proportional-share scheduler introduced by Waldspurger [168], in which resources are allocated to a process

in proportion to its number of tickets. The primary difference in our scheduler is that processes receive the same amount of resources regardless of their computational behavior (*i.e.*, when they run versus when they sleep). This is accomplished by giving exhaustible tickets [171] to processes that relinquished the processor in the past and did not receive their proportional-share. With more tickets, these processes are then scheduled more frequently in the future; thus, processes that communicate (and subsequently sleep) at different rates are scheduled fairly.

1.4.2 Conditional Two-Phase Waiting

The second key to implicit coscheduling is the behavior of processes that are waiting for communication and synchronization to complete. Processes can achieve coordinated scheduling by simply deciding to spin or to block when waiting for an operation; To determine the correct action, each process applies its knowledge of the current scheduling state of the cluster to the scheduler's cost model. Conditional two-phase waiting incorporates this knowledge.

In this dissertation, we introduce conditional two-phase waiting as a generalization of *two-phase waiting* [134]. In the first phase of two-phase waiting, the process spins for some baseline amount of time while waiting for the desired event; 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.

Good performance with implicit coscheduling requires picking the correct amount of baseline and conditional spin-time in the first phase of the waiting algorithm. In previous research, the desired spin-time for two-phase waiting has been calculated with competitive analysis, assuming that waiting times are chosen by an adversary [85, 86, 102]. However, in our environment, it is not appropriate to assume that waiting times are adversarial. Instead, processes can influence future waiting times advantageously by acting in a cooperative manner. We have found that there are three factors to consider in conditional waiting.

- **Baseline Spin:** Processes should wait for the expected worst-case completion time of the operation that occurs when the arrival of a message triggers the scheduling of a remote process. Spinning long enough to maintain coordination increases the likelihood that future waiting times will also be low, which improves communication and synchronization performance.

- **Conditional Spin:** Processes should spin longer when receiving messages. At some threshold incoming message rate, it is beneficial for the waiting process to remain scheduled, even though it is not making forward progress, because it is handling incoming messages and allowing remote cooperating processes to make progress.

- **Long Waiting Times:** In some cases, by relinquishing the processor rather than waiting for slow operations to complete, a process may achieve better performance with implicit coscheduling than is possible with explicit coscheduling. These cases

occur when network latency or the internal load-imbalance of the application is high relative to the cost of a context switch.

1.5 Organization

In the following two chapters, we cover the background material for this thesis. Specifically, in Chapter 2 we describe the philosophy of implicit systems and give examples of implicitly-controlled systems and implicit information in other areas. In Chapter 3, we discuss previous research in scheduling communicating processes in multiprogrammed distributed and parallel systems.

In the next three chapters, we discuss implicit coscheduling in detail. In Chapter 4 we describe our model of the system and the available implicit information. We discuss the requirements of the local operating system scheduler in Chapter 5 and describe extensions to an existing scheduler with the desired properties. In Chapter 6 we present the two-phase waiting algorithm employed by processes when they communicate and derive the desired amount of spin-time.

In the final set of chapters we cover the performance of implicit coscheduling for a variety of workloads and environments. After describing our simulation environment in Chapter 7, we present our simulation results in Chapter 8. In Chapter 9 we describe our implementation on the U.C. Berkeley NOW cluster; we present measurements of our system in Chapter 10. We discuss our conclusions and extensions for future research in Chapter 11.

Chapter 2

Implicit Systems

"Where there's smoke, there's fire." – English Proverb

Most interesting systems contain multiple components that interact with one another to implement a common service. To interact in a cooperative manner, each component usually requires information about the state and behavior of other components. Constructing such systems is often quite complicated, due to the components' lack of perfect knowledge of remote components.

The idea behind an *implicitly-controlled* system is that rather than explicitly contacting other components to obtain information, components infer the state of other components from local observations of naturally-occurring events. Rather than obey commands from a component with complete control over the system, each component reacts independently to the forces that are influencing it.

In this chapter, we describe the properties and utility of implicit information for building complex services, especially services in distributed systems. We begin by defining the collection of components in a system, their common structure, and the interfaces for exchanging information between components. We then describe implicitly-controlled systems and implicit information. We conclude by citing examples of systems that are implicitly controlled or leverage implicit information.

2.1 Constructing Traditional System Services

Hardware and software *systems* are collections of *components*, or modules, that interact with one another to implement a common set of services. Complex systems are divided into multiple, interacting components for a variety of reasons. First, modules can hide unnecessary details, simplifying the conceptualization and maintenance of the overall system [137]. Second, components encapsulate a particular implementation, allowing designers to optimize for that implementation while preserving compatibility when underlying assumptions change. Finally, physical limitations may restrict the size of an individual component; for example, in hardware systems, only a certain number of transistors may fit on a single chip.

Components exist at many different levels in both hardware and software systems. For example, in a hardware system, each chip can be a considered an individual component. Multiple chips can also be joined together to form higher-level components, such as the CPU and memory subsystems within a workstation. Finally, each workstation can form a component in a parallel or distributed system.

Likewise, in a software system, each procedure can be considered a component. Multiple procedures can be joined together to form an application or a library. The user application, run-time libraries, and operating system are all individual components on a single workstation, yet the collection of software on each node can act as a single component in a distributed system. In this dissertation, we focus on distributed software systems containing large numbers of similar components.

2.1.1 Control Structure

Cooperating components implement a common service by communicating with one another. Communication can be divided into *data* and *control* traffic [88, 164]. Data traffic is that which is inherently required to implement the service, even if implemented by a single component. Control traffic represents all additional traffic, such as that required to obtain the internal state of another component or to direct another component in its actions.

In general, the components that control the decisions of the system determine the *structure* of the system. The control and communication structure of components can be classified into two main groups: master-slave and peer-to-peer.

With the *master-slave* structure, components behaving as *masters* directly command the actions of other components, the *slaves*. The most straight-forward design is *centralized*, where a single master controls all slaves. However, a centralized master can have severe performance and reliability restrictions in a distributed system; therefore, a number of variations exist. First, the design can be modified slightly such that there is a *parallel master*, a group of components acting as a unified master. Second, in a *hierarchical* design, a component may be both a slave to higher-level nodes and a master to lower-level nodes. While these more complicated structures can improve performance and reliability, they may still break down at some scale; for very large-scale systems, or when the state of the system is changing at such a rapid pace that a master cannot determine quickly enough what is best for the entire system, an alternative is needed.

In a *peer-to-peer* design, each autonomous component is responsible for optimizing its specific part of the problem. This design is inherently more scalable and reliable than master-slave approaches. The primary challenge of a peer-to-peer design is for each component to obtain an accurate view of the state of the global system. We now discuss traditional ways in which components gather such information.

2.1.2 Obtaining Information

Regardless of the structure of the system, to gather information, components usually communicate through well-defined *explicit interfaces*. For example, a local component

sends a message to a remote component, querying it about a particular variable. However, a number of drawbacks exist to this explicit approach.

First, not all relevant information may be obtainable from the interface. In software systems, whether information is available depends primarily upon whether the component designer anticipated this requirement and provided the interface. However, in hardware systems, the amount of information that can be transferred between components has physical limitations as well, related to the number of available data lines and pins.

Second, each interface must have an associated error model. Components must be able to handle all possible errors when communicating with another component. While this can be a difficult problem in sequential systems, it can be an unsolvable problem in a distributed environment due to the potential failure of both remote components and the communication mechanisms themselves in unexpected ways at unpredictable times. Thus, dealing with errors adds a significant burden on the programmer.

Third, the cost of accessing the interface may be prohibitive. Obtaining local information in a sequential software system may involve only a nominal cost, such as executing a library function or calling into the operating system; however, in any physically distributed system, there will be a latency associated with accessing remote components. The overhead of obtaining the information in this case may exceed its benefit.

Fourth, obtaining the control information may consume resources that are required for the data transfers of the service. This is primarily a factor in networked systems, where the control messages consume available bandwidth. We desire an approach that does not contend for the very resources that it is trying to control or allocate.

Finally, the information from the interface may be out-of-date by the time it is received. Once again, this is primarily a consideration in distributed systems. If there is a delay between the time that information is sent from a remote source and the time that information is locally accessible, then the information may have changed in the interim. Maintaining information consistency is especially difficult when clients must coordinate information from multiple remote sites.

2.2 Implicitly-Controlled Systems

Due to the cost and complexity of explicitly contacting other components for control information, we propose that components should only communicate the necessary data information and that control information should be exclusively inferred from naturally-occurring local events. A system which infers all of its control in such a manner is an *implicitly-controlled system*, or more succinctly, an *implicit system*. While the data communication may be structured in either a master-slave or a peer-to-peer fashion, we believe that an implicit system is more naturally suited to a peer-to-peer design. We now discuss how components can obtain control information without explicitly contacting other components.

2.2.1 Traditional Sources of Information

A large body of theoretical work has examined the problem of inferring information and making decisions in complex systems. Some research has focused on systems where

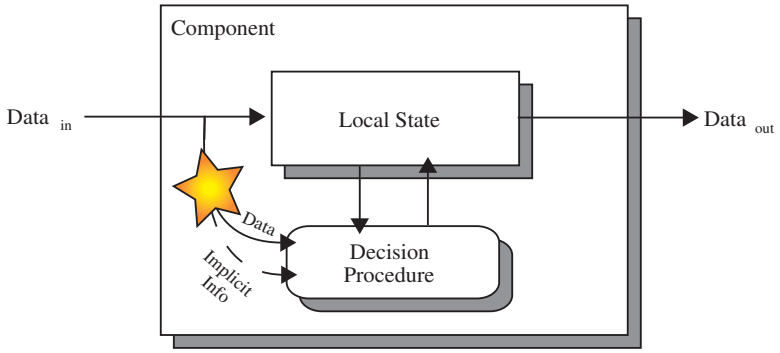


Figure 2.1: **Model of an Implicitly-Controlled System.** *Components that are implicitly-controlled do not receive explicit control messages from other components in the system. Instead, components infer control information from naturally-occurring data transfers.*

no communication occurs between components – an extreme example of an implicitly-controlled system. For example, *game theory* focuses on predicting the actions of adversaries in the absence of any communication [129]. Similarly, *decision theory* provides a model for optimizing the utility of decisions based on uncertain information that has been quantified with a probability measure [15]. Finally, *team decision theory* combines the previous two approaches, stressing the distributed nature of the decision makers [114].

Components can also infer the state of remote components if they have common knowledge [71]. *Common knowledge* is the strongest form of knowledge in a distributed system and consists of those facts that all components know, and that all components know that all components know, *ad infinitum*. Such knowledge can be practically obtained in distributed systems only when the facts are static and predefined, for example, when components are required to conform to a standard or are known to be identical to one another.

Other research, most notably distributed problem solving (DPS) within artificial intelligence, assumes multiple autonomous agents are solving a single problem in a cooperative but decentralized fashion. Much of the work in this field is directly relevant to implicit systems, such as modeling the levels of knowledge of distributed agents [71, 117, 159] and understanding of the utility of information as it ages [13, 138].

2.2.2 Implicit Information

Implicit information is a powerful additional source of knowledge in implicitly-controlled systems. A local component can infer the state of remote components from a combination of arriving data traffic and *implicit information*. Implicit information is available from observing characteristics of local events that are outside of their defined interface. The observed local events must occur naturally as part of the system; thus, implicit information from remote components cannot be requested on demand. An example of a naturally-occurring local event is the arrival of a data request from a remote node. Exam-

ples of implicit information that can be observed about such an event are its completion time and the rate of incoming requests. Figure 2.1 diagrams an implicit system leveraging implicit information.

Implicit information may be interesting for its own sake, with no additional information. However, we believe implicit information is most useful when it is combined with other knowledge; for example, knowledge of how the remote components behave. This combination may allow a component to infer the state in which the remote component must reside to have produced this event.

Implicit information differs from *hints* in that a hint can be incorrect [93]. Implicit information itself is never wrong; the implicit information carried by the naturally-occurring event is something directly observable. However, components may still derive incorrect inferences from implicit information. Therefore, just as a hint must be correct the majority of the time in order to be useful, so must the inference from the implicit information. Implicit information also differs from piggy-backed information: it must occur naturally as part of the data transfer.

Systems that react to implicit information are naturally *adaptive* to current conditions. We note that if a component knows that other components react in a certain manner to implicit information, the component may *manipulate* implicit information to produce a desired response. For example, in Section 2.3.3, we examine a proposal for network routers that drop packets to manipulate the congestion mechanisms within implicitly-controlled TCP endpoints.

For a system to rely entirely on implicit information, the interesting state or behavior in remote components must be observable in local events. When it is not the case that critical events can be inferred with implicit information, then explicit control messages are required. Furthermore, to make conclusive inferences from only implicit information, the local observations must occur simultaneously with the change in remote state and leave no room for misinterpretation. Therefore, when implicit information is only correlated with interesting remote events, but not an entirely reliable predictor, implicit information is more successful in improving performance than in guaranteeing correctness.

To reiterate, in an implicit system, the only communication across components is the data movement that is inherent to the service; an implicit system does not need to leverage implicit information, it simply must not explicitly communicate control messages across components. Alternatively, in an *explicit system*, some communication occurs strictly to convey control information across components – either to inform or query components of remote state or to direct components in their actions. An explicit system may be adaptive or may leverage implicit information to improve performance, but, by definition, requires explicit control messages.

2.3 Examples

Many systems exist in computer science that either are implicitly-controlled or leverage implicit information; however, they have tended to do so in an *ad hoc* manner. One of the goals of this thesis is to explore the common characteristics of such systems and to develop a systematic methodology for building implicit systems. In this section, we

describe a few examples that occur in single-node, multiprocessor, and networked systems.

2.3.1 Single-Processor Systems

Memory Subsystem

Our first example considers the interactions between two components in a single workstation: a user application running on the CPU and the memory subsystem (consisting of the cache and main memory). Due primarily to the limited available bandwidth between the CPU and the memory subsystem, the existing interfaces between the two components are rather restricted. For example, the application can request that a memory location be read by specifying the address of that location; the memory subsystem in turn supplies the data from the specified address. No messages are sent between components other than those inherently necessary to transfer data; thus, the two components act as an implicitly-controlled system. Note that cache architectures that include explicit instructions to prefetch data and to lock specified memory lines in or out of the cache are not strictly implicit systems.

Due to the limited interface, the memory subsystem optimizes its performance by inferring additional control information from the data requests it receives. Optimizing the performance of the memory subsystem equates to caching those memory locations which will likely be requested in the near future [150]. However, caching the correct data requires knowledge of the future behavior of the application, for which there is no explicit interface. To predict which data will be requested in the future, the cache controller assumes there is temporal and spatial locality in the access stream and infers future requests from the requests it has received in the past. As a result of this inference from a single explicit data request, multiple words of data are prefetched into a single block of the cache.

Likewise, the application running on the CPU may obtain better performance if it has knowledge of the underlying memory architecture (*e.g.*, the amount of physical memory; the line size, associativity, and number of lines in each cache; and the number of entries and associativity in the translation-look-aside buffer (TLB)). However, most systems do not contain interfaces that supply information about the cache or memory architecture. Therefore, the application must infer this data by observing available implicit information.

We know of two benchmarks that leverage implicit information to infer the characteristics of the memory subsystem. First, Saavedra-Barrera [145] measures the completion time of memory accesses with different reference patterns to determine the characteristics of the caches and the TLB. Second, the authors of NOW-Sort [9], measure CPU usage to determine the amount of available physical memory. Technically, an application that must run a benchmark to determine this information is not implicitly-controlled. To be truly implicit, the application must not generate any new memory references simply to observe them; instead, the application should measure only those references inherent to its correct operation.

Covert Channels

Implicit information is well-known in the field of security, where information that flows through a medium not intended for the transfer of information is said to flow through *covert channels* [42, 92, 103]. Most covert channels encode information in a physical phenomenon: a simple covert channel is the running time of a program; more complex channels exploit other resource usage patterns, such as the amount of electric power consumed. For example, a malicious user with knowledge of an application may be able to infer some secret information from the running time of that application.

2.3.2 Multiprocessor Systems

Cache-Coherence Protocols

We next consider a shared-memory multiprocessor, where the relevant components are each of the processors, their caches, and main memory. The only inherent data that may be transferred between each processor, its cache, and main memory are the specification of whether this operation is a read or write, the memory address, and the data that is read or written. One of the advantages of caching data is that it reduces the amount of traffic on the memory bus. However, the presence of multiple copies of a single memory location introduces the problem of stale data, when different processors see different values for the same memory address.

In early multiprocessor systems, data was kept consistent across caches with explicit methods by transmitting additional information. For example, an additional high-speed bus could be used to broadcast all write addresses [30]. Alternatively, by removing the abstraction of individual components in the system, the operating system could have complete knowledge of the behavior of all caches and explicitly ensure that inconsistencies do not occur [132]. Finally, by tagging memory locations, processors could explicitly acquire ownership of a memory location before entering a critical section. However, all three solutions introduce a significant amount of complexity beyond that which is required for inherent data transfer and limit the scalability of the system.

In the solution proposed by Goodman [68], cache controllers snoop inherent traffic on the memory bus to infer the state of other caches in the system. Each cache memory block has an associated state, which is updated according to actions of the processor as well as by traffic on the memory bus. The state of each block determines whether or not a processor has exclusive ownership of a memory location, and thus relies on only implicit control.

The design of the cache coherency algorithm in multiprocessors is a rare example of an implicitly-controlled system performed for correct behavior, not only optimized performance. However, an important assumption for the correct operation of this algorithm is that each cache controller cooperates and behaves in this same manner. For example, if one controller does not invalidate a block when another cache broadcasts data, then inconsistencies occur in the data. Further, special care is required in the implementation of the bus protocol to ensure that all remote components have a chance to react before a local action is performed.

Resource Allocation with Virtual Machine Monitors

Disco is a virtual machine monitor that enables multiple operating systems to run on a multiprocessor [26]. Such an approach significantly reduces the implementation complexity of operating system support for reliable and scalable operation. However, one of the drawbacks is that information for efficient resource allocation is hidden from the virtual machine monitor.

For example, the monitor has the responsibility of allocating one of the virtual processors to each of the physical processors. However, the monitor lacks explicit information about which instructions are executing on each virtual processor: for example, the instructions for the idle loop of the operating system and for spin-waiting are indistinguishable from instructions for useful process computation. As a result, the monitor can not make an informed decision about which virtual processor to schedule.

In this situation, the implementors found that additional information could be inferred from naturally-occurring events. Specifically, the MIPS processor contains a reduced power consumption mode; the commodity operating system, IRIX 5.3, enables this mode whenever the system is idle. Since, the virtual machine monitor catches this call, it can infer that no useful work is being performed on this virtual processor and schedules another virtual processor.

Load-Balancing

A common problem in distributed systems is evenly balancing a workload of many jobs across a set of nodes. One of the steps to solving this problem is to determine the current load across the system. Typical approaches for gathering this information are to periodically exchange load information or to query a fixed number of nodes on demand. However, neither approach is ideal. First, both approaches incur overhead to determine load, beyond that of simply placing the job. Second, both approaches may not have an accurate view of the cluster: the first approach may leverage out-of-date information while the second may not know the state of the entire system.

When a single front-end is responsible for placing the jobs across worker-nodes, an alternative exists for obtaining load that uses only implicit information. As described for a cluster-based web-page server [135], given that the front-end must hand-off the incoming TCP connection to the responsible worker-node, the front-end can infer the load on the worker-nodes from the number of active TCP connections. This is a simple and efficient approach: it leverages information that is already available locally, it accurately reflects the load of the worker-nodes, and it involves no extra overhead.

2.3.3 Networked Systems

Ethernet

Perhaps the simplest example of a distributed system with implicit information consists of nodes transmitting data over an Ethernet network [22, 122]. Ethernet is a Carrier Sense, Multiple Access with Collision Detect (CSMA/CD) local area network. In such a

network, multiple nodes are plugged directly into a shared link; when one machine transmits a message, the signal is broadcast over the entire network.

The challenge of managing this shared-medium network is that only one node can transmit data at a time without corrupting the transmissions of all nodes. One approach to guaranteeing mutual exclusion would be to require each node to explicitly gain control of the link before transmitting, as in a token-ring based system [34]. Instead, nodes leverage implicit information for a much simpler, decentralized implementation.

Whenever a node has data to send, it waits until the line is idle and then transmits immediately. While the node sends its data, it also simultaneously watches the line to observe if any collisions with other transmissions occur. Thus, implicit information exists in the form of corrupted data, from which the sender can infer that another node is transmitting simultaneously. If this is the case, the sender performs exponential back-off as it tries to send the data again.

This algorithm uses implicit information to obtain the correct transmission of data with a very easy implementation; however, a number of drawbacks must be noted. First, fair operation depends upon each of the colliding senders performing the same back-off algorithm; if some nodes retransmit with a smaller interval, they will obtain a greater share of the resources. In the worst case, a single malicious node can prevent all others in the system from communicating. Second, Ethernet works best in lightly-loaded conditions; typically, if utilization exceeds 30%, much of the network capacity is wasted by collisions.

TCP Congestion Avoidance

The TCP/IP network protocol requires a different solution for *congestion avoidance* in wide-area networks; that is, to operate in the regime of low delay and high throughput. The challenges in such an environment are many. First, large-scale networks such as the Internet clearly require a decentralized approach. Second, the resource demands across senders are bursty, varying over time. Third, sending additional messages to describe the state of the network is unacceptable since they consume a portion of the resource they are supposed to allocate. Finally, there are no defined interfaces for transmitting feedback between nodes.

In the well-known TCP congestion control algorithm by Van Jacobson [80], clients leverage implicit information in the form of round-trip time and packet loss to infer the state of the network. The round-trip time average and variation are estimated from past measurements and help determine the expiration time of a local timer; if a given message response does not return before the timer expires, the packet is labeled as dropped and must be retransmitted. The key insight within the TCP algorithm is that packet loss in wired systems is almost always due to network congestion; when the network is congested, clients should reduce the amount of data that they are trying to send. As a result, when a dropped packet is detected, the TCP algorithm multiplicatively reduces its current window size. This congestion avoidance algorithm has prevented congestion collapse in the Internet since its inception.

As noted by Jain [81], by observing only round-trip time and packet loss, the network is treated as a *black-box*:

> "Black-box schemes have no *explicit* feedback and are therefore also called *implicit* feedback schemes." (emphasis in the original)

Jain notes further that implicit feedback increases the amount of available information, and is therefore useful even in networks that already have explicit feedback.

However, due to the limit of control that can be accomplished from the edges of the network, the TCP congestion avoidance mechanisms are not sufficient to provide good service in all circumstances. Therefore, a complementary protocol has been recently proposed for routers in the Internet. With *Random Early Detection (RED)* [59], routers manipulate implicit information in order to implicitly control the TCP congestion avoidance algorithm. By randomly dropping packets as a function of the average queue length of packets, routers can signal senders to reduce their sending rate. As long as packet flows are responsive to the implicit information inherent in congestion, RED is expected to help increase utilization of the Internet.

Deadlock Detection and Recovery

Deadlocks occur in networks when packets cannot advance toward their destinations because resources held by other packets are requested in a cyclic pattern. Traditionally, networks using worm-hole routing have relied upon *deadlock avoidance* strategies in the design of routing algorithms, which limits the routes that packets can travel. Alternatively, *deadlock recovery* strategies can use fully adaptive routing and thus potentially outperform techniques using deadlock avoidance.

To detect deadlock, the entire graph formed by the messages must be examined for cycles. When a cycle is found, one of the messages must be killed to break the cycle. A simple, implicitly-controlled, deadlock detection mechanism exists that uses a set of heuristics and local information provided by the existing flow-control signals on each router [106]. To ensure that a small number of messages (ideally, one message) in the cycle are flagged as deadlocked, the root message first must be identified, where the root is the last message of the cycle to arrive.[1] This algorithm identifies those messages that are blocked on the root with only local information by marking messages that were initially waiting for busy, but non-blocked channels, that later became blocked. To determine this state transition, routers observe implicit information in the form of the rate of progress of each message. To remove the cycle in the graph, the messages that are blocked on the root message are killed.

2.4 Summary

Building services from multiple, interacting components is a complex task. To interact with one another in a cooperative manner, components require information about the state and behavior of other components; however, components do not inherently have this perfect knowledge. Explicitly contacting other components to obtain this information has a number of drawbacks: namely, the desired information may not be available, errors

[1] In the case of simultaneously arriving "root" messages, multiple messages in a single cycle can still be marked for deadlock recovery.

may occur, the cost may be prohibitive, critical resources may be consumed, or the information may be out-of-date by the time it is received. Thus, the construction of cooperative systems can be greatly simplified if implicit information is employed.

Most work in distributed problem solving has focused on making effective decisions given available information, not on the sources of that information. In fact, in the words of Casavant and Kuhl [29]

> "the components of a distributed computation must *explicitly* exchange messages to share information regarding total state" (emphasis in the original).

In this chapter, we have argued that *implicit information*, or information from observing characteristics of local events outside of their defined interface, helps components share information regarding the total state of the system. In an *implicitly-controlled* system, rather than explicitly contacting other components to obtain information, components infer remote state by observing naturally-occurring local events. Rather than obey commands from a component with complete control over the system, each autonomous component reacts dynamically and independently to the forces that are influencing it; often, the components rely on some common knowledge to advance the system toward a shared goal. Implicit information is particularly useful in such circumstances because it provides additional information about the state of the system at little or no additional cost.

It is our belief that implicit information is particularly suited to peer-to-peer systems containing autonomous components. In such systems, no component knows the state of the entire system; thus, each component is responsible for optimizing the problem from its perspective. When components are completely autonomous, it is likely that some components will only support the minimum interface required for implementing the service (*i.e.*, those parts of the service inherent in the transfer of data). Rather than degenerate to the capabilities of the "least common denominator" of the system, smart components can leverage implicit information that emanates even from components conforming only to the minimum specification. Thus, we predict that implicit information will become more prevalent and essential as distributed systems increase in scale and complexity.

In this chapter, we have described a number of systems that leverage implicit information or that are implicitly controlled. The most developed example is the congestion avoidance algorithm in TCP/IP [80, 81]; a proposal even exists for having routers manipulate the implicit information to which TCP reacts [59]. Many of the examples of implicit information and implicit control occur in networked systems, but examples can be found in many systems containing multiple components or modules.

Chapter 3

Scheduling Background

The work in this dissertation builds from research performed in a number of different areas within distributed systems and multiprocessor scheduling. To set the context of implicit coscheduling, we discuss some of these areas in this chapter. We begin by describing the convergence of distributed and parallel systems into today's networks of workstations (NOWs). In the next section, we describe the requirements that this new environment and its evolving workloads place on the scheduling policy. We believe that a mixed approach of *space-sharing* and coordinated *time-sharing* provides the best throughput and response-time for general-purpose workloads. We discuss two popular approaches for time-sharing: *local scheduling* and *explicit coscheduling*. We analyze the drawbacks of both approaches for networks of workstations and general-purpose workloads, as well as some of the current proposals for reducing these weaknesses.

3.1 Evolution of Clusters

Networked computers have been a continually evolving platform since the early 1980s. In this section, we briefly describe how advances in communication technology have instigated the convergence of loosely-coupled *distributed systems* and traditional *massively parallel processors* (MPPs) into today's tightly-coupled clusters.

The earliest collections of machines in the 1980s were usually loosely-coupled distributed systems with relatively poor network performance. The workloads in such environments tended to consist of interactive jobs executing on a user's desktop machine plus computationally-intensive jobs allocated to idle, remote machines. Due to the expense of finding available remote cycles and of migrating jobs if a workstation's owner returned, desktop systems usually performed only batch-processing of long-running sequential applications. A large proliferation of operating systems, run-time environments, and languages were created to support such computation.

The specific scheduling problems that were solved in distributed systems included providing transparent remote execution and load balancing with migration. Early research efforts solved these problems by implementing a distributed operating system tailored for a specific hardware platform; examples of such operating systems include V [162], Nest [2], Butler [130], Sprite [44], Amoeba [123], Amber [32], Eden [14], and Clouds [41], Accent [142],

Charlotte [58], and Locus [140]. More recent research efforts have implemented specific functionality on top of the vendor-supplied operating system running on each machine in the distributed system. Many of these run-time environments are still in use today and some systems have been incorporated into commercially available software; for example, Condor [104] now forms the basis of IBM LoadLeveler, while Utopia [178] has become the Load Sharing Facility (LSF) from Platform Computing.

Parallel programming in distributed systems did not become popular until the 1990s. Part of the reason for this slow evolution has been the focus on using idle cycles of workstations located on user's desks [1, 7, 10, 49, 100, 125, 163]. Furthermore, the shared-medium Ethernet interconnections of the past and the high-overhead of standard messaging protocols were prohibitive to all but the most coarse-grain parallel applications.

Recent advances in low-latency, high-bandwidth switches [20] and low-overhead message-passing software [167, 110] have enabled communication performance to more closely resemble that of massively parallel processors (MPPs). By connecting these switches and commodity workstations (or personal computers), one can build incrementally-scalable, cost-effective, and highly-available shared servers [4, 11, 19, 37, 75, 143, 154, 175]. We use the term *clusters* to refer to these collections of more tightly-coupled machines.

Over the years, a number of parallel programming languages and run-time environments have been developed for distributed systems. Some of the earliest programming environments focus on coarse-grain parallel applications; for example, p4 [108], Data-parallel C [127], PVM [63], PARFORM [28], and MPI [161]. Later systems use objects which can be migrated between idle nodes and invoked regardless of location; for example, Emerald [84] and CHARM [146]. Finally, the most sophisticated systems, developed more recently, adjust the level of parallelism in the application to dynamically match the number of available machines, such as Piranha [64], Cilk [17, 18], and CARMI (Condor Application Resource Management Interface) [141].

Clusters are capable not only of running fine-grain parallel applications written in traditional parallel programming languages, but also of supporting multiple users running workloads typically reserved for large shared-memory servers. These workloads are expected to contain compute-bound, I/O-bound, interactive, and multimedia jobs that may be serial, client/server, or parallel. Therefore, new techniques are required for clusters to fairly and efficiently schedule this general-purpose, developmental workload. In the next section, we present the requirements of the cluster operating system scheduler.

3.2 Requirements

For clusters to be attractive to a wide group of users, every service in the system should have the following four characteristics:

- **Autonomous:** As the size of the cluster grows, different organizations are likely to own different components of the cluster. Since organizations tend to want to maintain control over their resources, each component should determine its own actions, participating in the system because it is in its best interest to do, not by imposition.

- **Reconfigurable:** Workstations should be able to join and leave the cluster dynamically without restarting any system services.
- **Reliable:** The system must tolerate the failure of any node; while applications running on the failed node may not survive, the system as whole must continue.
- **Scalable and Absolute Performance:** Implementing a service in a distributed environment should not degrade the absolute performance of the service on a single node; further, the performance and the correct operation of the service should scale through the size of the system.

After the basic functionality is supported, the service should be implemented such that it provides fair and efficient performance across users and jobs. When the service is the scheduling of long-running communicating processes, this criteria can be more precisely defined.

- **High Throughput:** Achieving high throughput for frequently-communicating parallel applications has been shown to require that communicating processes are scheduled simultaneously across workstations.
- **Fair Allocation:** The amount of resources allocated to a job should be independent of its communication behavior. Likewise, since the cluster is a shared environment with many competing users, resources should be allocated across users fairly, independent of the number of jobs each user runs.

The primary goal of implicit coscheduling and of this thesis is to obtain fair, coordinated scheduling for communicating processes in traditional parallel applications. However, scheduling a general-purpose workload with client-server, I/O-bound, and interactive processes has a number of additional requirements.

- **Dynamic Identification:** Communicating processes must be identified at run-time to support client-server applications and processes that communicate with a wide set of processes over their life-time. Static identification is suitable only for traditional parallel applications.
- **Fast Response Time:** Interactive jobs should receive quick response time, while negligibly harming the throughput of long-running parallel jobs.
- **Efficient Resource Usage:** Jobs that are waiting for events (*e.g.*, disk or user I/O) should relinquish the processor in order to not waste resources.

In the next section, we describe in detail the ability of various scheduling approaches to meet this collection of design and performance goals.

3.3 Cluster Scheduling

The scheduling of parallel jobs in a multiprogrammed environment has long been an active area of research. The scheduling of multiprogrammed, parallel workloads is a challenging problem because performance is highly dependent upon many factors: the workload, the parallel programming language, the algorithm processes use when waiting for communication or synchronization to complete, the local scheduler, and the machine and network architecture. For the interested reader, Feitelson has written a comprehensive summary of research in this area [52].

Parallel scheduling is usually decomposed into two interdependent steps. The first step determines the number of processors to allocate to a parallel job and the placements of processes on those processors. The second step dispatches those allocated processes over time.

A hybrid approach combining space-sharing and coordinated time-sharing of competing processes has a number of advantages over pure space-sharing with one job per processor. The most significant advantage of time-sharing is that no changes to the programming model are required for good performance; in particular, the popular Single-Program-Multiple-Data (SPMD) programming style with message-passing can be used. Pure space-sharing techniques that achieve good response time and throughput require that applications are malleable to the number of available processors, a non-trivial programming task.

A large number of studies have focused on the allocation step of parallel job scheduling [31, 33, 66, 70, 101, 118, 119, 120, 126, 139, 149]. In this dissertation, we focus on the second step of time-sharing processes over time. Two popular methods exist for time-sharing competing processes: *local scheduling* and *explicit coscheduling*. When processes are locally scheduled by the operating system on each workstation, frequently communicating processes exhibit unacceptably poor performance. Explicit coscheduling improves the performance of communicating processes by coordinating scheduling across workstations. We discuss these two previous approaches in detail, as well as a recent third proposal, *dynamic coscheduling*.

Numerous studies have compared the performance of explicit coscheduling to a variety of space-sharing techniques along several different metrics. In general, coscheduling gives better response time than space-sharing, especially for jobs with large resource requirements [53, 173] or performing I/O [144]. Studies have also shown that coscheduling is a relatively good policy for system throughput [31, 35, 70, 101, 105]. Performance of coscheduling can be further improved if processes can be migrated between nodes to better fill available time slots [148, 173]. Finally, anecdotal evidence suggests that system utilization can be improved with coscheduling: when the scheduler on the LLNL Cray T3D was changed from variable partitioning to gang-scheduling, the average utilization nearly doubled from 33% to 61%; additional tuning led to weekly utilizations above 96% [82].

3.3.1 Local Scheduling

With local scheduling, the operating system scheduler running on each workstation in the cluster independently schedules the processes that have been allocated to it; a process

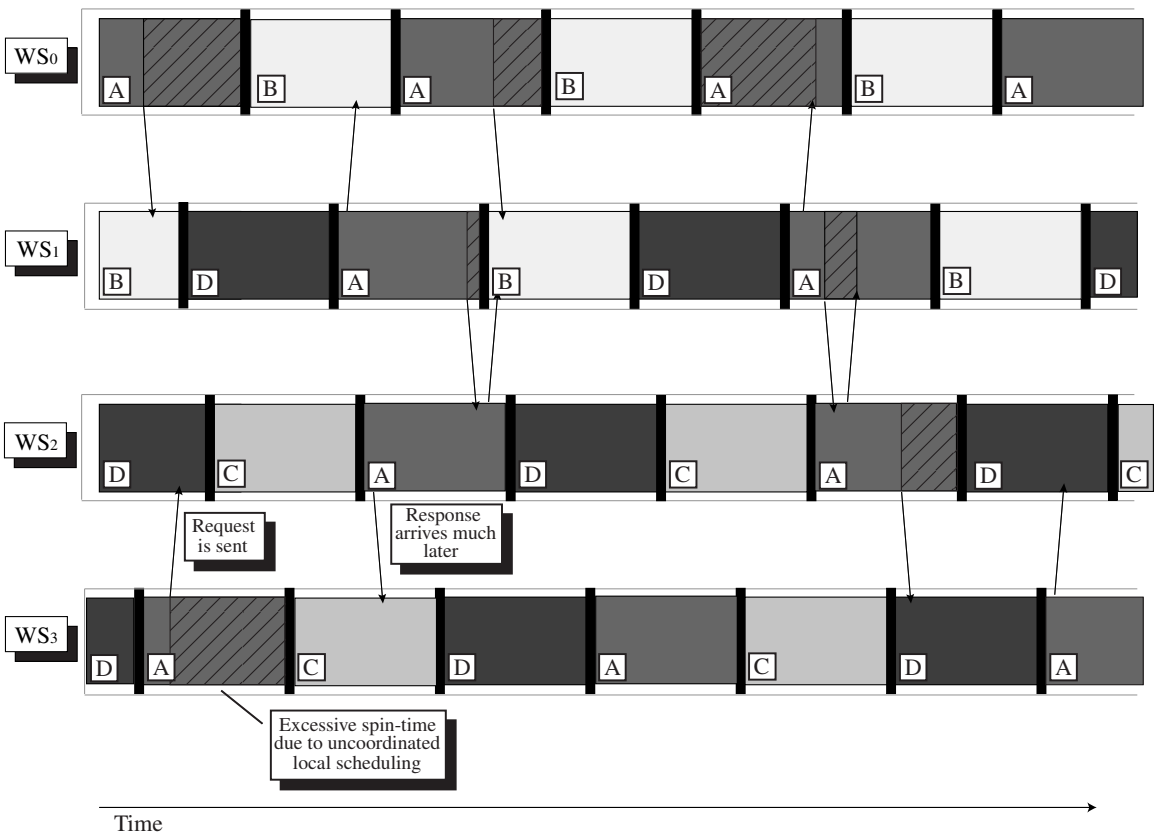


Figure 3.1: **Local Scheduling.** *Four communicating jobs (A, B, C, and D) are allocated to four workstations. If the processes composing those jobs are scheduled by independent operating system schedulers on each workstation, then the communicating processes are not scheduled simultaneously across workstations. As a result, if a process spin-waits after sending a reply message until the reply arrives, CPU time will not be used effectively.*

that communicates with other processes in the system is scheduled as if it were completely independent. The problem with local scheduling is that *fine-grain* applications experience poor performance because their scheduling is not coordinated across workstations. We loosely define a fine-grain application as an application that communicates frequently (*i.e.*, on the order of once every $100\mu s$) and uses small messages. An example of local scheduling is shown in Figure 3.1, where one process sends a request message to a process on a remote node that is not currently scheduled. If the request requires a response, then the sending process must wait (a potentially long time) until the destination process is scheduled for the response to be returned.

A process waiting for a response from a remote node has three options for waiting. First, the process can *spin-wait* until the response arrives, handling incoming messages while it waits. 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* [134]: in the first phase, the process spin-waits some amount of time, S, for the desired response; if the response does not arrive, then the process blocks in the second phase.

To determine the optimal waiting algorithm, the system is typically modeled as follows. While the process spin-waits, it pays a cost in wasted processor cycles equal to the time that it spin-waits. If the process blocks, it pays the fixed cost of a context-switch to be scheduled again when the event completes. Under this model, the optimal behavior is for the process to spin-wait if the completion time of the operation is less than the time for a context-switch; otherwise, the process should block. Since waiting times are typically large with uncoordinated local scheduling, previous researchers have found that blocking immediately has the best performance. We discuss these performance results for the three waiting algorithms in more detail.

Spin-Waiting

To measure the performance of local scheduling with spin-waiting on parallel applications, we use a technique called *direct simulation* on a 64-node Thinking Machines' CM-5 [98]. Direct simulation measures the run time of a parallel program executing under artificial conditions that emulate a multiprogrammed environment. In our experiments, the processes of a single parallel program are periodically interrupted and forced to execute an idle loop; the interruptions occur independently across the processes. To the parallel program, these disturbances appear as competing processes using a time-slice. Messages which arrive in this interval are buffered in a small, fixed-size buffer until the parallel process is resumed. By measuring the execution time of the parallel application, we can observe the slowdown induced by the simulated competing processes.

We composed a suite of five parallel applications, written in Split-C [39] for direct simulation. The first, `cholesky`, performs LU factorization on symmetric, positive-definite matrices. `Column` is an implementation of the column-sort algorithm [48, 97]. The program `em3d` simulates the propagation of electro-magnetic waves through objects in three dimensions on an unstructured mesh [39]. Another sorting algorithm is implemented in `sample` sort [16, 48]. Finally, `connect` uses a randomized algorithm to find the connected

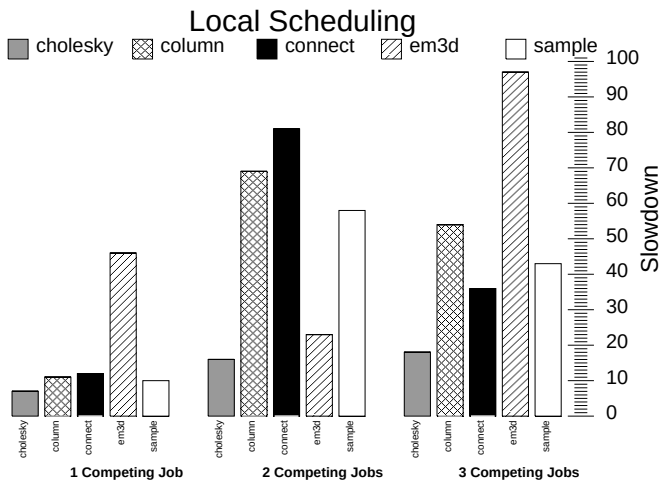


Figure 3.2: **Simulated Performance of Local Scheduling versus Coscheduling.** *One real parallel application and one to three simulated parallel jobs are scheduled in a round-robin fashion with a 100ms time quantum on a 64-node CM-5. Slowdown is the ratio of the measured locally-scheduled execution time to the ideal coscheduled time.*

components of a graph [107].

Figure 3.2 shows the slowdown of scheduling each application in a round-robin fashion while varying the number of simulated competing jobs between one and three. The reported slowdown is relative to the execution time of the applications when ideally coscheduled. Our results show that the performance of locally-scheduled parallel applications is incredibly poor with spin-waiting. When resources are shared between two parallel applications, each application is slowed down by at least a factor of eight. Even the applications that communicate infrequently, such as `cholesky` and `column`, exhibit poor performance with local scheduling and spin-waiting. The application that synchronizes the most frequently, `em3d`, is slowed down the most significantly, *i.e.*, by nearly 50 times.

As the number of competing parallel applications is increased from two to four, the execution times increase for two reasons. First, when communicating with a destination process, the likelihood that the destination process is not scheduled increases with more competing processes. Second, if a destination process is not scheduled, then the time remaining before that process is scheduled increases.

In summary, spin-waiting interacts very poorly with processes that are locally scheduled [7, 35, 55, 99, 109]. When the scheduling of processes is not coordinated, processes often do not receive the response from the remote node until their next time-slice. As a result, the process spins idly for the remainder of its time-slice without making progress. With spin-waiting, communication often completes in the same amount of time as the duration of a time-slice, rather than in the inherent round-trip time of the network.

Block Immediately

When processes are locally scheduled and block-immediately when waiting for a response to a communication request, communication tends to progress at the rate of a context-switch, instead of a time-slice. Thus, the performance of local scheduling can be improved significantly when processes block immediately rather than spin-wait for communication [35, 55].

For frequently communicating and synchronizing applications, the performance of local scheduling with immediate blocking is still much worse than with coscheduling and spin-waiting. However, for applications that communicate infrequently, immediate blocking with local scheduling may actually be superior to coscheduling, depending upon the amount of time communication operations require to complete. In general, when the waiting time for an event is high relative to the cost of relinquishing the processor, better system throughput is achieved if processes block immediately.

Previous research has investigated the trade-off of blocking-immediately and spin-waiting on applications performing only barriers [55, 56]. In their analysis, the waiting time of the barrier is strictly the amount of *load-imbalance* in the application, where load-imbalance is the time between the arrival of the first and last process to reach a barrier synchronization; the cost of relinquishing the processor is equal to the time for a context-switch. With these assumptions, when the load-imbalance is less than the context-switch cost, processes should be coordinated (*e.g.*, with explicit coscheduling) and spin-wait; otherwise, processes should be scheduled independently and block immediately.

Nevertheless, in most well-tuned applications on modern networks, the waiting time at a communication operation is significantly less than the context-switch cost of the local operating system. Subsequently, processes that communicate frequently should be scheduled in a coordinated manner and should stay scheduled while waiting for communication operations to complete.

Two-Phase Waiting

Since the waiting time with local scheduling is not known until after the operation completes, determining how a process should wait for an event to complete is an on-line problem. Competitive analysis is an attractive technique for bounding the worst-case performance because it requires no knowledge of the input distribution. Theoretical results that have applied *competitive analysis* to two-phase waiting have found that when the spin-time in the first phase equals the penalty of blocking then the worst-case performance is within a factor of two of the optimal off-line algorithm given the same input distribution [86]; that is, the algorithm is *competitive* with a factor of two.

The reasoning for this is simple. If the waiting time, C, is less than B, then the cost of the on-line strategy and the optimal off-line strategy are equivalent: each idly spins for C, wasting processor resources. If the completion time is greater than B, then the on-line algorithm pays a cost of $2B$ while the off-line algorithm blocks immediately and pays only B.[1]

[1]This result holds regardless of the power of the adversary that picks the observed waiting times: whether against an oblivious adversary (one who constructs the sequence of events based on a description of the be-

However, we observe that competitive analysis is not an appropriate technique for cooperating processes because the input distribution of waiting times is not fixed: the waiting times can be impacted by the behavior of the waiting process. Since two different waiting algorithms may produce different distributions of input values, the performance of the two algorithms cannot be compared competitively. Specifically, a waiting algorithm that spin-waits for the penalty of blocking before relinquishing the processor is not guaranteed to perform within a factor of two of the optimal algorithm.

However, because experimental results have not clearly contradicted the theory, one may be misled into thinking that the adversarial assumptions of a fixed distribution hold. A common application of two-phase waiting is when processes compete for shared locks in a multiprocessor [70, 85, 102]; however, even this mundane scenario violates the adversarial assumptions of competitive analysis. Since different behavior by the waiting process can cause processes to be scheduled in different orders, the relative timing of critical sections and thus waiting time for future locks may be altered.

Measurements performed by Karlin et. al. [85] verify that different waiting algorithms produce different lock-waiting times; this is shown in two ways. First, their analytic results on a fixed distribution of waiting times do not match their experimental results in which the waiting algorithm can alter the future inputs. For example, in their work, given a fixed input distribution, the `Hanoi` application performs better with a spin-time of half the context-switch cost than with a spin-time equal to the context-switch cost; in the real environment with feedback, the performance is reversed. Second, spinning for the context-switch cost does not always perform within a factor of two of the optimal algorithm. Our interpretation of their experimental data indicates that with the optimal algorithm, only 10 seconds are spent in synchronization, but with a spin-time equal to the context-switch cost, 52.7 seconds are spent.

One of their hypotheses for these discrepancies matches ours: the waiting strategy changes the relative timing of events across threads and subsequently the lock-waiting times. However, in this environment, it is unlikely that the waiting process can systematically bias or even predict the impact of its behavior on future waiting times.

Little attention has been paid to the use of two-phase waiting for processes communicating with one another in a distributed environment such as ours. In one study [56], Feitelson and Rudolph find that local scheduling and two-phase waiting with a spin-time equal to the local context-switch cost performs poorly relative both to local scheduling with immediate blocking and to explicit coscheduling with spin-waiting; performance is nearly two times worse than with immediate-blocking and ten times worse than with coscheduling.

In this environment, two-phase waiting was not found to substantially affect the distribution of waiting times. Due to their assumptions about the local operating system scheduler, the application workload, and the amount of spin-time, scheduling is always uncoordinated and waiting times remain high. Thus, few of the communication operations complete while the process is spinning in the first phase; as a result, for each communication operation, the sending process first spins for a time equal to the context-switch and then

havior of the algorithm), an adaptive on-line adversary (one who generates an event based on the algorithm's reaction to previous events), or an adaptive off-line adversary (one who generates events based on the algorithm's reactions to all events).

relinquishes the processor, paying another context-switch. The performance of two-phase waiting is competitive within a factor of two of blocking immediately, which is the optimal algorithm given this distribution of high waiting times.

However, Feitelson and Rudolph find that two-phase waiting with local scheduling performs almost ten times worse than with coscheduling and spin-waiting for many fine-grain applications. Two-phase waiting does not have competitive performance relative to coscheduling because the distribution of waiting times is not the same for the two algorithms. Therefore, the performance of two-phase waiting with uncoordinated scheduling cannot be compared to that of spin-waiting with coordinated scheduling.

As we discuss in Chapter 4, the key to good performance with local scheduling and two-phase waiting is to achieve coordinated scheduling. With coordinated scheduling, the waiting algorithm will react to smaller waiting times [176, 177]. With smaller waiting times, spinning for some amount of time before blocking improves performance relative to blocking immediately.

3.3.2 Explicit Coscheduling

To improve the performance of fine-grain parallel applications, *coscheduling* ensures that communicating processes are scheduled simultaneously across different processors [134]. Performance of fine-grain applications is improved relative to local scheduling, because when one process communicates with another, the destination process is scheduled and can promptly reply. Since the parallel job is given the impression that it is running in a dedicated environment, processes usually spin-wait for responses.

Implementations of explicit coscheduling require that a global context-switch is performed simultaneously across processors; this ensures that communicating processes are scheduled simultaneously across workstations. Ousterhout suggested three algorithms for grouping communicating processes: a matrix algorithm, on which most implementations are based, and two algorithms based on a sliding window. The matrix algorithm is shown in Figure 3.3 for four parallel jobs (A, B, C, and D) allocated on four workstations. Within the matrix, each row designates a different workstation and each column a new time-slice. When allocating a parallel job with P processes, a column with P empty slots must be found. In the remainder of our discussion, we assume that the matrix algorithm is used.

If the communication layer of the system cannot support the case where a message destined for one process arrives when another process is scheduled, then a variant of coscheduling is required: *gang-scheduling*. With coscheduling, the operating system may schedule any subset of processes at any given time; with gang-scheduling, the scheduling requirements are more strict in that only complete sets of communicating processes are scheduled at all times. In this dissertation, we assume that the communication layer can handle multiple communicating processes, and therefore consider forms of coscheduling.

Coscheduling Implementations

Coscheduling has a number of practical, attractive qualities, as reflected by the large number of implementations on MPP systems, shared-memory multiprocessors, and networks of workstations. Some of the implementations of coscheduling were supplied by

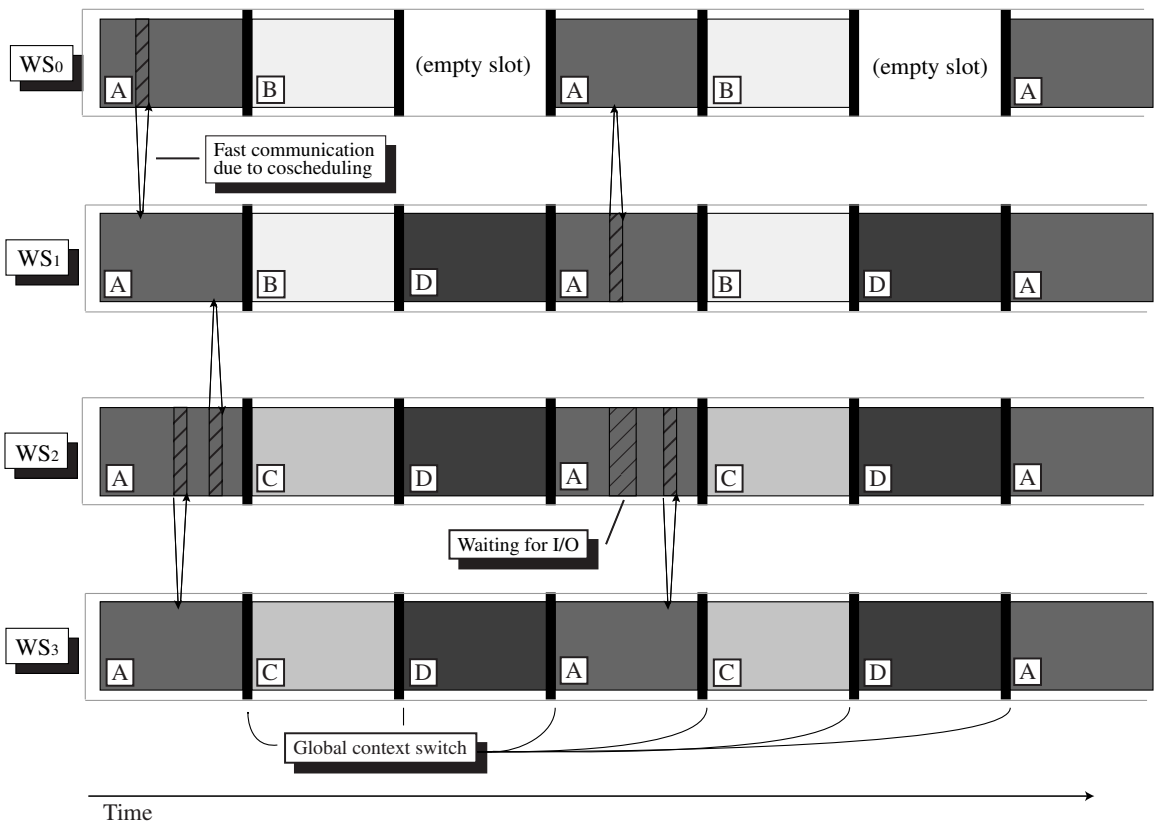


Figure 3.3: **Explicit Coscheduling Matrix.** *Four parallel jobs (A, B, C, and D) are allocated to four workstations. With explicit coscheduling, or gang scheduling, the system guarantees that communicating processes are scheduled simultaneously across workstations. As a result, when network latency and load-imbalance are low, processes can spin-wait for message replies without wasting significant amounts of CPU. However, to scheduled processes under such constraints, empty slots may exist in the global schedule where no process can run simultaneously with the other processes in its job. Further, processes must spin-wait when they perform I/O, not allowing another process to perform useful computation.*

the commercial vendors responsible for the operating systems, while others were added later by researchers desiring better performance. For example, MPP systems with gang scheduling include: the Concentrix Alliant FX/8 [160], the Connection Machine CM-5 from Thinking Machines [98], the Cray T3D at Lawrence Livermore National Laboratory [82], the Political Scheduler for the Cray T3E [91], the Meiko CS-2, OSF/1 on the Intel Paragon, and SHARE on the IBM SP-2 [62]. Shared-memory systems with coscheduling support include Medusa on CM* from Carnegie Mellon University [134], the BBN Butterfly at the University of Rochester [35], MAXI on the Makbilan multiprocessor at the Hebrew University of Jerusalem [55], the BBN TC2000 [82] at LLNL, and IRIX on the family of Silicon Graphics multiprocessor systems. Finally, a number of research projects have implemented coscheduling on clusters: SCore-D (or DQT) from Real World Computing on clusters of SparcStation 20s and Pentium PCs connected with Myrinet [76, 77, 78]; GLUnix from U.C. Berkeley on UltraSPARCs with Myrinet [65]; and as part of the SHRIMP project from Princeton on Pentium PCs [133].

The overhead for a global context-switch in each system depends upon a number of factors. On small-scale shared-memory machines, global context-switch overheads are comparable to those on a single processor. For example, the global context-switch requires only $500\mu s$ on the 16 processor BBN Butterfly [35] and $200\mu s$ on the 15 processor Makbilan multiprocessor [55]; as a result, time-slices near $100ms$ can be used effectively. However, even implementations on shared-memory machines are not always scalable: a global context-switch on the 126 processor BBN TC2000 at LLNL requires on the order of milliseconds; therefore, the system uses a ten second time-slice [82].

The global context-switch time on MPPs tends to be much larger than a context-switch on a single processor due to the complexities of the distributed environment, the larger scale of processors, and (in some cases) the need to flush the network of messages for strict gang-scheduling. For example, on the CM-5, between 5 and $10ms$ were required to switch between processes [27]; as a result, time-slices near one second were used. To amortize context-switch costs on the Intel Paragon, time-slices could be up to 24 hours long [79]. Machines which do not support virtual memory paging have even higher overheads; for example, the LLNL implementation on the T3D must swap the entire resident process on every context-switch, requiring about one second per preempted processor, or about one minute for a 64-processor job.

The implementations of coscheduling on networks of workstations often rely upon the operating system on each local node to perform the global context-switch. One common technique is to use UNIX signals to stop and start the processes as desired. As a result, the time for a context-switch is relatively large and increases with the number of workstations. For example, in SCore-D, overheads of $3ms$, $15ms$, $30ms$, and $45ms$ were measured on 1, 2, 4, and 8 processors, respectively, running Solaris with a $200ms$ time-slice [78]. A later implementation reduced context-switch overhead to $45ms$ on 36 SparcStation 20s and $12ms$ on 32 Pentium PCs [77]. A large portion of the global context-switch time is due to the cost of preempting the network to ensure that no messages are in transit. Other implementations, such as GLUnix [65], which rely upon the communication layer to separate messages across processes, should have lower overheads.

Despite its popularity and its advantages, explicit coscheduling has a number of

disadvantages. Some of the drawbacks concern its construction: a master-slave implementation severely limits the ability of coscheduling to scale with more workstations, to adapt to reconfigurations of the participating workstations, to tolerate faults, and to provide autonomous operation. The more severe concern is its ability to handle general-purpose workloads, in particular, client-server applications, interactive processes, and jobs that page or perform I/O. We now discuss each of these disadvantages in more detail.

Scalability, Reliability, Reconfiguration, and Autonomy

Many explicit coscheduling implementations are constructed such that a master process running on a single processor controls the timing of the global context-switch across the system. The control structure of a centralized master has a number of inter-related drawbacks.

First, the master may form a performance bottleneck as the number of nodes in the system grows. As the scale of the system increases, the time to propagate the global context-switch across all processors is expected to increase. Also, since the master must be aware of the state of the entire system, adding more jobs or more nodes may make computing the optimal schedule more difficult.

Second, the master represents a single point-of-failure in the system. Such a design may have been acceptable in a traditional supercomputer, where all components could be placed in a controlled environment to minimize failures. However, the operation of an entire cluster cannot be dependent on a single machine.

Third, current implementations of explicit coscheduling assume that the nodes in the cluster remain static over the lifetime of the service. In a mature cluster, nodes must be able to enter and exit the system while taking full advantage of system services. Finally, the nodes in the coscheduling matrix must act as slaves to the master, running only the designated parallel job.

Distributed Hierarchical Control (DHC) [54] is an implementation scheme that improves the first two problems. In DHC, machines are grouped into a hierarchical tree structure, where masters at higher levels control context-switches across larger groups of machines. Scalability is improved, because no component requires full knowledge of the system. Presumably, subtrees within the system are robust to failures in other portions of the system.

However, even with DHC, machines cannot act on their own accord, but must 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. Nodes may also not act in their own best interest by running sequential applications. While this approach may be acceptable in systems that are under the same administrative domain, it is not feasible as clusters expand to multiple sites in the wide-area. Therefore, DHC does not satisfy our goal of autonomy for each machine.

Static Identification of Processes

Explicit coscheduling requires that communicating processes be identified statically, *i.e.*, prior to their execution. This requirement has three main drawbacks.

First, static identification requires that communicating processes be grouped pessimistically. That is, if two processes ever have a phase of joint communication, then they must always be coscheduled. If the number of such processes is greater than the number of processors, explicit coscheduling cannot be used, even though there may be sufficient resources to support the processes actively communicating with one another at any given time.

Second, client-server applications are not easily supported. When a client process is started, it may not know *a priori* which server processes it will contact and engage in communication. Furthermore, the collections of clients and servers may change dynamically over time.

Third, applications that link with libraries performing communication cannot be effectively coscheduled with traditional methods. For example, a traditional parallel application that leverages a parallel file system may consist of two independent threads located on each workstation in the cluster. The set of threads in the parallel file system may also be performing communication, which should be coscheduled separately from the application.

Clearly, communicating jobs should be identified at run-time, an idea originally proposed by Feitelson and Rudolf in 1992 and published in 1995 [56]. In their approach, the communication rates between sets of processes are recorded and those that exceed a threshold are marked as belonging to the same "activity working set" by a master process. These sets are then placed into a matrix schedule and explicitly coscheduled. While run-time identification fixes the problems of statically identifying processes, this approach is still not ideal: communication between processes must be explicitly monitored, a global schedule must be constructed for all processes by a master, and simultaneous context-switches must be coordinated across workstations. Run-time identification has been simulated, but not yet implemented.

Workloads with Interactive Jobs and I/O

For a network of workstations to act as a general-purpose compute server, a scheduling approach that can effectively handle a mixed workload is required. To most efficiently allocate the resources in the cluster, parallel applications should dynamically share workstations with interactive, sequential processes [45, 128]. However, few studies have examined the performance of explicit coscheduling with interactive applications or with applications performing I/O [111, 144].

Interactive jobs typically perform a small amount of computation after waiting a longer interval for input from the user. Similarly, I/O-bound jobs may perform minimal computation between phases of transferring data from disk. Therefore, while neither class needs to be scheduled often, both should be scheduled promptly when they have work to do. Prompt scheduling of interactive and I/O-bound jobs shortens response time and keeps the disk subsystem utilized, as desired. Priority-based preemptive schedulers have been specifically designed to handle such workloads [50, 67, 96].

Due to their disparate scheduling requirements, interactive and I/O-bound jobs do not mix well with parallel applications that are explicitly coscheduled. Clearly, an interactive user will not be productive when allocated a long time-slice in a fixed round-robin

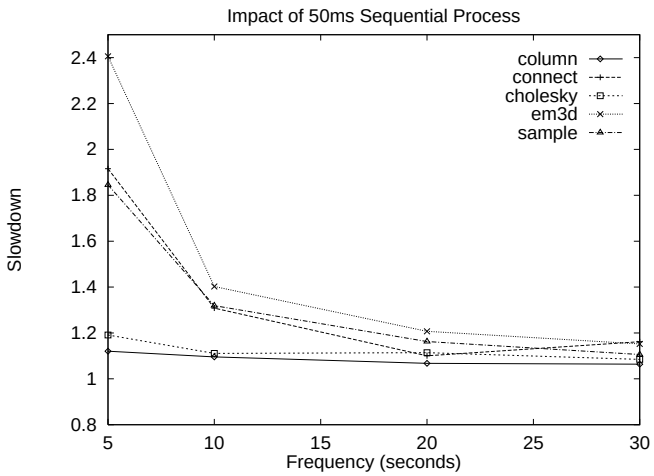


Figure 3.4: **Impact of Interactive Processes on Parallel Jobs.** *Each of the parallel applications is periodically interrupted at an random time on one of 64 nodes of the CM-5. The 50ms interruption simulates a short-lived process arriving on one of the workstations in the cluster. During this interruption, messages arriving for the parallel program are buffered and the sending process must spin-wait for a longer interval. Along the x-axis, the inter-arrival time of the interrupting processes on each workstation is varied between 5 and 30 seconds. The y-axis is the slowdown of the process with interruptions versus that in a dedicated environment.*

coscheduling matrix [73]. Studies have shown that coscheduling I/O-intensive applications can severely hurt performance, since the disk subsystem is not utilized in the time-slices when CPU-bound parallel jobs are scheduled [95]. Due to the fact that page faults are not correlated across processes of a parallel application [174], even parallel applications that page memory to disk can exhibit slowdowns when coscheduled [27].

On the other hand, the alternative approach of scheduling interactive jobs whenever they have computation to perform harms the performance of fine-grain parallel applications. To measure this effect on parallel applications that are coscheduled and spin-wait during communication, we again use direct simulation [7]. In these experiments on five Split-C applications, we simulate a sequential, interactive job being scheduled on one of the workstations by periodically interrupting one of the nodes for $50ms$. Figure 3.4 shows that most parallel applications measure a noticeable increase in execution time when another process is randomly scheduled. One application, `em3d`, is slowed down by more than a factor of two with only a $50ms$ interruption on each workstation every 5 seconds.

In summary, coscheduling does not mix well with interactive processes and applications performing I/O. If jobs are forced to run only during a predefined time-slice in the coscheduling matrix, the response time of interactive jobs and the throughput of I/O-bound jobs suffers. On the other hand, if interactive jobs or paging activity interrupt coscheduled applications, then the performance of fine-grain parallel applications suffers.

3.3.3 Dynamic Coscheduling

Due to the weaknesses of both local scheduling and explicit coscheduling, a new approach is needed that not only coordinates communicating processes across workstations, but also dynamically identifies communicating processes and allows workstations to autonomously react to the needs of a mixed workload. *Dynamic coscheduling* is a recent proposal that shares this set of goals [153]. In dynamic coscheduling, coordinated scheduling is achieved by scheduling the process for which an arriving message is destined, modulo certain fairness criteria. The major drawback to dynamic coscheduling is that it requires that the fairness criteria be selected by hand for each workload. It is also unknown whether dynamic coscheduling works for more than one communicating process per workstation.

The first models and simulations of dynamic coscheduling were performed at a coarse level of detail [153]. An analytical model of two competing jobs showed that if at least several hundred messages are sent to random destinations per time-slice, then the steady-state probability is that one of the two jobs is always coscheduled in its entirety. The major simplification of their model is that jobs spontaneously context-switch independently across workstations, representing both involuntary context-switches (*e.g.*, expiration of time slices) and voluntary context-switches (*e.g.*, blocking due to waiting for communication). Further, their model assumes that context-switches and communication events occur instantaneously.

An implementation of dynamic coscheduling with Illinois Fast Messages (FM) [136] is described for WindowsNT [25] and for Solaris 2.4 [152, 151] and coined FM-DCS. However, 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. The priority of the destination process is raised if the following equation holds:

$$2^E(T_{\text{Current}} - T_{\text{ThreadLastScheduled}} + C) \geq Q \cdot J,$$

where Q is the length of a time-slice, J is the number of jobs on the workstation, and E and C are tuned by hand for each workload. This implementation of FM-DCS appears 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.

The WindowsNT measurements are brief; the workload consists of a single ping-pong latency benchmark between a pair of processors [25]. More measurements are performed for the Solaris environment on seven SPARCstation-2 workstations: in addition to the ping-pong latency test, a barrier microbenchmark and a Laplace equation solver are examined. The authors find that FM-DCS performs better than local scheduling with two-phase waiting and a fixed spin time of $1.6ms$. However, their comparison to two-phase waiting and a fixed spin-time is not equivalent to a comparison of implicit coscheduling.

Therefore, while dynamic coscheduling is very similar to implicit coscheduling in its goals, the two approaches are somewhat different. Dynamic coscheduling focuses on the receiving process, while we will see that implicit coscheduling focuses on the sending process. With dynamic coscheduling, the remote process is forced to schedule processes whenever a message arrives for it and it is fair to do so. With implicit coscheduling, the sending process uses implicit information to infer the scheduling state of remote processes;

with this knowledge, it can optimize its own behavior as it waits for responses from remote processes.

3.3.4 Fair Allocation across a Shared Cluster

Finally, we review previous research in allocating a fair proportion of resources to competing users in a shared environment. Active users should not be able to obtain more resources by running more jobs or by running jobs with certain communication characteristics. While numerous researchers have investigated the problem of allocating a fair share of resources on a single workstation [60, 69, 72, 74, 87, 155, 170], little work has been performed for parallel jobs in clustered environments. For example, current run-time systems for fair allocation within clusters only support the space-sharing of communicating processes, with fairness enforced over relatively long intervals; in Condor, fairness is obtained by allowing users who have executed fewer jobs in the past to preempt users who have run more jobs [124].

Micro-economic systems can provide more precise allocation [57, 112]. In these systems, a server sells its available resources to the client that offers the highest bid. A major drawback to this approach is that, due to the high overhead of holding auctions, clients must bid for the exclusive space-shared rights to resources for the life-time of their processes. Thus, clients must estimate their run-time.

Due to the complexity of building micro-economic systems, very few systems have been implemented. One implementation, Spawn [169], reveals the high overhead of allocating resources by bidding: in a system with nine nodes, an auction takes roughly 6.2 seconds, forcing a 60-second time quantum to amortize the overhead. Furthermore, because the authors only consider coarse-grain applications, such as Monte-Carlo simulations, they do not provide a mechanism for allocating multiple nodes simultaneously to a single client. A more recent simulation study supports fine-grain parallel applications by allowing applications to purchase time on multiple nodes simultaneously [156].

3.4 Summary

In this section, we have discussed the requirements of the scheduling policy for general-purpose workloads on clusters of workstations. The scheduling strategy should be scalable, reconfigurable, reliable, and autonomous, as should any system service in a cluster. The most important criteria when time-sharing communicating processes is that cooperating processes are scheduled simultaneously across different workstations.

Without coordinated scheduling, processes that communicate at a frequent rate exhibit poor performance. The precise performance that is seen with local scheduling depends upon the waiting algorithm employed while the process waits for communication operations to complete. For example, when processes spin-wait at communication events, communication tends to proceed at the rate of a time-slice; when processes block-immediately, communication proceeds at the rate of a context-switch. Previous researchers who have studied the performance of two-phase waiting, where processes spin for a context-switch before relinquishing the processor, have found that performance is slightly worse than when

processes block immediately [56]. However, as we will show in Chapter 6, by choosing spin-time correctly, processes remain scheduled in a coordinated fashion.

Explicit coscheduling ensures that communicating processes are scheduled simultaneously across different processors [134]. Performance of fine-grain applications is improved relative to local scheduling, because when one process communicates with another, the destination process is scheduled and can promptly reply. Since the parallel job is given the impression that it is running in a dedicated environment, processes usually spin-wait for responses. Implementations of explicit coscheduling require that a global context-switch is performed simultaneously across processors; therefore, explicit coscheduling is difficult to build in a scalable, reconfigurable, or reliable manner. More importantly, workstations under the control of explicit coscheduling cannot act autonomously and cannot respond adequately to general-purpose workloads containing client-server, interactive, or I/O-intensive applications.

Chapter 4

Implicit Coscheduling Framework

Implicit coscheduling is an example of an implicitly-controlled system that leverages implicit information. In this chapter, we begin by describing our model for the system and its components. We then present the implicit information that implicit coscheduling leverages: message round-trip time and message arrival rate. For each piece of implicit information, we discuss the probable scheduling state on the remote nodes and the local action that will coordinate communicating processes.

4.1 Components of System

Networks of workstations contain multiple levels of components, each optimized to solve its own part of a task. Each workstation in the cluster behaves as an independent component and each workstation itself contains multiple components: the hardware itself, the communication software, the set of user processes and applications, and the local operating system scheduler. We describe each of these components in turn. The relevant parameters of our system are summarized in Table 4.1; the parameters for the workloads are shown in Table 4.2.

4.1.1 Machine Architecture

The lowest-level component in the cluster is the physical hardware that composes a *node*. Each node is a complete computer, consisting of a powerful microprocessor, cache, and large DRAM memory. In recent systems, these machines may be commodity workstations [4, 11, 19, 37, 75], commodity PCs [143, 154, 175], or commodity processors with special-purpose communication support [98, 90, 147].

Our current analysis of the performance of implicit coscheduling makes two assumptions about the machine architecture that are not necessarily true for all clusters. First, we assume a single processor per node (and, therefore, use the term node, processor, and workstation interchangeably). Second, we assume that each machine in the cluster is roughly identical in performance.

4.1.2 Message-Layer

Each workstation in the cluster is connected with a switched, high-bandwidth network, the topology of which is not specified and not pertinent to our analysis. Processes communicate with one another with a light-weight communication layer, such as Active Messages [167]. We use the LogP model [40] to describe the performance of the message-layer:

- L: an upper bound on the *latency*, or delay, incurred in communicating a message that contains a small, fixed number of words from the source node to the target.
- o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
- g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g is the available per-processor communication bandwidth.
- P: the number of processor/memory modules.

In all cases, 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. In our analysis and simulations, we assume that receiving processes are informed of message arrivals through *asynchronous interrupts*. Conversely, in our implementation, processes must *poll* the network explicitly to determine if a message is waiting; an interface does exist to wake a process waiting for a message arrival. We will see that this difference does not noticeably impact our performance results.

4.1.3 User Processes

A set of communicating processes is called a *job* or *application*, interchangeably. This job may be a dynamic collection of communicating processes (*e.g.*, a server and multiple clients) or a predefined, static collection (*e.g.*, a traditional parallel application).

We refer to the set of processes within a job as *cooperating* processes, while the others in the system are *competing* processes. We assume cooperating processes are placed on different nodes in the system, but competing processes may time-share the same node.

User jobs are the most important components in our system. In the end, it is the run-time of jobs that users care about, not the absolute performance of the machine hardware, the operating system, or the communication layer. For implicit coscheduling, the most important characteristics for determining performance are the communication primitives a job employs and the collection of jobs in the current workload.

Communication Primitives

In our model, a job contains four types of communication operations. These operations were chosen to closely match those operations found in programs with a global address space, such as those written in Split-C [39].

- **Request-response:** a round-trip message between a pair of processes (*e.g.*, a `read` or `write` operation). The requesting process must wait until it receives the response before it can proceed with its remaining computation. Note that, in the general case, the sending process may perform work or initiate additional requests while waiting for a response by separating the request from the later synchronization point, as in a split-phase operation (*e.g.*, separating the `get` and the `sync` in Split-C). Although our analysis ignores this intervening work (under the assumption that the amount of work is small), applications can still use split-phase operations with implicit coscheduling.

- **One-way Requests:** a one-way message sent to a destination process (*e.g.*, a `store` operation). The sending process can continue executing independent of the actions of the receiving process.

- **Synchronization:** messages synchronizing a set of cooperating processes. A participating process cannot continue until it has been notified that all cooperating processes have reached the synchronization point. In message-passing programs, such as those using MPI [161] or PVM [63], synchronization occurs between pairs of processes performing `sends` and `receives`. However, for simplicity in our analysis, we consider only synchronization operations across all processes within a job (*e.g.*, a `barrier`).

As initially discussed in Section 3.3.1, when a communicating process must wait for a remote condition before it can continue to make forward progress, it has three options. First, the process can *spin-wait* until the condition is true. Second, the process can *block immediately*, relinquishing the processor so that a competing process can be scheduled. Third, the process can use *two-phase waiting* [134]: with two-phase waiting, a process spins for some amount of time, S, and if the response arrives before the time expires, it continues executing. If the response is not received within the interval, the process voluntarily relinquishes the processor so a competing process can be scheduled. Thus, two-phase waiting is a generalization of the spin-wait and the immediate-block algorithms, where the spin times are $S = \infty$ and $S = 0$, respectively.

4.1.4 Application Workload

The set of J jobs running in the cluster compose the application workload. The important parameters for characterizing a job j in that workload are the following:

- **Processes:** The number of processes, p_j, in job j. The processes are numbered 0 to $p_j - 1$.

- **Communication Interval:** the average time, c_j, between request operations on the sending process.

- **Synchronization Interval:** The average time, g_j, between barriers across all processes.

- **Load-Imbalance:** the difference in arrival time across participating processes at a synchronization operation. More precisely, $v_{i,j}$ is the load-imbalance observed by

process i of job j at a given synchronization operation; *i.e.*, $v_{i,j}$ is the difference in arrival time between the slowest process of job j and process i. V_j is the worst load-imbalance observed across all processes, *i.e.*, $V_j = \max(v_{0,j}, v_{1,j}, ..., v_{p_j-1,j})$ or the difference in arrival time between the slowest and the fastest process.

When no possibility for confusion exists, the subscript j is dropped from each of the parameters.

While not an essential requirement for implicit coscheduling, we assume that each process contains a fixed amount of work irrespective of how the processes are scheduled. Furthermore, the job is not expected to be malleable, or to adjust to the number of available workstations. Thus, if a job consists of p processes, it is allocated a constant $P = p$ workstations throughout its lifetime. We focus on two placements of processes to workstations: *identical placement*, where process 0 of each job is placed on workstation 0, process 1 of each job on workstation 1, and so forth through process $p - 1$ of each job on workstation $p - 1$; and *random placement*, in which a randomly-selected process from each parallel job is placed on each workstation.

4.1.5 Operating System Scheduler

Each workstation in the cluster runs its own copy of a local operating system. For implicit coscheduling, the only relevant service of each operating system is the component which schedules processes over time. We assume that the scheduler time-slices between competing jobs after some interval, Q. The details of this allocation depend upon the properties of the scheduler as described later in Chapter 5. Regardless of the policy, each process transitions through three scheduling states:

- **Scheduled:** The process is currently scheduled on the processor. This process will remain scheduled until one of three circumstances occurs: the process voluntarily sleeps, it is preempted by a higher-priority process, or its time-slice expires.
- **Runnable:** The process is waiting to be run while a competing process is scheduled. Runnable processes wait on the *ready queue*.
- **Sleeping:** The process cannot be scheduled because it is waiting on an event, such as a message from a remote process or I/O. Sleeping processes wait on a *sleep queue*. A competing process may be currently scheduled, or the processor may be idle. We also refer to this state as *blocked*.

The *context-switch* time is the cost of scheduling a new process. We denote the time required to schedule a process after a message-arrival as W. To simplify our terminology, we often use the phrase "context-switch" when we specifically mean the case where a process is woken and scheduled after a message-arrival.

4.2 Components of Implicit Coscheduling

The components in our system that implement implicit coscheduling are the operating system schedulers on each workstation and the communicating processes themselves.

Variable	Description
L	network latency
o	overhead
g	gap
P	number of processors
W	wake-up from message arrival
Q	duration of time-slice

Table 4.1: **System Parameters.** *The table summarizes the relevant network, machine architecture, and operating system parameters in our system.*

Variable	Description
J	number of jobs in workload
p_j	number of processes in job j
c_j	average interval between communication operations
g_j	average interval between synchronization operations
V_j	load-imbalance across processes in job
$v_{i,j}$	load-imbalance of process i

Table 4.2: **Workload Parameters.** *The table summarizes the relevant parameters to describe the workload running in the cluster.*

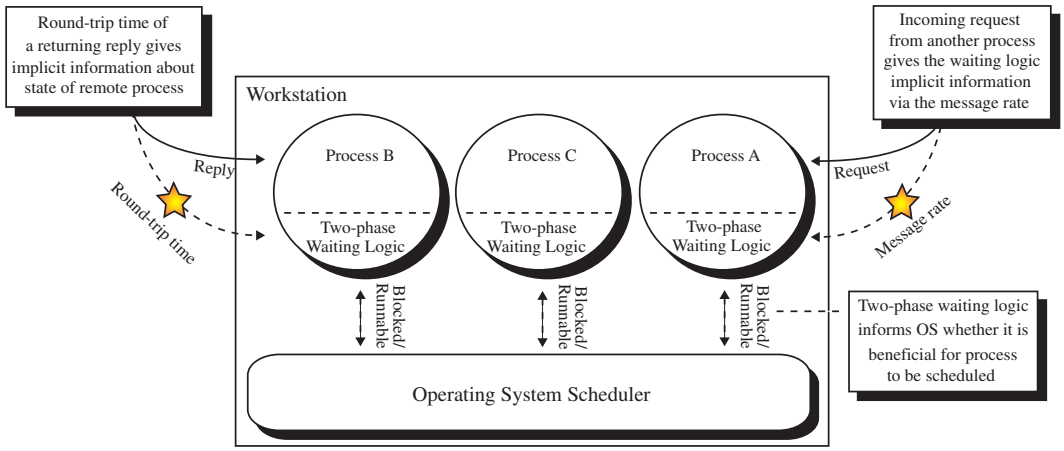


Figure 4.1: **Transfer of Information in Implicit Coscheduling.** *Processes that are part of a communicating job receive implicit control information from the processes they are currently communicating with. This information is used as part of the two-phase waiting logic within each communicating process. 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.*

The internals of these two components are described in Chapters 5 and 6, respectively. In this section, we briefly discuss the flow of information between these cooperating, but independent, components; this transfer of information is summarized in Figure 4.1.

4.2.1 Interaction between Processes and the Scheduler

From the perspective of the communicating processes, the primary responsibility of the local scheduler is to export a cost model. With this cost model, each process understands how frequently (or when) it will be scheduled as a function of its CPU usage pattern; that is, as a function of how frequently (or when) the process sleeps.

To determine if it is beneficial to be scheduled, each communicating process combines its knowledge of the scheduling state of the parallel job with this cost model. If the process determines that it is beneficial to run, then it places itself on the ready queue; this is the default state. If the process determines that it is not beneficial, it places itself on the sleep queue by sleeping on an event (*e.g.*, a message arrival); once the state of the system changes and the process determines that it is beneficial to run (*e.g.*, cooperating processes are scheduled on remote machines), the process wakes up and moves itself to the ready queue. By relying on only this small, well-defined interface between the process and the scheduler (*i.e.*, sleeping and waking), each component can independently optimize its performance in a modular fashion.

4.2.2 Interaction between Communicating Processes

This section presents the information available between sets of cooperating processes for determining the scheduling state of the parallel job. We show that each communicating process can observe response time and message arrival rate to infer the probable scheduling state of cooperating processes on remote nodes.

In our system, coordinated scheduling is not required for correctness, but is desired for improved performance; therefore, the fact that the scheduled process has changed on a remote node does not have to be immediately or definitively propagated across the cluster. For improved performance, it is only required that an event that is correlated with the remote scheduling state is propagated, from which other nodes can infer the likely remote state.

Such a correlated event occurs when processes communicate: the arrival of a message indicates that the remote process that sent this message was scheduled in the recent past. Thus, a message arrival allows a local process to infer that the remote process is likely to still be scheduled. By observing two pieces of implicit information associated with communication (the response time of request messages and the arrival rate of incoming requests), processes that communicate frequently can determine whether their scheduling is currently coordinated and act accordingly. Processes that do not communicate will not be able to determine whether their scheduling is coordinated, but these jobs do not required coordinated scheduling for good performance.

We now describe these two pieces of implicit information in more detail. For each piece of implicit information, we describe the implied scheduled state and the corresponding action the local process should take for coordinated coscheduling.

Response Time

The first piece of implicit information available to a local process is the round-trip time of a request-response message. This time provides feedback concerning the process scheduled on a remote node, since a process must be scheduled to return a response. Therefore, receiving a "fast" response indicates to the local node that the cooperating remote process was recently scheduled, while a "slow" response (or, more precisely, not receiving the response in a designated interval) indicates that the destination is probably not currently scheduled.

To maintain coordination with cooperating processes the local process should remain scheduled when it receives a "fast" response; when coordinated, the benefit of running generally exceeds the cost charged by the operating system scheduler. However, the local process should relinquish the processor when remote processes are not scheduled, as indicated by "slow" responses; when not coordinated, the cost of holding onto the processor generally exceeds the benefit. The mechanism which achieves both of these goals is two-phase waiting with the "correct" spin-time. With the correct spin-time, processes spin for a time that is approximately equal to that required for a "fast" response. The key is selecting this correct spin-time, as described in Chapter 6.

When the inference of the scheduling state of a remote node is out-of-date or simply incorrect, two-phase waiting still provides the correct behavior for the local process.

For example, when a process receives a fast response, the remote process may no longer be scheduled (*i.e.*, it was descheduled after sending the message);[1] when a process receives a slow response, the remote process may be scheduled but ignoring the network. However, in both situations, the decision of the local process still optimizes performance. If the response was fast, it is inconsequential that the remote process is no longer scheduled – the local process can continue to run productively until it communicates again, in which case it rechecks the scheduling condition. If the response was slow, it is inconsequential that the remote process is currently scheduled – the local progress should relinquish the processor since it is unable to make forward progress.

The preceding discussion showed how spinning in the first phase of the two-phase waiting maintains scheduling coordination when it already exists. Waking up again after blocking in the second phase can also help instigate scheduling coordination. Coordination develops under two scenarios, both of which involve multiple processes waiting for the same process to return a response.

The first scenario occurs when multiple processes synchronize at an operation such as a barrier. If scheduling is uncoordinated, then some of the processes arrive at the barrier earlier than others; these early processes eventually sleep. When the last process reaches the synchronization point, all participating processes are notified. These notification messages are sent nearly simultaneously and thus received nearly simultaneously by the participating processes. When this response arrives, each process is woken and placed on the ready queue; each process is then scheduled, if it is fair to do so (which is a function of the local scheduler and the other local competing processes). Thus, programs containing frequent synchronization operations are likely to dynamically coordinate themselves.

The second scenario occurs when, over time, multiple processes send requests to a destination process that is not scheduled; since the process is not scheduled, the requests are temporarily buffered. Each requesting processes eventually sleep. When the critical process is scheduled, it will handle each of its buffered messages and send back responses. The requesting processes are then likely to receive the responses, to be woken, and to be scheduled in quick succession (once again, if fair to do so). Thus, programs in which all processes are dependent upon the same set of processes are also likely to become dynamically coordinated.

Request Arrival

With the implicit information conveyed by the response time for requests, each process optimizes its local performance. However, for the best performance for the job as a whole, processes must not only optimize their local performance, but also that of cooperating processes. The second event, the receipt of a request from a remote node, provides implicit information about the scheduling requirements of cooperating processes.

We described previously that processes that have chosen to sleep while waiting for a message response should be returned to the ready queue when the response arrives.

[1] This scenario is more likely to occur if the network latency, L, is large relative to the time-slices, Q, used by the local schedulers; in current systems, the time-slices (between $20ms$ and $200ms$) dominate message latencies (near $10\mu s$), so this is expected to occur infrequently.

However, it is also the case that these processes should be returned to the ready queue when any *request* message arrives. In general, it is beneficial for a process to be scheduled if it is handling incoming requests and enabling other cooperating processes to make progress.

The correct action of a process when a request arrives depends upon the current state of the process. If the process is spin-waiting, then its spin-time is temporarily extended so that the process remains scheduled longer. We call this extension to two-phase waiting, *conditional two-phase waiting*. Spinning longer keeps processes coordinated on the expectation that more messages will arrive in the future. The desired amount of additional spin-time is analyzed in more detail in Chapter 6.

If the process is sleeping when a request arrives, then it is woken, and placed on the ready queue; when the process is scheduled, it will immediately handle the request, thus improving the response time measured by the remote sender. The local process will then also reset its spin time before sleeping again, with the expectation that the processes will remain coordinated and more messages may arrive in the future.

The actions described above are taken regardless of whether the remote process actually requires an immediate response to make forward progress. In fact, there are numerous situations where a sending process can make forward progress without a response [46, 95, 109]; for example, the request could be a one-way message storing data, or the first segment of bulk transfer, or a notification that this process has reached a barrier before other processes have done so. However, when a request message arrives, there is no way for the local process to know whether it must send back a response without first being scheduled and inspecting the message.

Chapter 5

Local Scheduler

Implicit coscheduling requires a local scheduler with a cost-model against which user processes can optimize. In this chapter, we describe the needed functionality in the local operating system scheduler running on each node of the cluster. We also describe the strengths and weakness of two types of schedulers. The first type, based on *multilevel feedback queues*, is often used in practice. The strength of these popular schedulers is that they provide a compromise between several metrics: quick response time for interactive jobs, high throughput for compute-intensive jobs, and fair allocations for all jobs. The second type, a *proportional-share scheduler*, is more popular in the research community than in practice. The strength of proportional-share schedulers is their ability to precisely control the amount of resources allocated to each process. It is for these schedulers that we describe simple extensions that meet the needs of implicit coscheduling.

5.1 Requirements

For implicit coscheduling to guarantee forward progress, ensure efficient utilization, and provide fair allocation across different jobs and users, the operating system scheduler must have the following three properties:

- **Preemption.** To guarantee that the set of communicating processes can all make forward progress, it may be necessary to preempt a running process and dispatch a competing process; a round-robin scheduler guarantees that processes will not deadlock. Performance may be improved if a running process can also be preempted by a competing process with more urgent work to perform (*e.g.*, handling a message).

- **Amortized Context-Switches.** The duration of a time-slice should be long relative to the cost of a context-switch to amortize the cost of obtaining coordination.

- **Fair Cost-Model.** The local scheduler should export a well-defined cost-model so that processes use the CPU only when it is beneficial. Further, with a well-defined cost model, higher-level policies can be built that provide fair allocations to competing users across the cluster.

These three qualities of the local scheduler are desirable not only for communicating processes within implicit coscheduling, but also for general-purpose workloads on a single machine. For example, preemptable schedulers improve the response time of interactive processes. Minimizing context-switch costs is a consideration in sequential operating systems as well. Finally, interactive processes waiting for user input or I/O-intensive jobs waiting for disk requests are often compensated for the time they do not use the CPU. We now discuss these three requirements in more detail for implicit coscheduling.

5.1.1 Preemption

In certain circumstances, the local operating system scheduler must preempt the currently scheduled process for both correctness and improved performance. To see why preemption is required for correctness, consider the following simple scenario. There are two competing jobs, A and B, each containing two processes; the processes are allocated such that processes A_1 and B_1 are placed on workstation 1 and processes A_2 and B_2 are on workstation 2. We assume that for a job to complete, each of its processes must have completed, and that neither process voluntarily relinquishes the processor. If the processes arrive at the workstations such that process A_1 is scheduled first on workstation 1 and process B_2 is scheduled first on workstation 2, then the system is in a deadlock if neither process can be preempted. The only way to break this deadlock is either for workstation 1 to schedule process B_1 or for workstation 2 to schedule process A_2. A preemptable scheduler that eventually schedules each of its runnable processes (such as a round-robin scheduler) guarantees that all jobs make forward progress, regardless of their interdependencies.

Preemption at one key moment can significantly improve the performance of implicit coscheduling: when a process becomes runnable due to a message arrival. There are two reasons for this meritorious effect. First, if the arriving message is a request that requires a response, then prompt scheduling reduces the waiting time of the requesting remote process. Second, if multiple processes are receiving messages from the same sending process, then prompt scheduling increases the likelihood that dynamic coordination will be achieved across all processes.

5.1.2 Amortized Context-Switches

A process must pay the cost of a context-switch every time that it is scheduled; therefore, preemptions should not occur too rapidly. The implication is that the default time-slice of the scheduler should be large relative to the context-switch cost. This is significantly more critical within implicit coscheduling than in sequential systems; communicating processes lose coordination whenever a time-slice expires and regaining this coordination typically incurs multiple context-switches on all workstations. In practice, we have found that a default time-slice of $100ms$ is sufficient to amortize a $200\mu s$ context-switch.

However, when a message arrives for an unscheduled process, the time-slice may be terminated prematurely. To ensure that message arrivals cannot incur an unbounded number of context-switches, the scheduler should only preempt the running process in favor of a newly runnable process when it is "fair" to do so; otherwise, the scheduler should wait for the current time-slice to expire.

5.1.3 Fair Cost-Model

To increase the utilization of the system, the scheduler should dispatch those processes that are able to use the CPU most effectively; *i.e.*, it should not select processes that are idly spin-waiting or frequently context-switching. This goal can be accomplished if the scheduler exports a well-defined cost-model to user processes. One such model is that all processes are given the same proportion of the CPU averaged over some time interval, regardless of how they utilize the CPU; *e.g.*, whether they use the CPU to spin-wait, to context-switch, or to perform useful computation. We refer to this cost-model as fair.

Given this cost-model, each process can determine for itself when it is beneficial to run. If each process is guaranteed the same amount of the CPU regardless of when it executes and if each process has a constant amount of work independent of when it is scheduled, then each process can minimize its execution time by running in two specific cases. First, the process should run when it has useful local computation to perform and thus is making forward progress. Second, under certain conditions, the process should run when it is idly waiting for a remote event to complete: if the time the process wastes by running and spin-waiting is less than the time the entire job wastes if the process blocks, then it should continue to hold onto the processor.

A process notifies the local scheduler that it is beneficial to be scheduled by remaining runnable and spin-waiting; it notifies the scheduler that it is not beneficial to be scheduled by blocking. 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 Chapter 6. We now discuss the extent to which these qualities are present in existing schedulers.

5.2 Multilevel Feedback Queue Schedulers

The objective of a *multilevel feedback queue scheduler* is to fairly and efficiently schedule a mix of processes with a variety of execution characteristics. The queues in the system are ranked according to priority. By controlling how a process moves between queues, the scheduler can treat processes with different characteristics appropriately. Processes waiting in higher priority queues are always scheduled over those in lower priority queues; processes at the same priority are usually scheduled in a round-robin fashion. These schedulers are also called *dynamic priority-based schedulers*.

In this section, we discuss the extent to which a prototypical priority-based scheduler, the Solaris Time-Sharing scheduler, meets the three requirements of implicit coscheduling. We will see that while this scheduler is preemptable, it does not always amortize the cost of context-switch over a sufficiently long time-slice and does not provide a precise cost-model.

5.2.1 Overview of Solaris Time-Sharing Scheduler

The Time-Sharing (TS) scheduler in Solaris [51] is based on Unix System V Release 4 [67]. In the TS scheduler, the priority of a process is lowered after it consumes its time-slice; its priority is raised if it has not consumed its time-slice before a *starvation interval* expires.

Thus, compute-bound jobs filter down to the lower priorities, where they are scheduled less frequently (but for longer time-slices.) Alternatively, interactive jobs propagate to the higher priorities where they are scheduled whenever they have work to perform, on the assumption that they will soon relinquish the processor again.

The durations of the time-slices, the changes in priorities, and the starvation interval are specified in a user-tunable dispatch table. In the default dispatch table, shown in Table 5.1, the priority of jobs ranges from a high of 59 down to 0. In Solaris 2.6, time-slices begin at $20ms$ at the highest priority and gradually increase to $200ms$ at the lowest priorities. Generally, the priority of a process decreases by 10 levels after it consumes its time-slice; the priority of a process is increased to 50 or above if the process sleeps for a specified interval or is marked as starving. However, due to the configuration of the Solaris 2.6 default dispatch table (*i.e.*, the starvation interval is set to zero), the priority of every process is raised once a second, regardless of whether it is actually starving.

5.2.2 Preemption

In general, multi-level feedback queue schedulers preempt processes in two circumstances. First, a process is preempted whenever its time-slice expires. Second, a process is preempted if a higher-priority process exists, which occurs in two circumstances: either the priority of a runnable process is raised above that of the currently scheduled process or a higher priority process wakes and transition to the runnable state. As expected, the Solaris time-sharing scheduler is preemptable under these circumstances, as desired.

5.2.3 Amortized Context-Switches

Since time-slices in the Solaris TS scheduler vary from $Q = 20ms$ at the high priorities to $Q = 200ms$ at the low priorities, the more time a process spends at high priorities, the shorter its time-slice and the more time it wastes achieving coordination. The average time-slice given to a process depends not only on its own computation characteristics, but also on the number of competing jobs in the system. Because the priority of very process is raised once a second and lowered only when the process completes a time-slice, a process in competition with more processes will complete fewer time-slices in the one second interval. Therefore, with more jobs in the system, processes spend more time executing at high priorities and receive shorter time-slices. Our experiments will show that the short time-slices at high priorities negatively impact the performance of implicit coscheduling when there are many competing jobs.

5.2.4 Fair Cost-Model

In general, multi-level feedback queue schedulers do not export a precise cost-model to user-level processes. Thus, a process may receive more or less of the CPU over a time interval depending upon how frequently the process blocks. Multi-level feedback queue schedulers tend to provide a coarse level of fairness across jobs with different computation characteristics. However, because these schedulers attempt to compromise between several

Current Priority Level	Time-Slice Q (ms)	Expire Priority	Sleep Priority	Starve Priority	Starve Interval (s)
0	200	0	50	50	0
...	...	...	...	...	...
9	200	0	50	50	0
10	160	0	51	51	0
...	...	...	...	...	...
19	160	9	51	51	0
20	120	10	52	52	0
...	...	...	...	...	...
29	120	19	52	52	0
30	80	20	53	53	0
...	...	...	...	...	...
34	80	24	53	53	0
35	80	25	54	54	0
...	...	...	...	...	...
39	80	29	54	54	0
40	40	30	55	55	0
...	...	...	...	...	...
44	40	34	55	55	0
45	40	35	56	56	0
46	40	36	57	57	0
47	40	37	58	58	0
48	40	38	58	58	0
49	40	39	58	59	0
50	40	40	58	59	0
51	40	41	58	59	0
...	...	...	...	...	...
58	40	48	58	59	0
59	20	49	59	59	32000

Table 5.1: **Solaris 2.6 Time-Sharing Default Dispatch Table.** *Priorities in this class range from a low of 0 to a high of 59. The time-slice is the length of time a process at this priority level will be scheduled (assuming that the process does not voluntarily relinquish the processor or that a higher-priority process does not preempt it). After a process consumes its time-slice, its priority is degraded to that designated in the third column. The fourth and fifth columns show the priority the process is given if the process is sleeping or does not consume its time-slice when the starvation interval expires, as designated in the final column.*

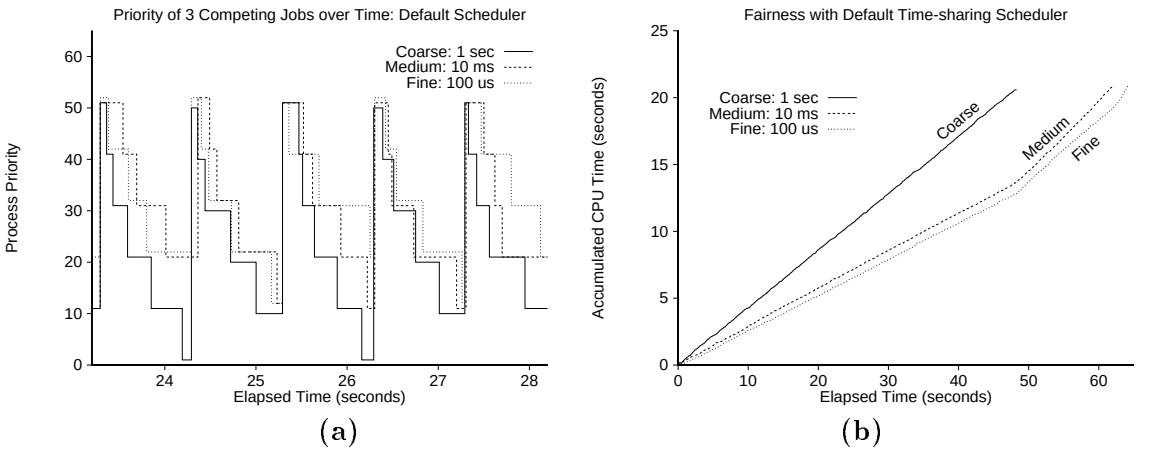


Figure 5.1: **Measured Fairness with Solaris Time-Sharing Scheduler.** *Three bulk-synchronous jobs communicating with the NEWS pattern and no load-imbalance are time-shared on 16 workstations. To capture this data, we ran the three parallel jobs with implicit coscheduling in our system, while recording all changes to the priority of each process on a single workstation.*

desirable metrics (*e.g.*, response time for interactive jobs and throughput for compute-intensive jobs), they do not compensate jobs that voluntarily relinquish the processor in a precise manner.

The Solaris time-sharing scheduler approximates fair allocations by decreasing the priority of a job the more that it is scheduled. Therefore, a job that is runnable relatively infrequently remains at a higher priority and is scheduled over lower priority jobs. However, because the allocation history of each process 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.

This behavior is illustrated in Figure 5.1.a for three competing parallel jobs that communicate at different rates. The figure shows a snapshot over a five second execution interval. As a job executes and consumes its time-slice, its priority is lowered ten levels. Note that the priority of the coarse-grain job drops more quickly since it consumes its time-slices more rapidly. However, every second the priority of all three jobs is reset to level 50 or higher. The impact of this scheduling policy on the relative execution times of the three applications is shown in Figure 5.1.b: because the coarse-grain application acquires more CPU time, it finishes its work earlier than the other applications, even though all three jobs require the same amount of time in a dedicated environment.

Finally, priority-based schedulers are not good building blocks for providing a fair allocation of the shared resources in the cluster across competing users. Numerous studies have shown that these schedulers are difficult to tune and to understand [131]. Since the scheduler makes decisions with no notion of which processes belong to which users, users running more processes naturally receive more resources. Although higher level policies have been built on top of priority-based schedulers to allocate resources fairly [72, 74, 87],

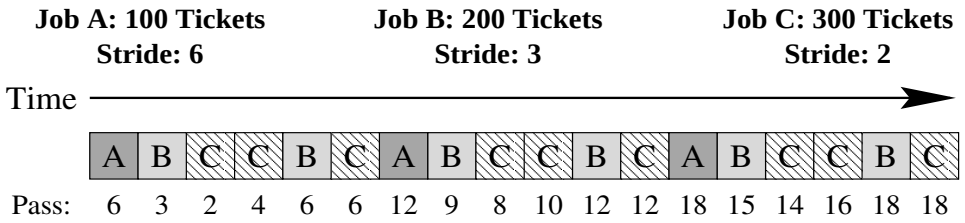


Figure 5.2: **Basic Stride Scheduling.** *Three jobs (A, B, and C) with different ticket allocations are running on a single workstation. Each job has a* `stride` *and a* `pass` *associated with it. The* `stride` *of a job is inversely proportional to the number of tickets in the job. At each scheduling interval, the job with the minimum* `pass` *is scheduled and its* `pass` *is incremented by its* `stride`*.*

these *fair-share schedulers* remain difficult to tune and provide fairness only over relatively long intervals.

5.3 Proportional-Share Schedulers

The idea behind a *proportional-share scheduler* is to allocate resources to processes in proportion to their relative number of shares [60, 69, 83, 155, 158, 170]. Such schedulers are useful for processes that require a fixed allocation of resources, such as multimedia applications. We focus on *stride scheduling* [168, 171] because it is easy to understand and implement, has been well described and analyzed in the literature, and supports modular resource management. We begin by the describing the basic system as introduced and developed by Waldspurger and discussing the degree to which it fulfills the needs of implicit coscheduling. Then, the bulk of this section presents our extensions to provide fair allocations to processes that relinquish the CPU and fair allocations across the cluster.

5.3.1 Overview of Stride Schedulers

Stride scheduling is a deterministic allocation algorithm that encapsulates resource rights with tickets. Like *lottery scheduling* [170], resources are allocated to competing clients in proportion to the number of *tickets* they hold. For example, if a client has t tickets in a system with T total tickets, the client receives t/T of the resources. A *client* may be either a user or a process, depending upon the context.

In stride scheduling, 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 5.2, a `pass` associated with each process is incremented by the `stride` of the process each time it is scheduled for a fixed time-slice; the process with the minimum `pass` is selected each quantum. 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`.

5.3.2 Preemption

Proportional-share schedulers preempt a process after its time-slice has expired and dispatch another process. However, since these schedulers have not been targeted for general-purpose workloads, they do not have a general notion for preempting a process after an event completes (*e.g.*, a message arrival). Fortunately, modifying a stride scheduler to switch to a process that has just woken is not difficult: the new process is scheduled if its `pass` is lower than the `pass` of the currently scheduled process.[1] It also requires small modifications to ensure that the preempted process is only charged for the amount of the time-slice that it was able to consume.

5.3.3 Amortized Context-Switches

Due to the constant-length time-slices in stride-scheduling, it is trivial to select a time-slice duration, Q, to amortize the costs of obtaining coordination and context-switching. In our simulations and implementation, we use a time-slice of $Q = 100ms$.

5.3.4 Fair Cost-Model

While proportional-share schedulers do provide a very 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. Although the need for interactive jobs to receive *time-averaged* fairness rather than *instantaneous* fairness was observed by Waldspurger in [172], no proposal was presented for allowing temporarily inactive processes to catch up to their desired allocation.

To ensure that jobs that voluntarily relinquish the processor receive their proportional-share of resources, the basic stride scheduler must be extended to fairly allocate resources over a longer time interval. We describe two extensions that compensate processes for the time that they voluntarily relinquish the CPU. Both policies build upon *exhaustible tickets*, proposed by Waldspurger [172]; exhaustible tickets are tickets with an expiration time. In general, the number of exhaustible tickets and the expiration time are chosen such that the process has caught up to its desired proportional-share of resources when the exhaustible tickets expire.

In our first approach, *system credit*, processes are given exhaustible tickets from the system when they awake; this leads to a simple implementation, but processes relinquish control over their precise allocation. In our second approach, *loan & borrow*, exhaustible

[1] As when a process exits the system before consuming its entire time-slice, this change implies that the amount of time allocated to a process must be precisely tracked and that its `pass` value is incremented accordingly.

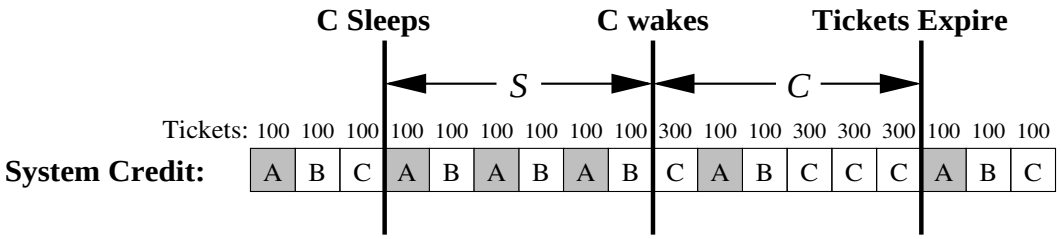


Figure 5.3: **Stride-Scheduling with System Credit Extension.** *Three jobs with equal ticket allocations are competing for resources. Jobs A and B are compute-intensive and do not relinquish the processor; job C is an interactive job. Job C temporarily exits the system, and sleeps for an interval S; in the time-interval C, job C catches up for the time it missed. All jobs receive their proportional-share of 6 allocations over the entire interval of 18 time-units*

tickets are traded among competing clients; by keeping the number of tickets in the system constant, precise service rates can be guaranteed to clients.

System Credit

The idea behind the *system credit* policy is that after a process sleeps and awakens, the operating system scheduler calculates the number of exhaustible tickets for the process to receive its proportional share over some longer interval.[2] This policy is easy to implement and does not add significant overhead to the scheduler on sleep and wakeup events. Figure 5.3 shows an example.

Consider a workload with three competing jobs, A, B, and C, each with an equal number of tickets, t, backed by independent currencies. In the beginning, each job is active and therefore receives 1/3 of the CPU. When job C sleeps for an interval S, jobs A and B are temporarily given $t/(T-t)$ (*i.e.*, 1/2) of the resources instead of t/T (*i.e.*, 1/3). When job C awakens, we pick some number of exhaustible tickets, e, and an expiration interval, C, such that over the interval $S+C$, job C receives its desired proportion of resources, t/T. Obviously, it is also required that jobs A and B receive their proportion of resources over this same $S+C$ interval.

The implementation of the *system credit* approach is simple. When a process goes to sleep, 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 a job to receive $t/T \cdot (S+C)$ units of service over the interval C, the job must have $(t+e)$ of the $(T+e)$ tickets in the system, where the number of exhaustible tickets e is as follows:

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

[2]Processes exiting the system for multiple time quanta are distinct from processes relinquishing the processor before the end of their quantum. In basic stride scheduling, processes consuming only a fraction of their quantum are charged for only the amount they used; they receive their full proportional share if and only if they are active again at the time of their next allocation.

$$e = \frac{tTS}{CT - (S+C)t}, \text{ if } C > \frac{St}{T-t}$$

We now must choose a compensation period, C, over which to allocate the e exhaustible tickets. A number of different constraints exist. First, if C is longer than the remaining execution time of the process, the job will not acquire its share of resources before it terminates. Second, if the process continues to alternate between run and sleep intervals, then C should be less than the average run time – otherwise the process will not be able to use its exhaustible tickets at the same rate it accumulates them. More formally, a process cannot receive its proportional-share if the ratio of its sleep time to its run time exceeds the ratio of the number of competing tickets to its number of tickets, t.

$$\frac{S}{R} > \frac{T-t}{t}$$

Our current approach chooses C so as to simplify the calculation for the number of exhaustible tickets. There are three possible cases, depending upon the relative value of the number of client tickets, t, and system tickets T. In the simplest case, when $2t < T$, we choose $C = S$, and simplify the equation for the number of exhaustible tickets as follows.

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

When $2t \geq T$ and $t \neq T$, we select C and e as follows.

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

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

If a process relinquishes the processor while it still has exhaustible tickets, the ticket count along with the remaining interval before they expire are recorded. When the process later receives additional exhaustible tickets, the two sets must be combined, which requires that each has the same expiration interval. If the expiration interval of a set of exhaustible tickets, e_1, is modified from one expiration interval, C_1, to a new interval, C_2, the number of exhaustible tickets, e_2, must also be changed.

$$\begin{aligned} \frac{e_1}{T+e_1} \cdot C_1 &= \frac{e_2}{T'+e_2} \cdot C_2 \\ e_2 &= \frac{T'e_1C_1}{TC_2 + e_1(C_2 - C_1)} \\ e_2 &\approx \frac{e_1C_1}{C_2}, \text{ if } T' = T \text{ and } e_1(C_2 - C_1) \ll TC_2 \end{aligned}$$

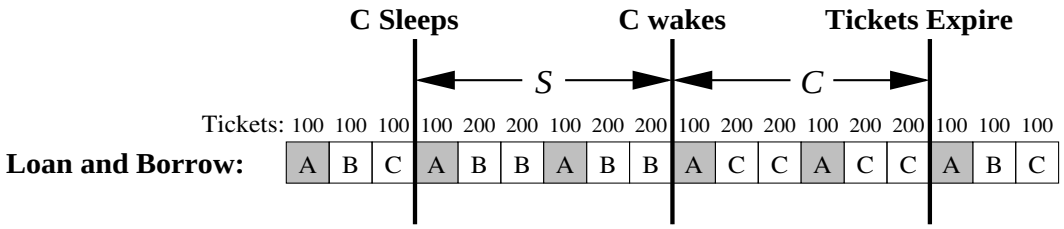


Figure 5.4: **Stride-Scheduling with Loan & Borrow Extension.** *Three jobs with equal ticket allocations are competing for resources. Job A desires a constant service rate, job B is compute-intensive and willing to borrow tickets, and job C is an interactive job. Job C temporarily exits the system, and sleeps for an interval S; in the time-interval C, job C catches up for the time it missed. All jobs receive their proportional-share of 6 allocations over the entire interval of 18 time-units; however, only this policy also supports jobs with precise allocation requirements; job A is always scheduled at least 1 out of every 3 time units regardless of the activity of other jobs.*

In our implementation, we assume that the number of system tickets at the current time, T', is equal to the number of system tickets in the past, T, and that $\epsilon_1(C_2 - C_1) \ll TC_2$.

An unstated assumption of the *system credit* extension is that there is only one process sleeping and obtaining credit at a time. The problem when multiple processes re-enter the system is that each process calculates its exhaustible tickets independently of the simultaneous (or later) calculations of competing processes. Since processes do not account for the increase in T due to other processes also introducing exhaustible tickets, jobs may acquire too little of the resources. Correcting for these errors stretches the *system credit* policy beyond its function as an algorithm that is simple to implement with little overhead. Thus, this extension performs most accurately when few processes are simultaneously leaving and joining the system.

Load & Borrow

In our second policy, *loan & borrow*, when a process temporarily exits the system, other processes can borrow these otherwise inactive tickets. The *borrowed tickets* have an unknown expiration time (*i.e.*, when the process rejoins the system). When the sleeping process wakes after S time units, it stops loaning tickets and is paid back in exhaustible tickets by the borrowing processes. In general, the lifetime of the exhaustible tickets, C, is equal to the length the original tickets were borrowed, S. The newly awakened process has positive exhaustible tickets in this interval, C, while the borrowing process has negative exhaustible tickets. An example is presented in Figure 5.4.

The advantage of the *loan & borrow* approach relative to the *system credit* approach is that the total number of tickets in the system is kept constant over time; thus, clients can accurately determine the amount of resources they receive. For example, applications that desire a constant service rate or interactive processes that want their full share when active, may decline to borrow tickets. On the other hand, a compute-bound process which is concerned only with throughput can choose to borrow all available tickets. The disadvantage

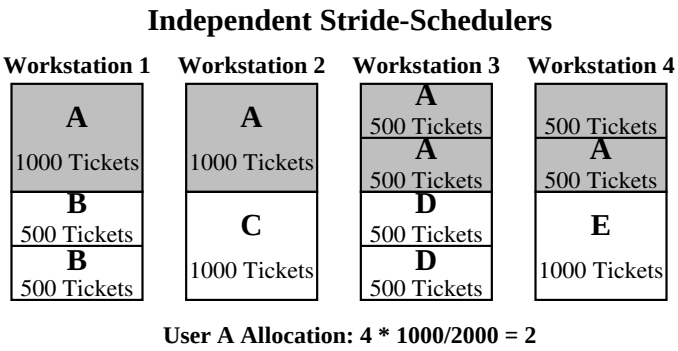


Figure 5.5: **Fairness in Cluster with Independent Stride Schedulers.** *With independent proportional-share schedulers, users do not necessarily receive their expected share. In this example, users A, B, C, D, and E each have 1000 tickets in the base currency; in a cluster of four machines, each user should receive 4/5 of the time on one workstation. With the current placement of two users per workstation, if each scheduler independently calculates tickets then each user receives half of a workstation. Therefore, user A receives more resources by running on all of the workstations.*

of *loan & borrow* is that the implementation introduces an excessive amount of computation into the scheduler on every sleep and wake-up event.

When a process goes to sleep, it first saves its exhaustible tickets and expiration date. Then, the process checks if there are active processes that are willing to borrow exhaustible tickets; all active processes that have specified that they are willing are initially placed on this list; processes are removed from the list when they sleep.[3] If a process is unable to lend all of its tickets, the remaining tickets are temporarily dropped from the system.

When the process awakens, it calculates the exhaustible tickets that it is owed by each of the borrowers. The lending process may have exhaustible tickets saved from a previous sleep/run cycle; therefore, the new and old tickets must be converted into one collection with the same expiration date. Likewise, each borrower may have multiple sets of exhaustible tickets that must be converted. Further, if a process goes to sleep while it is borrowing tickets, the process must give back the borrowed tickets to the lender and record the amount of time the tickets were used. Likewise, when a process wakes, it should check if there are any sleeping processes who were unable to loan all of their tickets and borrow from them accordingly.

5.3.5 Scheduling in the Cluster

Stride scheduling can provide an excellent building-block for higher-level policies. *Currencies* allow clients to distribute tickets in a modular fashion [170]. By assigning one currency per user, a proportional-share of resources on a workstation can be allocated to each user; the user can in turn allocate a proportional-share of her resources to her processes.

[3] Processes also may not borrow as many exhaustible tickets as they have base tickets; otherwise, a process could be left with negative tickets in the pay-back period.

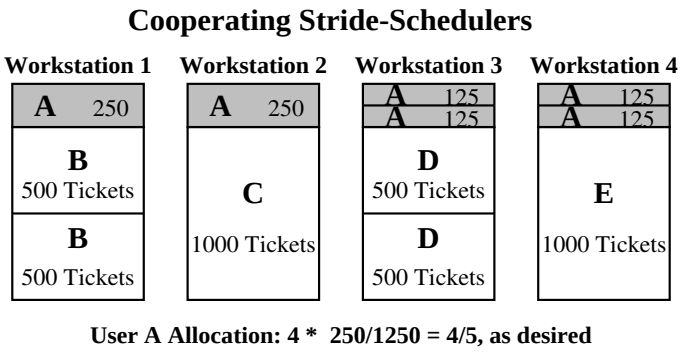


Figure 5.6: **Fairness in Cluster with Cooperating Stride Schedulers.** *If each scheduler knows the number of tickets issued across the cluster, resources can be allocated fairly across users. If each scheduler knows that user A has 6 jobs, each with an equal number of tickets, (or the total number of issued tickets), then the tickets can be appropriately converted. As a result, user A receives 1/5 of the time on each of the four workstations, as desired.*

For example, if a user with 100 tickets in the *base currency* allocates 300 tickets in her user currency to one job and 100 tickets in her user currency to another job, then the jobs have 75 and 25 tickets each, respectively, in the base currency. Different users can be allocated different proportions of the CPU simply by issuing them different ticket amounts in the base currency.

One might think that running a proportional-share scheduler on each workstation in the cluster would give a proportional-share of the total resources to each user; however, this is not sufficient for two reasons. First, each scheduler must know the number of tickets that have been issued in each currency. Second, the number of base tickets allocated on each machine must be equal.

Tracking Issued Tickets

If independent stride schedulers are run on every node, then the schedulers do not know the total number of tickets issued in a user's currency across the cluster and cannot convert tickets in the user's currency to the correct number in the base currency. Thus, as shown in Figure 5.5, users running jobs on more workstations receive more of the total resources. Fortunately, the solution is simple: each local scheduler should be periodically informed of the number of tickets issued in each currency. With this information, each scheduler can correctly calculate the base funding of each local job. The resulting allocations are shown in Figure 5.6.

In our prototype, a *parallel ticket server* tracks the number of tickets issued by each user across the cluster and informs each stride scheduler of the value of each currency. The ticket server is a user-level, multi-threaded program run as root on every workstation. Running the ticket server as a user-level program has a number of attractive features in the presence of failures. First, if one of the nodes in the system crashes, the Active Message layer simply returns messages directed to that node as undeliverable; the data stored on the

failed node can be then reconstructed.[4] Second, if the ticket server crashes, the behavior of the stride schedulers degenerates to the case with no global information: jobs continue to run, but clients are no longer guaranteed their proportional share of the shared cluster.

The parallel ticket server is implemented as a collection of local ticket server processes located on each of the nodes in the cluster. The processes communicate with one another using AM-II. To update and distribute the ticket counts of each client throughout the cluster, each stride scheduler periodically contacts its local ticket server. When a process enters or exits the scheduling class, the scheduler writes the user's id and change in allocated tickets, (`uid`, Δt), to a memory region that is shared with a pseudo-device driver, `SScomm`. This driver shuttles information between the scheduler in the kernel and the local user-level ticket server.

Since many processes are expected to be short-lived, unnecessary overhead is incurred if every process enter and exit causes the ticket server to recalculate and redistribute the value of each currency. Therefore, the ticket server only periodically polls the `SScomm` module to obtain the list of (`uid`, Δt) pairs. The drawback of only periodically updating ticket counts is that users are not charged for tickets allocated to jobs which start and complete within one of these intervals; thus, a user may cheat the system by running many very short jobs. If this is found to cause fairness problems in practice, then the ticket server must adjust for such past allocations.

The parallel ticket server communicates amongst itself to collect and update the number of ticket issued in each currency across workstations. Each user that has ever run a job in the cluster has a home node (calculated with a simple hash function on the `uid`), which records the current number of allocated tickets. When the home node is updated, the current ticket count is distributed to the local ticket servers and corresponding `SScomm` modules across the cluster. Each `SScomm` module directly updates the data structures in the local stride scheduler. Thus, when the stride scheduler allocates the next process, the appropriate number of tickets in the base currency are used.

Balancing Tickets

Knowing the number of tickets issued in each currency is still not sufficient to ensure that the correct share of resources is allocated to each client. After a job has been placed, tickets compete only with other tickets on the same machine. The result is that tickets on nodes with less competition are worth more than tickets on nodes with more competition. This effect is very similar to that in load-balancing, where jobs on machines with fewer competing jobs receive more processing time.

To guarantee that clients receive their proportional-share, not only must the system track the number of tickets issued in each currency, but the number of tickets per machine must also be balanced. It is simple to show that this condition is sufficient. Consider the case where there are P machines and a total of T base tickets, where user u has T_u tickets. If there are an equal number of tickets, T_p, on each machine, then by definition $T_p = \frac{T}{P}$. If the T_u tickets of user u are distributed across the nodes in the cluster such that $T_{u,p}$ tickets are on node p, then user u receives the desired allocation of $P \cdot \frac{T_u}{T}$:

[4] The functionality to reconstruct data is not implemented in our prototype.

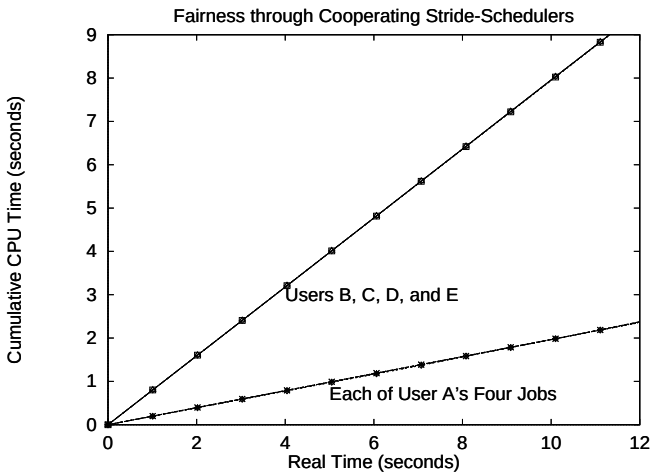


Figure 5.7: **Measured Fairness with Ticket Server and Stride Scheduler.** *Five competing users are running compute-bound sequential jobs on a cluster of four workstations. User A is running one job on each of the four workstations, while the other users are running only one job each. The parallel ticket server informs the stride scheduler running on each workstation of the number of tickets user A has allocated across the cluster, so that user A receives only 1/5 of each workstation, as desired.*

$$\frac{T_{u,1}}{T_p} + \frac{T_{u,2}}{T_p} + \frac{T_{u,3}}{T_p} + ... + \frac{T_{u,P}}{T_p} = \frac{T_u}{T_p} = \frac{T_u}{\frac{T}{P}} = P \cdot \frac{T_u}{T}$$

Placing jobs to balance tickets is similar to placing jobs to balance load. For example, processes may be placed in a fixed location once when they created, or they may be migrated over their lifetime to improve performance. Likewise, ticket information can be kept in a centralized location or may be distributed. We do not address these problems in this dissertation, but leave them as future work. We now briefly discuss our approach to informing each scheduler of the number of tickets issued to each user.

Performance

Figure 5.7 shows the results of running the ticket server with the SSC local scheduler in a very simple experiment. In the workload, five competing users are running collections of compute-bound jobs on a cluster of four workstations, where the number of tickets per machine is balanced. As illustrated in Figures 5.5 and Figures 5.6, user A is running jobs on all four workstations, while users B, C, D, and E are contained to a single workstation. With the parallel ticket server, each scheduler is notified of the number of tickets user A has allocated throughout the cluster; as a result, user A only receives 1/5 of each of the four workstations, while the other users receive 4/5 of one workstation. Thus, as desired, each user receives their fair proportion of the shared resources, regardless of the number of jobs the user runs.

5.4 Summary

In this chapter we have shown that the operating system scheduler running on each workstation in the cluster must have three properties to efficiently support implicit coscheduling. First, the scheduler must preempt processes both at the end of a time-slice and also when a message arrives. Second, the default time-slice should be of a sufficient duration to amortize the overhead of obtaining coordination across machines. Third, the scheduler should provide a well-defined, fair cost-model to the user processes.

In our model, there is a strict separation of functionality between the processes and the scheduler. This approach allows the local scheduler to optimize for its own performance criteria (*e.g.*, response-time or throughput). Due to this separation, the scheduler knows nothing about parallel jobs or message arrivals.

Alternatively, the scheduler could be made more aware of events in user processes; for example, a message arrival could automatically trigger the scheduling of the destination process. This is the approach used in dynamic coscheduling [151], where a message arrival increases the priority of a process if it is fair to do so. We do not believe that the local scheduler should be given specific knowledge of communicating processes for the following reasons.

- **Fairness:** Specific policies are required to ensure that processes that receive more messages are not scheduled more frequently, thus receiving more than their fair share of resources.
- **Potential Thrashing:** Multiple competing processes could be sending messages to the same node, causing the scheduler to thrash between processes; specific policies are required to ensure that this does not occur.
- **Unnecessary Scheduling:** In the case of one-way messages and bulk messages, it may not be necessary for the destination process to be scheduled for the sending process to make forward progress.
- **Encapsulation Breaking:** The local scheduler must be modified to understand message arrivals.

In our approach, when a message arrives and the local process thinks it is beneficial to be scheduled, it simply moves itself to the ready queue. However, a disadvantage of this strategy is that a process may not always be scheduled immediately when an important message arrives; instead, the local scheduler has complete responsibility for allocating resources across competing processes. By keeping to this predefined interface, coordinated scheduling can be achieved with almost any local scheduler. Thus, we believe that the benefits of modularization outweigh the costs.

In practice, priority-based schedulers are often used for general-purpose workloads. Unfortunately, the time-sharing scheduling within Solaris does not meet all of the requirements of implicit coscheduling. First, under high load, each process spends too much of its execution time at high priorities, which have short time-slices. Second, the Solaris scheduler not provide a precise or fair cost-model to processes; therefore, jobs that frequently relinquish the processor do not receive their fair share of resources.

Nevertheless, due to the popularity of priority-based schedulers, we show the performance of implicit coscheduling with this scheduler for many of the experiments in this dissertation. When the number of jobs is small and the scheduler is not required to enforce fair allocation (*i.e.*, when scheduling processes with the same communication characteristics), the Solaris time-sharing scheduler provides respectable performance.

Proportional-share schedulers, such as a stride scheduler, are designed to allocate resources to processes in proportion to their shares, or tickets. However, since these schedulers have not been targeted to general-purpose workloads, they are generally neither preemptable nor fair to jobs that frequently relinquish the processor. We have introduced the *system credit* and the *loan & borrow* extensions which support time-averaged fairness for jobs that relinquish the CPU. Due to its greater simplicity, we use the *system credit* extension of stride-scheduling as our building-block in the remainder of this dissertation.

Chapter 6

Cost-Benefit Analysis of Waiting

In this chapter, we describe the cost-benefit analysis that each process performs when waiting on a communication event. We show that communicating processes can achieve coordinated scheduling by simply deciding to spin or to block when waiting for a remote operation; the actions that minimize the costs to the parallel job also tend to lead to coordinated scheduling.

To calculate the cost of spinning or blocking, each process applies its knowledge of the current scheduling state of the cluster to the scheduler's cost model. We begin by showing that conditional two-phase waiting is a suitable mechanism for efficiently implementing this analysis. We then derive the appropriate spin-times within the algorithm as a function of system and application parameters.

6.1 Conditional Two-Phase Waiting

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 exceed 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, the process can determine the extent to which it is coordinated by observing two pieces of implicit information.

First, the waiting process can determine if it is coordinated with the other processes participating in the current communication or synchronization operation (*e.g.*, the destination process in a request-response operation). This is accomplished by observing how long the operation takes to complete; if the operation completes in the "expected" time when all cooperating processes are scheduled, the process simply infers that the cooperating processes are running. Second, the waiting process can determine if it is coordinated with the remaining processes in the job (*e.g.*, all of the processes excluding the destination of the request-response operation). This is accomplished by observing the incoming message rate; if a message arrives from a sending process, the waiting process infers that the sender is also running.

Conditional two-phase waiting, a generalization of two-phase waiting, allows the process to leverage implicit information to dynamically calculate the cost of spinning versus

blocking. Unlike traditional two-phase waiting where the spin-time, S, is determined before the process begins waiting, with conditional waiting the process may increase its spin-time based on events that occur while the process is waiting. In our implementation, there are two components of spin-time: *baseline* and *conditional*. The baseline mount, S_{Base}, is the minimum time the process spins and is determined before the operation begins. The conditional amount S_{Cond}, is the additional spin-time based on spontaneous events.

These two components of spin-time correspond naturally to maintaining coordination with the two disjoint 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 many systems, two-phase waiting is viewed as a compromise between immediate-blocking and spin-waiting; for a given operation, either immediate-blocking or spin-waiting is optimal, depending on the waiting time, and two-phase waiting simply bounds the worst-case performance. In implicit coscheduling, conditional two-phase waiting may actually be superior to both immediate-blocking and spin-waiting. When a process is waiting, the cost of spinning versus blocking depends upon the dynamic rate of arriving messages. The best performance may be achieved when the local process spin-waits initially, but then blocks when the incoming message rate declines.

Note that conditional two-phase waiting differs from *adaptive two-phase waiting* [86] in which the spin-time varies from operation to operation, but is predetermined before the current operation begins. With conditional waiting, the process gathers information while it is spinning that helps it to evaluate the state of the system and to more accurately choose how long to spin for the current operation. Thus, given an oracle of this dynamic information, one could replace conditional waiting with adaptive waiting.

In the remainder of this Chapter, we describe the following three factors of spin-time:

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. This minimum waiting time is the *baseline spin*.

2. **Conditional Spin:** If a process waits the baseline amount and determines that the participating 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 relinquishes the processor. *Conditional spin* is the interval in which the waiting process must receive a message to justify keeping the process scheduled.

3. **Large Waiting Times:** In general, each process assumes that coordinated scheduling is beneficial. However, a process should wait less than the baseline spin if it expects that an individual communication operation will require a long time to complete, due

to high network latency or load-imbalance. In such cases, the process predicts that the cost of idly waiting for this particular response is greater than the cost of losing coordination.

6.2 Maintaining Coordination with Destinations

The amount of time a process must wait for a communication event to complete depends upon whether the cooperating remote processes are also scheduled. When the scheduling of processes is coordinated, waiting times are much lower than when scheduling is uncoordinated. While the process cannot directly control when it is scheduled, it can influence whether scheduling remains coordinated when it is already in such a state. Our analysis of spin-time begins by assuming that coordinated scheduling across communicating processes is beneficial; we examine the conditions under which this assumption is true in Section 6.4.

Processes remain coordinated for future communication events by spinning an appropriate amount of time at the current communication event. Processes stay coordinated by spinning for the expected worst-case time the operation requires when all involved processes are scheduled; this is the *baseline spin* amount, denoted S_{Base}. 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 remote process. The worst-case completion time occurs when the request message triggers the scheduling of the remote process. If the operation does not complete in this time, the local process infers that the processes are not coordinated and blocks, thus bounding the amount of time the process wastes with spinning.

S_{Base} differs for each type of communication operation, and can be either modeled or measured with a simple microbenchmark running in a dedicated environment. In this section, we describe models of S_{Base} using LogP for our three types of operations: request-response messages, one-way messages, and barriers.

6.2.1 Request-Response

We first evaluate the baseline spin amount, S^R_{Base}, for request-response messages. To begin, we examine the amount of time that passes between when the sending process initiates the request and when it receives the response, given that the receiving process is scheduled throughout the operation. As illustrated in Figure 6.1, the time for this operation, S^R_{Base}, can be modeled very simply with LogP [40]. The local sending process first spends time o in overhead injecting the request into the network. The request travels through the network for L time units, at which point it arrives at the receiving node. Assuming that there is no contention with other messages, the request is handled by the receiver after another o time units. The receiver spends time o returning the response, which arrives L units later at the initial sending node; the sender handles the response in o units.

Thus, T^R_{Sched} is the minimum time for a short request-response message to complete if all participating processes are scheduled and attentive to the network.

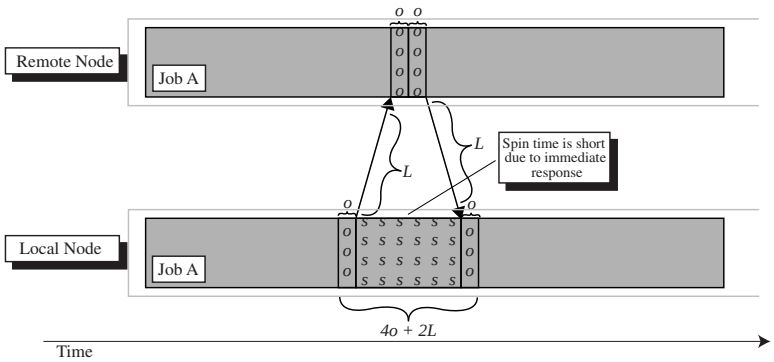


Figure 6.1: **Time for Request-Response Message with Remote Process Scheduled.** *The diagram illustrates T^R_{Sched}, the amount of time a local process must wait for a reply if the remote destination process is scheduled when the request message arrives. If the remote destination process is already scheduled, then it immediately handles the request and promptly returns the reply to the waiting process.*

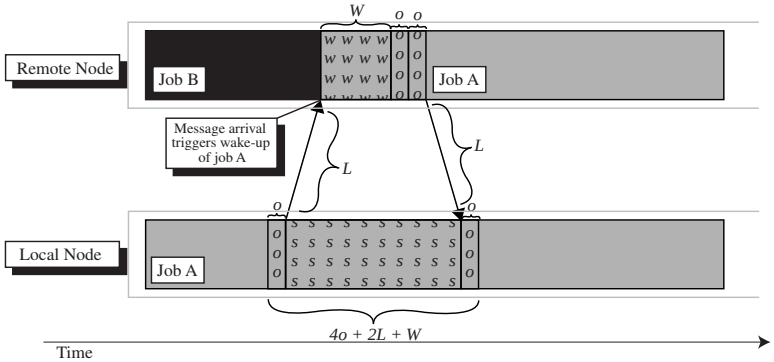


Figure 6.2: **Time for Request-Response Message with Triggering of Remote Process.** *The diagram illustrates $T^R_{Trigger}$, the amount of time a local process must wait for a reply if the request message triggers the scheduling of the remote destination process. In this case, the local process must spin-wait the additional time, W, while waiting for the remote process to be scheduled by its local operating system scheduler.*

$$\begin{aligned} T^R_{Sched} &= o + L + 2o + L + o \\ &= 2L + 4o \end{aligned}$$

However, it is often the case that the remote process is not scheduled before the request arrives, but that the request message triggers it to be scheduled. This scenario is illustrated in Figure 6.2. This case occurs under the following condition. If the remote process is blocked waiting on a response from another process, then any arriving message will wake the process and place it on the ready queue; the remote process will then be scheduled by the local scheduler if it is fair to do so (or, if the processor was idle). Therefore, if the request triggers the scheduling of the remote process, then the expected round-trip time, $T^R_{Trigger}$, is longer than if the process were already running by the amount of a context-switch, W, on the remote node.

$$\begin{aligned} T^R_{Trigger} &= 2o + L + W + 2o + L + o \\ &= 2L + 4o + W \end{aligned}$$

S^R_{Base} is the time that the local process should spin at request-response operations to ensure that it remains scheduled if the remote process is also scheduled. 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:

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

6.2.2 One-Way Requests

A process sending a one-way request does not require a response from the destination process before it continues its computation. Subsequently, there is no response for which the sending process must wait and no corresponding baseline spin time.[1]

6.2.3 All-to-all Synchronization

We now analyze the baseline time which processes must spin-wait at a barrier to ensure processes remain coordinated, S^B_{Base}. We analyze a barrier as a representative all-to-all synchronization operation without loss of generality. For a barrier to complete, not

[1]Note that even though the performance of the sending process may be oblivious to whether scheduling is coordinated, the destination process may be sensitive. For example, if the destination process was blocked on a request-response message, then incoming one-way requests will waken it and place it on the ready queue. The destination process may then be scheduled at a cost of W to handle this message, even though it was not immediately required. With coordinated scheduling, the destination process would not pay a context-switch for incoming requests and might receive better performance. The impact of arriving messages is analyzed in Section 6.3.

only must all involved processes be scheduled, but each process must also have completed the work that preceded the barrier. We consider both the case where processes have been coordinated since the last barrier and the case where the arrival of the last process triggers the scheduling of the participating processes.

If processes have been coordinated since the last barrier, then the time for the barrier to complete on process $p \epsilon 0..P-1$ is the sum of the time for a barrier with no load-imbalance, T^B_{Sched}, and the load-imbalance, v_p, observed by process p. We describe how to model T^B_{Sched} in this section; we discuss approaches for approximating load-imbalance in Section 6.4.2.

The time for the base barrier with no load-imbalance depends on how the barrier is constructed. In our simulations and prototype implementation, each of the processes participating in the barrier sends a message to a root process; we assume the root is process $p = P-1$ of the P participating processes. When the root has received P such messages, it broadcasts a barrier-completion message back to the waiting processes. Rather than communicating in a hierarchical tree structure, all processes communicate directly with the root, so that one unscheduled process does not affect the completion of the descendent processes in its subtree.

The time for T^B_{Sched} is illustrated in Figure 6.3. Assuming each of the P processes arrives at the barrier simultaneously, then each spends time o in overhead injecting a message to the root process; these messages all arrive at the root process L time units later.[2] Due to contention, the root requires time $(P-1)\max(o,g)+o$ to handle the P 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 the rank of that process in the order in which processes are notified. From the perspective of process p, time $p\max(o,g)$ is required to notify the previous processes; then, another o time units are required for its notification to be injected into the network, L units for the notification to propagate through the network, and o units to handle the notification.[3]

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

Next, we consider the case where the scheduling of all processes has not been precisely coordinated since the previous barrier. In such a case, it is possible that the completion of the barrier will trigger the scheduling of the participating processes, in which case, the processes arriving late at the barrier should remain scheduled by spin-waiting. The required spin-time depends upon the number of processes that arrive late at the barrier, P_{Late}.

Figure 6.4 depicts the scenario where all but one of the processes have reached the barrier and sleep while waiting for notification. When the last $P_{Late} < P$ processes reach the barrier, each spends time o injecting its message into the network; after L time

[2] The equation could be trivially modified to account for the fact that the root process either does not have to send a message, or at least that its message does not need to travel through the network; handling this message is likely to be overlapped with the latency of the other $P-1$ messages.

[3] Again, the root process does not have to wait L time units for the notification message to arrive.

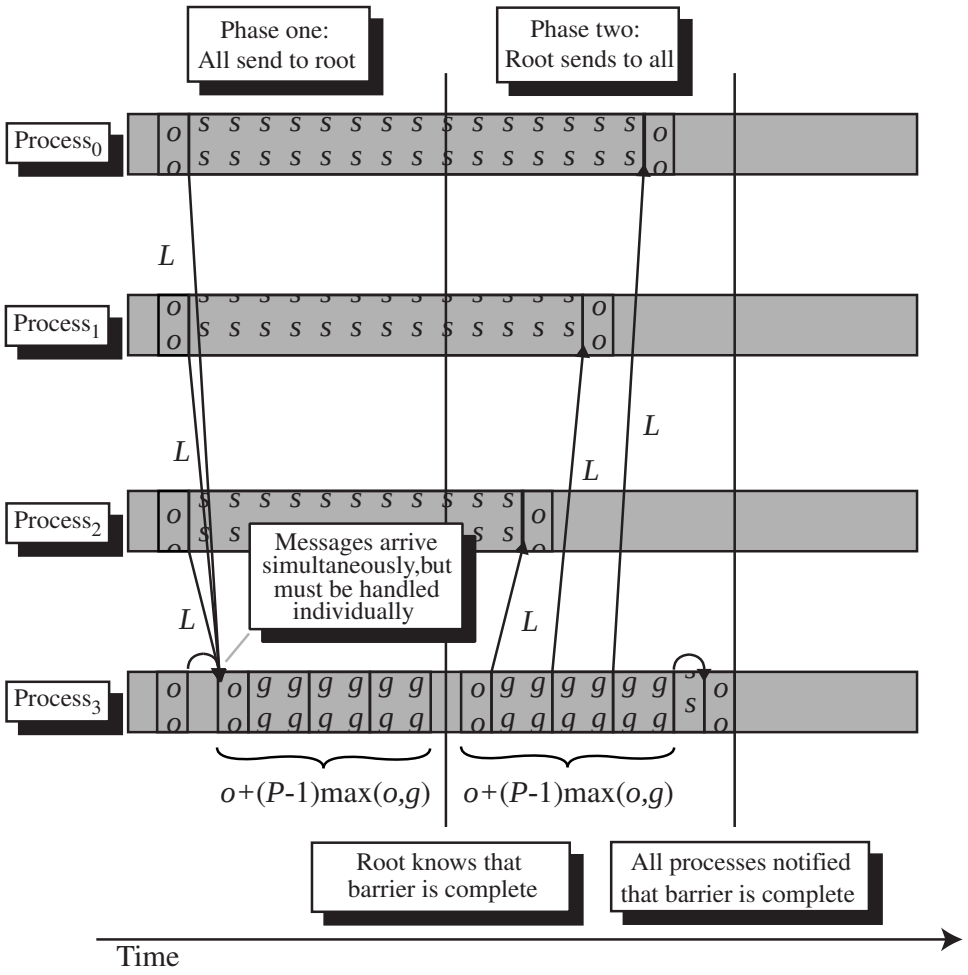


Figure 6.3: **Time for All-to-all Synchronization with Processes Scheduled.** *The diagram shows T^B_{Sched}, the time for a barrier with no load-imbalance to complete if all participating processes are scheduled in a coordinated fashion. The diagram illustrates T^B_{Sched} from the perspective of the last process to be notified that the barrier has completed.*

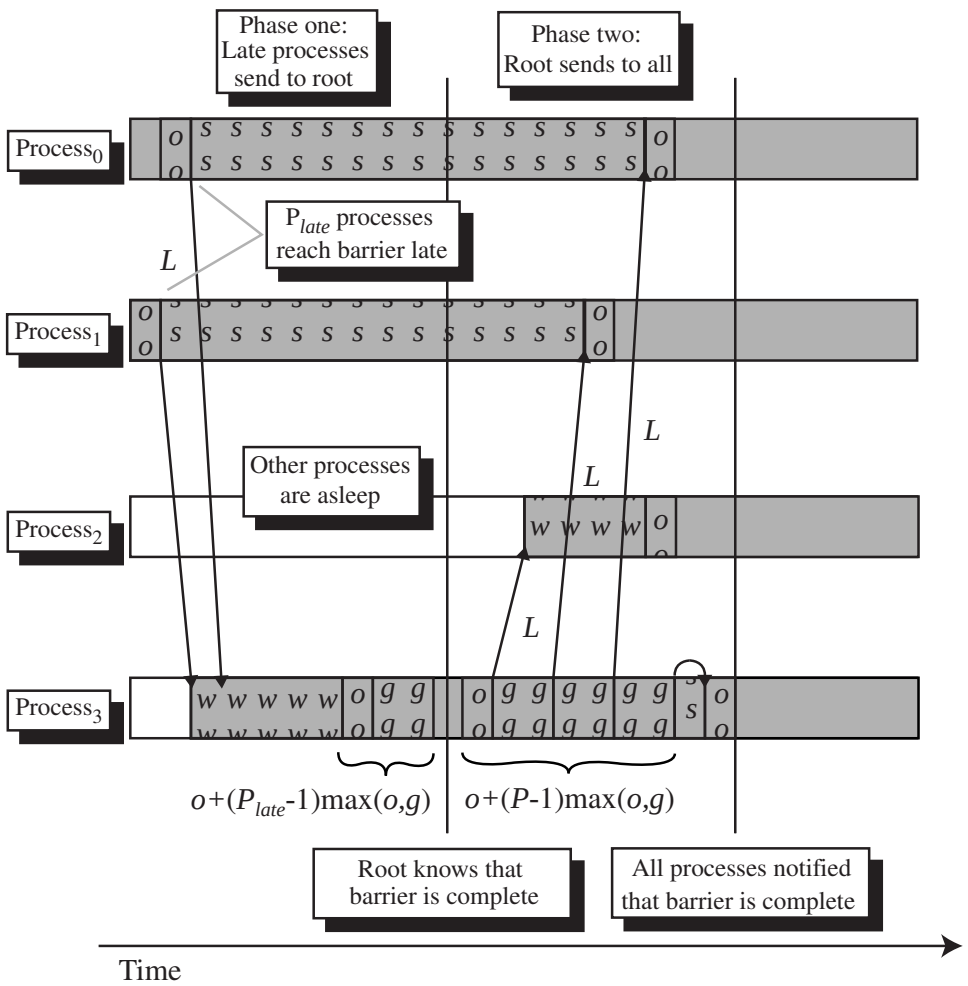


Figure 6.4: **Time for All-to-all Synchronization with Triggering of Root Process.** *The diagram illustrates $T^B_{Trigger}$, the amount of time the last process notified must wait if the root process was not scheduled when the last set of P_{Late} notification messages arrived.*

units the messages arrive at the root process. In the worst-case, the root process has already blocked, requiring a context-switch to schedule it, at a cost of W; after the root is scheduled, it handles the P_{Late} message sequentially. All participating processes are then notified as before. Thus, from the perspective of the P_{Late} processes, the operation requires time $T^B_{Trigger}$ as derived below.[4]

$$\begin{aligned} T^B_{Trigger}(p) &= o + L + W + (P_{Late} - 1)\max(o, g) + o + p\max(o, g) + o + L + o \\ &= 2L + 4o + W + (p + P_{Late} - 1) \cdot \max(o, g) \end{aligned}$$

Although the number of late processes, P_{Late}, is not known in practice, $T^B_{Trigger}$(p) can be bounded since P_{Late} must be less than P.

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

$S^B_{Base}(p)$ is the time that process p should spin-wait at a barrier before relinquishing the processor. As with request-response messages, the waiting process does not spin for the overhead of sending the request and receiving the response on the local processor.

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

The implementation may be simplified by having each of the processes wait for $S^B_{Base}(p)$ as if it were the last process to be notified, that is $S^B_{Base}(p = P - 1)$.

$$S^B_{Base}(P - 1) = 2L + 2o + 2(P - 1) \cdot \max(o, g) + W$$

A process should always spin the baseline amount, S^B_{Base}, at a barrier. Whether or not the process should also spin for the amount of load-imbalance, v_p, predicted for this barrier, is discussed in Section 6.4.2.

6.2.4 Discussion

In general, when estimating the baseline spin amount for a given communication operation, it is better for a process to err on the side of waiting too long. If a process underestimates baseline spin, the process will relinquish the processor on every communication

[4] The time for the barrier from the perspective of the $P - P_{Late}$ processes that arrived early is not relevant, since they have voluntarily relinquished the processor.

operation and scheduling will become uncoordinated. For fine-grain applications, the performance will be significantly worse than with coscheduling and spin-waiting, and slightly worse than if the process had blocked immediately. Alternatively, if a process overestimates baseline spin, the effects are not too detrimental since the higher spin time is only paid when processes are not currently coordinated, which should be rare. This intuition is verified in our simulation results, shown in Section 8.2.

6.3 Maintaining Coordination with Senders

After spinning S^R_{Base} for request-response messages, a process can conclude that the destination process is not currently scheduled. After spinning S^B_{Base} for barriers, a process can determine that the cooperating processes have not been entirely coordinated since the last barrier. If a process is unable to make forward progress because it is waiting on an unscheduled remote process, then it should usually be descheduled. However, if the process is handling incoming messages, then keeping the process scheduled can benefit both itself and the processes communicating with it.

When an incoming message arrives, the waiting process may be in either the first or second phase of the two-phase waiting algorithm. If the process is blocked in the second phase, then the process is made runnable and placed on the ready queue. Once the process is scheduled and handles the request message, it may be worthwhile for the process to remain scheduled by continuing to spin. At this point, the process should not restart its two-phase algorithm by spinning the baseline amount, S_{Base}, because the process is not waiting to learn if processes are coordinated. Instead, whether in the first or second phase of waiting, the process should conditionally spin as long as a benefit exists for the job as a whole. A process can predict the benefit by monitoring the information implicitly available from the incoming message rate.

In this section, we examine the interval, T_{Cond}, in which a waiting process must receive a message for conditional spinning to be beneficial. The local process may be waiting on any type of communication operation. Though the type of the operation on which the local process is waiting is inconsequential, the type of the arriving message is not. We consider three different types of message arrivals: requests that require responses, one-way requests, and notifications that a process has reached a synchronization point.

6.3.1 Incoming Request-Response

We begin by calculating the interval T^R_{Cond} when the arriving message is a request that requires a response. In our discussion, we consider a pair of processes. First, there is the local process that has previously communicated with a remote process that is not currently scheduled; the local process is performing two-phase waiting while its communication operation completes, but has exceeded the baseline spin amount. Second, there is a remote process that is sending a request to the local process; the remote process must wait until the response is returned before it can proceed. To evaluate T^R_{Cond} we compare the cost to both processes when the local process spin-waits for its own operation versus when the local process blocks.

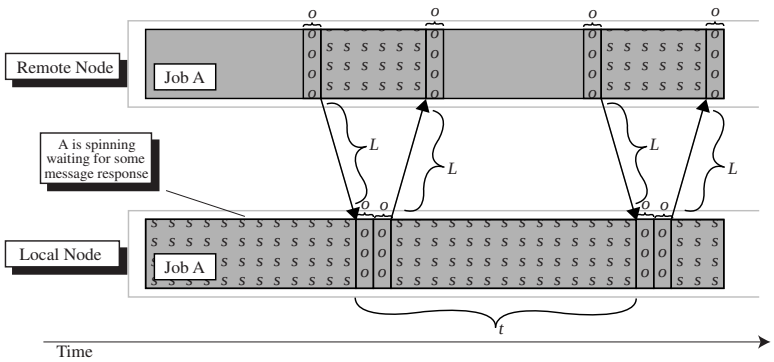


Figure 6.5: **Cost of Incoming Request-Response Messages if Local Process Spin-Waits.** *To help calculate T^R_{Cond}, this figure illustrates the cost to the local (receiving) process and the remote (sending) process when the local process spin-waits between message arrivals. We assume a constant time t between message arrivals.*

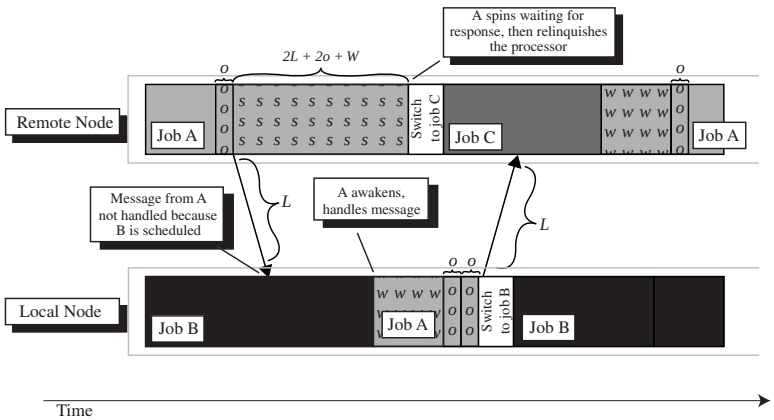


Figure 6.6: **Cost of Incoming Request-Response Messages if Local Process Blocks.** *To help calculate T^R_{Cond}, the figure illustrates the cost to the pair of processes when the local process blocks while waiting for message arrivals.*

We begin with the case where the local process spin-waits on its own communication operation and remains scheduled, as shown in Figure 6.5. After the local process begins spinning (whether in the first or second phase), a request arrives t units later; thus, the local process spins idly for time t to handle this one message. The remote process spends o injecting the request, waits $2L + 2o$ for the response to be returned, and spends another o handling the response. The combined cost to the pair of processes when the local process spins is thus $2L + 4o + t$.

In the second case, shown in Figure 6.6, the local process blocks before the request arrives. The remote process spends o sending, and then unsuccessfully spins $S_{RBase} = 2L + 2o + W$ 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.[5] 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 as long as the interval between messages is less than or equal to T^R_{Cond}. [6]

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

6.3.2 Incoming One-way Requests

We next determine the interval, T^O_{Cond}, in which a one-way request must arrive for the local process to remain scheduled. Since the sending remote process can proceed in its computation without waiting for a response, its performance is identical regardless of whether the local process is scheduled. Therefore, our analysis considers only the impact on the local process.

Figure 6.7 shows the case where the local process is spinning when the one-way request arrives. The local process spins idly for time t to handle this one message. In

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

[6] This comparison assumes that paying a cost on one workstation is equivalent to paying that cost on another workstation. For this to be true, neither process must form the critical path for the system. For example, if the sending process on a remote machine determines the performance of the job as a whole, then the local receiving process should always spin to minimize the waiting time on the remote sender. However, if the local process is the critical component, then only its costs should be examined when determining whether to spin or block.

$$\begin{array}{rrcr} & \text{Cost of Local Process Spinning} & = & \text{Cost of Local Process Blocking} \\ & t & = & 2o + W \\ \Longrightarrow & T^R_{Cond_{LocalCritical}} & = & 2o + W \end{array}$$

Due to the difficulty of determining which process is along the critical path and because we believe this inaccuracy is small for our workloads, we ignore this adjustment in our analysis. However, for client-server applications where the performance of the server is likely to be the critical factor, these adjustments may need to be taken into account.

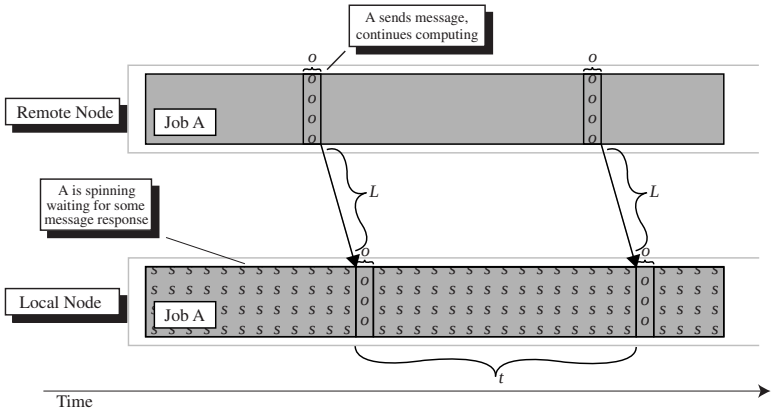


Figure 6.7: **Cost of Incoming One-Way Requests if Local Process Spin-Waits.** *To help calculate T^O_{Cond}, this figure illustrates the cost to the local (receiving) process when it spin-waits between message arrivals. We assume a constant time t between message arrivals.*

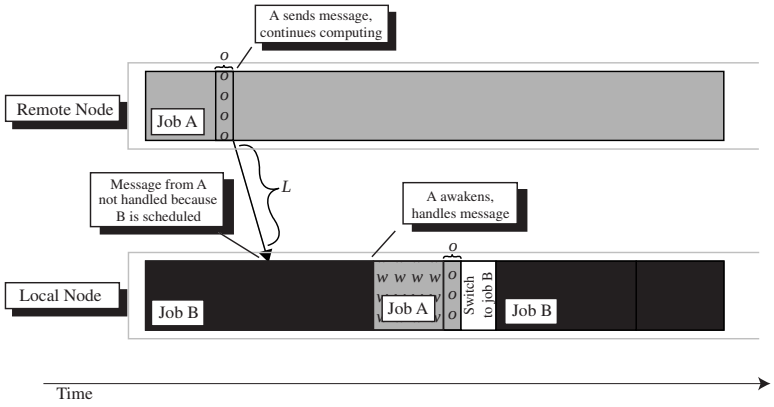


Figure 6.8: **Cost of One-Way Requests if Local Process Blocks.** *To help calculate T^R_{Cond}, the figure illustrates the cost to the local process when it blocks while waiting for incoming messages.*

the second case, shown in Figure 6.8, the local process blocks before the one-way request arrives. Since, the local process must still be scheduled to handle the one-way request, it pays a cost of W to be scheduled on the message arrival and o to handle the request. Thus, the cost to the local process when it blocks is $o + W$.

Comparing these two costs, we see that one-way messages must arrive more frequently than request-response messages to justify keeping the local process scheduled when it is unable to make forward progress.

$$\begin{aligned} \text{Cost of Spinning} &= \text{Cost of Blocking} \\ t &= W + o \\ \Longrightarrow \quad T^{O}_{Cond} &= W + o \end{aligned}$$

6.3.3 Incoming Synchronization Messages

In our implementation of a barrier operation, only the root process receives messages signifying that a cooperating process has reached the barrier. Since scheduling is not coordinated, the synchronization operation will not complete in the amount of time the sending process expects, regardless of whether the root process spins or blocks (unless this is the P-th process to arrive at the barrier): in either case, the sending process spins the baseline amount, S^{B}_{Base}, and then blocks. As with one-way messages, only the local process (in this case, the root process) is affected by whether it spins or blocks.

$$\begin{aligned} \text{Cost of Spinning} &= \text{Cost of Blocking} \\ t &= W + o \\ \Longrightarrow \quad T^{B}_{Cond} &= W + o \end{aligned}$$

6.3.4 Discussion

The preceding analysis assumes that a 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 arrivals are expected to be bursty (or steadily decline as more processes reach barriers and complete their communication phase), the average rate over a relatively short interval should be used. For example, if a message has arrived in the last T_{Cond} interval, the process should spin for another interval $S_{Cond} = T_{Cond}$, continuing until no messages are received in an interval. At the next two-phase wait, the process should reevaluate the benefits of spinning longer based on the current arrival interval.

We note that the arrival of synchronization messages at the root process cannot be used to predict the future arrival of such messages: each process can have only one such message outstanding at a time, and since scheduling is uncoordinated, there is no correlation with arrival times across different processes. This suggests that the arrival of synchronization messages should be ignored when calculating whether a waiting process should spin longer. However, to simplify the implementation, processes can choose to treat all arriving messages identically (*e.g.*, as request-response messages), with little negative impact expected.

6.4 Optimizations for Long Waiting Times

The preceding analysis of spin-time assumed that it is always desirable to keep processes coordinated while waiting for communication operations to complete. In general, coordinated scheduling is beneficial when processes communicate frequently and wait only a short time for communication to complete. However, it may not be advantageous for processes to wait the entire time for an operation when the expected waiting time is large.

First, as wait time increases, so does the amount of time a process must spin idly before concluding that scheduling is uncoordinated. Second, as waiting time increases, the probability that time-slices will expire before the operation completes increases, thus decreasing the potential for keeping processes coordinated. Finally, at some point, the penalty for idly spinning while waiting is higher than the penalty for losing scheduling coordination across cooperating processes; at that point, a competing job's useful computation should be overlapped with the waiting time.

Waiting time within a communication operation has two components. The first component of waiting time is dependent upon the system architecture, in particular, the latency of the network. In systems with high network latency relative to context-switch cost, local scheduling with immediate blocking is superior to coscheduling with spin-waiting regardless of the characteristics of the application. Since processes know *a priori* that waiting time will always be high, processes should assume that scheduling is always uncoordinated and use immediate blocking rather than two-phase waiting.

The second component of waiting time depends upon the characteristics of the application: the amount of load-imbalance at a synchronization point. Applications with load-imbalance may benefit from sometimes allowing processes to be scheduled independently. However, in such applications, some communication operations still benefit from coordinated scheduling (such as short request-response operations); in these cases, we must estimate the cost of uncoordinated scheduling on those communication events that occur before coordination is regained.

6.4.1 Network Latency

In this section, we evaluate the regime of systems where network latency is high enough that independent scheduling of processes with immediate blocking is superior to coordinated scheduling. This coordinated scheduling can be achieved with either explicit coscheduling and spin-waiting or implicit coscheduling with two-phase waiting. We first examine the cost of a communication operation that incurs no additional waiting beyond the base cost of the system: a simple request-response message. We then show that most current clusters of workstations operate in the regime where coordinated scheduling is desired for fine-grain applications.

Our calculation compares the time for the request-response message in two scenarios: first, when all processes block immediately and are not coordinated, and second, when all processes spin-wait and are coordinated. Figure 6.9 shows the cost to the system for the simple request-response message when processes block immediately after sending a message. In this scenario, the sending process spends time o in overhead injecting the message into the network, after which it immediately relinquishes the processor and a competing process

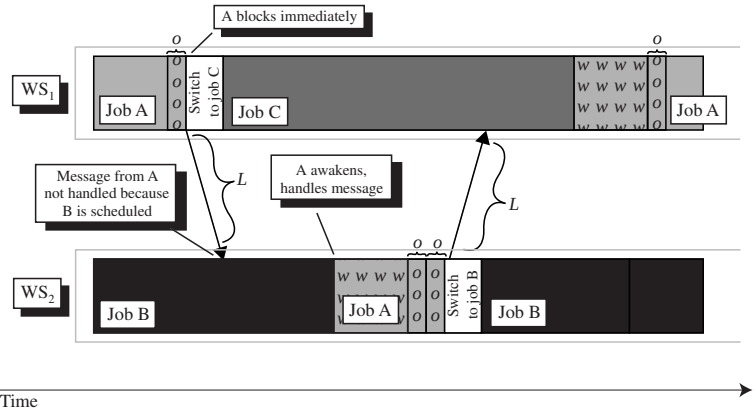


Figure 6.9: **Cost with Network Latency when Processes are Uncoordinated and Block.** *To help derive L_{Block}, this figure illustrates the cost of a request-reply when processes are scheduled independently and block immediately rather than spin-wait for communication to complete.*

is potentially scheduled. The request travels through the network for L time units, but this time is not charged to any process in the system. When the request arrives at the remote workstation, the destination process is unlikely to be scheduled. If this process is blocked on a request of its own, then it will be woken and placed on the ready queue; at some point it will be scheduled, and pay a cost of W for the context-switch.[7] Once the destination process is running, it handles the request in o time units and sends back the response in another o units. When the response arrives at the original sender L time units later, this process must also pay W to be scheduled and o to handle the message. The total cost to the two processes with independent scheduling is the time spent in overhead and context-switching: $4o + 2W$.

Figure 6.10 shows the cost when processes spin and remain coordinated while sending messages. As before, the sending process first spends time o in overhead injecting the request into the network; however, rather than relinquish the processor at this point, the process waits for the response by idly spinning. The request travels through the network for L time units, at which point it arrives at the receiving node. Since the receiving process is currently scheduled, it immediately handles the request and returns the response in a total of $2o$ units; the response then arrives L units later at the initial sending node. Since the sending process is also currently scheduled, it immediately handles the response at a cost of o. Under these circumstances, the total cost to the two processes is the time spent in overhead touching messages, $4o$, plus the time the sending process spent spin-waiting, $2L + 2o$, for a total of $2L + 6o$.

Comparing the two costs allows us to determine the range of systems for which coordinated scheduling with spin-waiting should be used. When network latency, L, is less than L_{Block} then coordinated scheduling should be used; when L exceeds L_{Block}, then

[7] In the case where the destination process is receiving messages from multiple senders, it is possible for the cost of W to be amortized over multiple requests.

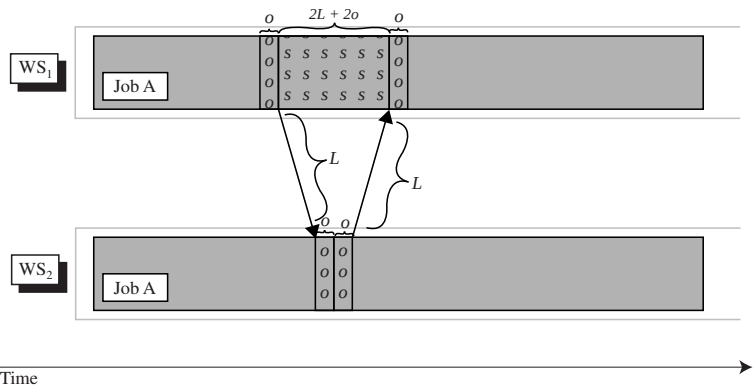


Figure 6.10: **Cost with Network Latency when Processes are Coordinated and Spin-Wait.** *To help derive L_{Block}, this figure illustrates the cost of a request-reply for the sending and receiving processes when processes are coordinated and spin while waiting for communication to complete.*

immediate blocking should be used. L_{Block}, is calculated as follows.

$$\begin{aligned} \text{Cost of Spinning} &= \text{Cost of Blocking} \\ 2L + 6o &= 4o + 2W \\ L &= W - o \\ \Longrightarrow \quad L_{Block} &= W - o \end{aligned}$$

6.4.2 Load-Imbalance

We now evaluate the amount of load-imbalance within an application for which processes should not remain coordinated. This analysis differs from that for network latency in two primary ways. First, the amount of load-imbalance must be predicted and can subsequently be incorrect; therefore, processes must be conservative in their choice of spin-time. Second, some communication operations (*e.g.*, short request-response operations) still benefit from coordinated scheduling; therefore, processes must also predict the cost of uncoordinated scheduling on those events that transpire before coordination is regained.

V_{Block} denotes the amount of load-imbalance for which it is advantageous for a process to block before the barrier completes rather than spin-wait for the expected completion time of the barrier. To calculate V_{Block} we compare the cost of spin-waiting for the expected completion time of a barrier with load-imbalance v_p versus the cost of spin-waiting only S^B_{Base} and then blocking.[8] Note that we do not compare to the cost of blocking-immediately at the barrier; since our prediction of load-imbalance may be incorrect, processes always spin-wait for at least the minimum amount, S^B_{Base}, so that they remain coordinated when there is actually only a small amount of load-imbalance.

We begin by considering the cost to the root process and a process p with a load-imbalance v_p when all processes are coordinated and spin-wait through the barrier. When

[8] Processes waiting at a barrier will always conditionally spin S_{Cond} if the incoming message rate supports its, regardless of the relative values of v_p and V_{Block}.

process p arrives at the barrier, it spends o injecting the message into the network. The barrier message reaches the root process after L time units.[9] After time V has passed, all processes have sent messages to the root. The root process then notifies each of the participating processes, spending time o on the notification for process p. Process p receives the notification in $p\max(o,g)+o+L$ time units and handles it in another o. Thus, the total cost to the two processes is $4o+2L+v_p+p\max(o,g)$.

We next consider the cost to these two processes when processes are not coordinated and process p does not wait until the barrier is completed. The arriving process p spends o sending the message to the root and then spins S^B_{Base} waiting for a potential response. If the root process has already blocked, then the root pays W to be woken and o to handle this message. After all of the uncoordinated processes have arrived at the barrier, the root spends o injecting the notification message for process p. When the message arrives, the local process pays $W+o$ to be woken and to handle the message. At this point, the total cost to the system is $4o+2W+S^B_{Base}$.

However, when a process blocks prematurely, the scheduling of cooperating processes becomes uncoordinated, which adversely impacts future communication operations and their waiting times. Therefore, there is some additional penalty to losing coordination, C_{Uncoor}, which must be included in the cost of blocking.

$$\begin{aligned}
\text{Cost of Spinning for Load-Imbalance} &= \text{Cost of Spinning Minimum} \\
4o+2L+v_p+p\max(o,g) &= 4o+2W+S_{BBase}+C_{Uncoor} \\
v_p &= C_{Uncoor}+3W+2o+(P-1)\cdot\max(o,g) \\
\Longrightarrow \quad V_{Block} &= C_{Uncoor}+3W+2o+(P-1)\cdot\max(o,g)
\end{aligned}$$

To summarize, if the predicted load-imbalance, v_p, is less than V_{Block}, then the process spins for the expected completion time of the barrier, $S^B_{Base}+v_p$. For higher values of load-imbalance, the process spins only S^B_{Base} before relinquishing the processor.

Penalty for Losing Coordination

To calculate the cost of losing coordination, C_{Uncoor}, we determine the additional overhead, K, that is incurred on every communication operation involving an unscheduled process and the number of such operations, M, that are expected to occur before all processes regain coordination. Our estimations for both K and M are approximations of what we have observed in practice.

We begin by considering the state of each of the P processes when the barrier completes; each of these processes had previously voluntarily relinquished the processor. When the barrier notification messages are sent to the participating processes, each of the processes is woken and placed on its local ready queue. Whether each process is scheduled depends upon whether it is fair to schedule the process and the characteristics of the local scheduler; therefore, only some fraction, $1/f$, of the P runnable processes are initially scheduled.

[9] We do not include the overhead of handling any of the messages in the cost to the root process, since this overhead may be overlapped with spin-waiting if the root is not on the critical path; *i.e.*, if $v_{\text{root}=p-1} > Po$.

To simplify our discussion, we assume that once one of the P cooperating processes is scheduled instead of its competing processes, it remains fair for it to be scheduled. This is a reasonable assumption as long as the process makes little progress relative to the length of a time-slice and as long as competing processes that have acquired less of the resources do not later become runnable. We refer to this process that will be scheduled when it is runnable as the *active* process on that workstation; the competing processes on that workstation are said to be *inactive*. After being scheduled, each of the P/f cooperating processes is, by definition, the active process on that workstation.

Each of the P/f processes sends messages to destination processes as long it remains scheduled. Some of the messages are sent to scheduled processes and others to non-scheduled processes, which may be inactive or active. We discuss these three cases in turn.

In the first case, the request is sent to a scheduled process. Since the destination process is scheduled, it promptly handles the request and returns the response. The sending process remains scheduled while waiting for the response and continues communicating with other processes. Thus, for this case, there is no additional overhead to losing coordination at the previous barrier.

In the second case, the request is sent to one of initial $P - P/f$ inactive processes. Since the inactive destination process already received the barrier-completion message and was not scheduled, it must not be fair to schedule that process and the new request will not schedule it either. Thus, sending a request to an inactive process causes the sender to relinquish the processor: the sending process spins for the baseline spin amount (*i.e.*, an extra W beyond the minimum round-trip time) and then voluntarily relinquishes the processor (if incoming messages are not arriving at an interval less than S_{Cond}). However, the sending process remains the active process on that workstation and is scheduled promptly again when it receives requests.

At some point in the future, it becomes fair to schedule the inactive destination process; *i.e.*, the destination process becomes active. The destination process handles the original barrier-completion message and all waiting requests, sending back the appropriate replies. When the reply arrives at the original requester, it is scheduled promptly at a cost of W, and continues sending more messages. Therefore, the cost to the system for each request sent to an inactive process is $K = 2W$, paid entirely by the sending process. Meanwhile, the destination process is now active and scheduled and begins sending its own requests.

In the final case, the request is sent to a non-scheduled, but active processes. Since the active process will always be scheduled when it has work to do, the arriving message wakes the active process and triggers its scheduling at a cost of W for the context-switch. The sending process spin-waits the additional W units for the response to be returned and remains scheduled. Thus, every message sent to an unscheduled, active process incurs an extra $K = 2W$.

In summary, all processes eventually become active and scheduled; however, each request-response message sent to an unscheduled process incurs a cost of $K = 2W$. The total cost to the system for losing coordination depends upon how many messages are sent to unscheduled processes before coordination is regained. With our assumptions, coordination

is regained when it becomes fair to schedule the last cooperating process in the job. Since processes continue to communicate as long as they eventually receive responses, a process may incur this cost of K per message until it sends a message to this last, critical, inactive process.

The number of messages, M, that a process sends before sending to this critical process clearly depends upon the communication pattern of the application. If the process has some knowledge of its communication pattern, then it can bound the number of messages that may be sent to unscheduled processes and thus calculate the penalty of losing coordination. However, estimating the behavior of many communication patterns is not understood at this time. For example, in communication patterns with locality, some processes may communicate exclusively with a subgroup of active processes and thus never send to the inactive process and never block; these processes may then pay the additional cost of $K = 2W$ for every communication operation until the next barrier.

Therefore, given no knowledge of the communication pattern, we assume that processes send messages to random destinations. In this case, as with any all-to-all pattern, each process is expected to send $M = \frac{P}{2}$ messages to active processes before sending to the critical process. If we assume that none of the active processes are scheduled when the message arrives, then the penalty a process is expected to pay for losing coordination is:

$$\begin{aligned} C_{Uncoor} &= K \cdot M \\ &= 2W\tfrac{P}{2} \\ &= W \cdot P. \end{aligned}$$

While this discussion made a number of simplifying assumptions, simulation results in Section 8.4 show that it works reasonably well in practice for a variety of communication patterns.

Predicting Load-Imbalance

The preceding analysis assumed that the load-imbalance, v_p, was known for each process p arriving at a barrier operation. While the actual amount of load-imbalance cannot be known until after the barrier completes, a number of approaches exist for estimating load-imbalance. For example, programmers could annotate the barriers in their application with predictions for the expected load-imbalance per process. Alternatively, the application could be executed once in a dedicated environment and each barrier could be automatically annotated for future executions. However, automatically calculating load-imbalance for the current run with no setup is clearly the best solution.

Our proposal for approximating load-imbalance at run-time is to measure past waiting times at barriers and predict future load-imbalance from the distribution of samples. Depending upon the expected behavior of the application, each unique barrier in the application can have its own history and predictions, or this information can be aggregated over all barriers. Further, statistics and predictions can be made independently for each process in the application, or as a collection. If the distribution of load-imbalances, v_p, differs for each process p, then the prediction for load-imbalance should be made independently for each process. However, if the distribution of load-imbalance is similar for all

processes, then the root process can obtain a better estimation of V with fewer samples and can reduce storage and computation costs.

The primary observation for our approach is that the root process of the barrier can observe the arrival time of each process and calculate either each v_p (or V) as the difference in arrival time between process p (or the first process) and the last process. The load-imbalance predicted for the next barrier can then be propagated to the other processes as part of the barrier-completion message. With this approach, the root process can predict a coherent set of load-imbalances for each process, such that either all processes block or all processes spin. However, because processes are occasionally uncoordinated, the measured load-imbalance may be higher than the *inherent* load-imbalance of the application; that is, the load-imbalance measured in a dedicated environment. The correct method for adjusting these waiting times depends upon whether the lack of coordination is *transient* or *consistent*.

A transient lack of coordination occurs as a natural side effect of implicit coscheduling: when a time-slice expires on one node, communicating processes see a wave of uncoordinated scheduling. To deal with transient uncoordination, large waiting times that are statistical outliers should be removed from the distribution of samples. A heuristic that we have found to work well in practice is to remove the largest 10% of the samples. The remaining wait times are used to predict future load-imbalance, as described below.

A consistent lack of coordination occurs when processes repeatedly block rather than wait for communication operations to complete. In this case, the measured waiting times are consistently inflated and are not correlated with the inherent load-imbalance. As described previously, processes block rather than spin for the expected waiting time when load-imbalance is high. Therefore, when consistent uncoordination exists, the actual load-imbalance is generally irrelevant. However, if the application alternates between phases with high load-imbalance and phases with low load-imbalance, then the low inherent load-imbalance will be inflated when measured and the processes may never become coordinated. If this is a concern, then processes must not block immediately at a barrier and must instead spin for at least the minimum amount, $S^B_{Base} + V_{Block}$, which keeps processes coordinated when there is a small amount of load-imbalance.

Our current method for predicting future load-imbalance from past history is simplistic, but easy to implement. An application may use any prediction scheme that it desires without impacting other applications that are using the default scheme provided by the system. In the default scheme the root process approximates v_p (or V) for the next instantiation of this barrier as follows. The root disregards the top 10% of the collected $N = 100$ samples and then uses the highest remaining sample as the next prediction. The algorithm adapts to changes in load-imbalance over the lifetime of the application by randomly replacing old measurements with new samples. Before the application has gathered N samples, processes spin for $S^B_{Base} + V_{Block}$ to keep jobs coordinated so that the inherent load-imbalance can be measured.

6.5 Summary

By choosing whether to spin or block at a communication operation, each process communicates with the local scheduler whether or not it is beneficial to be scheduled. In

Variable	Description	Equation
S_{Base}	**baseline spin**	S^R_{Base} or S^B_{Base}
S^R_{Base}	read baseline spin	$\max(T^R_{Sched}, T^R_{Trigger}) - 2o$ $= 2L + 2o + W$
T^R_{Sched}	request-response time (remote process scheduled)	$2L + 4o$
$T^R_{Trigger}$	request-response time (remote process triggered)	$2L + 4o + W$
S^B_{Base}	barrier baseline spin	$\max(T^B_{Sched}, T^B_{Trigger})$ - 2o $\approx 2L + 2o + (2P-2) \cdot \max(o,g) + W$
T^B_{Sched}	barrier time (remote processes scheduled)	$2L + 4o + (2P-2) \cdot \max(o,g)$
$T^B_{Trigger}$	barrier time (remote processes triggered)	$2L + 4o + (2P-3) \cdot \max(o,g) + W$
S_{Cond}	**conditional spin**	S^R_{Cond} or S^O_{Cond} or S^B_{Cond}
S^R_{Cond}	request-response conditional spin	$3W + 2o$
S^O_{Cond}	one-way request conditional spin	$W + o$
S^B_{Cond}	barrier conditional spin	$W + o$
L_{Block}	blocking latency	$W - o$
V_{Block}	blocking load-imbalance	$C_{Uncoor} + 3W + 2o + (P-1)\max(o,g)$
C_{Uncoor}	penalty for lost coordination	$\approx W \cdot P$

Table 6.1: **Conditional Two-Phase Waiting Parameters for Implicit Coscheduling.** *The table shows the variables and equations for the time a process should spin before blocking.*

general, processes spin when the scheduling of cooperating processes is coordinated across workstations; under these circumstances, communication performance is determined by the characteristics of the network. On the other hand, process block when scheduling is uncoordinated; relinquishing the processor allows other competing processes to perform useful work, but implies that communication performance is limited by the speed of a context-switch.

Two-phase waiting allows processes to dynamically spin-wait for an operation to complete when processes are coordinated and to block otherwise. Each component of spin-time and its relationship with various system parameters is summarized in Figure 6.1. In this chapter, we discussed three heuristics for determining the spin-time within the *conditional two-phase waiting* algorithm.

First, a process waiting for a communication event to complete should spin long enough to ensure that its scheduling stays coordinated with cooperating processes when already in such a state. By maintaining coordination, the process not only optimizes the performance of the current operation, but also increases the chances that future communication events will complete quickly. By spinning the *baseline* amount, the local process ensures that it remains scheduled until the response is expected to return; in the worst-case, when the arrival of the current message triggers the scheduling of the remote process, the local process must wait for the context-switch of the remote process.

Second, a process that waits the baseline amount and determines that cooperating processes are not scheduled should remain scheduled when receiving messages. If the interval between arriving messages is less than the conditional interval, a greater cost would be incurred if the waiting process blocked than if it stayed scheduled. Since the interval before the next message arrives cannot be known, it is predicted from past arrival rates.

Finally, a process can optimize its performance beyond that achievable with explicit coscheduling by relinquishing the processor prematurely when waiting times are known to be very high due to network latency or load-imbalance. When high waiting time is due to network latency, processes block immediately at all communication events and are always uncoordinated. However, when waiting time is high for only a particular communication operation, coordinated scheduling may remain beneficial for the other communication events within the application; therefore, before relinquishing the processor at a barrier, processes must estimate the penalty of losing scheduling coordination on later operations.

Chapter 7

Simulation Environment

In this chapter, we describe our event-driven simulator, `SIMplicity`, which simulates the execution of communicating processes under a variety of scheduling approaches. We describe how `SIMplicity` matches the model of our system initially described in Section 4.1. In particular, we discuss the high-level machine architecture, the message-layer, the user processes, the application workload, and the operating system schedulers.

Performing simulations in addition to measuring an implementation of implicit coscheduling has a number of benefits. First, the simulator enables us to study the effect of many parameters on implicit coscheduling performance, such as the local operating system scheduler, the waiting algorithm, the communication layer, and the parallel workloads. Not only are we able to explore a large parameter space in a short amount of time, but we can also study the effect of system parameters that cannot be changed in the real world. Second, it gives us control over all layers of the system. Third, with the simulator, we can unobtrusively view statistics from all layers. Finally, it allows us to study implicit coscheduling before the implementation of the Active Message layer supported multiple communicating processes.

7.1 Machine Architecture

The system-level architecture within `SIMplicity` consists of a fixed number of workstations that are dedicated to running the parallel jobs. Each workstation contains a single processor and is identical to the others in the cluster. In our simulations, the number of workstations has been fixed at 32. The workstations are connected together with a simple high-performance network, whose characteristics are described in Section 7.2.

7.2 Message Layer

The message layer within `SIMplicity` incorporates the performance of the physical interconnect, as well as the lowest level of software. The most important parameter that can be varied in this layer is network latency (L), as defined by the LogP model [40]. We assume that a messages arrives in exactly L time units (even when a process sends a message

System	Latency (μs)	Source
Cray T3D	2	[6]
TMC CM-5	6	[167]
Intel Paragon	6	[38]
FDDI	6	[115]
Myrinet	11	[38]
Fore ATM	33	[166]
Switched Ethernet	52	[89]

Table 7.1: **Measured Network Latency.** *The table shows the network latency of recent MPPs and Networks of Workstations.*

Variable	Description	Value
L	network latency	$10\mu s$
o	overhead	0
g	gap	0
P	number of processors	32
W	wake-up from message arrival	$50\mu s$ and $200\mu s$
Q	duration of time-slice	SSC : $100ms$ TS : from $20ms$ to $200ms$ (Table 5.1)

Table 7.2: **System Parameters in Simulations.** *The table summarizes the relevant network, machine, and operating system parameters in our simulator.*

to itself) and no congestion exists in the network or at endpoints. Finally, messages are always transmitted reliably and received in-order.

To configure L in our simulator, we examined the latency of current networks. Table 7.1 shows that network latency varies significantly in current systems. For most of our experiments, we choose $L = 10\mu s$ to closely match our implementation.

We do not examine the effect of packet sizes, message buffering, flow control, or network bandwidth (g) in our measurements. Furthermore, we do not model processing overhead (o) on either the sender or the receiver, due to the number of additional events that would be generated for every simulated message. These LogP values are summarized in Table 7.2. Overhead on the sending side can be modeled by simply increasing the amount of the computation that each process performs before each communication event. We do not expect that these effects will significantly impact the performance of implicit coscheduling.

Within `SIMplicity`, a process is notified of a message arrival through an interrupt. If the process is already running when the message arrives, then the message is handled immediately with no cost to the system. If the process is on the ready queue, then the

message is buffered until the process is scheduled. Finally, if the process is sleeping, it is woken and placed on the ready queue; whether the process is scheduled depends upon the characteristics of the local operating system scheduler.[1]

7.3 User Processes

At the highest level, `SIMplicity` is driven by the arrival of a parallel job at a specified time and requiring a certain number of processes, where the number of processes must be less than or equal to the number of workstations in the system. In our experiments, all jobs arrive at the same time and request 32 processes, matching the number of workstations.

Two types of operations exist within a parallel application: computation and communication. Computation is modeled as a constant amount of work, regardless of how the job is scheduled; no memory effects or I/O events are simulated. Since overhead, o, is not simulated, a process can handle incoming message requests while computing without increasing this fixed time.

7.3.1 Communication Primitives

Our model contains three communication primitives.

- **Request-Response:** A request-response message pair is denoted as a `read` operation. When a read is initiated, a request is sent from the source process to the destination; at this point, the source process begins the two-phase waiting algorithm. After L time units, the request message arrives at the destination process; when the destination process is scheduled, it handles the message and returns the response immediately. When the response arrives, the requesting process continues with its computation. Thus, a read requires exactly $2L$ units when all processes are scheduled.
- **One-Way Requests**: These operations are not employed in our simulated workloads.
- **All-to-all Synchronization:** The only synchronization operations in our simulated workloads are `barriers`. When a process arrives at a barrier, it sends a message to the root process. When the root has received P such messages, it calculates the load-imbalance of the current barrier and predicts the load-imbalance for the next barrier, as described in Section 6.4.2. This prediction, along with the barrier-completion message, is broadcast in a linear-fashion to the waiting processes. A process at a barrier waits until it has received this completion message before continuing.

[1] Note that the behavior of `SIMplicity` in the experiments presented in this dissertation differs in two primary areas from those originally presented in [47]. First, in the previous simulations, a sleeping process was given kernel priority (above 59) when a message arrived; thus, this process was almost always promptly scheduled. Second, the process immediately slept again if the arriving message was not the event for which the process was waiting.

Variable	Description	Equation	Value (when $W = 50\mu s$)
S_{Base}	**baseline spin**	S^R_{Base} or S^B_{Base}	
S^R_{Base}	read baseline spin	$\max(T^R_{Sched}, T^R_{Trigger}) = 2L + W$	$70\mu s$
T^R_{Sched}	request-response time (remote process scheduled)	$2L$	$20\mu s$
$T^R_{Trigger}$	request-response time (remote process triggered)	$2L + W$	$70\mu s$
S^B_{Base}	barrier baseline spin	$\max(T^B_{Sched}, T^B_{Trigger}) = 2L + W$	$70\mu s$
T^B_{Sched}	barrier time (remote processes scheduled)	$2L$	$20\mu s$
$T^B_{Trigger}$	barrier time (remote processes triggered)	$2L + W$	$70\mu s$
S_{Cond}	**conditional spin**	S^R_{Cond} or S^B_{Cond}	
S^R_{Cond}	request-response conditional spin	$3W$	$150\mu s$
S^O_{Cond}	one-way request conditional spin	—	—
S^B_{Cond}	barrier conditional spin	$3W$	$150\mu s$
L_{Block}	blocking latency	W	$50\mu s$
V_{Block}	blocking load-imbalance	$\approx 20 \cdot W$	$\approx 1ms$

Table 7.3: **Conditional Two-Phase Waiting Parameters for Simulation.** *The table summarizes the variables and equations for the time a process should spin before blocking.*

7.3.2 Waiting Algorithm

With `SIMplicity` we investigate three actions that a process can take when waiting for a response from a remote process. The waiting process may block-immediately, spin-wait, or perform a two-phase algorithm. In the majority of our experiments for implicit coscheduling, we investigate variations of the two-phase waiting algorithm, where we independently vary the amount of baseline spin and conditional spin for read and barrier operations.

Table 7.3 summarizes the ideal spin-times within our simulation environment as derived in Chapter 6. Since the simulator does not model overhead, o, or gap, g, most of the equations are simplified significantly.

We note that `SIMplicity` does not model the best approach for predicting whether a message will arrive in the conditional interval. The simulator does not record the arrival time of messages; therefore, when a process waits at a communication event, it does not know when the last message arrived and, thus, cannot readily predict if a message will arrive in the next S_{Cond} interval. As a result, when waiting, the process spins $\max(S_{Base}, S_{Cond})$, while recording whether a message arrives. If a message arrives, the process determines that it is beneficial to remain coordinated for another S_{Cond} interval. Since $S_{Cond} > S_{Base}$, processes pay an additional cost of $S_{Cond} - S_{Base}$ when a message does not arrive and scheduling is not coordinated. Thus, the simulator achieves slightly worse performance than the ideal implementation of implicit coscheduling.

7.4 Application Workload

Because the space covering the parameters we have introduced is too large to explore exhaustively, we fix a number of the workload and system characteristics as follows, unless otherwise noted:

- Each workload contains exactly three competing jobs.
- Each job is identical in its communication characteristics.
- Each job arrives at the same time.
- Each job contains 32 processes, matching the number of workstations.

Our performance metric is usually the slowdown of the last job in the workload to complete relative to an idealized model of explicit coscheduling. We compare to explicit coscheduling because we want to show the ability of implicit coscheduling to achieve coordinated scheduling. Thus, 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.

To explore the sensitivity of implicit coscheduling over a range of controlled communication parameters, we simulate two styles of synthetic parallel jobs: bulk-synchronous

and continuous communication. The bulk-synchronous style is common in many applications with bursty communication. However, because the bulk-synchronous applications are relatively easy to schedule, we also examine applications that communicate continuously.

Bulk-Synchronous Communication

A *bulk-synchronous* application alternates between phases of computation and communication, as shown in Figure 7.1. The behavior of a job is defined by several parameters: the mean length of a computation phase phases (g), the load-imbalance across processes within a computation phase (V), the time between reads within a communication phase (c), the number of communication phases to execute before termination, and the communication pattern. The computation time for each process is chosen independently from a uniform distribution, $(g - V/2, g + V/2)$. The time between communication operations, c, is fixed at $8\mu s$, while the number of phases is selected such that the execution of one application in a dedicated environment requires approximately 10 seconds of simulated time.

We investigate three different patterns of communication: `Barrier`, `NEWS`, and `Transpose`. The `Barrier` pattern consists of only one barrier and thus contains no additional dependencies across processes; the other two patterns consist of a sequence of reads surrounded by two barriers. In `NEWS`, a grid communication pattern is used; each process reads from its four nearest neighbors. The communication phase of `Transpose` contains P reads, where on the i-th read, process p reads data from process $(p + i) \bmod P$; therefore, each process depends on all other processes.

Figure 7.2 shows the characteristics of our synthetic benchmark programs as a function of the communication pattern, granularity, and load-imbalance. Each bar shows the percentage of time spent computing, communicating, or synchronizing (averaged over all processes) when the job is run in a dedicated environment. The communicating and synchronizing costs are the time spent spinning while waiting for an event to complete.

Examining the breakdowns of each bar, we see several trends as the communication characteristics of the programs are varied. First, as the load-imbalance in the programs increases (*i.e.*, moving to the right within a group of six bars), the time spent spinning during synchronization increases. Second, for the `NEWS` and `Transpose` programs, as the computation granularity decreases (*i.e.*, moving to the right across groups of bars), the communication-to-computation ratio of each program increases.

Continuous Communication

In the *continuous communication* synthetic benchmarks, there are no distinct phases of computation and communication; instead, each process reads from a cooperating process every c time units. As with the bulk-synchronous workload, the average time between barriers is represented by g and the load-imbalance across processes by V; processes compute and communicate for a period of time chosen independently from the interval $(g - V/2, g + V/2)$. This behavior is illustrated in Figure 7.3.

With the continuous-communication benchmarks, we examine two communication patterns: `NEWS` and `Random`. With the `NEWS` pattern, processes continuously communicate

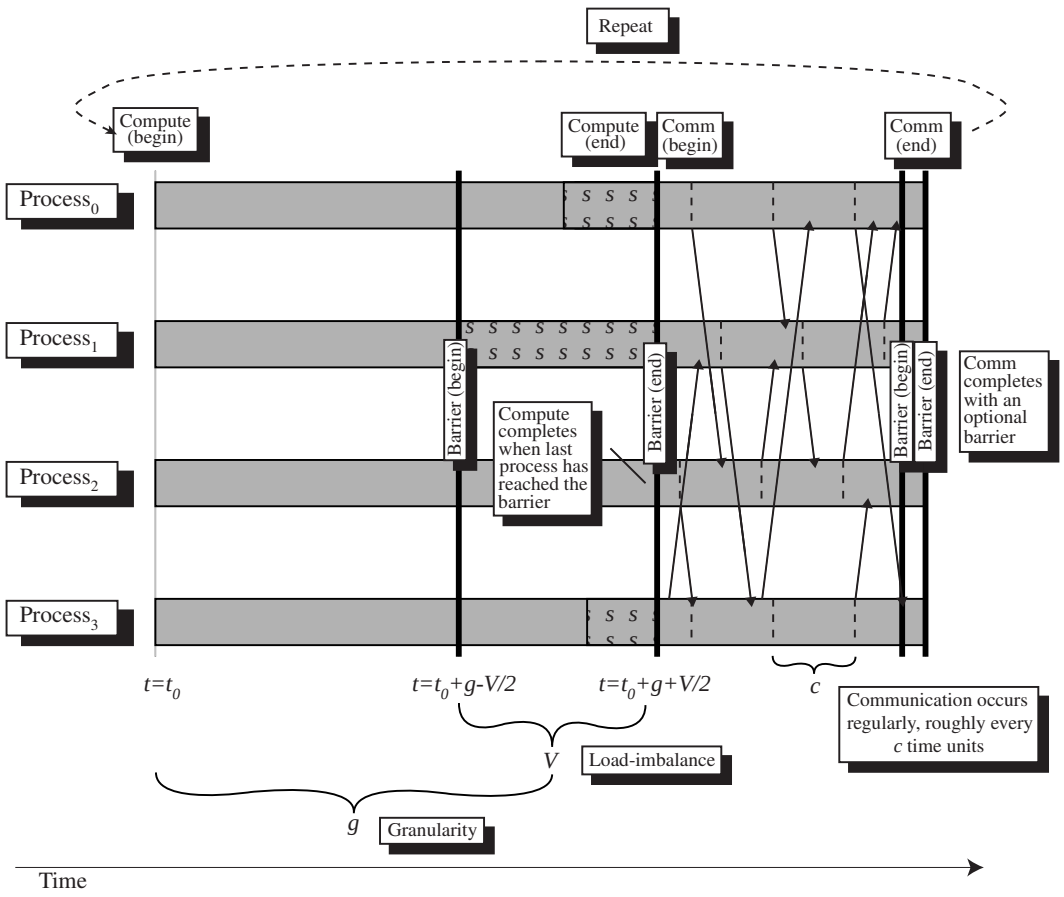


Figure 7.1: **Model of Bulk-Synchronous Synthetic Parallel Applications.** *Each process of a parallel job executes on a separate processor and alternates between computation and communication phases. Processes compute for a mean time g before executing the opening barrier of the communication phase; 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.*

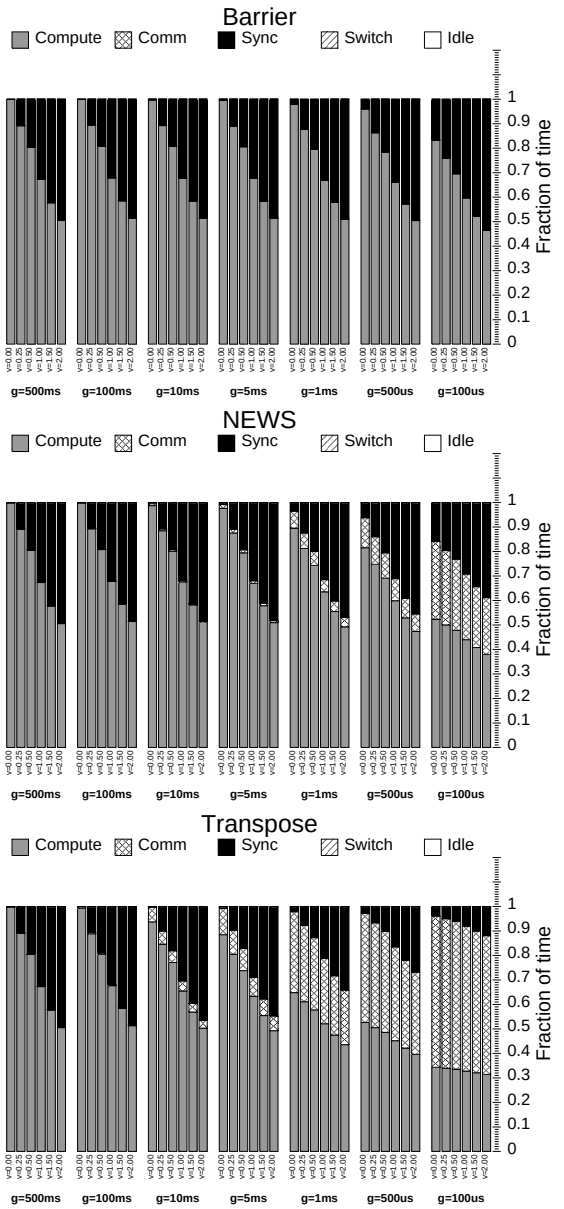


Figure 7.2: **Characteristics of Bulk-Synchronous Benchmarks.** *The three graphs show the breakdown of execution time for the three communication patterns. Along the* x *axis, the computation granularity, g, and the load-imbalance, V, are varied. Within a group of six bars, each job has the same computation granularity. Groups on the left are relatively coarse-grained; groups on the right are more fine-grained. Within each group, the load-imbalance is increased as a function of the granularity: from $V = 0$ to $V = 2 \cdot g$. The fraction of execution time is divided into three components (computing, communicating, and synchronizing). Note that the* `Barrier` *program performs no communication. This data assumes a network latency of 10μs.*

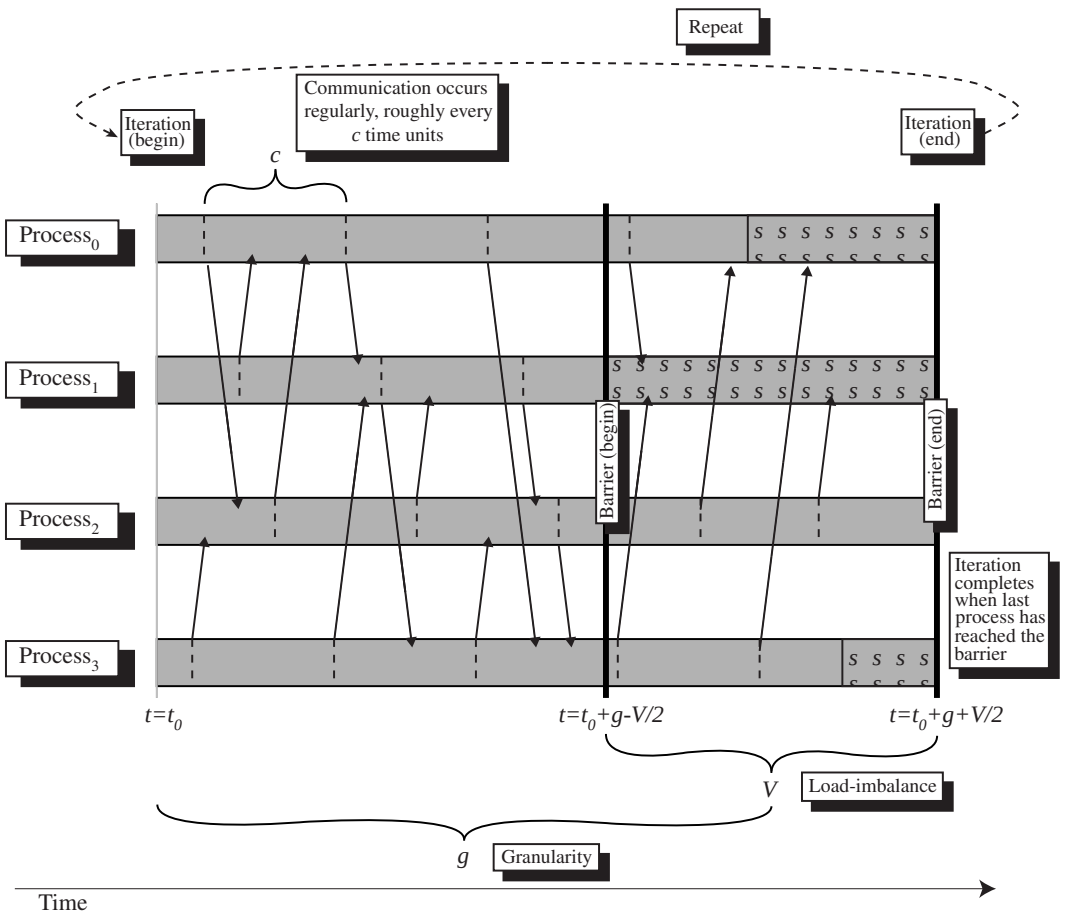


Figure 7.3: **Model of Continuous-Communication Synthetic Parallel Applications.** *Each process continuously communicates with other cooperating processes running on other processors. The average time between communication requests for a given process is c time units. Each process also periodically performs a barrier; the mean time between barriers is g, chosen uniformly from the interval* $(g - V/2,\ g + V/2)$.

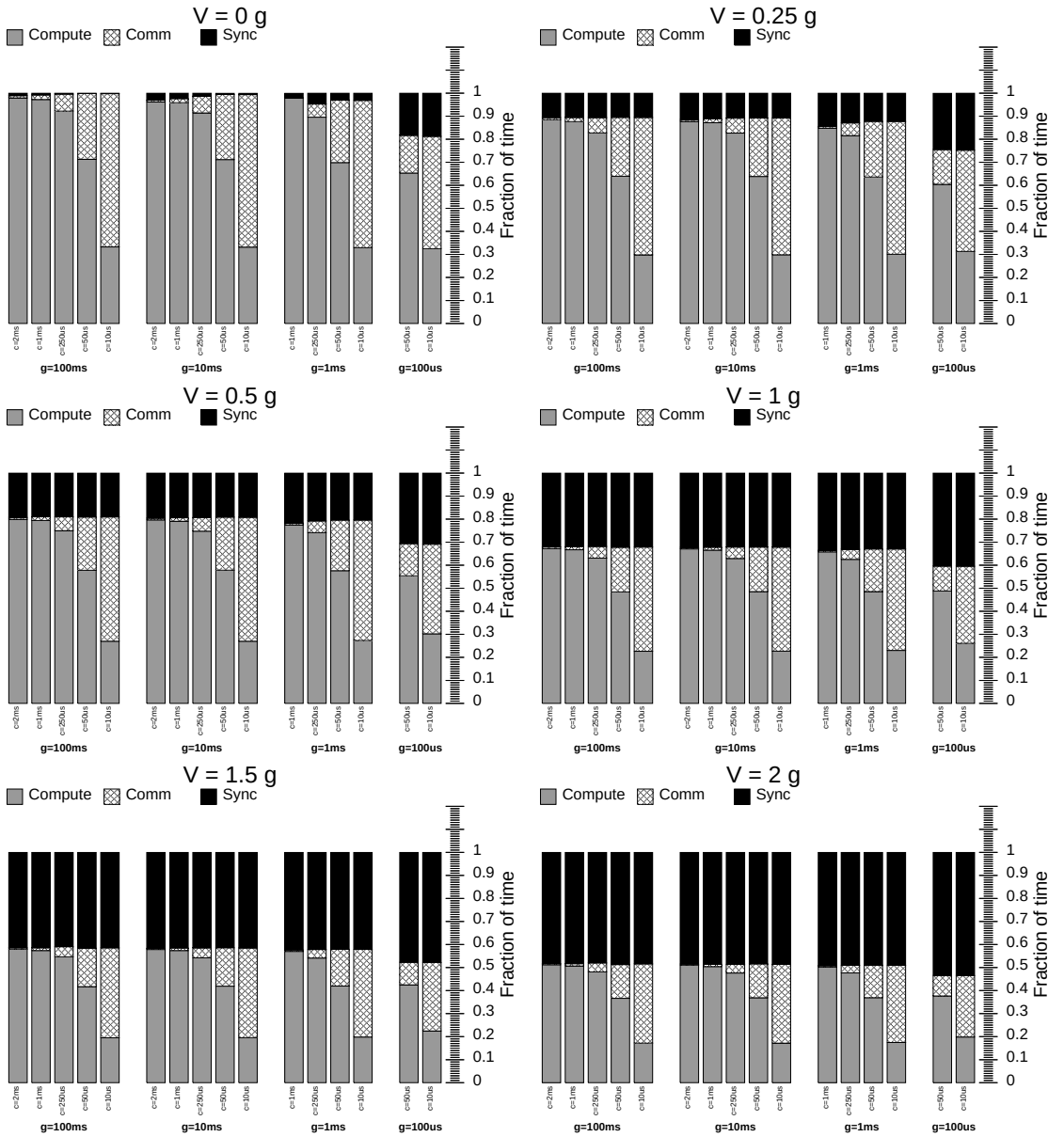


Figure 7.4: **Characteristics of Continuous-Communication Benchmarks.** *The six graphs show the breakdown of execution time for r* `NEWS` *communication pattern as the load-imbalance, V, is varied. Along the* x *axis, the average time between barriers, g, and time between read operations, c, is varied. Groups of bars on the left are relatively coarse-grained; groups on the right are more fine-grained. The fraction of execution time is divided into three components (computing, communicating, and synchronizing). This data assumes a network latency of 10μs.*

with each of their four nearest neighbors in a fixed rotation. In the `Random` pattern, processes communicate with a destination process chosen at random from the P cooperating processes (including itself).

Figure 7.4 illustrates the communication characteristics of the `NEWS` workloads; measurements with the `Random` pattern look identical. We note the division of execution time as the parameters are changed. First, as the amount of load-imbalance in the application increases (*i.e.*, as V increases across graphs), the time waiting for synchronization increases. Second, the smaller the interval between communication events (*i.e.*, as c decreases), the more time is spent waiting for communication relative to time spent computing. Finally, as the amount of computation between barriers decreases (*i.e.*, as g decreases), the relative amount of time spent performing barriers increases slightly.

7.5 Operating System Scheduler

We constructed the local scheduling component of `SIMplicity` to closely match that of the Solaris [67] scheduler in both functionality and structure; in fact, significant portions of our code are adapted directly from Solaris 2.4 source. Scheduling in Solaris, as in all SVR4-based schedulers, is performed at two levels: class-independent routines and class-dependent routines. Class-independent routines are those that are responsible for dispatching and preempting processes. Class-dependent routines are those that are responsible for setting the priority of each of its processes.

In this section, we first describe the functionality of the operating system scheduler that is independent of the scheduling class. We then describe the functionality that is specific to each of our three scheduling classes (*i.e.*, time-sharing, stride scheduling with system credit, and explicit coscheduling).

7.5.1 Class Independent Functionality

The class independent routines have three basic responsibilities. First, the process with the highest priority must be dispatched and the state of the preempted process must be saved. Second, the class-dependent routines must be notified of the state of its processes. Finally, processes must be moved between priorities as specified by the scheduling classes.

When dispatching processes, an important parameter of the operating system scheduler is the time to context-switch to a new process. In the simulations, this is a constant amount of time. We assume that waking a process on a message arrival, W, is equal to the basic context-switch cost, incurring no additional cost at other layers of the system. To configure W, we examined the context-switch costs for a variety of current operating system schedulers. Table 7.4 shows that the cost of context-switching varies significantly. Since the value of W has a large impact on scheduling performance, we chose two values: a moderate cost of $50\mu s$ to roughly match that of our implementation and a higher cost of $200\mu s$ to stress the performance of implicit coscheduling.

The scheduling classes should be informed whenever the state of one of its processes changes: at creation, termination, and when a blocked process becomes runnable. The scheduling class is also informed every $10ms$ if one of its processes is currently scheduled.

Machine	OS	Cost (us)	
		(0 KB)	(64 KB)
6600-990	AIX 3.x	13	17
PowerPC	AIX 4.x	16	119
K210	HP-UX B.10.01	17	31
MIPS R10K	IRIX64 6.2-no	21	25
P5-133	FreeBSD 2.2-C	27	55
P6	Linux 1.3.37	6	32
Alpha	Linux 1.3.57	10	13
8400	OSF1 V3.2	14	18
Alpha	OSF1 V3.0	53	96
UltraSPARC	SunOS 5.5	14	30
Average		19.1	56.0

Table 7.4: **Measured Context-Switch Cost.** *This table shows the context switch costs of various modern systems, based on data found in [121]. The left column shows the cost of a context-switch when each of two null processes switch back and forth. The column on the right displays the cost when each of the two processes have 64 KB of cache state that they continuously access; in that case, the cost is noticeably higher.*

We make several pessimistic assumptions about the synchronization of these notification events across processors. 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.

A process is moved between runnable priorities as specified by its scheduling class. When a process sleeps on any event (*e.g.*, communication, synchronization, or disk I/O), it is placed on a list of blocked jobs and is not scheduled, regardless of its priority.

7.5.2 Scheduling Classes

Each process in the system belongs to exactly one scheduling class. By default, SVR4 officially supports a time-sharing class and a real-time class; however, new scheduling classes can be added as necessary. We briefly describe our implementation of the default time-sharing class, as well as our extension of stride scheduling with system credit and explicit coscheduling.

Time-Sharing

The time-sharing class is an important scheduler to consider because it is often used in practice. In our experiments, we use the time-sharing scheduler (TS) whenever the results do not depend upon the ability of the local operating system scheduler to allocate resources

fairly; that is, whenever scheduling jobs with the same communication characteristics. Our implementation of the TS class is strongly based on the time-sharing class within Solaris 2.4 and matches the description in Section 5.2.

Stride Scheduling with System Credit

To fairly schedule applications with different communication characteristics, we simulate the extended stride scheduler described in Section 5.3. In this section, we describe two interesting issues that occur due to interactions with the class independent functions.

The first issue occurs because the class independent functions are based on priorities, whereas stride scheduling is not. The class independent functions dispatch the process with the highest priority. However, the goal of stride scheduling is to dispatch the process with the minimum `pass` value. To ensure that the class independent function schedules the desired process, the stride scheduling class raises the priority of the process with the lowest `pass` and lowers the priority of all other processes. Priority changes are required in three circumstances: when the time-slice of the scheduled process expires ($Q = 100ms$), when a process with a lower `pass` value than the scheduled process becomes runnable (*e.g.*, due to a message arrival), and when the running process voluntarily relinquishes the processor.

The second issue arises when accounting for the amount of CPU each process has consumed and subsequently incrementing the `pass` of the process. The class independent functions do not provide an interface for notifying a scheduling class when a process is scheduled or descheduled; instead, the scheduling class is notified of the running process at each $10ms$ clock tick. Within the clock tick routine, the stride scheduler increments the `pass` of the scheduled process by its `stride` and decrements the remaining ticks within the allocated time-slice. While this approach may lead to slight accounting inaccuracies if a process relinquishes the processor before the clock tick occurs, it has been found to be precise enough for implicit coscheduling.

Explicit Coscheduling

To compare the performance of implicit coscheduling, we have implemented a version of explicit coscheduling as a scheduling class within `SIMplicity`. To build on the class independent functions, the priority of the job to be coscheduled is raised to the highest in the system, while the priorities of the other jobs remain low. Our implementation is based on Ousterhout's matrix algorithm [134]. When explicitly coscheduled, processes always spin-wait for communication events to complete, as they would in a dedicated environment.

Our implementation of explicit coscheduling is optimistic in a number of areas. First, there is 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 ($Q = 500ms$) relative to the context-switch cost.

Chapter 8

Simulation Results

In this chapter, we evaluate variations of the conditional two-phase waiting algorithm and the local operating system scheduler on synthetic workloads. With our simulator, we confirm five main points presented in Chapter 6:

1. **Coordinated Scheduling:** Coordinated scheduling, such as that achieved with explicit or implicit coscheduling, is required for fine-grain jobs in current cluster systems. However, when network latency is greater than the context-switch cost, performance with implicit coscheduling can exceed that of explicit coscheduling.

2. **Baseline Spinning:** When processes spin for the expected completion time of communication operations, the scheduling of processes stays coordinated across machines.

3. **Conditional Spinning:** Processes should spin longer when receiving messages from remote processes to stay coordinated and to benefit the job as whole.

4. **Load-Imbalance:** The performance of applications with high load-imbalance can be improved if applications approximate the amount of load-imbalance at run-time.

5. **Local Scheduler with Fair Cost-Model:** A local scheduler that allocates resources fairly is required to handle jobs that communicate at different rates.

8.1 Scheduling Coordination

In Section 6.4, we argued that coordinated scheduling is superior to uncoordinated scheduling when the cost of context-switching between processes exceeds the latency of the network. In this section, we verify this result with our simulator for a range of workloads and system parameters.

8.1.1 Sensitivity to Network Latency

To verify the conditions under which coordinated scheduling should be used, we compare the performance of local scheduling with immediate blocking (*i.e.*, uncoordinated

scheduling) to explicit coscheduling with spin-waiting (*i.e.*, a form of coordinated scheduling); we do not consider local scheduling with spin-waiting, since it was shown in Section 3.3.1 to perform poorly for all workloads. We evaluate two different context-switch costs, 50 and $200\mu s$, and a range of network latencies from 10 to $300\mu s$. For simplicity, we use the Solaris Time-sharing scheduler as the local operating system scheduler. Our performance metric, slowdown, is the ratio of the workload completion time under local scheduling to the completion time under ideal explicit coscheduling.

Our initial workloads contain three applications running on 32 workstations. We examine both applications in which the amount of communication is balanced across all processes and applications in which it is not. To represent balanced communication, we measure continuous-communication applications. To represent unbalanced communication, we examine bulk-synchronous applications; bulk-synchronous applications perform many barrier operations for which the root process becomes the bottleneck.

Balanced Workloads

Figure 8.1 shows the slowdown of the balanced communication workload. In each of the applications, there are very few barriers ($g = 100ms$) and there is no load-imbalance ($V = 0$). Each process communicates with the `NEWS` pattern. Five lines are shown in each graph, each designating a different communication granularity, between $c = 10\mu s$ and $c = 2ms$. We make three observations from the figure.

First, as predicted for balanced workloads, coordinated scheduling should be used when network latency is less than context-switch time; uncoordinated scheduling should be used otherwise. The graphs verify that the slowdown of uncoordinated scheduling is greater than 1 when $L < W$ and less than 1 when $L > W$ for all communication frequencies and for both context-switch costs.

Second, because a context-switch is incurred for every communication operation with uncoordinated scheduling, a higher context-switch cost results in worse performance for a given network latency. For example, when $L = 1\mu s$ and $W = 200\mu s$, immediate blocking can result in slowdowns more than ten times worse than coscheduling; when $W = 50\mu s$, the relative slowdown decreases to under four.

Third, the two graphs show that applications with frequent communication are more sensitive to the scheduling policy than those with infrequent communication. However, because the same scheduling policy (either local scheduling or coscheduling) must be used for all jobs in the system, coordinated scheduling should be applied when the most sensitive jobs require it.

Unbalanced Workloads

We next evaluate the performance of bulk-synchronous applications with unbalanced communication. With the bulk-synchronous applications, each process performs a barrier every g units, communicates in a `NEWS` pattern after $c = 8\mu s$ of intervening computation, and then performs a second barrier; there is no load-imbalance across processes between barriers (*i.e.*, $V = 0$). Three lines are shown in each graph, varying the time between barriers, g.

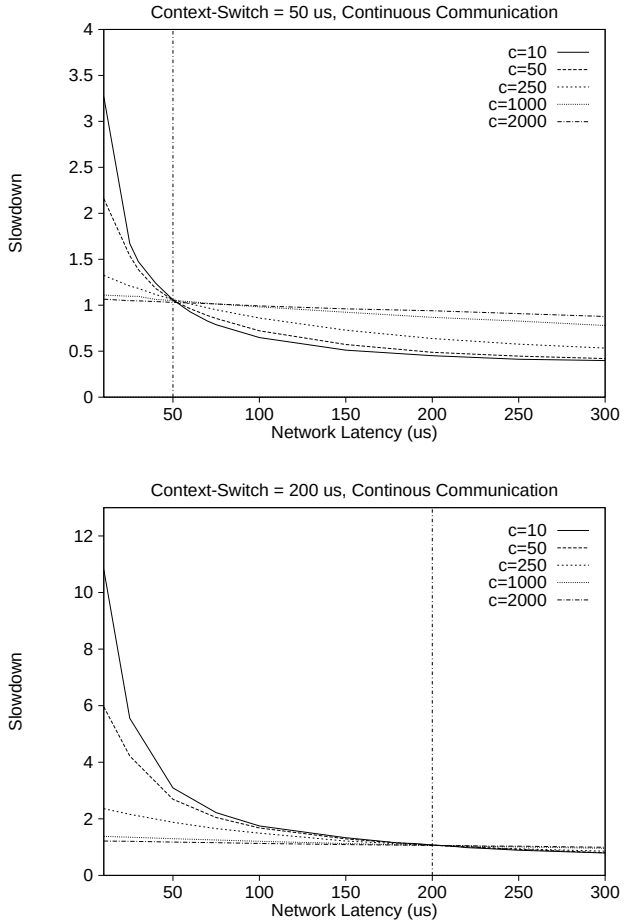


Figure 8.1: **Impact of Latency and Context-Switch Time on Continuous-Communication Workloads.** *Each workload consists of three competing jobs each containing 32 processes time-shared on a total of 32 workstations. Each of the three jobs follows the continuous communication model. The time between barriers, g, is 100ms, and there is no load-imbalance ($V = 0$). Each process communicates with its four nearest neighbors in a regular* `NEWS` *pattern. Each line in the graph designates a different communication granularity; the average time between reads, c, is set to 10, 50, 250, 1000, and 2000μs. The metric along the y-axis is the slowdown of the workload when scheduled independently with immediate blocking versus explicit coscheduling with spin-waiting. In each graph, network latency, L, is increased from 10μs to 300μs along the x-axis. The vertical lines designate the point where $L = W$. Note the change in scales in the y-axes across the two graphs. Context-switch time is changed in the two graphs from $W = 50\mu s$ to $W = 200\mu s$.*

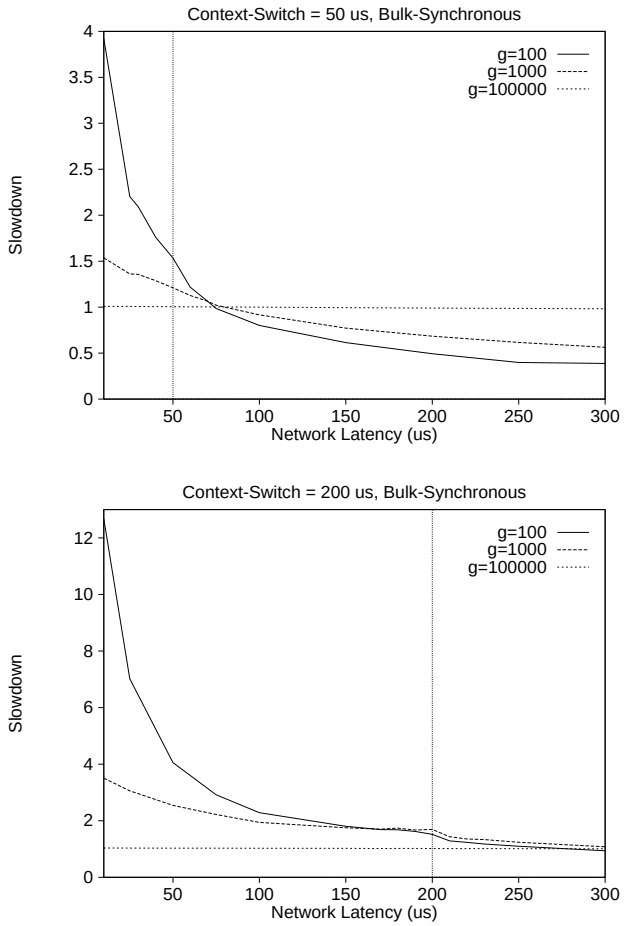


Figure 8.2: **Impact of Latency and Context-Switch Time on Bulk-Synchronous Workloads.** *Each workload consists of three competing jobs consisting of 32 processes each time-shared on 32 workstations. Each of the three jobs follows the bulk-synchronous communication model. Each line designates a different synchronization granularity; the time between barriers, g, is set to 100μs, 1ms, and 100ms. There is no load-imbalance ($V = 0$), and the time between reads to the four nearest-neighbors, is fixed at $c = 8\mu s$. The metric along the y-axis is the slowdown of the workload when scheduled independently with immediate blocking versus explicit coscheduling with spin waiting. Network latency, L, is increased from 10μs to 300μs along the x-axis. The vertical lines designate the point where $L = W$. Note the change in scales in the y-axes across the two graphs. Context-switch time is changed in the two graphs from $W = 50\mu s$ to $W = 200\mu s$.*

The two graphs in Figure 8.2 show that for the unbalanced workload, coordinated scheduling should not only be applied when the latency, L, is less than W, but also when L is slightly greater than W. The implication is that coordinated scheduling is more important for unbalanced workloads because it helps the overloaded process avoid a costly context-switch; such an imbalance is expected in client-server applications where the server is more loaded than the clients.

With our synthetic workload, the process forming the root of the barrier sends and receives more messages than other processes. When local scheduling is used, the root process performs a context-switch on every arriving message, and thus becomes the bottleneck of the job. Therefore, when network latency is slightly greater than the context-switch cost, better performance is achieved when work is moved away from the root process; that is, it is better for the leaf processes to pay a cost of $2L$ to spin-wait than for the root to pay W for each cooperating process.[1]

8.1.2 Sensitivity to Workload Parameters

For the remaining experiments in this section, we examine a richer set of bulk-synchronous and continuous-communication workloads, but fix network latency at $10\mu s$, which approximately matches the value in our implementation.

Bulk-Synchronous Communication

To better understand the scheduling requirements of the applications in our workload, we first compare the performance of local scheduling with immediate blocking to explicit coscheduling with spin-waiting. The graphs in Figure 8.3 show slowdown for the two context-switch costs as a function of the computation granularity and load-imbalance of the application for three different communication patterns: `Barrier`, `NEWS`, and `Transpose`. The breakdowns of each bar show the percentage of time spent in each of the five phases; note that no time is spent waiting for either communication or synchronization when processes block immediately.

As expected, coordinated scheduling is strictly superior for fine-grained jobs regardless of the communication pattern or the internal load-imbalance of the jobs. For example, the `Transpose` pattern synchronizing every $100\mu s$ when scheduled locally exhibits slowdowns nearly four times or 16 times worse than with explicit coscheduling, depending upon the context-switch cost. Context-switch cost has a large impact on the slowdown of local scheduling, since a context-switch is incurred on every communication and synchronization event.

The less communication in the workload, the less sensitive is performance to the scheduling algorithm. In fact, with infrequent synchronization (*e.g.*, a barrier is performed no more often than every $5ms$) and high load-imbalance (*e.g.*, one to two times the computation granularity), local scheduling with immediate blocking performs slightly better than coscheduling. These results are well understood: blocking immediately is beneficial

[1] This particular workload could be easily fixed to avoid the unbalanced contention at the root process; for example, a two-way tree barrier could be used instead of the current barrier. However, other unbalanced communication patterns will always exist.

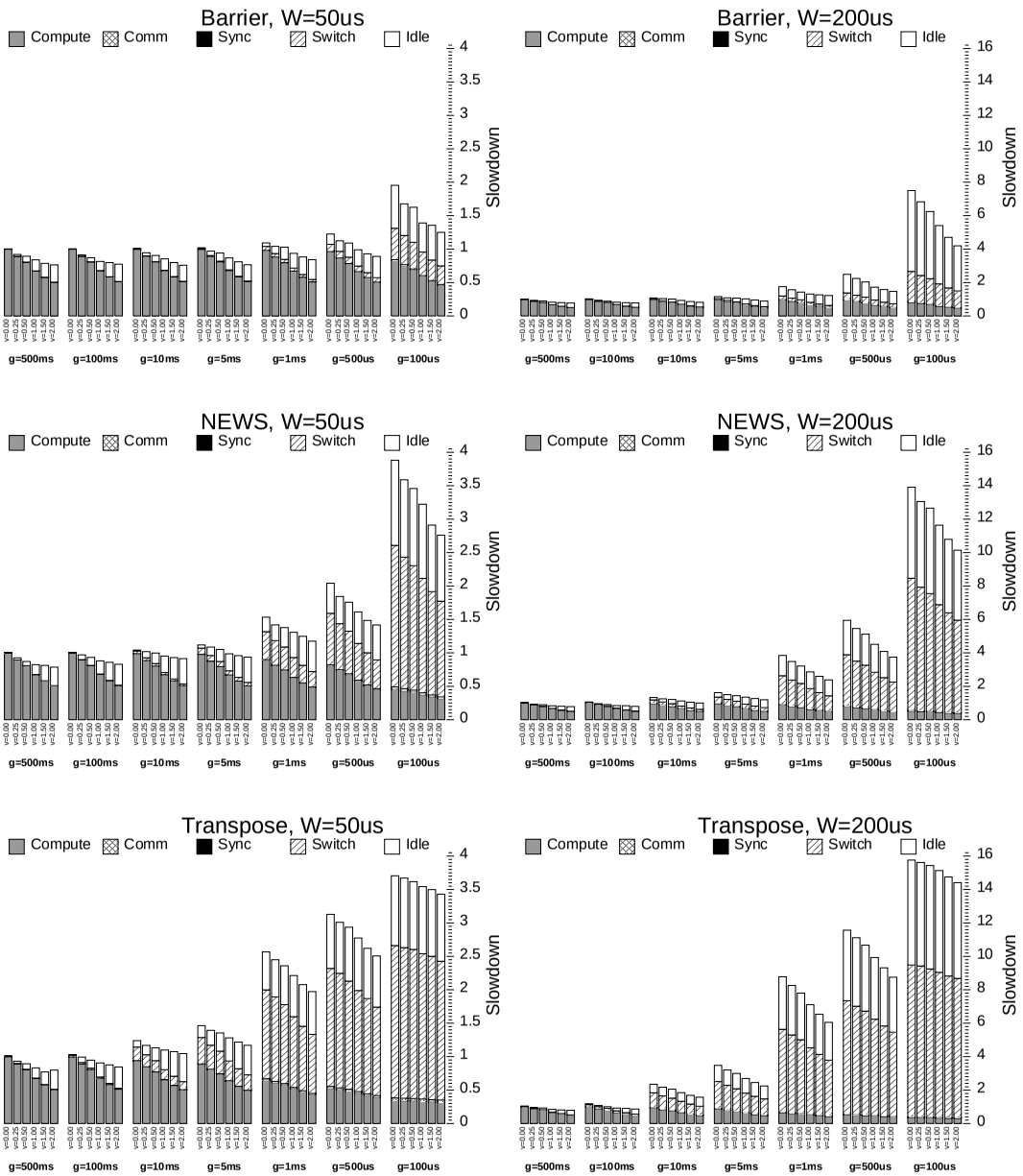


Figure 8.3: **Performance of Immediate Blocking for Bulk-Synchronous Programs.** *The graphs show the slowdown of uncoordinated scheduling with immediate blocking versus explicit coscheduling with spin-waiting for three communication patterns (*`Barrier`*,* `NEWS`*, and* `Transpose`*) and two context-switch costs,* $W = 50\mu s$ *and* $200\mu s$*. Network latency is set to* $L = 10\mu s$*. Along the* x *axis, the computation granularity,* g*, and the load-imbalance,* V*, are varied. Each group of six bars has the same computation granularity. Within each group, the load-imbalance is increased as a function of the granularity: from* $V = 0$ *to* $V = 2 \cdot g$*.*

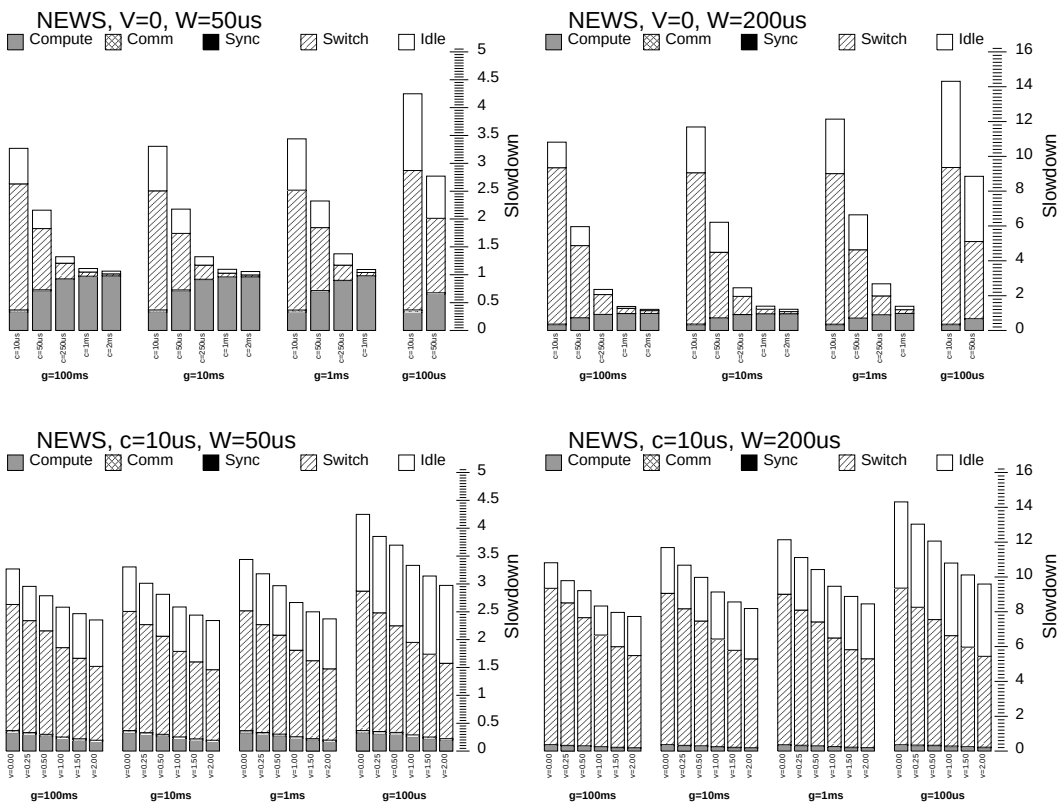


Figure 8.4: **Performance of Immediate Blocking for Continuous-Communication Programs.** *The graphs show the slowdown of uncoordinated scheduling with immediate blocking versus explicit coscheduling for the* `NEWS` *communication pattern and two context-switch costs,* $W = 50\mu s$ *and* $W = 200\mu s$*. In the top two graphs, load-imbalance is fixed at* $V = 0$*; along the* x *axis, the time between barriers,* g*, and the time between reads,* c*, are both varied. In the bottom two graphs,* c *is fixed* $10\mu s$ *and* g *and* V *are varied. In all graphs, network latency is set to* $L = 10\mu s$*.*

during load-imbalance because the processor can switch to and execute another process instead of spinning uselessly. The impact of load-imbalance is investigated in more detail in Section 8.4.

Continuous Communication

Figure 8.4 presents measurements for a workload containing three continuous-communication applications. Each process in each job continuously reads from each of its four nearest neighbors in a regular rotation[2], performing a barrier every g time units, which varies between $100\mu s$ and $100ms$. The first set of graphs varies the amount of time between reads, c, for jobs with no load-imbalance. The second set of graphs fixes c at $10\mu s$ and varies the amount of load-imbalance.

[2]The `Random` communication pattern has almost identical performance.

The results are very similar to those for bulk-synchronous applications in two respects. First, the higher the context-switch cost, the more sensitive is performance to the scheduling policy. Second, as shown in the first set of graphs, applications that communicate frequently (*e.g.*, $c = 10\mu s$) exhibit larger slowdowns with local scheduling than applications that rarely communicate (*e.g.*, $c = 1ms$).

However, unlike the bulk-synchronous workloads where the value of g determined the communication rate as well, performance is relatively insensitive to the value of g in the continuous-communication benchmarks; in these benchmarks, c strictly determines the communication rate. Furthermore, as shown in the second set of graphs, programs with a large amount of load-imbalance still require coordinated scheduling if they communicate continuously at a frequent rate.

8.1.3 Discussion

In this section, we have verified that coordinated scheduling (in the form of explicit coscheduling) is superior to uncoordinated scheduling (in the form of local scheduling with immediate blocking) for fine-grain applications on current clusters. Our simulation results confirm our analysis in Section 6.4.1 showing that coordinated scheduling is beneficial when network latency, L, is less than the context-switch cost, W.

This section has also helped to pinpoint the range of applications which are sensitive to the scheduling approach. In the remainder of this dissertation, among the bulk-synchronous jobs, we examine the `NEWS` pattern in more detail, focusing on more fine-grain applications (*i.e.*, $g = 100\mu s$ and $g = 1ms$). For the continuous-communication workload, we examine the range of communication rates where $c \leq 2ms$; to examine a workload that differs radically from bulk-synchronous applications, we examine continuous-communication jobs with relatively few barriers ($g = 100ms$). For simplicity, we begin with applications in both workloads that contain no load-imbalance (*i.e.*, $V = 0$).

8.2 Baseline Spin

The previous section showed that coordinated scheduling is important for fine-grain applications; we now must show that coordinated scheduling can be achieved with implicit coscheduling. In this section, we show that two-phase waiting with the correct baseline spin significantly improves performance relative to blocking immediately. We also show while spinning the baseline amount achieves performance similar to explicit coscheduling for bulk-synchronous applications, it is not adequate for continuously-communicating applications.

Throughout our experiments, we will see that a good predictor of performance is the percentage of communication and synchronization operations that complete successfully. We say that an operation is *successful* if it completes while the initiating process is still spinning in the first phase of the waiting algorithm. Thus, processes which are coordinated will successfully complete their remote operations.

8.2.1 Bulk-Synchronous Communication

Our first experiments in this section measure the performance of three competing bulk-synchronous jobs, while varying the baseline spin in the two-phase waiting algorithm. No pairwise spinning is performed in any of the experiments in this section. The performance metric, *slowdown*, is the ratio of the workload completion time with two-phase waiting and local scheduling to explicit coscheduling with spin-waiting. Each of the processes perform a barrier every $g = 100\mu s$ or $g = 1ms$, communicate with its four nearest neighbors after $c = 8\mu s$ of intervening computation, and then perform a second barrier; there is no load-imbalance (*i.e.*, $V = 0$). We continue to evaluate two different context-switch costs, $W = 200\mu s$ and $W = 50\mu s$ and a single network latency, $L = 10\mu s$.

Sensitivity to Read Baseline

To separately evaluate the impact of baseline spin on reads and barriers, we begin by varying the baseline spin for read operations while keeping the baseline spin for barriers fixed at its optimal point of S^B_{Base}. These results are shown in Figure 8.5 for the two context-switch costs. As shown, when processes waiting for remote read operations spin for only a short time (*i.e.*, less than T^R_{Sched}= $2L + 4o$= $20\mu s$), the performance of two-phase waiting is very poor, matching the performance of immediate blocking shown previously in Figure 8.2. As expected, as the baseline spin for reads is increased, performance improves dramatically at two distinct baseline spin amounts.

The first improvement in performance occurs when processes spin at least the round-trip time of the network at reads, T^R_{Sched}. For example, for an application synchronizing every $g = 100\mu s$ on a machine with a context-switch cost of $W = 200\mu s$, the performance with a baseline spin of $10\mu s$ is 11 times worse than explicit coscheduling, but is only 60% worse than explicit coscheduling with a baseline spin of $20\mu s$. Performance improves significantly in this region because spinning T^R_{Sched} keeps the initiating process coordinated when the remote process is scheduled before the request message arrives; in this region, the success rate of read operations increases to 97%.[3]

The second improvement in performance occurs when processes spin $T^R_{Trigger}$= $2L + W$. Spinning for an additional context-switch keeps processes coordinated when the arrival of the read request triggers the scheduling of the destination process on the remote machine. When processes spin this additional amount, the rate of successful reads increases from 97% to 99.9%. Due to this increase in successful operations when processes spin $T^R_{Trigger}$, performance improves to within 2% or within 10% of ideal explicit coscheduling, depending upon whether the context-switch cost is 50 or $200\mu s$.

Spinning for longer than $T^R_{Trigger}$ does not further increase the read success rate and, therefore, is not beneficial. In general, spinning longer hurts the overall execution time of the program, since the waiting process pays a higher penalty when the operation is not successful. However, in this case, longer spin times do not noticeably degrade performance

[3]When processes spin less than T^R_{Sched}, 15% of reads complete successfully; a read can appear complete in less time than the round-trip time of the network if the initiating process is involuntarily context-switched out while still spinning and is then rescheduled after the message arrives.

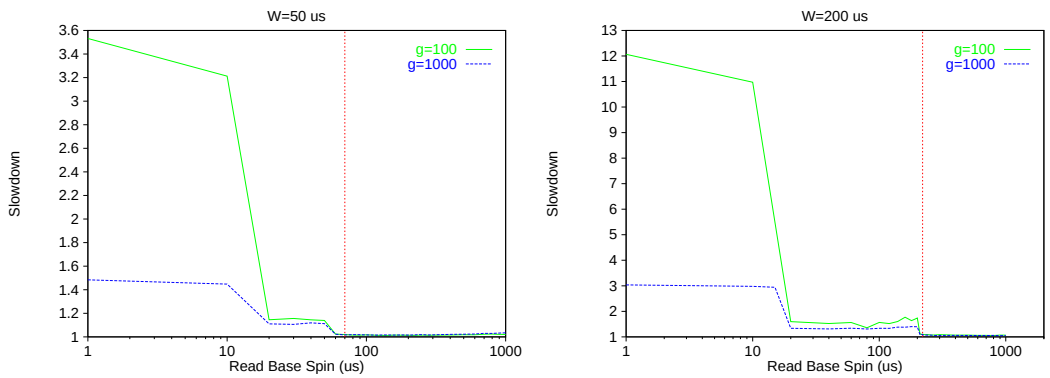


Figure 8.5: **Sensitivity to Read Baseline Spin for Bulk-Synchronous Programs.** The workload consists of three bulk-synchronous jobs on 32 workstations. The two lines designate different synchronization granularity; the time between barriers, g, is set to either $100\mu s$ or $1ms$. The vertical line in each graph designates the desired read baseline spin, $S^R_{Base} = 2L + W$. There is no load-imbalance ($V = 0$), and the time between reads after the barrier is fixed at $c = 8\mu s$. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the baseline spin for reads is varied between $1\mu s$ and $1000\mu s$, while the baseline spin for barriers is fixed at S^B_{Base}. When a process wakes after a message arrival other than the desired message, it spins for $S_{Wake} = W$. Network latency is fixed at $L = 10\mu s$. In Figure a, the context-switch time is set to $W = 50\mu s$; in Figure b, $W = 200\mu s$. Note the change in scale along the y-axes across the two graphs.

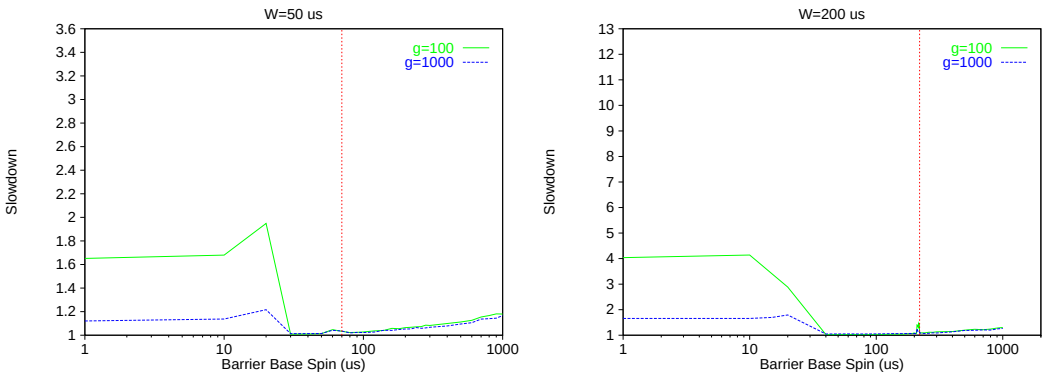


Figure 8.6: **Sensitivity to Barrier Baseline Spin for Bulk-Synchronous Programs.** *The environment and workloads are identical to that in the previous figure. Along the x-axis, the baseline spin for barriers is varied while the read spin time is kept constant at* S^R_{Base}. *The vertical line in each graph designates the desired barrier baseline spin,* $S^B_{Base} = 2L + W$ *In Figure a, the context-switch time is set to* $W = 50\mu s$*; in Figure b,* $W = 200\mu s$.

because most read operations complete successfully and, therefore, the larger spin penalty is paid for very few (0.1%) of the read operations.

Sensitivity to Barrier Baseline

The graphs in Figure 8.6 illustrate performance as the barrier baseline spin is varied while the read baseline spin is held constant at S^R_{Base}. Once again, two-phase waiting improves performance significantly compared to blocking immediately; however, waiting for $T^B_{Trigger}$=$2L + W$does not result in any benefits beyond waiting T^B_{Sched}=$2L$. If processes spin T^B_{Sched}, then 98% of the barriers complete successfully; as processes spin longer, the percentage of successful operations does not increase further. As a result, performance degrades slowly due to the extra spin time on the 2% of unsuccessful barriers.[4]

The slowdown when processes do not wait long enough for barriers to complete is not as severe as when processes do not wait for reads to complete. For example, with $W = 200\mu s$, blocking immediately at barriers leads to performance four times worse than explicit coscheduling, while blocking immediately at reads is 12 times worse. There are two reasons for this difference. First, each process performs twice as many read operations as barriers (*i.e.*, each set of four reads in the NEWS pattern is contained between two barriers); therefore, the context-switch cost when processes do not wait for a barrier to complete is paid for fewer operations. Second, whether a process waits for a read has a large impact on both the read and the barrier success rate, but whether a process waits for a barrier impacts only the barrier success rate.

Whether a read is successful determines if subsequent barriers are successful for the following reasons. If processes become uncoordinated at a read, then the processes will

[4] We do not fully understand the reason for the small area of poor performance in the region immediately preceding $T^B_{Trigger}$.

not reach the next barrier simultaneously and will not spin successfully through the barrier in the expected amount of time. This is indicated in our measurements by the fact that when the read baseline spin is less than T^R_{Sched}, the barrier success rate is only 30% even when processes spin S^B_{Base} at barriers. However, the barrier success rate increases to 95% when the read baseline spin is increased past T^R_{Sched} and to 98% when the read baseline spin is greater than $T^R_{Trigger}$. In summary, barrier success rates follow read success rates.

Conversely, read success rates are relatively independent of barrier spin times and barrier success rates. First, reads have less strict completion requirements. For a read to complete successfully, only the destination process must be scheduled and it does not need to have been scheduled for a particular length of time. For a barrier to complete successfully, not only must all the other $P-1$ processes in the job be scheduled, but they must have been coordinated since the last barrier. Second, an unsuccessful barrier can trigger the coordinated scheduling of the participating processes, allowing subsequent reads to complete successfully. Since all of the processes waiting at a barrier receive the barrier-completion message simultaneously from the root process, the cooperating processes may all then be woken and scheduled. Thus, regardless of the barrier baseline spin amount, read operations are successful 99.9% of the time as long the process waits S^R_{Base}.

8.2.2 Continuous-Communication Workloads

Our previous measurements showed that bulk-synchronous workloads can achieve performance within 2% or 10% of ideal, depending upon the context-switch cost, as long as processes wait the baseline spin amount at communication and synchronization events. In our next experiments, we show that no fixed baseline spin amount is adequate for applications that communicate continuously with infrequent barriers.

We examine a workload of three continuous-communication applications, each of which has infrequent barriers ($g = 100ms$) and no load-imbalance ($V = 0$). For both the `NEWS` and `Random` patterns, we measure five different communication rates. As with the previous simulations, we examine two different context-switch costs ($W = 200\mu s$ and $50\mu s$). In these experiments, we change the base time of reads and barriers simultaneously, rather than evaluating each in turn; since barriers are performed rarely, its baseline spin time has negligible impact on performance.

As shown in Figure 8.7, no amount of baseline spin achieves acceptable performance for fine-grain applications (*i.e.*, $c = 10\mu s$ and $c = 50\mu s$); the performance exhibited by some fine-grain applications remains 4.5 times worse than explicit coscheduling. As the baseline spin amount is increased, the performance of the fine-grain applications falls into three distinct regions.

In the first region, the baseline spin is less than $T^R_{Trigger}$, and slowdown is very high for all workloads. Unlike the bulk-synchronous workloads which exhibited a performance improvement when the baseline spin exceeded T^R_{Sched}=$2L$, the performance of continuous-communication workloads does not improve until processes spin at least $T^R_{Trigger}$=$2L+W$. After fine-grain applications spin at least $T^R_{Trigger}$, the percentage of successful read operations increases from less than 10% to about 80%; in this region, performance roughly doubles.

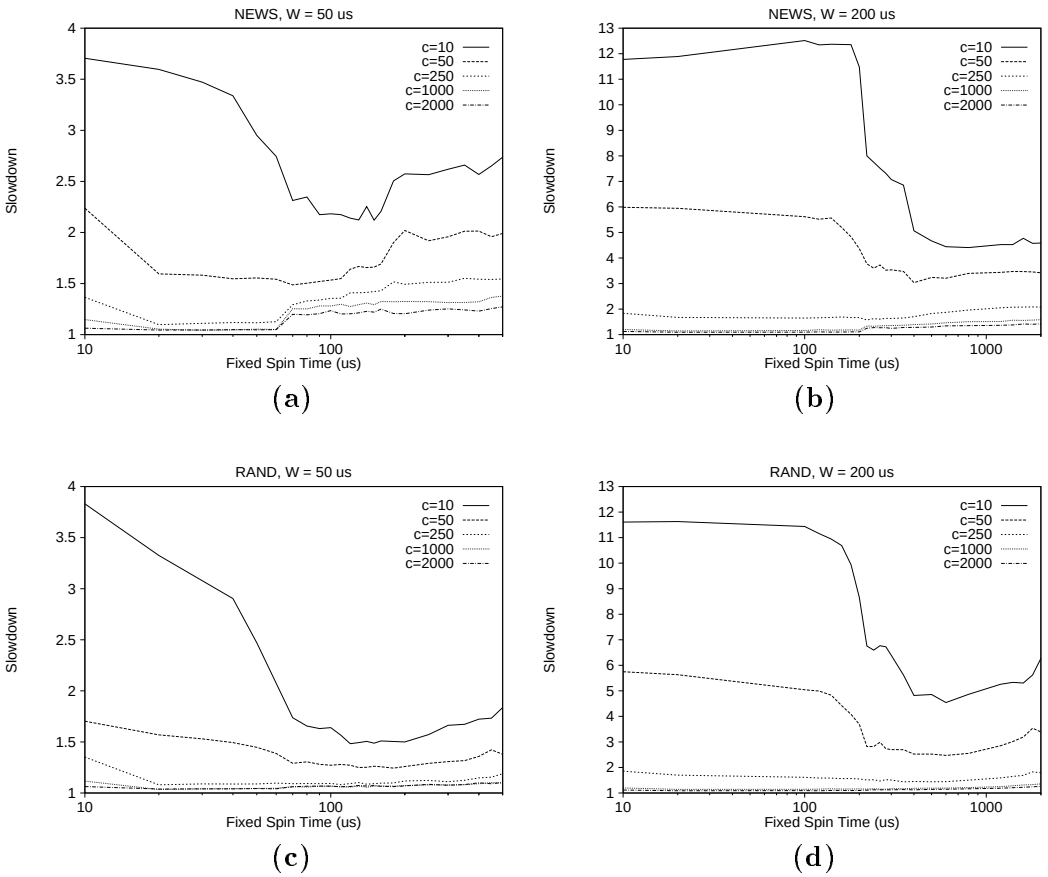


Figure 8.7: **Sensitivity to Baseline Spin for Continuous-Communication Programs.** *Three continuously communicating applications are time-shared on 32 workstations. The time between barriers is set to $g = 100ms$ with no load-imbalance ($V = 0$). Each line in the graphs designates a different communication granularity, c. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the baseline spin amounts for both reads and barriers are changed simultaneously. In Figures a and c, the context-switch cost is set to $W = 50\mu s$; in Figures b and d, $W = 200\mu s$; note the change in scale along the y-axis between the two sets. In the top two graphs, processes communicate in a repeated NEWS pattern; in the bottom two graphs, processes communicate with random destinations.*

In the second region, processes spin more than $T^R_{Trigger}$, but less than some optimal baseline spin amount. The graphs indicate that this optimal amount depends on the frequency of communication within the application: longer baseline spin times are more beneficial with more frequent communication. In this region, the percentage of successful operations increases very little and performance remains relatively constant.

Finally, the third region occurs as processes increase their baseline spin time up to and beyond the optimal baseline spin amount. At the optimal point, the success rates of operations increase suddenly from 80% to approximately 90%. As processes spin longer than the optimal amount, the percentage of successful operations may very gradually improve; however, overall performance degrades due to the extra penalty paid for spinning at unsuccessful operations.

8.2.3 Discussion

In conclusion, the effectiveness of baseline spin depends largely upon whether processes communicate in a bulk-synchronous or a continuous manner. For bulk-synchronous applications, when the read and barrier baseline spin amounts are set to S^R_{Base} and S^B_{Base} as derived in Section 6.2, the performance of three parallel jobs on 32 workstations is near to that of ideal explicit coscheduling. Even when applications are fine-grain (*e.g.*, synchronizing with barriers every $100\mu s$), and context-switch times are high relative to network latency, (*i.e.*, $W = 200\mu s$ and $L = 10\mu s$), implicit coscheduling performs within 10% of explicit coscheduling. When context-switch costs are lower (*i.e.*, $W = 50\mu s$), more closely matching the costs of current systems, slowdowns are within 2%.

On the other hand, workloads with frequent communication, yet few barriers, are difficult to schedule effectively. These workloads require coordinated scheduling, yet contain few barriers that force all of the processes to be scheduled simultaneously. For these workloads, a process must remain scheduled even when only partial coordination exists.

8.3 Conditional Spin

In this section we show that conditional spinning within the two-phase waiting algorithm improves the performance of continuously-communicating applications to an acceptable level relative to explicit coscheduling. With conditional spinning, a waiting process remains scheduled after unsuccessfully spinning the baseline amount, as long as the incoming message arrival rate justifies the cost of holding onto the processor.

8.3.1 Bulk-Synchronous Workloads

Since performance with only baseline spinning is nearly ideal for bulk-synchronous workloads, performance is not noticeably affected by conditional spin. In these workloads, communication is performed only in short, intense phases immediately following a barrier. Since the barrier tends to force all participating processes to be scheduled simultaneously, when a process performs communication, the destination is usually already scheduled and returns the response in the baseline spin amount. For this reason, conditional spinning is

rarely activated and does not impact performance; therefore, we do not show the performance of bulk-synchronous workloads with conditional spinning.

8.3.2 Continuous-Communication Workloads

In communication-intensive workloads where barriers are performed infrequently, it is often the case that only a subset of the cooperating processes are scheduled at a given time. Under such conditions, processes should maintain partial coordination as it develops. Thus, a process should stay scheduled when it is receiving messages, even if it is unable to make forward progress itself.

Figure 8.8 shows the slowdown of the continuous-communication workload with conditional spinning relative to explicit coscheduling. In these experiments, processes always wait the baseline amount at reads and barriers, as determined by S^R_{Base} and S^B_{Base}, respectively. The minimum message arrival rate required to conditionally increase spin time is increased along the x axis.

The graphs in Figure 8.8 show that conditional spinning dramatically improves performance relative to that achieved with the best baseline spin time shown in Figure 8.7. With the conditional spin time derived in Section 6.3 (and marked as a vertical line in the graphs), performance of all workloads and context-switch costs is within 35% of ideal explicit coscheduling (significantly better than the factor of 4.5 slowdown with the best baseline spin time). To explain the behavior of conditional spinning, we discuss the three performance regions exhibited as the conditional spin amount is increased in each graph.

In the first region, the conditional spin time is very small and messages do not arrive at a fast enough rate to conditionally increase spin time. Therefore, processes usually spin only the baseline amount. Subsequently, performance when the conditional spin amount is $10\mu s$ matches the performance in Figure 8.7 when the baseline spin is S^R_{Base}=$2L + W$.

In our measurements, a second regime denoted by a dramatic improvement in performance occurs when conditional spin time is greater than or equal to S^R_{Base}=$2L + W$. For example, the slowdown of the `Random` pattern with $c = 10\mu s$ on a system with $W = 200\mu s$ decreases from 4 times worse than coscheduling to only 2 times worse.[5] As the conditional spin time increases, messages can arrive less frequently for the process to conditionally spin longer, improving performance for applications with frequent communication. For each application, there is then a region between approximately $W + 2L$ and $6W$ where performance is relatively flat.

Close-ups for these regions are presented in Figures 8.9 and 8.10. The derived conditional spin amount, S^R_{Cond}=$3W$ is marked in each graph. Across all workloads and context-switch costs, the performance with a conditional spin time of $3W$ is within 35% of

[5] Due to the implementation of conditional spinning in our simulator, this improvement occurs suddenly at the point when conditional spin time equals S^R_{Base}=$2L + W$, rather than improving gradually as conditional spin time is increased. For all points where the conditional spin time is less than S^R_{Base}, the application must receive at least two messages within the fixed interval where the application spins the baseline amount. When the conditional spin time is greater than or equal to S^R_{Base}, only one message must arrive in the first interval for the process to conditionally spin longer. The ideal implementation which records the arrival time of messages is expected to improve more gradually.

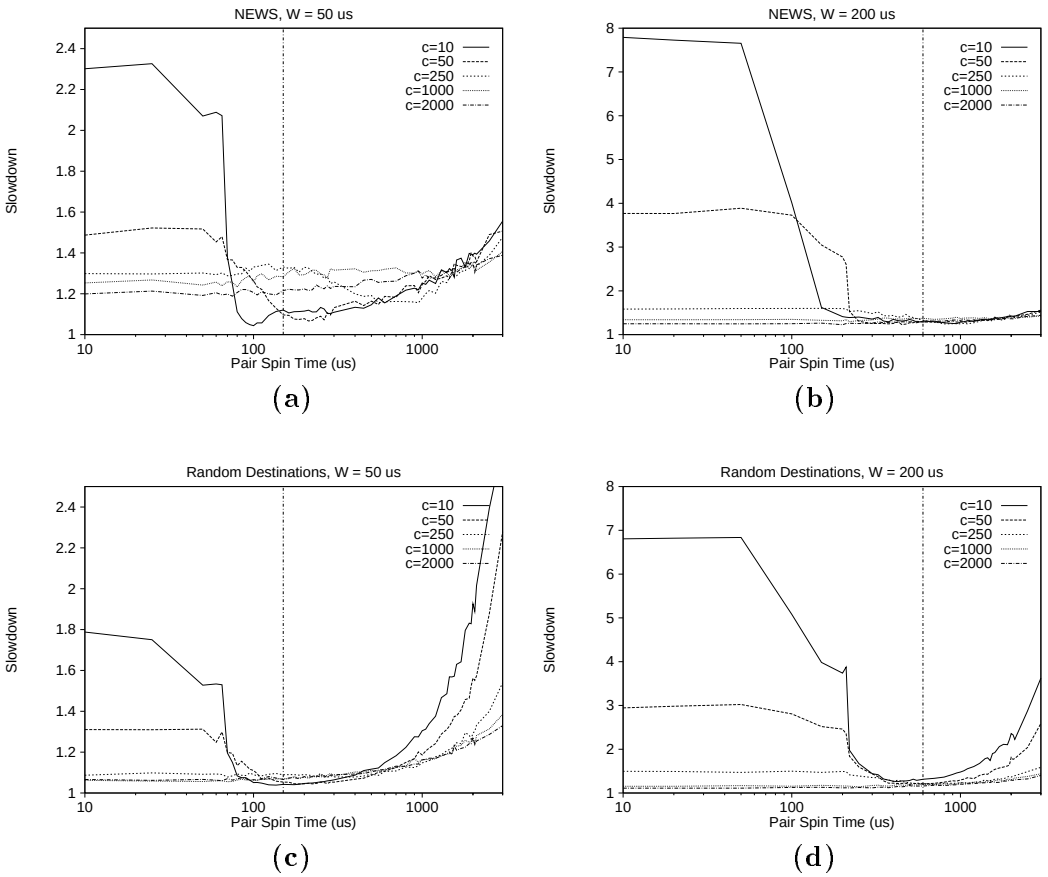


Figure 8.8: **Sensitivity to Conditional Spin for Continuous-Communication Programs.** *The workload consists of three continuously communicating jobs on 32 workstations. The time between barriers is set to $g = 100ms$ with no load-imbalance ($V = 0$). Each line in the graphs designates a different communication granularity, c. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting. Along the x-axis, the pairwise spin time for both reads and barriers are changed simultaneously. The vertical line marks $3W$, our chosen value of S^R_{Cond}. In Figures a and c, the context-switch cost is set to $W = 50\mu s$; in Figures b and d, $W = 50\mu s$; note the change in scale along the y-axis between the two sets. In the top two graphs, processes communicate in a repeated* `NEWS` *pattern; in the bottom two graphs, processes communicate with random destinations.*

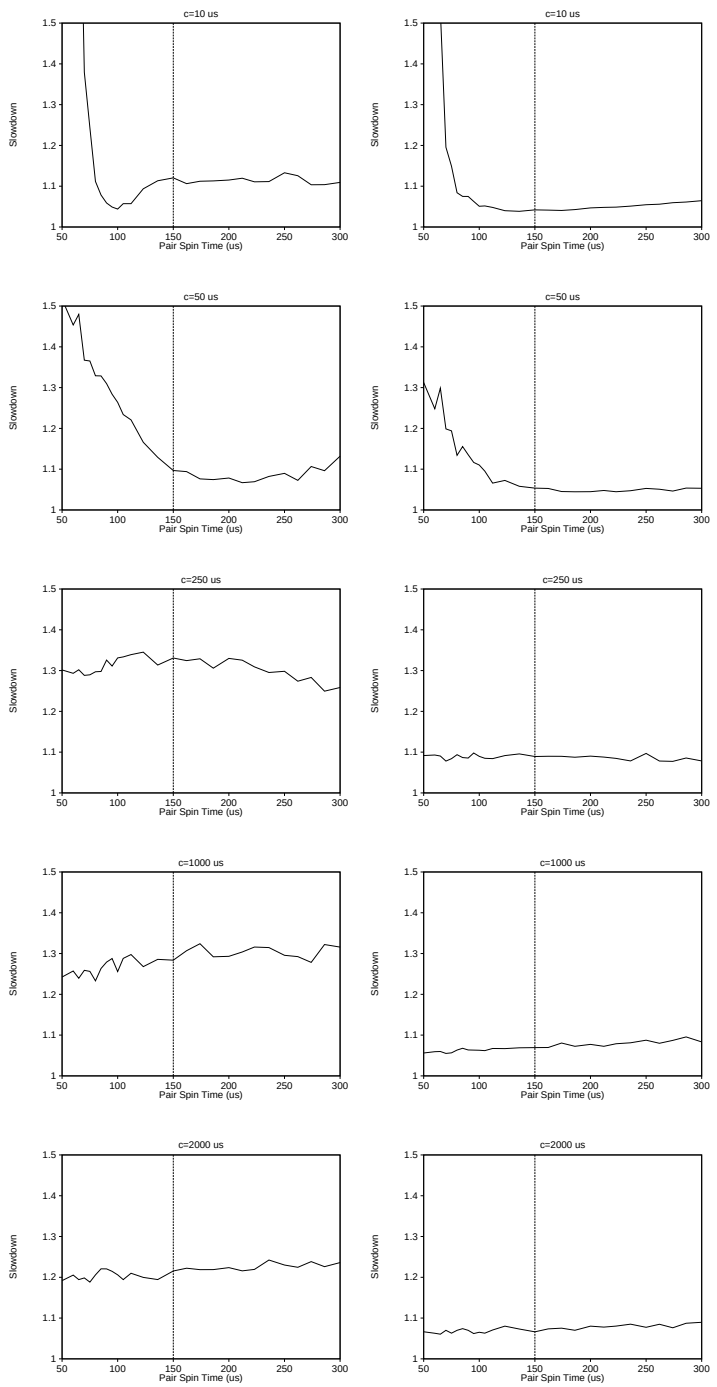


Figure 8.9: **Closeups of Sensitivity to Conditional Spin Amount ($W = 50\mu s$).**

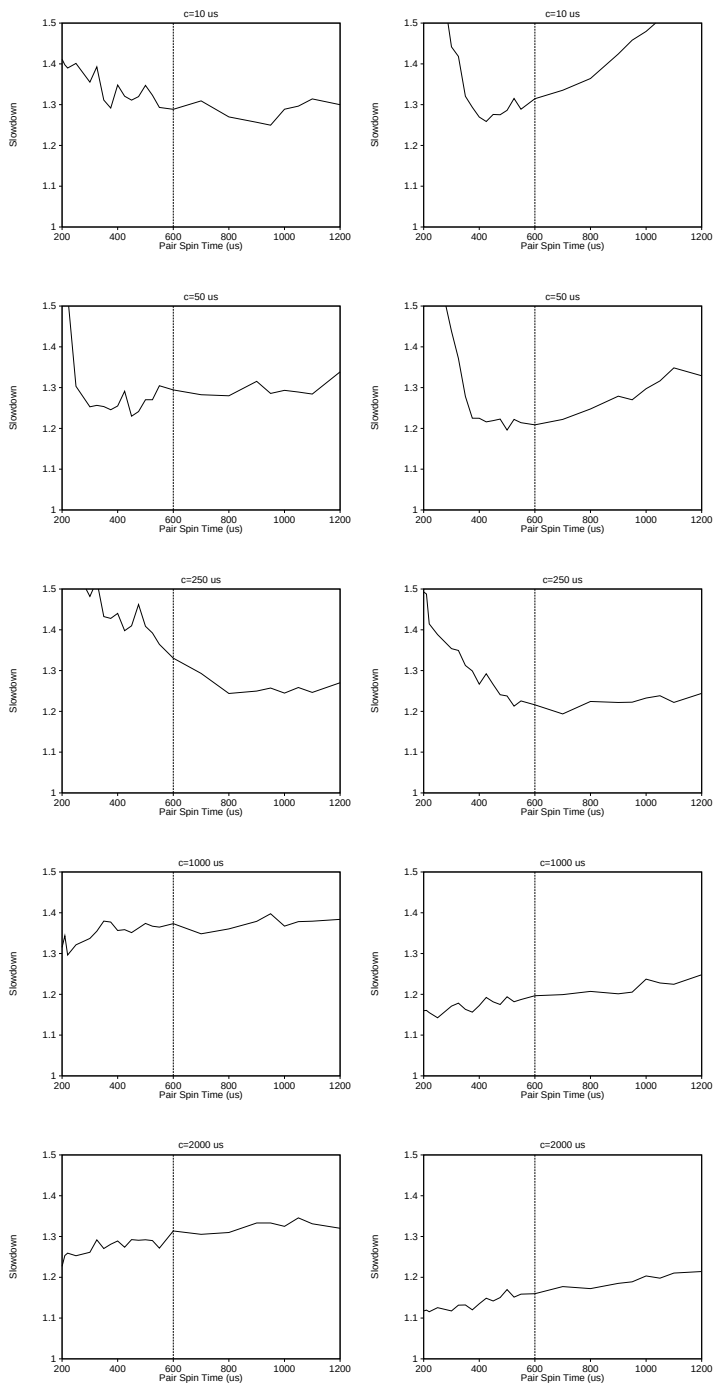


Figure 8.10: **Closeups of Sensitivity to Conditional Spin Amount** ($W = 200\mu s$).

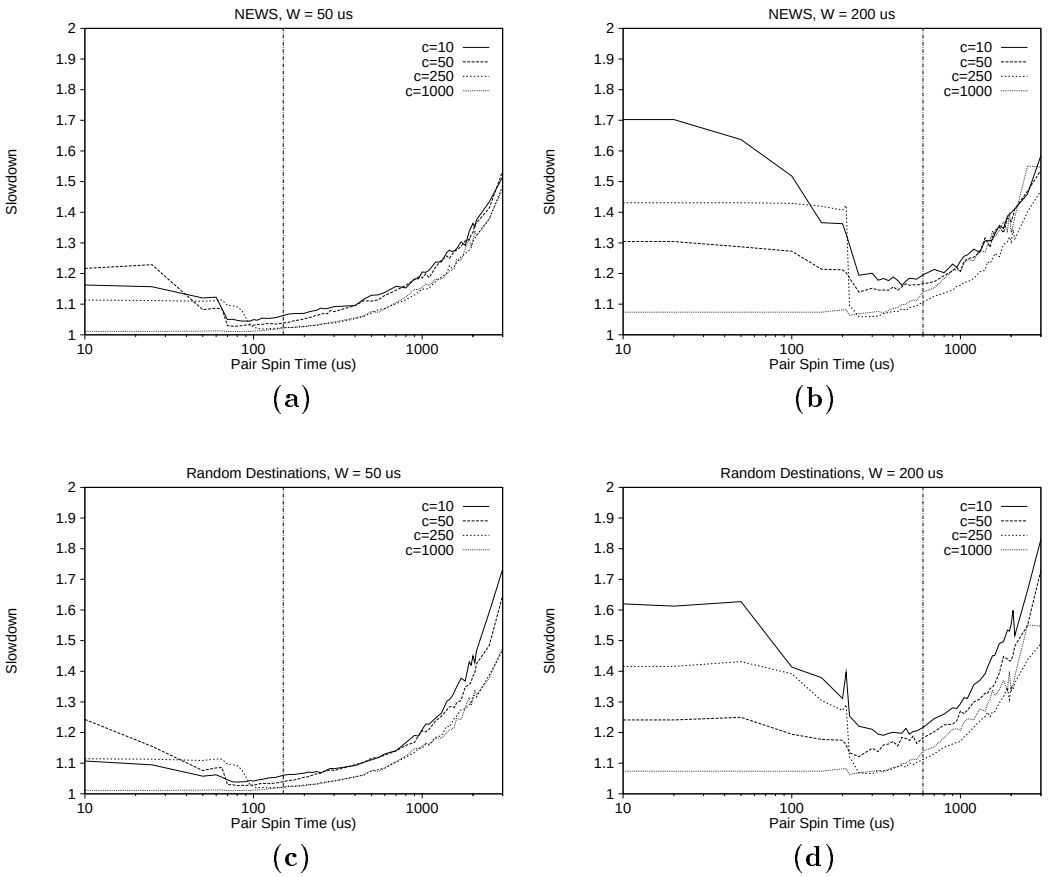


Figure 8.11: **Sensitivity to Conditional Spin for Continuous-Communication Programs with Frequent Barriers.** The experimental setup is identical to that in the preceding figures, except the time between barriers is set to $g = 1ms$ instead of $g = 100ms$. Again, the vertical line marks $3W$, our chosen value of S^R_{Cond}. Note that the point $c = 2ms$ is not shown because it is not possible for $c > g$ in this workload.

ideal explicit coscheduling. As one can easily see, $3W$ usually has comparable performance to the best conditional spin; in no case is the best performance more than 10% better than the performance with our derived conditional spin time. In some cases, the optimal spin amount is lower than $3W$, (*e.g.*, for with random destinations, when $c = 10\mu s$ and $W = 200\mu s$, the optimal point occurs when conditional spin time is $200\mu s$); in other case, the optimal conditional spin amount is higher than $3W$ (*e.g.*, with the `NEWS` pattern, when $c = 250\mu s$ and $W = 200\mu s$, the optimal point occurs with a conditional spin of $400\mu s$).

Returning to Figure 8.8, in the final region, conditional spin time is increased well past its optimal point; thus, processes continue spinning even when receiving messages relatively infrequently. In this region, the cost of spinning longer exceeds the benefit of maintaining partial coordination. Note that performance may degrade more noticeably for a given amount of conditional spin than baseline spin, because multiple consecutive conditional spins may be performed for a given communication operation.

Our final workload for this section, shown in Figure 8.11, examines continuous-communication applications where the time between barriers is reduced from $g = 100ms$ to $g = 1ms$. As expected, with more frequent barriers, the performance of implicit coscheduling is less sensitive to the amount of conditional spin. However, conditional spinning remains beneficial for fine-grain applications. For example, on a machine with $W = 200\mu s$, a conditional spin amount equal to S^R_{Cond}=$3W$ improves the performance of `NEWS` with $c = 10\mu s$ from 70% slower than explicit coscheduling to only 20% slower.

8.3.3 Discussion

The conclusion from this section is that all processes should spin at least the baseline amount when waiting for an event, and conditionally spin longer when receiving messages. Through this set of simulations, we have shown that this simple modification to the two-phase waiting algorithm significantly helps fine-grain applications that contain infrequent barriers. For example, with the baseline spin amount derived in Section 6.2, some fine-grain continuous-communication applications exhibit performance 8 times slower than ideal explicit coscheduling; with conditional spinning, performance is always within 35%.

In workloads where barriers are performed infrequently, it is often the case that only a subset of the cooperating processes are scheduled at a given time. Conditional spinning is beneficial in these circumstances for two reasons. First, conditional spinning helps processes maintain partial coordination as it develops. Second, conditional spinning dynamically adjusts to the increased penalty of blocking when receiving messages: it helps both the sending and receiving processes avoid an additional context-switch.

8.4 Load-Imbalance

To illustrate baseline and conditional spinning, the previous experiments assumed that the processes in each application were perfectly load balanced. As a result, when the scheduling was coordinated, processes could spin for S^B_{Base} at a barrier and complete the barrier successfully. In this section, we broaden our investigation to consider the impact of

load-imbalance on bulk-synchronous and continuous-communication applications.

We show two main results. First, when workloads contain large amounts of load-imbalance, implicit coscheduling can achieve better performance than explicit coscheduling with spin-waiting. Second, to obtain the best performance, processes must estimate the amount of load-imbalance across processes for each barrier.

8.4.1 Bulk-Synchronous Workloads

In this section we analyze the performance of bulk-synchronous applications with load-imbalance. Because our bulk-synchronous applications do not receive messages in the phase with load-imbalance, we can use baseline spin without conditional spinning. We begin by examining the sensitivity of the applications to the amount of baseline spin when we fix the time between barriers, g, and vary the load-imbalance, V.

To determine an interesting range for V, we note that our analysis in Section 6.4.2 determined that processes should block rather than spin when the load-imbalance exceeds a value between $3W$ and $(3+P)W$, depending upon the communication pattern after the barrier. To cover a wide portion of this space for context-switch costs of 50 and $200\mu s$, we focus on applications that have load-imbalances of $2ms$ or less. For an application to have $V = 2ms$, the time between barriers must be at least $1ms$; therefore, to measure programs that are the most sensitive to the scheduling effects, we set $g = 1ms$. Finally, since the cost of losing coordination depends upon the amount of communication, we examine workloads with the `Barrier`, `NEWS`, and `Transpose` patterns.

Figures 8.12 and 8.13 show the slowdown of implicit coscheduling as a function of baseline spin for context-switch costs of $W = 200\mu s$ and $W = 50\mu s$, respectively. Within each graph, the baseline spin time is increased from W to $3ms$. The graphs from left to right change the communication pattern, increasing the amount of communication: for the `Barrier` pattern, there are no reads; with `NEWS`, there are four reads; with `Transpose`, there are $P = 32$ reads. The graphs from top to bottom increase the load-imbalance in the program from $V = 0$ to $V = 2ms$.

On each graph, a vertical line indicates two critical spin times: the minimum baseline spin time, S^B_{Base}=$T^B_{Trigger}$, and the baseline spin time accounting for load-imbalance, $T^B_{Trigger} + V$. For each context-switch cost, communication pattern, and amount of load-imbalance, both of these spin amounts correspond to a local minima. Before discussing which baseline spin amount leads to the best performance, we describe the three performance regions in each graph.

In the first performance region, the baseline spin time is less than $T^B_{Trigger}$. In this region, processes are independently scheduled because few barriers complete successfully. For those applications with little load-imbalance or for those communication pattern with a second barrier (*i.e.*, `NEWS` and `Transpose` which have no load-imbalance at the second barrier), performance suffers because scheduling coordination is lost at the barrier and little waiting time is hidden. However, in those workloads where load-imbalance is high, implicit coscheduling can improve performance relative to explicit coscheduling (*e.g.*, the Barrier pattern with $W = 50\mu s$ and $V \geq 1ms$), as exhibited by the slowdown less than 1.

In the second region, the baseline spin amount is greater than or equal to $T^B_{Trigger}$,

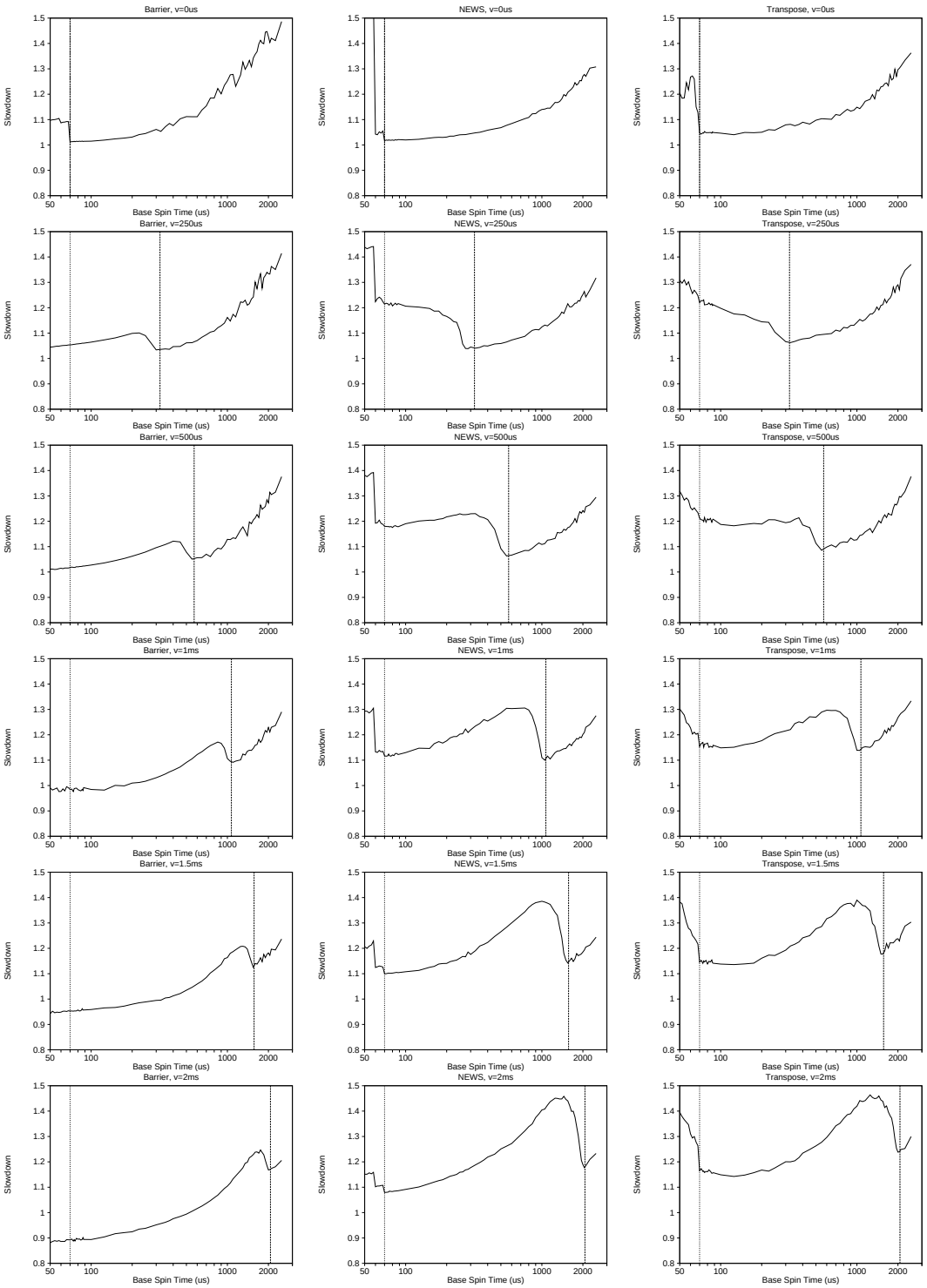


Figure 8.12: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance ($W = 50\mu s$).**

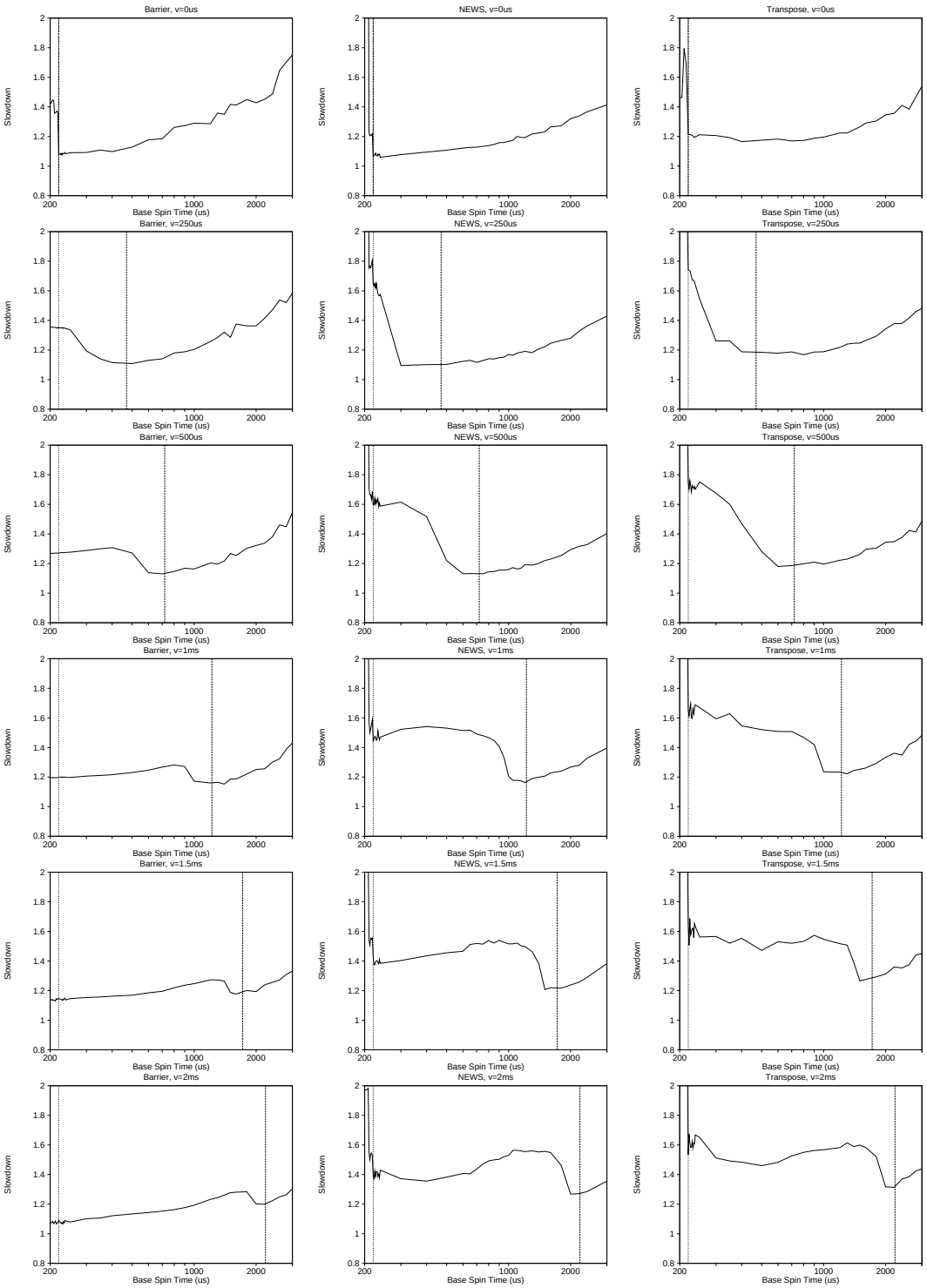


Figure 8.13: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance** ($W = 200\mu s$).

but less than $T^B_{Trigger} + V$. When processes increase their baseline spin to $T^B_{Trigger}$, performance improves dramatically because barriers with no load-imbalance now complete successfully. As processes increase their baseline spin time up to $T^B_{Trigger} + V$, performance slowly degrades. Spinning in this regime results in very few additional barriers completing successfully, and a large spin penalty is paid on every unsuccessful barrier.

In the third region, the process spins for $T^B_{Trigger} + V$ or longer. Once the process spins through the load-imbalance of the application, performance again improves as the majority of barriers complete successfully. The processes are now usually scheduled in a coordinated fashion. However, after spinning longer than $T^B_{Trigger} + V$, performance degrades, once more because the percentage of successful barriers does not increase and a larger spin penalty is paid for unsuccessful barriers.

Whether the best performance occurs when spinning $T^B_{Trigger}$ or $T^B_{Trigger} + V$ depends upon the ratio of V to W and the amount of communication in the application. For example, with the `Barrier` pattern, spinning for $T^B_{Trigger} + V$ is superior to spinning only $T^B_{Trigger}$ when $V \leq 5W$. However, as the communication pattern is changed and more communication is performed after the barrier, spinning for $T^B_{Trigger} + V$ continues to be beneficial for larger amounts of load-imbalance. For example, with the `NEWS` and `Transpose` patterns, spinning $T^B_{Trigger} + V$ remains superior as long as $V \leq 20W$. These simulation results are consistent with the analytical analysis presented in Section 6.4.2.

8.4.2 Continuous-Communication Workloads

In our next set of experiments, we vary the amount of baseline spin time when conditional spinning is employed and set to its optimal value, S_{Cond}. In these experiments we examine continuous-communication applications, setting the time between barriers to $g = 1ms$ and varying load-imbalance between $V = 0$ and $V = 2ms$. Rather than investigate multiple communication patterns, we instead examine only `Random` destinations and vary the communication interval, c.

Figures 8.14 and 8.15 show the slowdown for this workload as a function of baseline spin, for context-switch costs of 50 and $200\mu s$, respectively. Within each graph, the baseline spin time is increased from S_{Cond} to $3ms$. The amount of communication is increased across graphs from the left to the right; we examine $c = 250, 50$, and $10\mu s$. The graphs from top to bottom increase the load-imbalance in the program from $V = 0$ to $V = 2ms$.

The two points that are local minima on the graphs again correspond to possible baseline spin amounts. The first point is the left-most data point on the graph, which designates the minimum amount of baseline spin; due to our implementation of conditional spinning in the simulator the minimum baseline spin is S_{Cond}. The second point is $T^B_{Trigger} + V$, indicated by a vertical line. As with the bulk-synchronous workloads, whether the minimum baseline spin or $T^B_{Trigger} + V$ achieves the best performance depends upon the relative values of V and W and the amount of communication in the application.

For the most fine-grain applications (*i.e.*, $c = 10$ and $50\mu s$), spinning for the load-imbalance of the barrier is preferable. Due to the fact that messages are continuously arriving, a process achieves no benefit for relinquishing the processor when it reaches a barrier early. For example, consider applications with a load-imbalance of $2ms$ communi-

cating every $10\mu s$ on a system with a context-switch cost of $50\mu s$; this workload exhibits a slowdown of 35% if processes use the minimal baseline spin, but a slowdown of 16% if processes use a baseline spin of $T^B_{Trigger} + V$.

In general, to minimize worst-case performance, processes should spin for the load-imbalance of the barrier, up through $(3 + P)W$. For a system with $W = 50\mu s$, the worst-case slowdown that occurs when processes spin for $T^B_{Trigger} + V$ is 16%; for a system with $W = 200\mu s$, this increases to 26%. However, the performance of the workloads is still acceptable with the minimum baseline spin: for $W = 50\mu s$, the worst-case slowdown is 35% and for $W = 200\mu s$ it is 45%.[6]

8.4.3 Approximating Load-Imbalance

In our final set of experiments in this section, we show the performance of a more complete set of bulk-synchronous applications when load-imbalance is estimated at run-time. To predict the load-imbalance of a barrier, we use the algorithm described in Section 6.4.2. With this approach, the process at the root of the barrier records the time each participating process arrives at the barrier; arrivals that are late due to uncoordinated scheduling are removed from the sample. The root process then predicts that the load-imbalance of the next barrier will be equal to the largest remaining sample. This simple algorithm is sufficient for our synthetic applications because the maximum load-imbalance across processes varies little across the lifetime of the job.

The difficult part of the algorithm is in determining the amount of load-imbalance after which it is beneficial to spin-wait only the minimum amount rather than the full load-imbalance. In our prototype, the root process decides that processes should spin only the minimum amount when the predicted value of V exceeds $10W$. Clearly, a more sophisticated approach would determine this crossover value based on the amount of communication in the application, and thus the penalty for losing coordination.

Figure 8.16 shows the performance of this simple approach for the three bulk-synchronous communication patterns with a context-switch cost of $200\mu s$. As expected, all communication patterns with coarse-grained computation and high load-imbalance achieve better performance with implicit scheduling than with explicit coscheduling. At these points, processes learn to spin for only a short time and then relinquish the processor so that another process can be scheduled. The largest slowdown occurs when the load-imbalance in the program is near $g = 1ms$ and $V = 2g = 10W$, representing one of the workloads we examined in great detail in this section. Our final results indicate that we are able to achieve performance within 35% of coscheduling for all bulk-synchronous workloads containing three competing applications.

[6] We believe that the performance of applications with a large amount of load-imbalance will improve significantly if processes spin only the minimum amount, S^B_{Base}, instead of $\max(S^B_{Base}, S_{Cond})$ when waiting. This small modification will especially improve the performance of fine-grain applications since processes spin the extra amount on every read when processes are not coordinated.

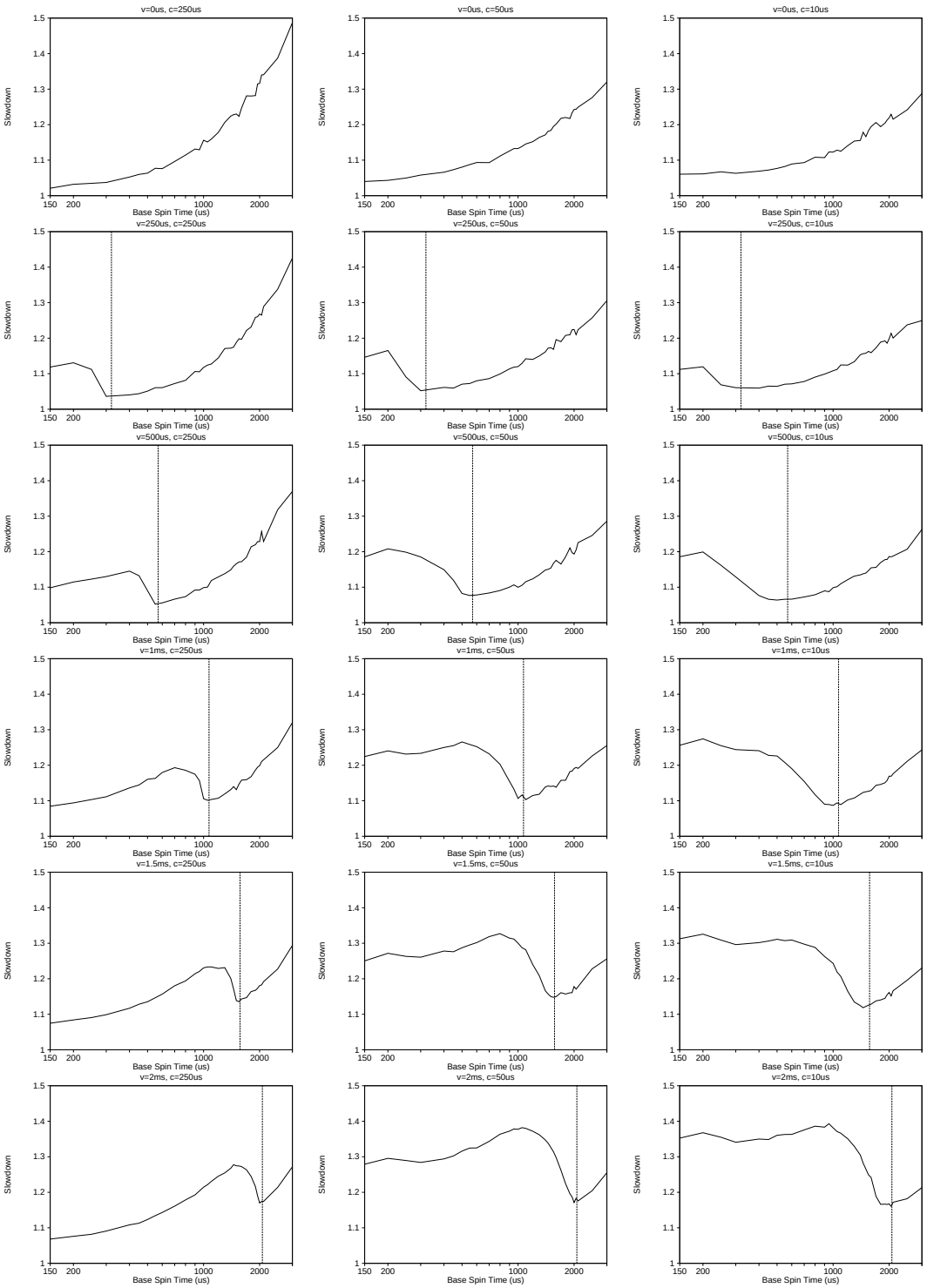


Figure 8.14: **Sensitivity to Baseline Spin for Continuous-Communication Programs with Load-Imbalance ($W = 50\mu s$).**

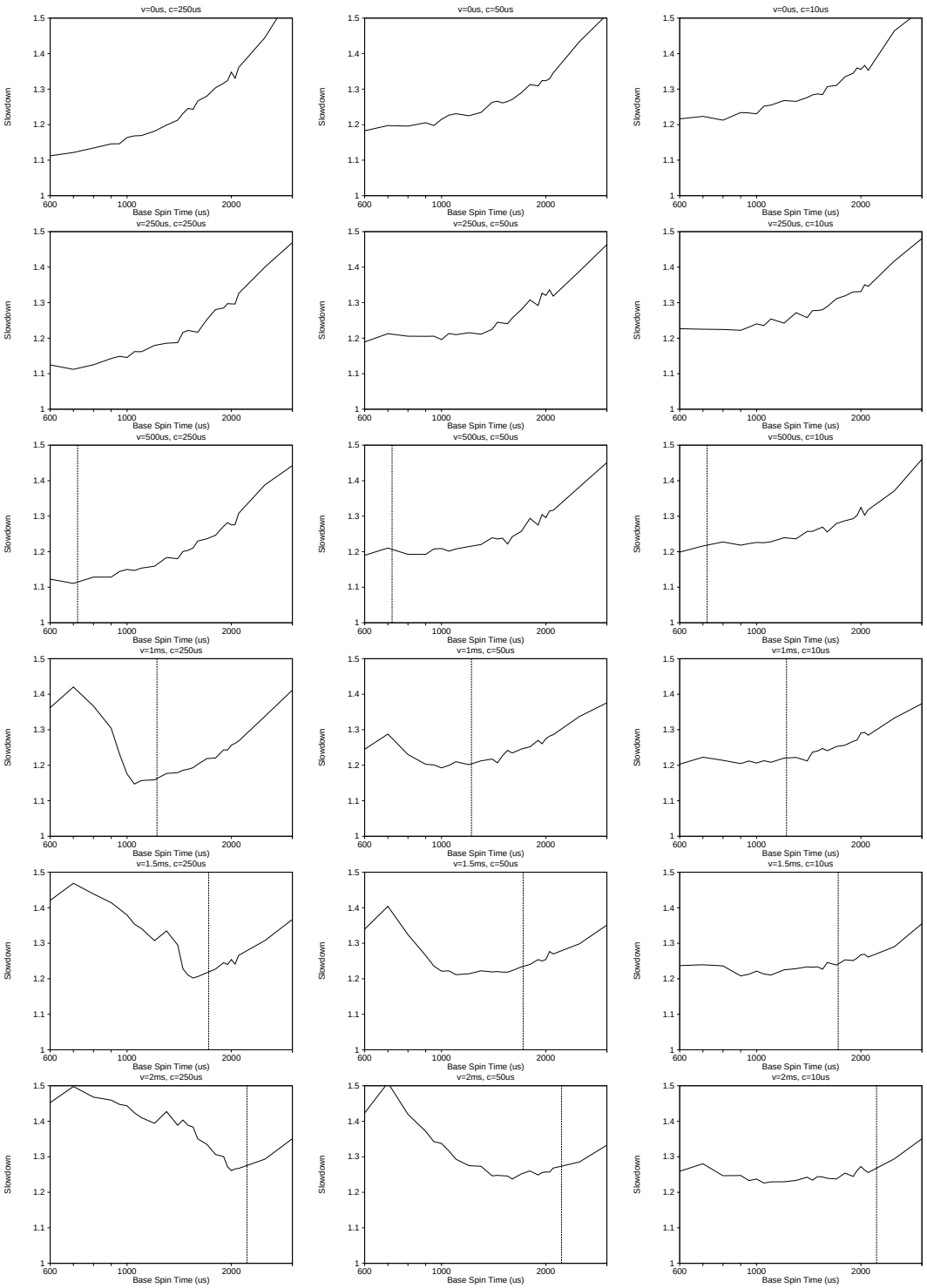


Figure 8.15: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs with Load-Imbalance** ($W = 200\mu s$).

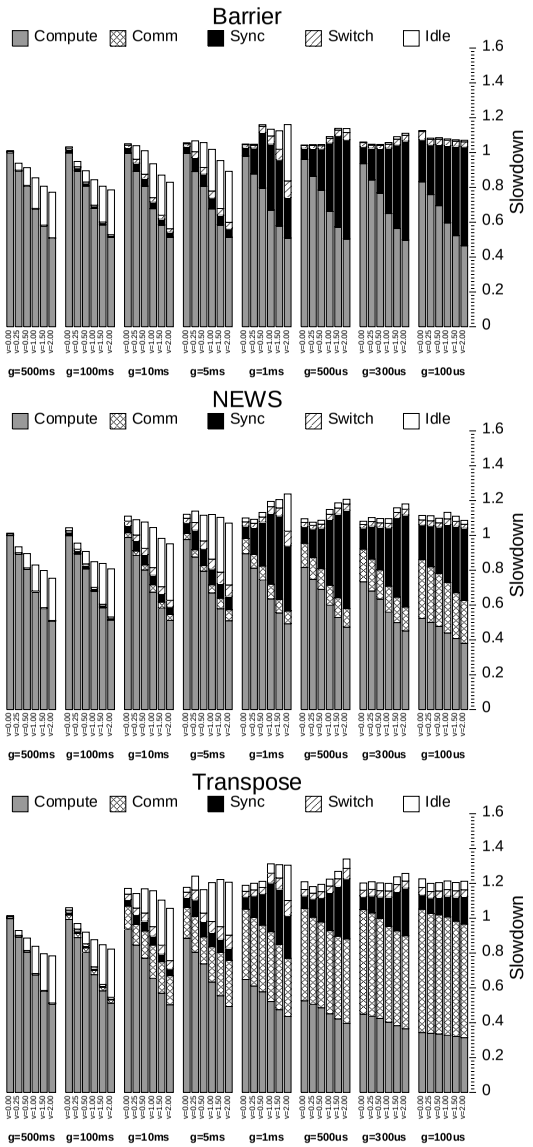


Figure 8.16: **Performance with Global Approximation of Load-Imbalance for Bulk-Synchronous Programs.** *The slowdown of implicit coscheduling with a run-time approximation of the amount of load-imbalance for each barrier relative to coscheduling with spin-waiting is shown. The root process of the barrier approximates load-imbalance by observing the arrival times at previous barriers and removing outliers due to scheduling irregularities.*

8.4.4 Discussion

In this section, we have seen that processes arriving at a barrier should spin-wait either only the minimum baseline amount, or should spin-wait for the minimum baseline amount plus the expected load-imbalance of the barrier. The spin time that leads to superior performance depends upon two factors.

- The amount of load-imbalance in the application relative to the cost of a context-switch.
- The amount of communication in the application.

Both of these factors determine the relative costs of losing coordination and of idly spin-waiting. For our workloads, we found that the best performance is achieved when processes spin for the expected load-imbalance, V, when V is less than 10 to $20W$. However, if the application is not able to approximate load-imbalance, performance is still acceptable if processes spin only the minimum baseline amount at all barriers, regardless of the load-imbalance.

The work in this section has shown that research in two related areas is still needed. First, more sophisticated algorithms for approximating load-imbalance at run-time would be helpful for more realistic applications. Second, to better determine whether a process should wait for a barrier with load-imbalance to complete, an approach is needed that predicts the cost of losing coordination, based on dynamic communication rates and patterns.

8.5 Local Scheduler

The previous experiments in this chapter assumed that the local operating system scheduler on each workstation was the default Solaris 2.4 Time-Sharing (TS) scheduler. This scheduler was sufficient for the previous workloads because each of the jobs had the same communication characteristics; therefore, the scheduler treated each of the processes similarly and there was no bias for or against any of the jobs. However, as described in Section 5.2, when the workload consists of jobs communicating at different rates, the Solaris TS scheduler is not fair.

In this section, we demonstrate that the Stride Scheduler with System Credit (SSC) is more fair for a wider range of workloads than the TS scheduler. We also investigate the impact of job placements on implicit coscheduling performance. Finally, we measure performance as the number of jobs in the system is increased from two to four.

8.5.1 Bulk-Synchronous Workloads

For diverse workloads, not only is a new local scheduler needed, but a new performance metric as well. The previous experiments measured the time required for a fixed workload of competing jobs to complete; however, this 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 in competition with continuously-running background jobs. In these workloads, we independently vary the communication granularity of both the examined job and the background jobs, producing a slowdown surface in our graphs.

The first workloads we examine contain a job in competition with either one, two, or three continuously-running background jobs. While all jobs in the workload communicate in the bulk-synchronous `NEWS` pattern and contain no load-imbalance, the granularity between barriers, g, is varied independently for the two sets of jobs and the resulting surface is plotted. Across the x-axis of each graph, the granularity of the evaluated job is increased from $50\mu s$ to $1s$, while across the y-axis the granularity of the background job(s) is increased. Thus, the points along the diagonal correspond to the workloads evaluated in the previous sections, where all jobs have identical communication characteristics. In all experiments, we examine only a system with a high context-switch cost of $W = 200\mu s$; therefore, the slowdowns we present are expected to be higher than those observed in practice.

The metric along the z-axis is the slowdown of the single measured job with implicit coscheduling relative to ideal explicit coscheduling. Contour lines along the base of each graph indicate the regions with a slowdown less than or equal to the amount indicated. 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 for a workload with a measured job A and a collection of competing jobs, if it exhibits near to the base slowdown with that workload; that is, slowdown is flat as the characteristics of the competing applications are changed.

In previous experiments, competing jobs were placed identically across all workstations; that is, process 0 of all the jobs is placed on workstation 0, and so on through process 31 and workstation 31. We begin our evaluation by placing jobs in this manner. However, our second set of experiments investigates performance when processes are randomly placed.

Identical Job Placement

With two-phase waiting, the natural tendency of implicit coscheduling is to bias against fine-grain jobs competing against more coarse-grain jobs, since fine-grain jobs relinquish the CPU more frequently. It is therefore the responsibility of the local schedulers to adjust for this unfairness.

Our first results, shown in Figure 8.17, examine one job in competition with two jobs for both local schedulers. Two observations are apparent from this figure. First, as expected with the Solaris Time-Sharing (TS) scheduler, for a fixed competing workload, slowdown increases as the interval between barriers in the measured job is decreased; slowdown decreases as the interval in the measured job is increased. Second, the Stride Scheduler with Credit (SSC) provides fair performance over a wider range of workloads than the TS scheduler.

We see that the fairness of the TS scheduler degrades slowly as the characteristics of the competing job are varied. For example, a bulk-synchronous job performing a barrier

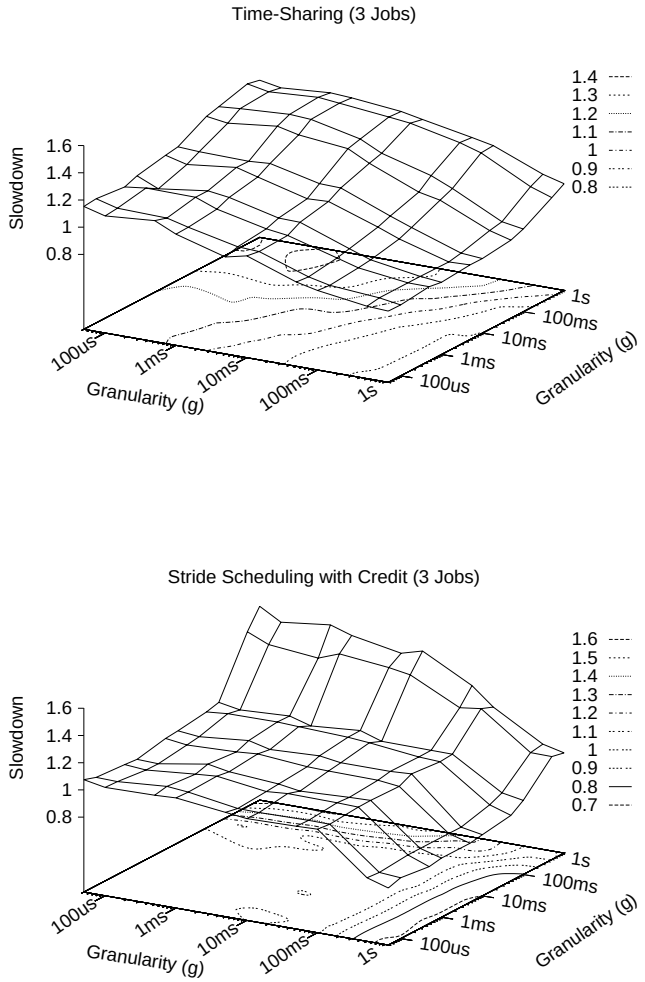


Figure 8.17: **Fairness for Bulk-Synchronous Programs with Identical Placement (3 Jobs).** *Three bulk-synchronous jobs communicating with the NEWS pattern and no load-imbalance are time-shared on 32 workstations. Along the x-axis, the time between barriers for the job under evaluation is varied between 50µs and 1s. Along the y-axis, the time between barriers is varied for the two continuously running competing job. The metric along the z-axis is the slowdown of the evaluated job implicitly coscheduled with the full two-phase waiting algorithm versus explicit coscheduling with spin-waiting. In the first graph, the Solaris Time-Sharing (TS) scheduler is used on each workstation; in the second, the Stride-Scheduler with System Credit (SSC) is used.*

every $g = 10ms$ exhibits a slowdown of 14% when competing against two other jobs that also synchronize every $g = 10ms$. However, when the measured job competes against background jobs that synchronize more frequently, the slowdown is less than 14%; in fact, in some cases, the measured job exhibits a speedup relative to its performance with explicit coscheduling (*e.g.*, when the $g = 10ms$ job competes against jobs with $g \leq 500\mu s$). Against jobs that synchronize less frequently, the observed slowdown is higher than 14% (*e.g.*, its slowdown increases to 34% against jobs synchronizing only once a second).

The SSC scheduler does a better job than the TS scheduler in fairly scheduling a diverse set of applications. The fairness of SSC falls into two distinct regions: slowdown is relatively flat as the communicating granularity of the competing jobs is varied across several orders of magnitude; however, after a certain point, scheduling becomes very biased toward coarse-grain jobs. For example, when the bulk-synchronous job with $g = 10ms$ competes against any two jobs with granularities between $g = 50\mu s$ and $g = 100ms$, the slowdown is between 1.0 and 1.09. However, when competing against very coarse-grain jobs (*e.g.*, when $g > 100ms$), slowdown increases rapidly: up to 1.6 times worse than with explicit coscheduling.

Closer inspection of the inflection point with SSC reveals that fairness severely degrades when the granularity of processes exceeds that of a time-slice, $Q = 100ms$. Against jobs whose synchronization interval, g, is larger than Q, competing fine-grain jobs do not acquire their fair share of resources even though they are given exhaustible tickets after relinquishing the processor. The problem appears to be that each competing coarse-grain job tends to run for g before the fine-grain job is able to coordinate itself and regain control of the cluster; once the fine-grain job is coordinated, it is scheduled for a time-slice. If the job has been allocated a sufficient number of exhaustible tickets, it should be scheduled for multiple time-slices; however, in our implementation, this does not occur. We believe that this performance problem is not fundamental to implicit coscheduling and could be fixed with more attention to the stride scheduler running on each workstation.

In Figure 8.18 we show the impact of increasing the number of competing jobs; in these graphs, the measured job competes against one, two, or three other jobs. As expected, the *base slowdown* increases gradually with more jobs. For example, with the TS scheduler and jobs synchronizing every $g = 50\mu s$, slowdown is about 1.1 with two jobs and 1.2 with four jobs. The SSC scheduler has slightly better performance as the number of jobs increases; for example, with SSC slowdown is only 1.05 with two jobs and 1.08 with four jobs.

Achieving coordination is more difficult with more jobs because the likelihood that each local scheduler picks a different runnable process increases. The difference in performance across the two local schedulers is primarily due to the lengths of the time-slices. With SSC, time-slices are always set to $Q = 100ms$ (although processes can be preempted by newly awakened jobs). With TS, time-slices range between $Q = 20ms$ at the highest priority down to $200ms$ at the lowest priorities. With shorter time-slices, a greater percentage of time is spent obtaining coordination. As described in Section 5.2.3, with the TS scheduler, the more jobs in the system, the more time each process spends executing at a higher priority with a shorter time-slice. By forcing jobs to run only at the lower priorities with $200ms$ time-slices, we were able to significantly improve the scalability of the

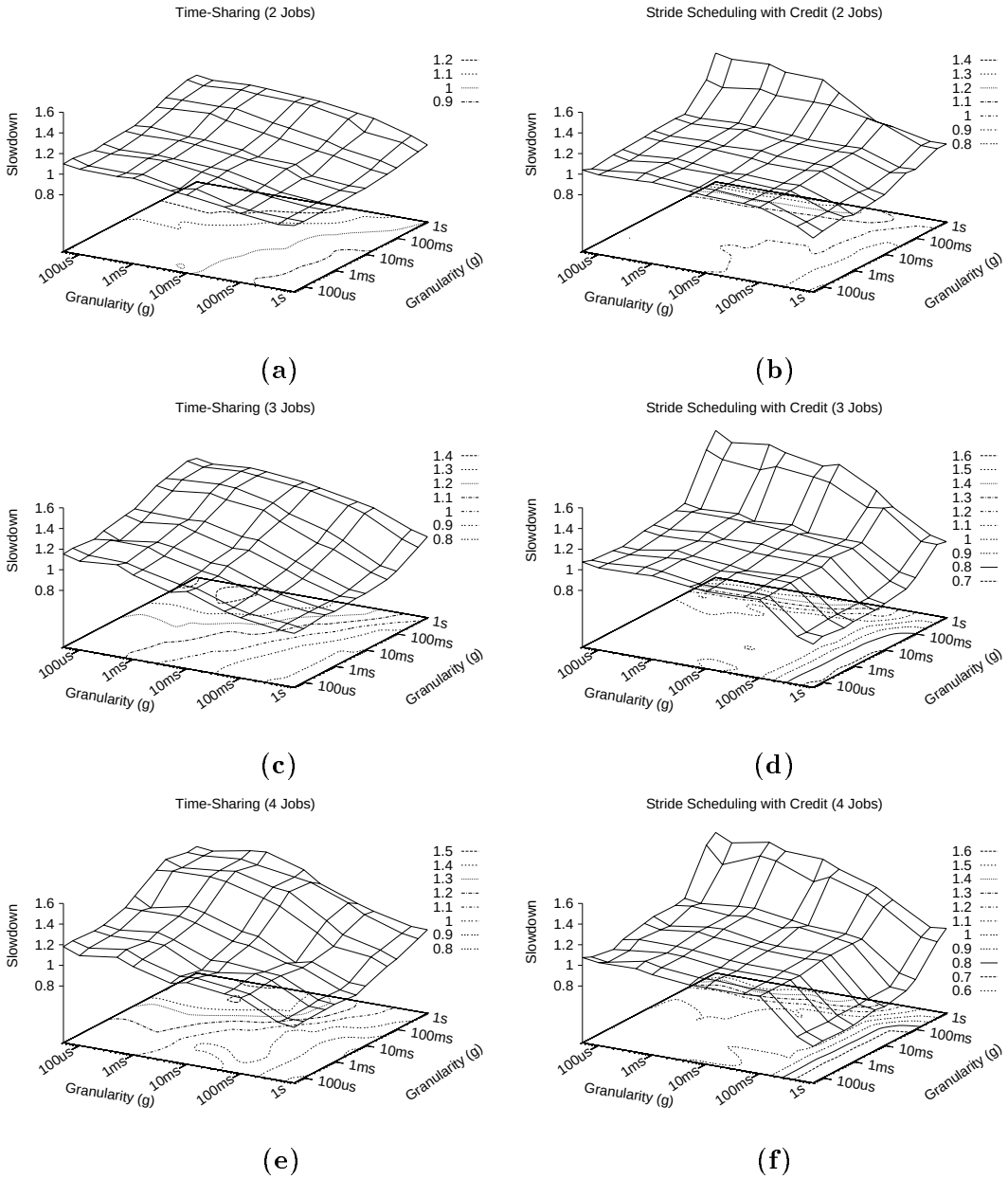


Figure 8.18: **Fairness for Bulk-Synchronous Programs with Identical Placement.** *Bulk-synchronous jobs communicating with the* `NEWS` *pattern and no load-imbalance are time-shared on 32 workstations. Along the x-axis, the time between barriers for the job under evaluation is varied between 50µs and 1s. Along the y-axis, the time between barriers is varied for a continuously running competing job. The metric along the z-axis is the slowdown of the evaluated job with implicit coscheduling versus explicit coscheduling. The number of jobs in the system is increased from two to four across graphs down the page. In the first graph of each pair, the Solaris TS scheduler is used on each workstation; in the second graph, the Stride-Scheduler with Credit is used.*

TS scheduler with more jobs.

Another trend that is evident from Figure 8.18 is that as the number of competing jobs is increased, fewer workloads are given fair scheduling. Clearly, it is more difficult for a fine-grain job to acquire its portion of the CPU when competing against additional coarse-grain jobs. However, SSC remains relatively fair as long as all jobs synchronize more frequently than a time-slice.

Random Job Placement

In our next experiments, we evaluate the impact of randomly placing the processes of each job across workstations. When processes are randomly placed, the process that forms the root of the barrier for each application is on a different workstation. As a result, there is no single workstation that acts an implicit master for the cluster.

Figure 8.19 shows the performance when processes are randomly allocated to workstations. The graphs show that random placement has very little impact on base slowdown or on fairness with the SSC scheduler. However, with the TS scheduler, random placement adversely affects base slowdown. For example, four competing jobs that synchronize every $g = 50\mu s$ experience a slowdown of more than 60% relative to explicit coscheduling with random placement; however, when the root processes were colocated on the same workstation, the jobs experienced a slowdown of only 18%. Once again, the majority of the effect is due to the shorter time-slices of the TS scheduler; performance can be significantly improved by increasing the length of all time-slices to $200ms$.

8.5.2 Continuous-Communication Workloads

In our next workloads, we consider applications that continuously communicate. We examine two distinct workloads: one in which processes synchronize with barriers every $g = 100\ ms$ and one in which processes synchronize every $g = 1s$. Each process is perfectly load-balanced and each communicates in a `Random` pattern every c time units. In these workloads, c is varied independently for the measured job (along the x-axis) and for the competing jobs (along the y-axis). In these experiments, the processes are always randomly placed across workstations.

We begin by evaluating the workload where the time between barriers is $g = 100ms$. Figure 8.20 shows the slowdown of the measured job for both the TS and SSC schedulers for two, three, and four jobs.

With the TS scheduler, as the number of competing jobs is increased, the base slowdown degrades. For example, when all processes communicate every $c = 50\mu s$, the base slowdown for two jobs is only 11% worse than explicit coscheduling; with four jobs, the base slowdown increases to 51%. Fairness degrades much more severely as the number of jobs is increased. Against one competing coarse-grain job, a job communicating every $c = 50\mu s$ exhibits slowdowns only 30% worse than explicit coscheduling; against three competing jobs, the slowdown is more than 3.5 times worse than coscheduling.

On the other hand, with the SSC scheduler, base performance and fairness remain near that of explicit coscheduling for a wide range of workloads. For example, with even four jobs in the workload, base slowdown is always within 15%. Further, because all processes

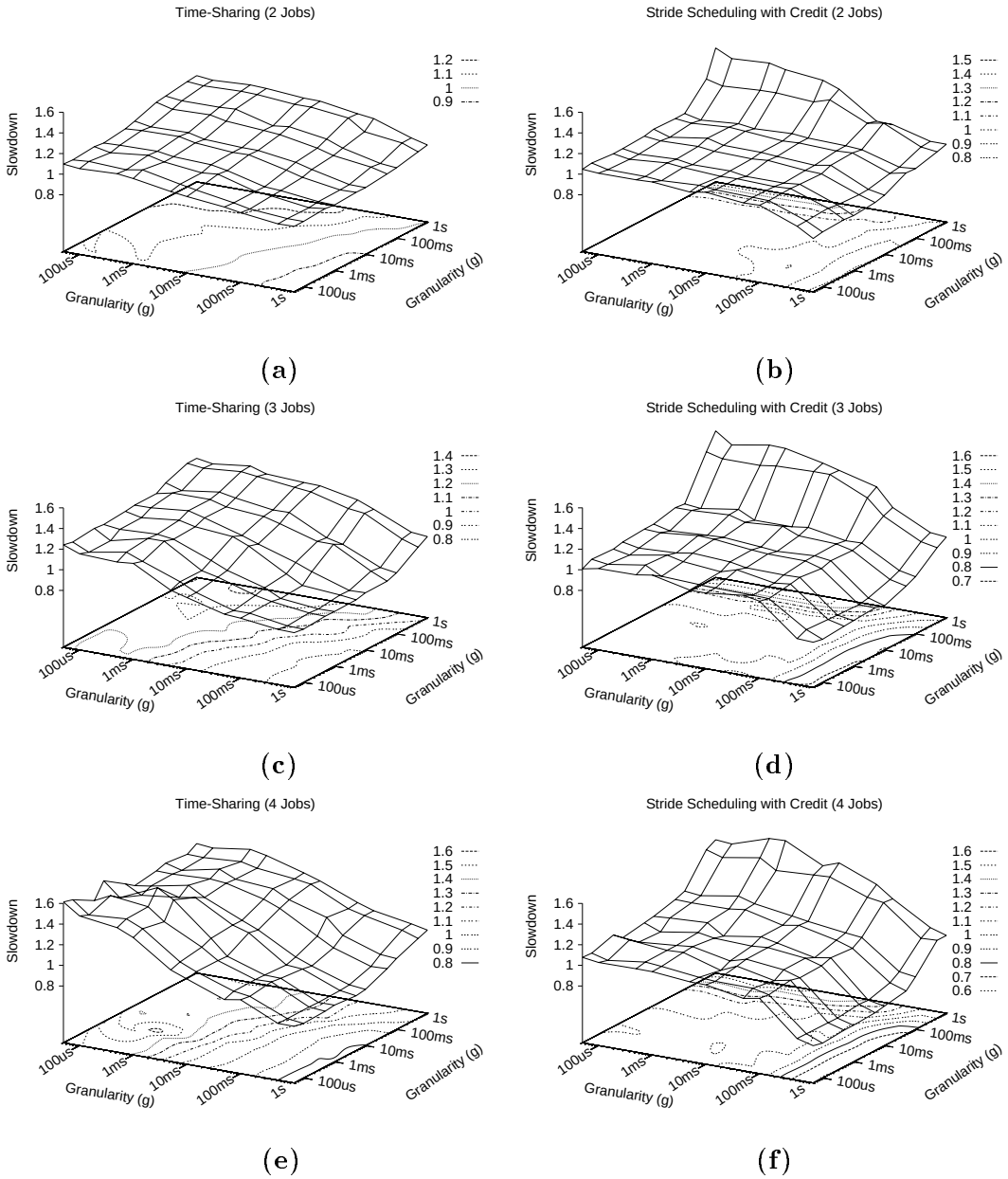


Figure 8.19: **Fairness for Bulk-Synchronous Programs with Random Placement.** *Bulk-synchronous jobs communicating with the NEWS pattern and no load-imbalance are time-shared on 32 workstations. Along the x-axis, the time between barriers for the job under evaluation is varied between 50µs and 1s. Along the y-axis, the time between barriers is varied for a continuously running competing job. The metric along the z-axis is the slowdown of the evaluated job with implicit coscheduling versus explicit coscheduling. The number of jobs in the system is increased from two to four across graphs down the page. In the first graph of each pair, the Solaris TS scheduler is used on each workstation; in the second graph, the Stride-Scheduler with Credit is used.*

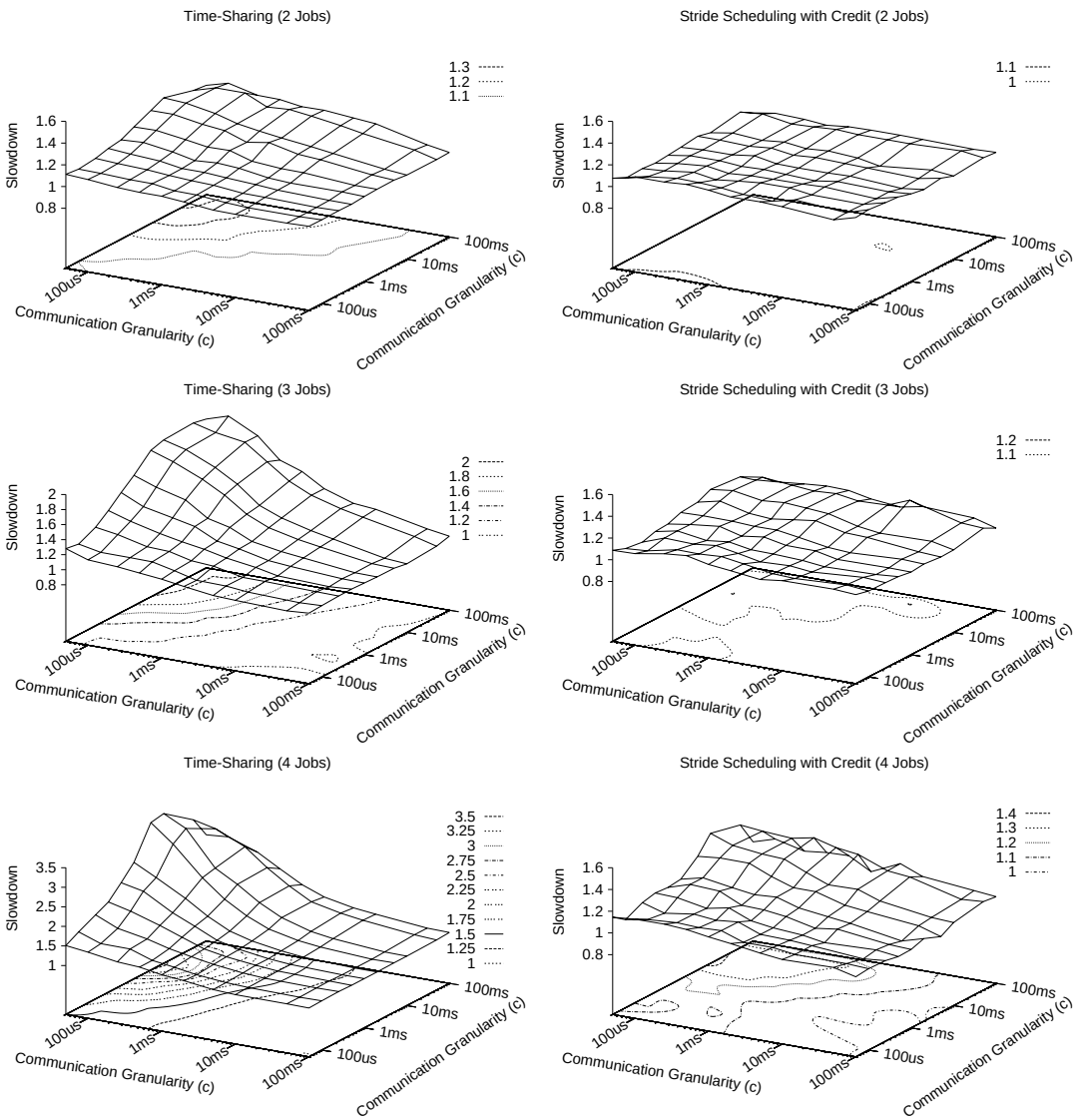


Figure 8.20: **Fairness for Continuous-Communication Programs ($g = 100ms$).** *Jobs continuously communicating with the NEWS pattern and no load-imbalance are time-shared on 32 workstations. The processes of each job are randomly placed across workstations. Along the x-axis, the time between reads for the job under evaluation is varied between 50µs and 100ms. Along the y-axis, the time between reads is varied for a continuously running competing job. The metric along the z-axis is the slowdown of the evaluated job with implicit coscheduling and the full two-phase waiting algorithm (ideal baseline and pairwise spin times) versus explicit coscheduling with spin-waiting. Note the change in scale as the number of jobs is increased with the Solaris Time-Sharing scheduler.*

synchronize with barriers every $g = 100ms$ regardless of their communication intensity, c, all programs in the workload must coordinate every time-slice, Q; therefore, the problem identified with our current implementation does not occur in this workload. As a result, for all communication granularities, the slowdown against one or two competing jobs is always within 20% of explicit coscheduling. Only when one relatively fine-grain job competes against three relatively coarse-grain jobs does slowdown approach 40%.

In our final set of experiments, we evaluate workloads that are particularly insidious for fair implicit coscheduling: applications that synchronize with barriers every $g = 1s$. Not only are workloads containing identical copies of such applications difficult to schedule due to the rarity of the coordinating barriers, but with the granularity greater than the time-slice, $Q = 100ms$, coarse-grain processes may receive more than their fair share of the resources, even with the SSC scheduler. Therefore, the results in Figure 8.21 illustrate the worst-case performance we are likely to see.

The TS scheduler exhibits both unacceptable base slowdown and scheduling biases for many workloads. For example, four fine-grain jobs with the same communication may be slowed down more than five times that of explicit coscheduling. Scheduling is also strongly biased by communication frequency; one job communicating every $c = 50\mu s$ that competes with coarse-grain jobs communicating every $c = 5ms$ is slowed down by more than seven times.

The SSC scheduler significantly improves both the base slowdown and fairness for this workload. For example, with even four jobs, the base slowdown is never greater than 20%. Fairness, although greatly improved relative to the TS scheduler, is still not acceptable. In a few workloads containing four jobs, a fine-grain job competing against more coarse-grain jobs experiences slowdowns more than three times worse than ideal. Once again, we believe that these biases can be removed by further the tuning the local stride schedulers running on each workstation.

8.5.3 Discussion

In this section, we have shown that the Solaris Time-Sharing (TS) scheduler is not adequate for fairly scheduling jobs that communicate at different rates. The Stride Scheduler with System Credit (SSC), which extends a stride scheduler [171] to be fair to jobs that voluntarily relinquish the processor, allocates resources more fairly for a wider range of workloads. While our current implementation remains biased towards jobs that synchronize less frequently than the time-slice of the scheduler, we believe that this problem is not fundamental to our approach.

This section also investigated two additional changes to the workloads. First, we measured the impact of job placement: identical processes are place on the same workstation or processes are placed randomly. When the process that acts as the root of barrier operations is placed on the same workstation as other root processes, that workstation acts as an implicit master of the cluster. When applications perform frequent barriers, then the process running on the implicit master directs the scheduling of all of the other workstations in the cluster. If root processes are not colocated, coordinated scheduling is harder to achieve. Second, we investigated the impact of adding more competing jobs to the

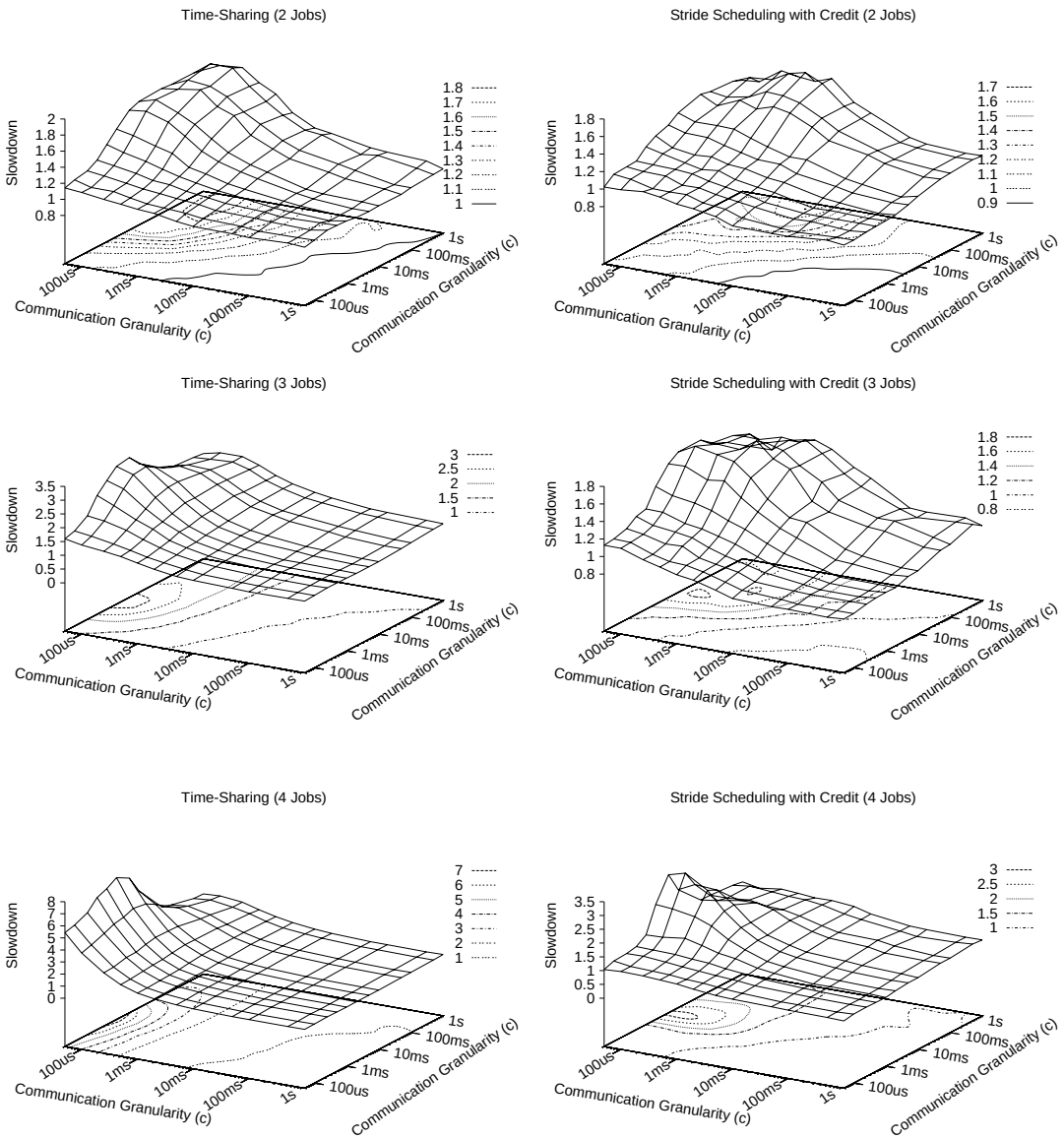


Figure 8.21: **Fairness for Continuous-Communication Programs** ($g = 1s$). *Jobs continuously communicating with the NEWS pattern and no load-imbalance are time-shared on 32 workstations. The processes of each job are randomly placed across workstations. Along the x-axis, the time between reads for the job under evaluation is varied between 50μs and 1s. Along the y-axis, the time between reads is varied for a continuously running competing job. The metric along the z-axis is the slowdown of the evaluated job with implicit coscheduling and the full two-phase waiting algorithm (ideal baseline and pairwise spin times) versus explicit coscheduling with spin-waiting. Note the change in scale as the number of jobs is increased with the Solaris Time-Sharing scheduler.*

workload. For both random job placement and additional jobs, the extra cost of achieving coordination can be amortized by using time-slices on the order of $100ms$. Due primarily to its longer time-slices, our SSC scheduler achieves better performance than the TS scheduler for these workloads.

8.6 Summary

In this chapter, we have verified through simulation that implicit coscheduling delivers fair and efficient performance for a wide variety of synthetic workloads. In our workloads, we have considered both bulk-synchronous applications and applications that continuously-communicate. We have analyzed performance sensitivity to three important application parameters: the rate of communication, the rate of synchronization, and the amount of load-imbalance internal to the application. We have also evaluated the impact of two system parameters: network latency, L, and the cost of a context-switch in the local operating system scheduler, W.

Implicit coscheduling can be configured to use either coordinated or uncoordinated scheduling, depending upon the relative values of L and W. Since coordinated scheduling achieves superior performance for fine-grain applications on current clusters, implicit coscheduling should be configured to use two-phase waiting at communication operations, rather than blocking immediately. There are two important components of two-phase waiting: baseline and conditional spinning.

Baseline spin within the two-phase waiting algorithm keeps processes coordinated. Our simulations have shown that baseline spin is sufficient to achieve respectable performance for bulk-synchronous applications and applications that synchronize frequently. Applications that rarely synchronize, yet communicate frequently, are difficult to schedule effectively. Such jobs require that processes remain scheduled when only partial coordination exists; this is achieved by using conditional spinning, where a process continues to spin-wait when receiving messages from other processes.

When workloads consist of jobs with similar communication characteristics, the Solaris Time-Sharing (TS) scheduler obtains acceptable performance. However, it does not fairly handle workloads where some jobs are more fine-grain than others and thus voluntarily relinquish the processor more frequently; in such workloads, fine-grain jobs do not receive their fair share of the resources. The stride scheduler with system credit (SSC) greatly improves the fairness of such workloads.

Chapter 9

Prototype Implementation

Although the simulation environment provided by `SIMplicity` is fairly detailed, it fails to capture all of the aspects of a complete system. Some of the simplifying assumptions of the simulator are necessary to reduce the time required to complete the simulation; for example, neither the overhead of sending messages or the cost of predicting load-imbalance is modeled. Other simplifying assumptions were made due to insufficient data from real systems; for example, the impact of daemon processes in the system is not modeled. Finally, in some cases, assumptions are simply not representative of the real world; for example, the simulations assumed that processes are notified of message arrivals with interrupts, whereas with our communication-layer, processes must explicitly poll the network.

To increase our confidence in implicit coscheduling, we have implemented a prototype version on the U.C. Berkeley Network of Workstations (NOW) cluster. In this chapter we describe that implementation, which only involves changes in the parallel language run-time layer and, optionally, a new scheduling module; no changes are required in the applications themselves. In our discussion, we focus on the requirements of the Active Message layer, the configuration of spin-time in the two-phase waiting algorithm within the Split-C run-time layer, the application workload, and the development of the operating system schedulers.

9.1 System Architecture

Our implementation of implicit coscheduling is performed on the U.C. Berkeley NOW cluster. Most of our experiments measure a cluster of 16 Ultra 1 Model 170 workstations, although some are performed on 32 workstations. Each machine contains a single 167 MHz UltraSPARC processor, a 512 KB off-chip second-level cache, and 128 MB of main memory. A diagram of each workstation is shown in Figure 9.1. Each workstation runs a copy of Solaris 2.6 [51], a modern, multi-threaded operating system based on SVR4 [67].

The workstations are connected with Myrinet, a switch-based, high-speed, local-area network, with links capable of bi-directional transfer rates of 160 MB/s [20]. Each machine has a single Myrinet card on the S-Bus, which is attached via cable to an eight-port switch; multiple switches can be linked together to form large, arbitrary topologies. The 105-node U.C. Berkeley NOW cluster is comprised of three 35-node clusters, each containing

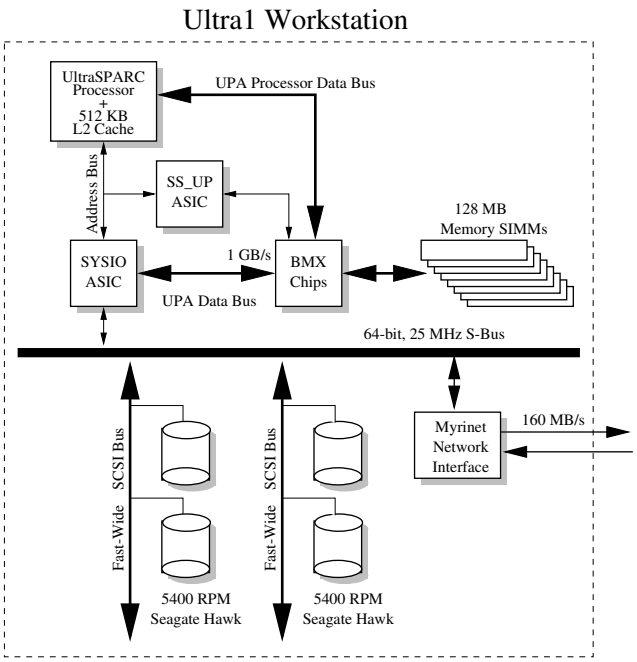


Figure 9.1: **Internals of Ultra 1 Workstation.** *The figure depicts the internal architecture of an Ultra1 workstation. The Network Interface Card is connected to the S-Bus of each workstation.*

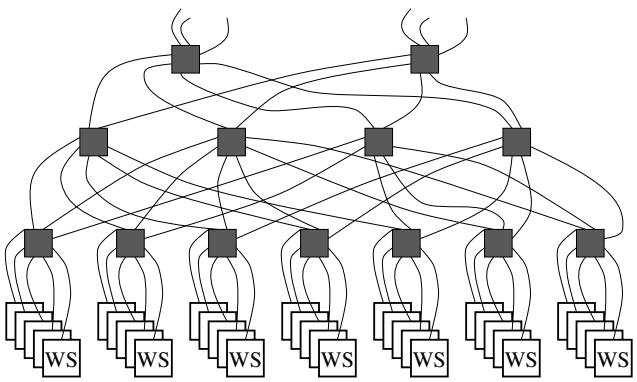


Figure 9.2: **Network Topology for Cluster of 32 Workstations.** *The figure shows 35 workstations connected with 13 eight-port Myrinet switches. Three such groups comprise the entire 105-node U.C. Berkeley NOW cluster.*

Variable	Description	Value
L	network latency	$9.8\mu s$
o	overhead	$3.6\mu s$
g	gap	$13.7\mu s$
P	number of processors	usually 16
W	wake-up from message arrival	$70\mu s$
Q	duration of time-slice	SSC : $100ms$ TS : from $20ms$ to $200ms$ (Table 5.1)

Table 9.1: **System Parameters in Implementation.** *The table shows the relevant network, machine, and operating system parameters in our system. L, o, and g are determined by the microbenchmark described in [38]. W is calculated from Figure 9.3.*

13 of these switches connected in a 3-ary, tree-like structure as shown in Figure 9.2.

9.2 Message Layer

Processes communicate in our system with AM-II [110], an extension of the Active Message paradigm [167]. The Active Message model is essentially a simplified remote procedure call that can be implemented efficiently on a wide range of hardware platforms. When a process sends an Active Message, it specifies a handler to be executed on the remote node. When the message is received, the handler executes atomically with respect to other message arrivals. AM-II extended previous versions of Active Messages on clusters [115] by handling multiple communicating processes, client-server applications, and system services. The LogP parameters of AM-II are shown in Table 9.1.

A fundamental difference from the simulations occurs in the manner in which processes are notified of a message arrival. In the simulations, processes received an asynchronous interrupt whenever a message arrived; the message was then immediately handled if the process was scheduled. In our implementation environment, a process is notified of a message arrival only when it touches the network: either by sending a message or by explicitly polling the network with `AM_Poll`. Thus, to a remote process waiting for a response, a process that is ignoring the network appears as if it were not scheduled. While this behavior does not match the assumptions of our model or of the simulations, we have not found it to adversely affect performance.

The parallel language run-time layer, described in the next section, has the responsibility of implementing the two-phase waiting algorithm for implicit coscheduling. To support the two-phase waiting algorithm, the underlying message layer must meet three requirements.

1. A waiting process must be able to voluntarily relinquish the processor until a message arrives.

2. To implement conditional spinning, a process must know if a message has arrived in a given interval.

3. The message layer must never spin-wait on a remote action; if the message layer reaches a remote condition for which it must wait before proceeding, control should be returned immediately to the parallel language run-time layer.

AM-II was designed to allow processes to relinquish the processor while waiting for a message to arrive. A waiting process simply calls `AM_SetAndWait` with the proper arguments; when a message arrives, the process is woken and eventually scheduled. When control is returned to the process, it explicitly polls the network to handle the message.

However, the initial version of AM-II did not meet the second two requirements of implicit coscheduling. First, AM-II did not initially contain a mechanism for processes to determine when a message arrived. Fortunately, the modification was straight-forward: AM-II now returns the number of messages handled in each call to `AM_Poll`, thus enabling conditional two-phase waiting.

Second, the initial version of AM-II contained circumstances under which the process spin-waits until a remote condition is satisfied: for example, when waiting for flow-control credits or queue space (because an outgoing message requires that the next slot in a fixed-length FIFO queue is free). For these circumstances, a recent version of AM-II provides a non-blocking interface that returns control to the calling process, while signifying that the message was not sent. However, this modification to AM-II was not made early enough to be incorporated into our current implementation of implicit coscheduling.

To ensure that AM-II does not spin-wait, the Split-C parallel language run-time layer ensures that those cases are never exercised. By guaranteeing that the next slot in a FIFO queue of 16 entries is available before sending a message, the parallel run-time layer can be assured that AM-II will not spin-wait. Therefore, 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. Note that due to the FIFO data structure used in AM-II, this condition is more strict than simply ensuring that there are no more than 16 outstanding messages. Without this precaution, implicitly coscheduled programs that send many small one-way messages to random destinations may spin-wait; this deficiency causes erratic performance, with some slowdowns five times worse than ideal explicit coscheduling.

9.3 User Processes

For our parallel language, we use Split-C [39], a parallel extension to C with operations for accessing remote memory built on Active Messages. We chose Split-C because many of its applications are communication intensive, and, therefore, sensitive to scheduling perturbations. In addition, it closely matches the model assumed in the simulations. In this section, we describe the communication primitives within Split-C and the implementation of the two-phase waiting algorithm.

9.3.1 Communication Primitives

Split-C contains a rich set of communication operations, much more so than those in our model or our simulations.

- **Request-Response:** `Read` and `write` operations access remote memory, requiring the requesting process to wait for the response. `Get` and `put` are *split-phase* forms of read and write, respectively, where the initiating process does not wait for the response until a later synchronization statement, `sync`. In all versions of the Split-C library, `reads` and `writes` are implemented simply as `gets` and `puts`, respectively, immediately followed by a waiting `sync` statement. Bulk transfer is provided for each of these communication styles.
- **One-Way Request:** A process may write a remote memory location without waiting for an acknowledgment with a `store` operation. The receiving process waits for a specific number of bytes to be written with `store_sync`.
- **All-to-All Synchronization:** In addition to `barriers` which synchronize all processes, there also exists a complete set of reduction and broadcast operations. These other synchronization operations are currently built on top of `store` operations; therefore, processes perform the two-phase waiting algorithm within each `store_sync` operation with no global knowledge of load-imbalance.

9.3.2 Waiting Algorithm

Previous implementations of the run-time library in Split-C assumed a dedicated environment (or explicit coscheduling), and, therefore, relied entirely on spin-waiting at communication and synchronization events. Modifying the Split-C run-time library for implicit coscheduling requires two simple modifications: ensuring that AM-II does not spin-wait and altering the operations that wait for replies (*i.e.*, `syncs`, `store_syncs` and `barriers`) to use two-phase waiting.

Implementation

In the original implementation of Split-C, barriers were constructed in a hierarchical tree. The problem with this notification structure for implicit coscheduling is that one unscheduled process delays the notification of all the processes in that subtree. Therefore, in our modified version of Split-C, `barriers` are implemented by having all processes send and receive notification messages directly to and from root process; this matches the model of barriers in our analysis and our simulations. As we will see later in this section, the linear tree barrier requires significantly more time than the hierarchical version when the cluster has more than eight nodes.

The two-phase waiting algorithm is identical for `sync`, `store_sync`, and `barrier` operations, with the exception of the amount of baseline spin. The basic algorithm operates as follows. In the first phase, the Split-C library polls the AM-II layer (while recording the number of incoming requests) until the event completes or up to the amount designated by S_{Base}. If the event completed, the Split-C library returns control to the user application.

If the event has not completed, the number of incoming requests is examined; our implementation does not differentiate between different types of arriving messages (*i.e.*, requests requiring responses versus one-way requests). If the average time between message arrivals is less than T_{Cond}, then the process spins for a single S_{Cond} interval. The process continues to spin for intervals of S_{Cond} as long as at least one message arrives in the interval.

If the event has not yet completed, the second phase begins: the process calls the AM-II function `AM_SetAndWait` to relinquish the processor. When a message arrives, the process wakes; when the process is scheduled, 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 process continues its computation; otherwise, the process spins a small fixed amount, W, once more recording message arrivals to potentially enable conditional spinning.

Configuration of Spin-Time

The conditional spin amounts are determined by the models presented in Section 6.3. However, to more accurately configure the baseline spin-time at `syncs` (and `store_syncs`) and at `barriers`, we run a small set of microbenchmarks rather than rely strictly on the LogP models. These microbenchmarks allow us to include the computation overhead of the Active Message handlers and to observe the full distribution of completion times. The result from running each benchmark in a dedicated environment is then fed back into the Split-C library. The spin-times in our implementation are summarized in Table 9.2.

To determine the baseline spin-time that processes should wait at a `sync` operation, we measure the round-trip time when short messages are exchanged between pairs of processes. To measure T^R_{Sched}, both the sending and receiving processes remain scheduled and spin-wait throughout the experiment. To measure $T^R_{Trigger}$, the receiving process relinquishes the processor by calling `AM_SetAndWait` before each message arrives; since there are no other active processes, the receiving process is immediately scheduled when the message arrives.

Figure 9.3 shows the cumulative distribution of round-trip times under the two scenarios for 1000 messages. The data indicates that when the receiver is scheduled, most operations complete within $T^R_{Sched} = 60\mu s$. Note that this is higher than predicted by the simple LogP model of $2L + 4o = 34\mu s$. When the receiver must be scheduled when the message arrives, most operations complete within $T^R_{Trigger} = 130\mu s$. We have found that the baseline spin time should be determined by the knee-of-the-curve, rather than the average completion time. In our implementation, the knee usually occurred at the time at which 95-97% of the operations had completed. Therefore, $S^R_{Base} = \max(T^R_{Sched}, T^R_{Trigger}) = 130\mu s$. Finally, the time to trigger the scheduling of a process on message arrival is calculated as $W = T^R_{Trigger} - T^R_{Sched} = 70\mu s$.

To determine the baseline spin for `barriers`, we have a corresponding benchmark. To simplify our implementation of the barrier waiting algorithm, all processes wait for the same value of S^B_{Base}, even though some processes are notified earlier than others. Therefore, we measure the completion time of the `barrier` from the perspective of the last process to be notified by the root. Once again, for T^B_{Sched}, we examine the case where all processes are

Variable	Description	Equation	Value
S_{Base}	**baseline spin**	S^R_{Base} or S^B_{Base}	
S^R_{Base}	read baseline spin	$\max(T^R_{Sched}, T^R_{Trigger})$	$130\mu s$
T^R_{Sched}	request-response time (remote process scheduled)	Figure 9.3	$60\mu s$
$T^R_{Trigger}$	request-response time (remote process triggered)	Figure 9.3	$130\mu s$
S^B_{Base}	barrier baseline spin	$\max(T^B_{Sched}, T^B_{Trigger})$	$450\mu s$
T^B_{Sched}	barrier time (remote processes scheduled)	Figure 9.4	$450\mu s$
$T^B_{Trigger}$	barrier time (remote processes triggered)	Figure 9.4	$300\mu s$
S_{Cond}	**conditional spin**	S^R_{Cond} or S^O_{Cond} or S^B_{Cond}	
S^R_{Cond}	request-response conditional spin	$3W$	$210\mu s$
S^O_{Cond}	one-way request conditional spin	W	$70\mu s$
S^B_{Cond}	barrier conditional spin	$3W$	$210\mu s$
L_{Block}	blocking latency	—	—
V_{Block}	blocking load-imbalance	$\approx 20 \cdot W$	$\approx 1.5ms$

Table 9.2: **Conditional Two-Phase Waiting Parameters for Implementation.** *The table summarizes variables and equations for the time a process should spin before blocking.*

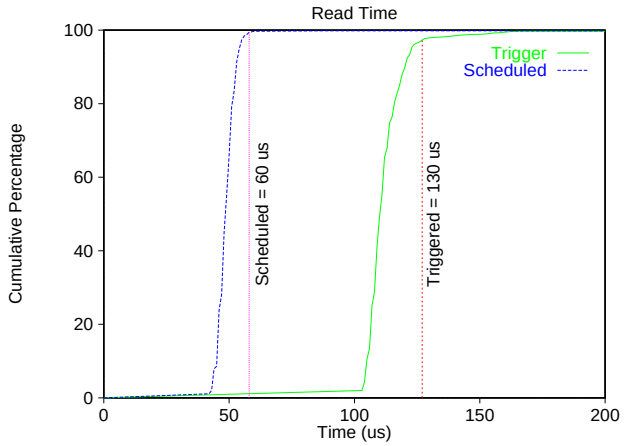


Figure 9.3: **Microbenchmark Results for Read Baseline Spin.** *The distribution of measured request-response times is shown for two cases: when the remote process is already scheduled when the request arrives and when the scheduling of the remote process is triggered by the arrival of the request message.*

scheduled throughout the experiment. For $T^B_{Trigger}$, we examine the case where all processes but one have relinquished the processor. To measure $T^B_{Trigger}$, all of the processes except the one that is notified last arrive at the barrier early and sleep while waiting for messages to arrive (*i.e.*, $P_{Late} = 1$). When the one late process arrives at the barrier, its notification message triggers the scheduling of the root process, which in turn notifies all processes. Since the barrier time depends upon the number of processors, this benchmark must be repeated for each cluster size.

Figure 9.4 shows the distribution of our barrier measurements for T^B_{Sched} and $T^B_{Trigger}$ on a range of cluster sizes between $P = 2$ and $P = 32$. When all processes reach the barrier simultaneously, as when measuring T^B_{Sched}, the root process spends a significant amount of time handling the notification messages ($(P-1)\cdot\max(o,g)$). When one process reaches the barrier much later than the others, the root has already handled the previous notification messages, and thus the late process sees only the cost of $W+o$ required for the root to handle this one notification message. Since $(P-1)\cdot\max(o,g)$ is greater than $W+o$ for clusters of size greater than eight, T^B_{Sched} is larger than $T^B_{Trigger}$on those clusters. Thus, $S^B_{Base} = T^B_{Sched} = 450\mu s$.

Our linear barrier implementation was chosen to avoid the problem where an unscheduled process prevents a subset of the parallel job from being notified that the barrier is complete. However, as Figure 9.4 shows, the drawback of the linear barrier is that its completion time increases linearly with the number of processes. On 32 processors, the barrier requires more than $800\mu s$ to complete.

The high cost of the barrier has an interesting interaction with implicit coscheduling. On clusters of approximately 32 nodes, better performance is achieved when processes block immediately at barriers rather than wait for the barrier to complete, even when there is no load-imbalance in the application. In fact, in this regime, implicit coscheduling

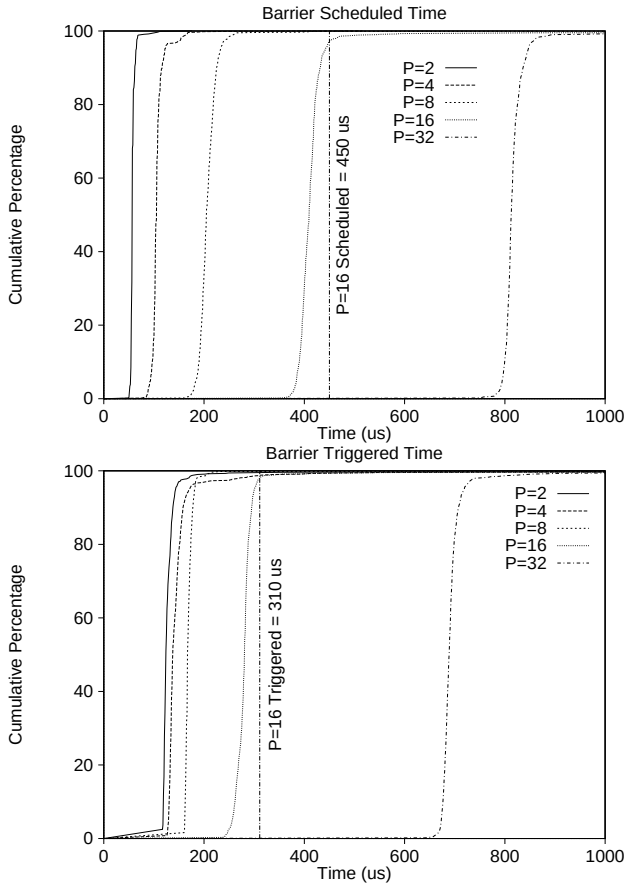


Figure 9.4: **Microbenchmark Results for Linear Barrier Baseline Spin.** *The top graph shows the distribution of barrier completion times (from the perspective of the process notified last) if all participating processes are coordinated and send notification messages simultaneously to the root process. The bottom graph shows the distribution when all but one process arrive at the barrier early and block until the barrier is completed; when the last notification message arrives at the root process, it is scheduled and sends responses to all processes. The number of participating processes is increased from two to 32 in both sets of measurements.*

with immediate-blocking achieves a speed-up relative to that of explicit coscheduling with spin-waiting.

As a result, for most of our measurements in the following chapter, we examine clusters of 16 nodes. For this cluster size, better performance is achieved when processes are coordinated at barriers than when processes block immediately, thus stressing the ability of implicit coscheduling to dynamically coordinate cooperating processes. However, future research must examine the interaction of implicit coscheduling with hierarchical-tree barriers in order to achieve good absolute performance on larger clusters.

New Split-C Interfaces

Finally, three new functions are added to Split-C so that user applications can wait for messages with the appropriate two-phase waiting algorithm. The most general function, `wait_until`, takes a pointer to an arbitrary function that must be satisfied before the wait function returns. The next two functions are optimizations for special cases that are expected to occur frequently: `wait_until_x` returns control when the given memory location contains the specified value; `wait_until_not_x` returns when the location contains any value other than the one specified.

```
void wait_until(int (*condition)());
void wait_until_x(volatile int *addr, int value);
void wait_until_not_x(volatile int *addr, int value);
```

9.4 Application Workload

We evaluate both synthetic and real applications in our implementation. Analyzing synthetic applications exposes the strengths and weaknesses of implicit coscheduling in a controlled environment. However, the synthetic programs do not change behavior over time and do not mix different communication styles; therefore, we also examine a set of seven Split-C applications with a variety of characteristics.

9.4.1 Synthetic Applications

The synthetic applications are designed to closely match the bulk-synchronous and continuous-communication benchmarks employed in the simulations. These synthetic applications allow us not only to evaluate our implementation in a controlled fashion, but also to compare our results to those predicted by the simulations. These benchmarks have the further advantage that they require only a small amount of memory, allowing us to run many competing applications without paging of virtual memory. The primary difference from the simulation versions is that we increase the number of iterations performed so that the Split-C applications require between 20 and 30 seconds to complete in a dedicated environment. We also evaluate performance with one-way `stores` and bulk messages.

Program	Description	Problem Size (per P)	Message Pattern and Style
`mm`	Matrix multiply	256 x 256	Regular all-to-all bulk-stores
`radix`	Radix sort	2M keys	Regular all-to-all bulk-stores
`radix:small`	Radix sort	128K keys	Random all-to-all stores
`fft`	Fast Fourier transform	512K points	Regular all-to-all bulk-stores
`fft:small`	Fast Fourier transform	256K points	Regular all-to-all stores
`em3d`	Electro-magnetic wave propagation	5000 nodes, 20 steps degree: 20, 40% remote distribution 3	Neighboring bulk-stores
`em3d:small`	Electro-magnetic wave propagation	5000 nodes, 10 steps degree: 20, 40% remote distribution 3	Neighboring reads
`em3d:init`	Initialization phase of `em3d`	–	Random all-to-all stores and reads

Table 9.3: **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. The initialization phase of the two versions of EM3D consume a significant portion of the execution times and have much different communication characteristics.*

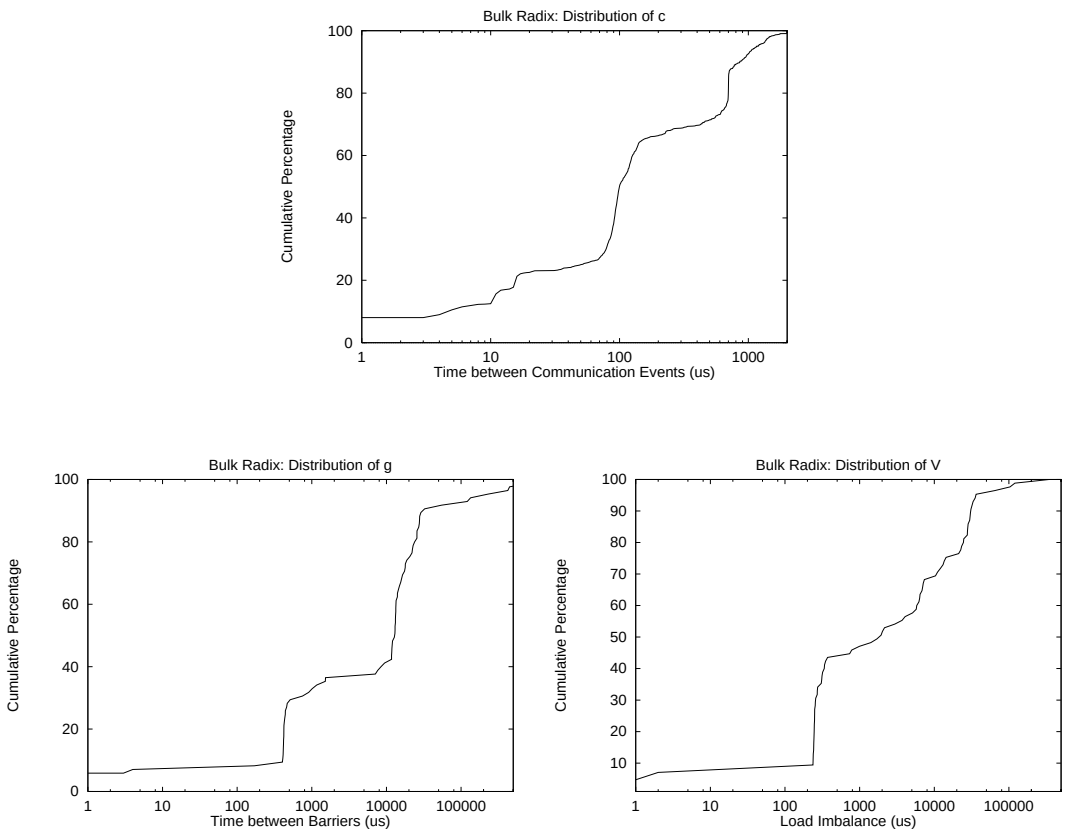


Figure 9.5: **Communication Characteristics of Radix.** *The graphs show the distribution of the time between communication events (c), barriers (g), and load-imbalance (V) for Bulk Radix Sort. The measurements were taken for one job of 16 processes running on a dedicated system. The data shows that there is approximately 100µs of computation between 70% of the communication operations. Radix performs only about 80 barriers in its lifetime.*

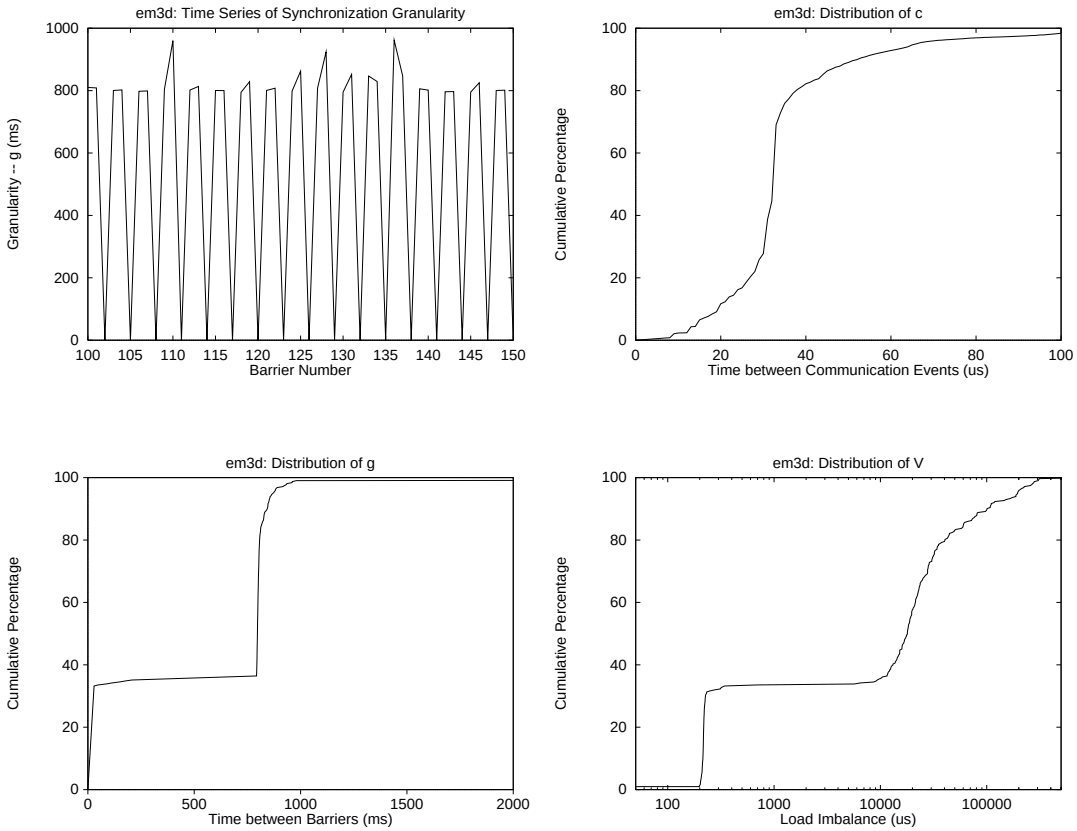


Figure 9.6: **Communication Characteristics of EM3D.** *The first graph shows the regular nature of EM3D, in which e-nodes and h-nodes are calculated in turn for a number of time-steps. The main loop of the application contains three barriers: one at the beginning of the time-step, one after performing the calculations for e-nodes, and one after the calculations for h-nodes. Thus, the average time for each e-node or h-node step across all processes is about 800ms. The remaining graphs show the distribution of the time between communication events (c), barriers (g), and load-imbalance (V). The barriers after the e-node and h-node calculations have between* $V = 10ms$ *and* $V = 300ms$ *worth of load-imbalance.*

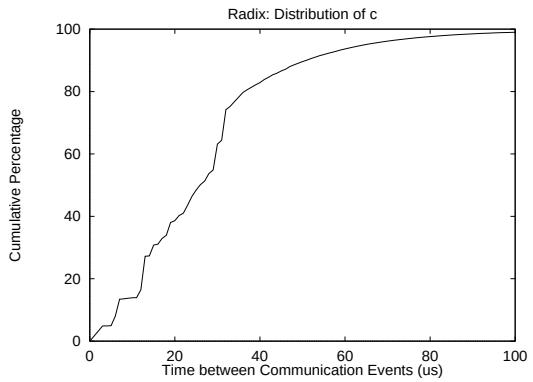
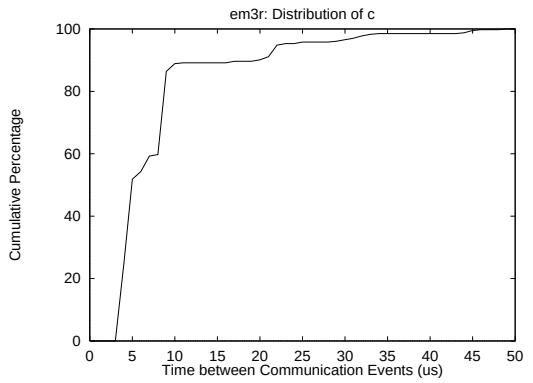
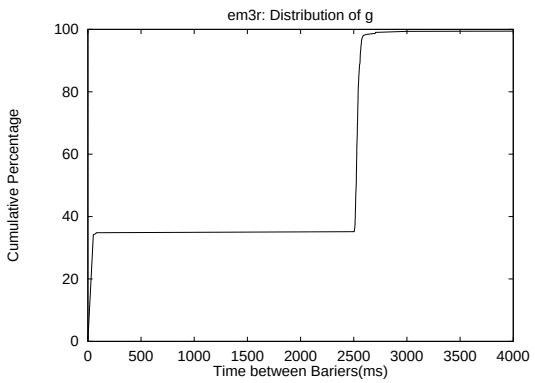


Figure 9.7: **Communication Characteristics of Radix:Small.** *Because this version of Radix Sort performs very few barriers, the graphs for g and V are not shown. The graph shows that there is only 30μs of computation (including the overhead of collecting the tracing information) between 75% of communication operations.*





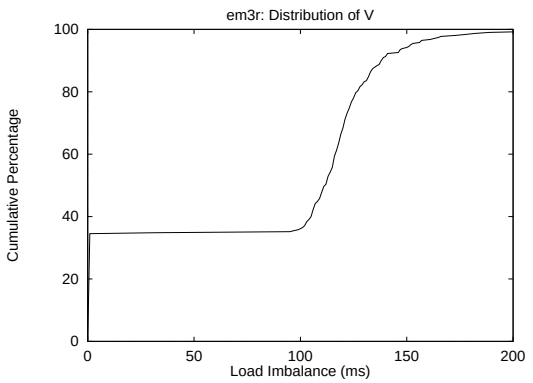


Figure 9.8: **Communication Characteristics of EM3D:Small.** *The version of EM3D:Small proceeds in time-steps like EM3D shown in Figure 9.6; however, communication occurs much more frequently ($c \approx 10\mu s$) and the time-steps are significantly slower ($g \approx 2.5s$).*

9.4.2 Real Split-C Applications

To evaluate a more varied set of applications, we measure the performance of seven Split-C programs written by different programmers. The applications include matrix multiplication, `mm`, two versions of radix sort, `radix` [3, 48], two fast Fourier transforms, `fft` [3, 40], and two versions of a model of electro-magnetic waves propagated in three dimensions, `em3d` [39]. The problem sizes, shown in Table 9.3, were chosen such that three copies of each application could fit into 128MB of memory without paging and such that each application required between 10 and 60 seconds in a dedicated environment. The only modifications made to any of the programs were to replace instances of spin-waiting in the `radix` sort programs with calls to the Split-C functions to implement two-phase waiting.

When two versions of an application exist, one copy has been optimized to communicate with large messages, while the other uses short messages. The seven applications exhibit a variety of communication characteristics. For example, in the bulk version of `em3d` there is $60\mu s$ of computation between most messages and $900ms$ between barriers, while in `em3d:small` there exists only $10\mu s$ between messages and nearly 3 seconds between barriers. Figures 9.5 through 9.8 show the distribution of the time between barriers (g), the load-imbalance across processes (V), and time between communication operations (c) for the two versions of `em3d` and `radix` sort.

9.5 Operating System Scheduler

Scheduling processes in a cluster contains two steps. First, processes must be allocated to nodes of the cluster. Second, the local operating system scheduler on each workstation must schedule its processes over time. Since the focus of this dissertation is on the second step, we describe these issues in the reverse order. We begin by describing the two local schedulers (*i.e.*, the default Solaris time-sharing scheduler and our implementation of stride scheduling with system credit) and our model of explicit coscheduling. We then briefly describe our approach to ensure that jobs receive a fair proportion of resources in the shared cluster.

9.5.1 Local Schedulers

Time-Sharing

When our implementation measurements do not require fairness across competing jobs (*i.e.*, when the workload contains jobs with similar communication characteristics) we use the Solaris 2.6 Time-Sharing scheduler (TS). The process scheduling modules in Solaris 2.6 are very similar to those in Solaris 2.4, as used in the simulations and described in Section 7.5.2; therefore, we do not repeat the description here.

Stride Scheduling

To fairly schedule applications with different communication characteristics, we have also implemented a Stride Scheduler with System Credit (SSC) as a new scheduling

class in Solaris. The advantages of adding a new scheduling class are mainly administrative. By using an already existing interface, we do not need source code for the kernel and can distribute our module to outside users. Development and distribution is also simplified since the scheduling module can be dynamically loaded and unloaded, without rebooting the machine. Specifying that a new job should belong to the stride scheduling class, or should be given t tickets, is easily performed with the `priocntl` command from the shell, or by forking off new processes from a process already in the stride scheduling class. The basic functionality of SSC matches that originally described in Section 5.3, however, a few details had to be changed due to the existing interface for scheduling classes in Solaris.

The basic function of the SSC module is to raise the priority of the process with the lowest `pass` and lower the priority of all other processes; in this way, the class-independent scheduling functions will dispatch the desired process. Allocation of a new process can occur in three circumstances: when the time-slice of the scheduled process expires ($Q = 100ms$), when a process with a lower `pass` value than the scheduled process becomes runnable (*e.g.*, due to a message arrival), and when the running process voluntarily relinquishes the processor.

The problem with this straight-forward implementation in Solaris, is that a per-process lock must be acquired before modifying the priority of a process. However, it is not always possible to acquire this lock. A clock-tick that expires every $10ms$ invokes the routine that determines when a time-slice has expired; however, this clock-tick runs at the highest system-level priority, and, therefore, can neither be preempted nor block. As a result, no locks can be acquired in the clock-tick routine, forcing a slight redesign of the approach used in the simulator and described in Section 7.5.2.

Rather than modify process priorities in the clock-tick routine (and potentially on every sleep and wake-up event), our current implementation changes the priorities of jobs only when a periodic timer expires every $Q = 100ms$. This timer runs at a lower system-level priority, and therefore can freely acquire and release locks (similar to the $1s$ timer in the Solaris TS scheduler). Thus, every $Q = 100ms$, a modified allocation routine is called that orders the priority of every process in reverse rank according to its `pass` value. Within this $100ms$ time-slice, the highest-priority process will always be scheduled when it is runnable; as before, its `pass` is incremented by its `stride` at every $10ms$ clock-tick. If the running process voluntarily relinquishes the processor, the process with the next highest priority is automatically dispatched with no additional computation. When the highest-priority process reawakens, it is given exhaustible tickets by the system, and automatically preempts the lower-priority process.

A second modification was also required in our implementation that did not exist in our simulations. Solaris does not immediately inform the scheduling class when a process exits; instead, this routine is called only when the process storage needs to be reclaimed; *e.g.*, when another process is started or after a thread reaper runs. The difficulty that arises is that the stride scheduler believes that the tickets belonging to the exited process are actively competing; consequently, the value of the other tickets funded by the same currency are artificially decreased. To fix this problem, when the stride scheduling allocation routine examines each process every $100ms$, it looks for processes which are marked as `free` or as a `zombie`, and deallocates its tickets.

9.5.2 Explicit Coscheduling

Rather than implement a version of explicit coscheduling to serve as a comparison point for implicit coscheduling, we model the explicit coscheduling performance of a workload by simply adding together the execution time of each application when run in a dedicated environment with spin-waiting at all communication and synchronization events. This approach represents the ideal explicit coscheduling performance because it does not capture any cache effects or additional overhead for global context-switching. For most measurements, we report the slowdown of the workload with implicit coscheduling relative to ideal explicit coscheduling. We repeat each experiment between five and ten times and report the average slowdown.

9.5.3 Job Placement

To place the parallel jobs across the nodes of the cluster, we leverage GLUnix, a Global-Layer UNIX [65]. GLUnix tracks the load on each machine in the cluster, redirects standard I/O, and provides job control. In our experiments, we specify the precise machine on which each process should be placed.

Chapter 10

Implementation Study

In this chapter, we have two goals as we present measurements for our implementation prototype. Our first goal is to compare the measurements of our prototype implementation to those predicted by our simulator. To facilitate this comparison, we evaluate synthetic applications with the same bulk-synchronous and continuous-communication characteristics as those studied in the simulations. The most significant change in our experimental workload is that we run on a cluster of only 16 workstations for most of our experiments, rather than the 32 workstations measured in the simulations.

Our second goal is to show the effectiveness of implicit coscheduling on a more thorough set of applications and workload combinations:

1. **Additional Communication Primitives:** one-way requests and bulk messages.
2. **Real Applications:** a collection of seven Split-C applications with a variety of communication primitives and message patterns.
3. **Scalability:** sensitivity to the number of competing jobs and workstations.
4. **Job Placement:** allocation of processes to workstations such that the load across workstations is not balanced.

10.1 Verification of Simulations

Our first experiments compare the performance of our implementation to that predicted by the simulator. We begin by verifying that the parameters for baseline and conditional spinning derived in Chapter 6 are appropriate for a range of bulk-synchronous and continuous-communication workloads. In these experiments we run with the Solaris Time-Sharing scheduler. In later experiments, we evaluate fairness across jobs with different communication characteristics; for these workloads we compare the Solaris Time-Sharing scheduler to our Stride Scheduler with System Credit.

10.1.1 Baseline Spin

In this section, we show the importance of baseline spinning for bulk-synchronous applications. Our first set of experiments confirm that the baseline spin amounts derived in Section 6.2 are optimal in our implementation for applications with no load-imbalance; these experiments closely duplicate the simulations performed in Section 8.2. Our second set of experiments show that the minimum baseline spin also performs well for applications with load-imbalance.

Sensitivity to Baseline Spin

In our first workloads, three bulk-synchronous jobs compete on 16 workstations. Each process performs a barrier every $g = 100\mu s$, then communicates with its four nearest neighbors in a `NEWS` pattern (with $c = 8\mu s$ of intervening computation), and finally performs a second barrier; these steps are repeated for approximately 20 seconds. Because we are validating the best *minimum* baseline spin-time, we examine applications with no load-imbalance (*i.e.*, $V = 0$).

To separately evaluate the impact of baseline spin on reads and barriers, we vary the baseline spin for reads while keeping the baseline spin for barriers fixed at its optimal point of $S^B_{Base} = 450\mu s$. No pairwise spinning is performed in these experiments. Figure 10.1 shows the slowdown of the workload with implicit coscheduling relative to the ideal model of explicit coscheduling.

These measurements agree with both our initial theory and our previous simulation results. That is, the performance of the workload improves at two points: when the baseline spin amount equals $T^R_{Sched} = 60\mu s$ and when the baseline spin amount equals $T^R_{Trigger} = 130\mu s$. As expected, performance is poor when processes spin less than T^R_{Sched}, because processes do not remain coordinated after a communication event. Thus, processes pay at least a context-switch for every communication operation. The best performance is achieved when processes spin at least $T^R_{Trigger}$, the completion time of a read when the request messages triggers the scheduling of the remote process; this point is marked on the graph. Performance degrades slowly as processes spin longer than $T^R_{Trigger}$, since processes pay a higher penalty in spin-time for those read operations that do not complete successfully.

While the general shapes of the curves from the simulation and implementation are similar, there are two obvious differences. First, in the simulations, as the baseline spin for reads was increased, the measured performance dramatically improved at precisely the points where the baseline spin equaled T^R_{Sched} and $T^R_{Trigger}$. In our implementation, since T^R_{Sched} and $T^R_{Trigger}$ represent distributions of completion times (shown in Figure 9.3) and not distinct points, performance improves more smoothly as baseline spin is changed.

Second, performance is less sensitive to spin-time in the implementation than in the simulations. For example, in the simulations, even with the lower context-switch cost of $W = 50\mu s$, performance was 3.5 times worse with immediate blocking than with explicit coscheduling. In the implementation, performance is only 70% worse with immediate blocking. This effect is discussed after our next set of experiments.

The next experiments measure performance as the barrier baseline spin is varied while the read baseline spin is held constant at S^R_{Base}. The results shown in the second graph

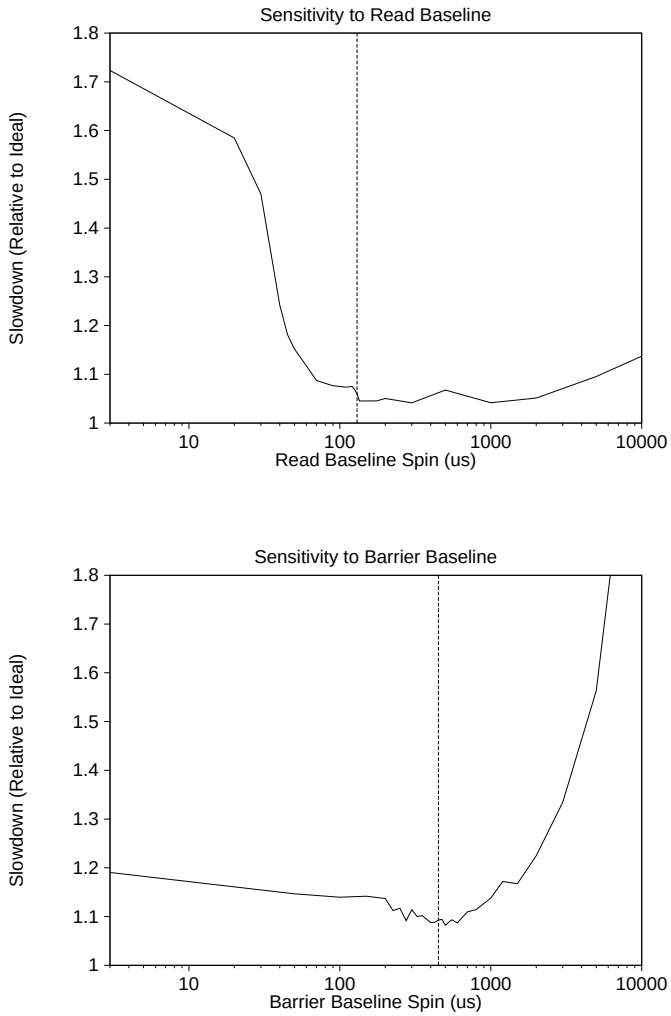


Figure 10.1: **Sensitivity to Baseline Spin for Bulk-Synchronous Programs.** *In the first graph, the baseline spin for reads is varied between 1μs and 1000μs along the x-axis, while the baseline spin for barriers is fixed at S^B_{Base}. In the second graph, the baseline spin for barriers is varied while the read spin time is kept constant at S^R_{Base}. The vertical line in each graph designates the baseline spin amount, S_{Base}, shown in Table 9.2. The metric along the y-axis is the slowdown of the workload when implicitly coscheduled with two-phase waiting versus explicit coscheduling with spin-waiting; the points represent the average of five runs. The workload consists of three bulk-synchronous jobs on 16 workstations. The time between barriers, g, is set to 100μs, there is no load-imbalance ($v = 0$), and the time between reads after the barrier is fixed at $c = 8\mu s$.*

of Figure 10.1, once again qualitatively match our previous simulation results. First, the best performance is achieved when processes spin approximately $S^B_{Base} = 450\mu s$. Second, as processes spin longer than S^B_{Base}, performance degrades severely. Because a noticeable percentage of barriers do not complete successfully regardless of spin-time, processes pay the penalty of extra spin time on a significant number of the barriers; this effect appears more prominent in our implementation than in our simulations because we examine baseline spin amounts up to $10ms$, rather than up to only $1ms$.

However, as with reads, when processes block immediately at barriers, the performance of our implementation is not as poor as that in our simulations. In fact, performance is only 20% worse than with explicit coscheduling. This performance discrepancy is due to the presence of message overhead, o, and gap, g, in the implementation, but not in the simulations. Due to o and g, barriers in our implementation require a significantly longer time than in the simulations. Since the time for a barrier on 16 processors ($S^B_{Base} = 450\mu s$) is significantly longer than a context-switch ($W = 70\mu s$), there is a smaller penalty for losing scheduling coordination. Slow barriers have similar performance implications as barriers with load-imbalance: there is an advantage to blocking before the barrier completes in order to allow competing processes to perform useful work.

In summary, the baseline spin amounts derived in Section 6.2 and shown in Table 9.2 achieve the best performance for a workload with bulk-synchronous applications with no load-imbalance. At read operations, processes should spin $S^R_{Base} = 130\mu s$ within the two-phase waiting algorithm before relinquishing the processor; at barrier operations, processes should spin $S^B_{Base} = 450\mu s$.

Sensitivity to Load-Imbalance

The previous experiments showed that losing scheduling coordination for bulk-synchronous applications with no load-imbalance had little penalty; there is even less relative penalty for losing coordination when there is load-imbalance across processes. In our next set of experiments, we show that spinning the minimum baseline amount at reads and barriers is sufficient for a wide variety of bulk-synchronous applications. In these workloads, we examine both the `NEWS` and `Transpose` communication patterns for a range of computation granularities (between $g = 100\mu s$ and $g = 100ms$) and levels of load-imbalance (between $V = 0$ and $V = 2 \cdot g$).

The measurements presented in Figure 10.2 show that for all workloads on 16 workstations, the slowdown with implicit coscheduling is no greater than 15% worse than ideal explicit coscheduling. Therefore, little additional benefit can be achieved by approximating load-imbalance at run-time for these workloads. These results closely match the predictions of the simulator for a system with $W = 50\mu s$ (shown in Figure 8.12), where the worst-case slowdown is within 25% when processes spin the minimum baseline amount. As predicted for a similar range of simulated workloads in Figure 8.16, applications with a large amount of load-imbalance (*e.g.*, $V > 20ms$) exhibit speedups relative to explicit coscheduling.

In summary, given the high costs of barrier operations relative to context-switches in our environment, it is not necessary for bulk-synchronous applications to approximate

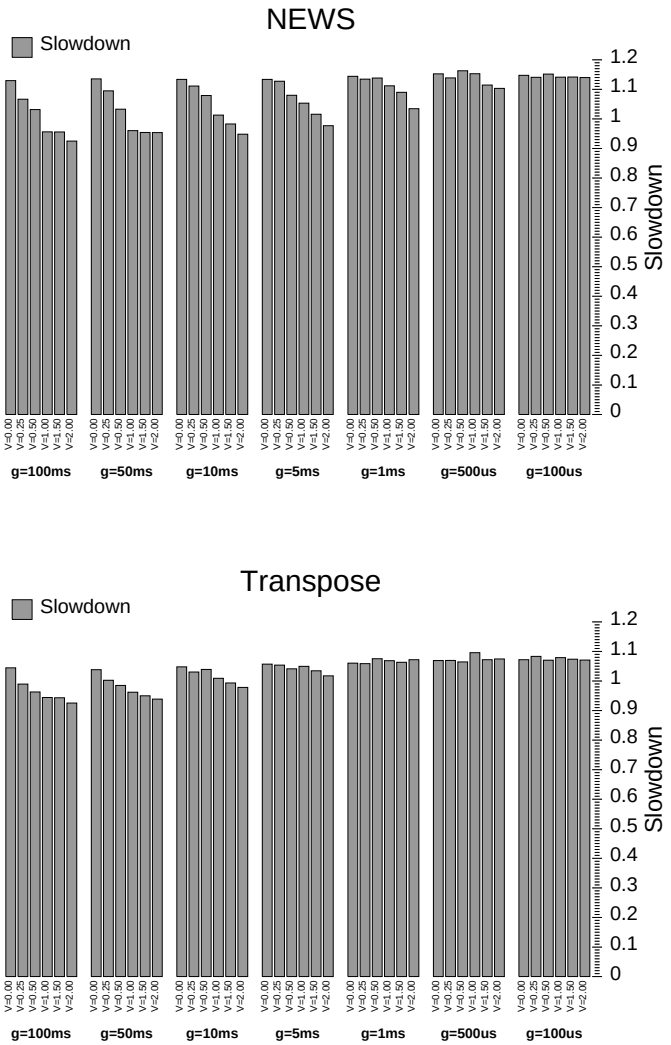


Figure 10.2: **Performance on Bulk-Synchronous Workloads.** *The graphs compare implicit coscheduling with the full conditional two-phase waiting algorithm (i.e., baseline and conditional spin amounts as specified in Table 9.2) to ideal explicit coscheduling with spin-waiting. The workload consists of three bulk-synchronous jobs on 16 workstations. Both the* `NEWS` *and* `Transpose` *communication patterns are examined; in each case, the time between read operations is fixed at* $c = 8\mu s$*. The time between barriers,* g*, is varied between* $100\mu s$ *and* $100ms$ *across the sets of bars. Within each set of bars, the load-imbalance across processes is varied from* $V = 0$ *to* $V = 2 \cdot g$*.*

load-imbalance at run-time. Instead, it is sufficient for processes to spin-wait the minimum baseline amount at barriers and lose coordination when load-imbalance exists. With this simple implementation of implicit coscheduling, bulk-synchronous applications perform within 15% of implicit coscheduling.

10.1.2 Conditional Spin

To show the importance of conditional spinning, we investigate applications that are more difficult to schedule: those with continuous communication in all phases. The simulation results in Section 8.3 showed that conditional spinning with S^R_{Cond} significantly improved performance. However, the simulations examined a relatively limited set of applications: the time between barriers was fixed at $g = 100ms$ and there was no load-imbalance across processes. For our implementation measurements, we expand our workload to examine a larger set of synchronization granularities (from $g = 100\mu s$ to $g = 100ms$) and two load-imbalances ($V = 0$ and $V = 1.5 \cdot g$).

Comparison of Waiting Algorithms

Figure 10.3 compares the performance of three different waiting algorithms: blocking immediately, two-phase waiting with baseline spinning, and two-phase waiting with baseline and conditional spinning. The spin amounts used in the two-phase waiting algorithms match those summarized in Table 9.2. In these workloads, processes repeatedly read from their four nearest neighbors in a regular `NEWS` pattern.

The trends in these graphs match those found in the simulations (Figure 8.4) in several respects. First, as expected, blocking immediately performs more poorly on the continuous-communication workload than on the bulk-synchronous workload. Second, the smaller the amount of computation between reads (*i.e.*, the smaller the value of c) and the less load-imbalance (*i.e.*, the smaller the value of V), the more performance suffers when processes block immediately.

However, once again, the simulations overestimate the penalty of blocking immediately relative to explicit coscheduling. For example, with $g = 100ms$ between barriers and $c = 10\mu s$ between reads, the performance of immediate blocking is roughly 3.25 times worse than explicit coscheduling in the simulations, but less than 2.75 times worse in our implementation. Nevertheless, in neither environment is blocking immediately a viable approach for processes that communicate frequently.

Figure 10.3 also shows that performing two-phase waiting with baseline spinning improves performance significantly over blocking immediately. When there is no load-imbalance in the applications, performance is always within 52% of ideal explicit coscheduling, in stark contrast to the 2.4 times slowdown measured in the simulations (Figure 8.7). As a result of this better than expected improvement, conditional spinning achieves less additional benefit. For example, at the point where $g = 100ms$, $c = 10\mu s$, and $V = 0$, conditional spinning improves the execution time of the workload from 52% worse than explicit coscheduling to within 28%.

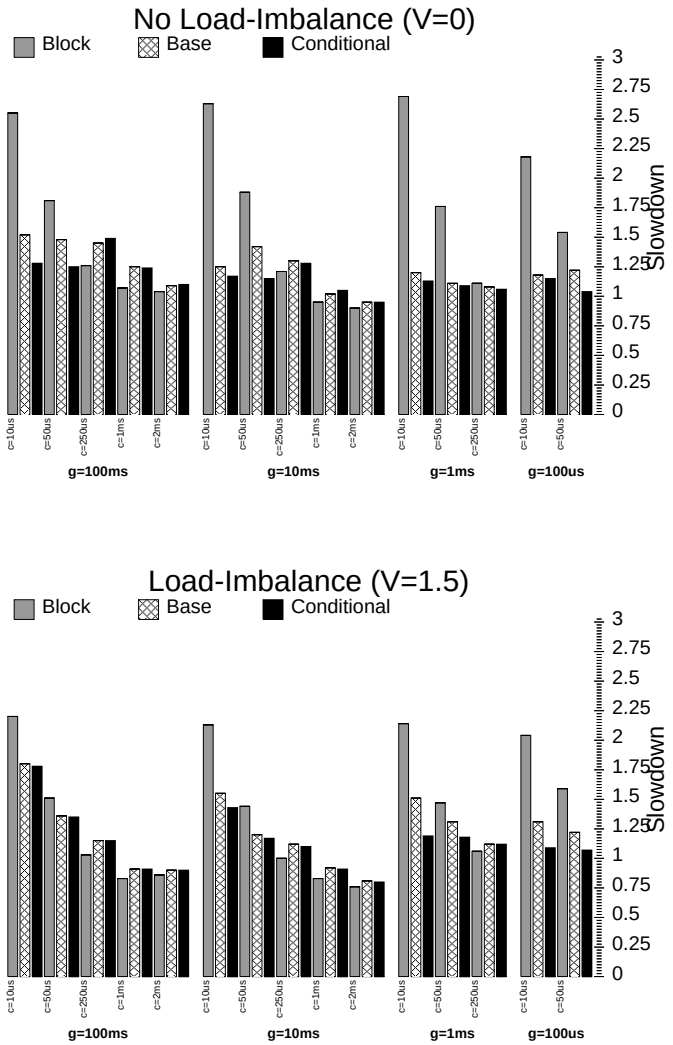


Figure 10.3: **Sensitivity to Waiting Algorithm for Continuous-Communication Programs.** *We compare the performance of the workloads when processes block immediately, two-phase wait with baseline spinning, and two-phase wait with both baseline and conditional spinning. The workload consists of three jobs running on 16 workstations. Each application communicates with request-response (read) messages in a* `NEWS` *pattern. The time between barriers is varied between* $g = 100\mu s$ *and* $g = 100ms$ *and the time between reads is also varied between* $c = 10\mu s$ *and* $c < g$*. The applications in the first graph have no load-imbalance (*$V = 0$*); in the second graph, the load-imbalance is* $V = 1.5g$*.*

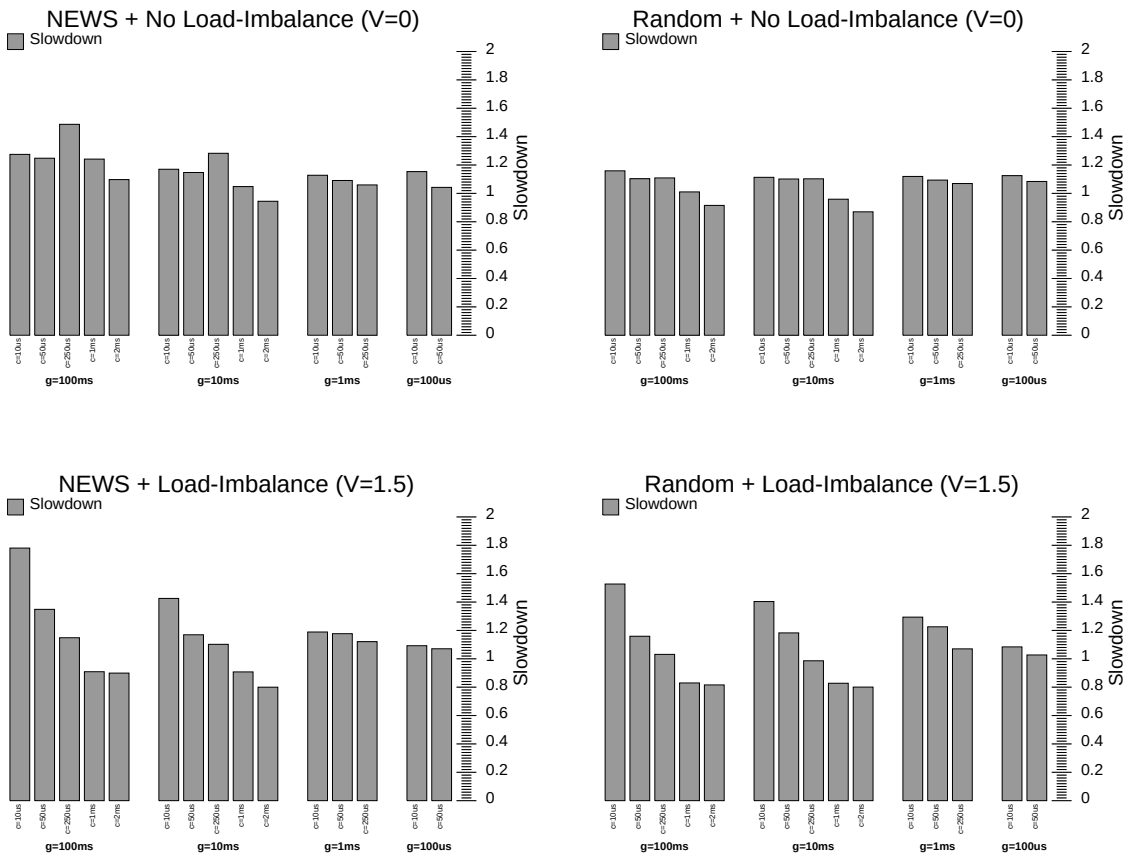


Figure 10.4: **Performance of Continuous-Communication Programs with Request-Response Messages.** *The workload consists of three jobs running on 16 workstations. Each application communicates with request-response (read) messages in either the* `NEWS` *or* `Random` *pattern. The time between barriers is varied between* $g = 100\mu s$ *and* $g = 100ms$ *and the time between reads is also varied between* $c = 10\mu s$ *and* $c < g$*. The applications in the first set of graphs have no load-imbalance (*$V = 0$*); in the second set of graphs, the load-imbalance is* $V = 1.5g$*.*

Conditional Spin Performance

Figure 10.4 explores the performance of conditional spinning in more detail, expanding the workload to include applications communicating in a `Random` pattern. There are four interesting facts to observe from these measurements.

First, conditional spinning is highly effective for all but a few extreme communication and synchronization rates. For example, performance is usually within 30% of explicit coscheduling for workloads with no load-imbalance. As expected from the bulk-synchronous applications, jobs with frequent barriers perform similarly to explicit coscheduling and jobs with significant load-imbalance and infrequent communication (*e.g.*, $c > 1ms$) exhibit speedups.

Our second observation is that 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 80% worse than explicit coscheduling. We believe that such workloads will not occur often in practice, because their performance will be poor even in a dedicated environment. To support these extreme applications, the two-phase waiting algorithm must predict the amount of load-imbalance in the application 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.

Third, we note that workloads that communicate in a `NEWS` pattern perform slightly worse than workloads that communicate in an all-to-all `Random` pattern. Applications in which all processes communicate with one another stay coordinated as a unit more effectively than applications in which processes communicate in subgroups. As described in Section 6.4.2, all-to-all communication patterns encourage either all or none of the processes to remain scheduled. This effect matches the simulation results in Figure 8.7 and the simulations reported in [47]: when processes did not spin the full baseline amount, processes communicating in an all-to-all `Transpose` pattern stayed coordinated through the communication phase more often than those communicating in the `NEWS` pattern.

Fourth, the `Random` pattern with infrequent barriers and infrequent communication (*i.e.*, $c = 2ms$) observes a speedup relative to explicit coscheduling, even with no load-imbalance. This effect is due to the fact that processes are only notified of message arrivals when they either send a message or explicitly poll the network. Even though each process performs about the same amount of computation between communication events, there is always some small amount of variation in the actual amount of intervening computation; that is, cooperating processes do not communicate with one another at precisely the same time. Because messages are only handled when the destination process touches the network, a sending process that is slightly slower than its destination must wait until the next time the destination process communicates for the request to be serviced. As a result, even in a dedicated environment, a sending process may spin-wait for more than $2ms$ for the reply. With such high waiting times at read operations, processes can achieve better performance by spinning only the baseline amount and relinquishing the processor if the response does not return in the expected interval.

Finally, the data indicates that conditional spinning is not effective when processes are communicating in the `NEWS` pattern at an interval of approximately $c = 250\mu s$ and

there are infrequent barriers. This communication rate has particularly poor performance because the time between arriving messages is just slightly longer than the interval required to activate conditional spinning, $S^R_{Cond} = 3W = 210\mu s$. As a result, conditional spinning is rarely activated for these workloads. Even worse, when conditional spinning is activated, it is unlikely that any messages arrive in the next interval, and thus the additional spin-time is wasted. Additional experiments (not shown) that evaluated values of S^R_{Cond} other than $3W$ produced only marginal improvements for these workloads and drastically harmed others.

In this section exploring conditional spinning, we have demonstrated that the performance of implicit coscheduling on continuous-communication applications is within 30% of explicit coscheduling for most workloads containing three jobs on 16 workstations. However, in a few extreme cases of very frequent communication coupled with large load-imbalance, applications exhibit slowdowns of nearly 80%. We want to stress that these workloads are unlikely to occur often in practice: even applications with excessively large amounts of load-imbalance perform adequately as long as processes compute for at least $50\mu s$ between messages.

10.1.3 Local Scheduler

In the previous measurements in this chapter we evaluated workloads containing multiple copies of the same application. As a result, fair allocation of resources is not an issue and the Solaris Time-Sharing (TS) scheduler is sufficient for evaluating the effectiveness of implicit coscheduling. However, as shown with the simulations in Section 8.5, the TS scheduler does not adequately handle workloads containing jobs with different communication rates, giving applications that communicate less frequently more than their fair share of resources.

In this section, we reproduce a subset of the simulation experiments that evaluated the fairness of the Solaris Time-Sharing scheduler and the Stride Scheduler with System Credit (SSC). In these measurements, as in the corresponding simulations, the performance metric is the relative execution time with implicit coscheduling versus explicit coscheduling for one job competing against a number of continuously-running background jobs. Rather than repeat the full combination of workloads examined in the simulations, we focus only on those workloads shown to be particularly problematic. That is, rather than vary the communication characteristics of both the measured job and the background jobs, we examine a single fine-grain job in competition with background applications whose computation granularities are varied from fine to coarse-grain.

Once again, we call the slowdown that a job experiences against jobs with identical communication characteristics the *base slowdown*. The local scheduler provides *fair* performance for a workload with a measured job A and a collection of background jobs if it exhibits slowdown near the base slowdown for job A. If the scheduler tends to favor the measured job within the workload, then its slowdown in that workload is less than its base slowdown; if the scheduler is biased against this measured job, its slowdown in that workload is larger than its base slowdown.

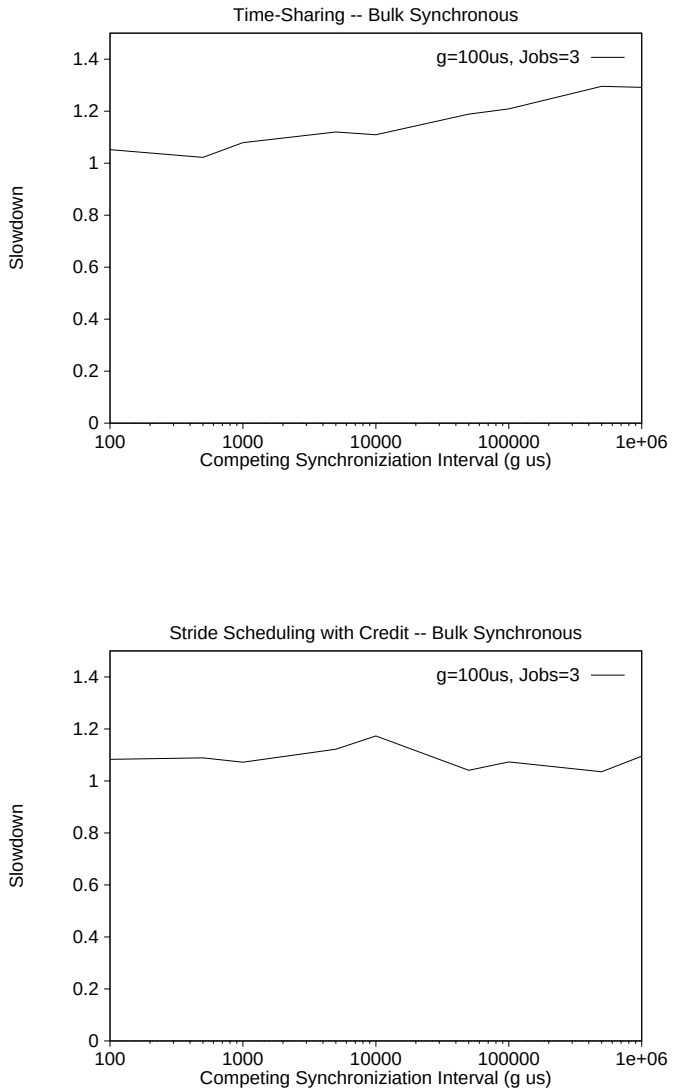


Figure 10.5: **Fairness with Bulk-Synchronous Programs.** *One measured job and two competing background jobs are allocated to 16 workstations. Only the slowdown of the one job is measured; the background jobs run continuously during the experiment. We evaluate bulk-synchronous workloads; the time between barriers in the measured job is fixed at $g = 1ms$, while g is varied in the background jobs up to $g = 1s$. All bulk-synchronous jobs have $V = 0$, $c = 8\mu s$, and communicate in the* `NEWS` *pattern. The first graph uses the default Solaris Time-Sharing scheduler on each workstation; the second graph uses our implementation of the Stride Scheduler with System Credit.*

Bulk-Synchronous Workloads

In Figure 10.5 we evaluate a bulk-synchronous workload containing three jobs on 16 workstations with both the TS and the SSC schedulers. In the measured job, the time between barriers is held constant at $g = 1ms$, while in the background jobs, the time between barriers is varied up to $g = 1s$. In these workloads, all jobs communicate in the `NEWS` pattern with $c = 8\mu s$ and no load-imbalance at barriers ($V = 0$).

Our measurements with the TS scheduler are similar to those predicted by the simulations in Figure 8.17. In both environments, the slowdown of the fine-grain measured job gradually worsens as the granularity of the background jobs is increased. For example, in the simulations when the time between barriers in the measured job is $g = 100\mu s$, the base slowdown is 12%; when competing against coarse-grain jobs where $g > 100ms$, slowdown increases to 40%. In our implementation, fairness degrades from base slowdown of 5% to a slowdown of 30%. We believe the slight improvement in performance is due to the fact that we assume a higher context-switch cost of $W = 200\mu s$ in the simulations.

More dramatically, measurements with the SSC scheduler show that slowdown is relatively flat as the granularity of the background jobs is increased. The simulations predicted that performance with SSC is fair until the computation time between barriers in the background jobs exceeds the length of a time-slice; at the point where $g > 100\mu s$, slowdown relative to explicit coscheduling increases dramatically from less than 10% to more than 60%. However, in our implementation, even when the granularity of the competing jobs is above $g = 100\mu s$, the slowdown of the fine-grain job remains relatively constant. We believe that our modification to the SSC scheduler that ranks the priority of each job over the $Q = 100ms$ interval, as described in Section 9.5.1, is responsible for this improvement.

Continuous-Communication Workloads

For our final fairness experiments, we evaluate the fairness of the two local schedulers on three continuous-communication workloads. In these workloads, we examine the effect of competing jobs communicating at different rates, but synchronizing with the same granularity; in the measured job, the time between reads is held constant at $c = 50\mu s$, while c is varied for the background jobs. We consider workloads with two different values of g ($100ms$ and $1s$); in each case, the jobs in the same workload have the same value of g. In all cases, there is no load-imbalance ($V = 0$), and processes read from remote processes in a `Random` pattern. For the continuous-communication workload with $g = 1s$, we evaluate both two and three background jobs.

The performance shown in Figure 10.6 illustrates that both schedulers perform better than the simulation predictions in Figures 8.20 and Figures 8.21. For example, with the TS scheduler, we predicted slowdowns nearly 2, 3.5, and 8 times worse than ideal explicit coscheduling for the three continuous-communication workloads (*i.e.*, with $g = 100ms$ for three jobs, $g = 1s$ for three jobs, and $g = 1s$ for four jobs). However, in the measurements of our system, the worst-case slowdowns are only 1.3, 1.75, and 2.10 worse than explicit coscheduling. We believe the dramatic difference in these results is due to the fact that we assume a significantly higher context-switch cost of $W = 200\mu s$ in the simulations.

More importantly, the SSC scheduler is surprisingly fair for all measured work-

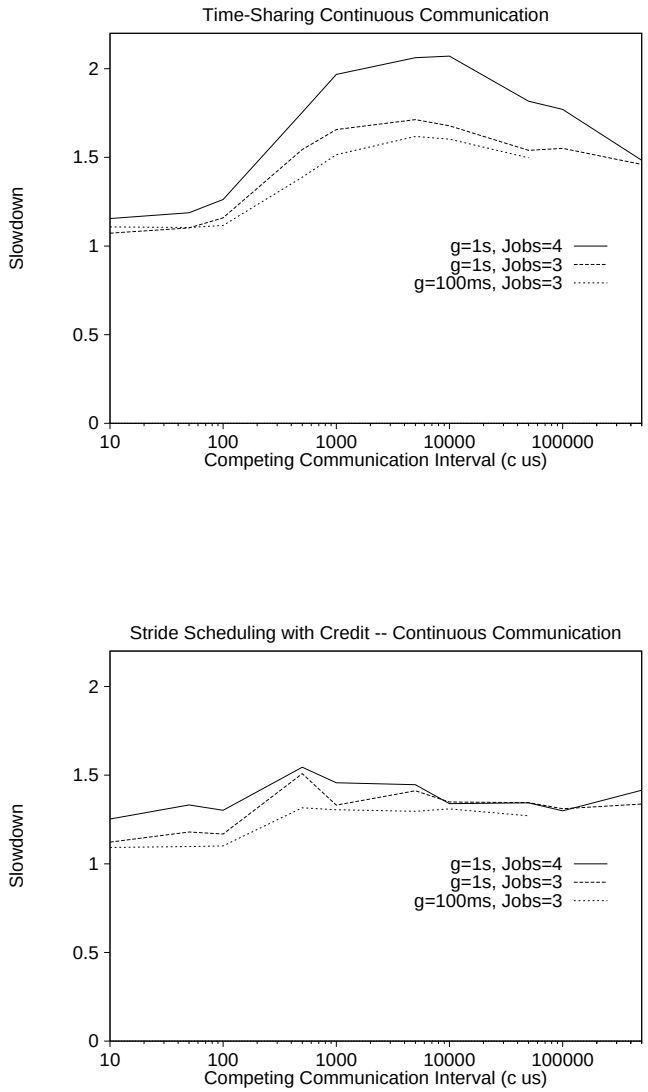


Figure 10.6: **Fairness with Continuous-Communication Programs.** *One measured job and two or three competing background jobs are allocated to 16 workstations. Only the slowdown of the one job is measured; the background jobs run continuously during the experiment. We evaluate continuous-communication workloads where barriers are rarely performed. In each workload, the values of g are identical for all jobs (either $g = 100ms$ or $g = 1s$), 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 or three background jobs. The first graph uses the default Solaris Time-Sharing scheduler on each workstation; the second graph uses our implementation of the Stride Scheduler with System Credit.*

loads. Even with four jobs in the system and processes synchronizing only once a second, the slowdown of the fine-grain measured job is within 55%. This represents a significant improvement relative to the 3.5 times slowdown predicted by the simulations for some continuous-communication workloads. It would be interesting to incorporate the SSC implementation changes into the simulator to confirm that it fixes the problems seen there.

10.2 Range of Workloads

For the remainder of our experiments, we evaluate workloads containing jobs with the same communication characteristics and thus return to the Solaris Time-Sharing scheduler. In this section, we measure workloads that differ significantly from those evaluated in our earlier simulations. We begin by expanding our communication primitives to include one-way requests (*i.e.*, stores) and bulk messages. Next, we evaluate the performance of a set of real Split-C applications. We then stress the scalability of implicit coscheduling, increasing both the number of competing jobs and the number of workstations. Finally, we test the robustness of our approach to different layouts of processes across workstations.

10.2.1 Additional Communication Primitives

Our previous measurements assumed that the only communication operations were short request-response messages, *i.e.*, reads. In our next experiments, we show that one-ways requests (*i.e.*, stores) and bulk request-response messages also perform well with implicit coscheduling.

One-Way Request Messages

In our next experiments, we investigate the performance of applications performing one-way requests, or stores. In these workloads, we again examine three continuous-communication `NEWS` applications competing on 16 workstations; the applications are identical to those in the previous experiments, except for the replacement of store operations for reads.

Note that in these experiments, we use a variation of our performance metric: we calculate the slowdown of the workload with implicit coscheduling relative to the sum of the execution times when each job is run individually with implicit coscheduling (as opposed to run individually with spin-waiting). This difference is required because, for reasons not fully understood, one job containing frequent store operations (*e.g.*, $c = 10\mu s$) run in a dedicated environment performs twice as slowly with spin-waiting as with two-phase waiting.[1] If we were to compare implicit coscheduling to spin-waiting, our performance on fine-grain applications would exhibit a significant speedup.

Figure 10.7 shows our performance on a range of workloads for two different conditional spin times, $S^O_{Cond} = 3W$ and $S^O_{Cond} = W$. In Section 6.3, we showed that

[1] We hypothesize that the improvement results because the version with two-phase waiting forces the sending process to back off when contention occurs at destination processes.

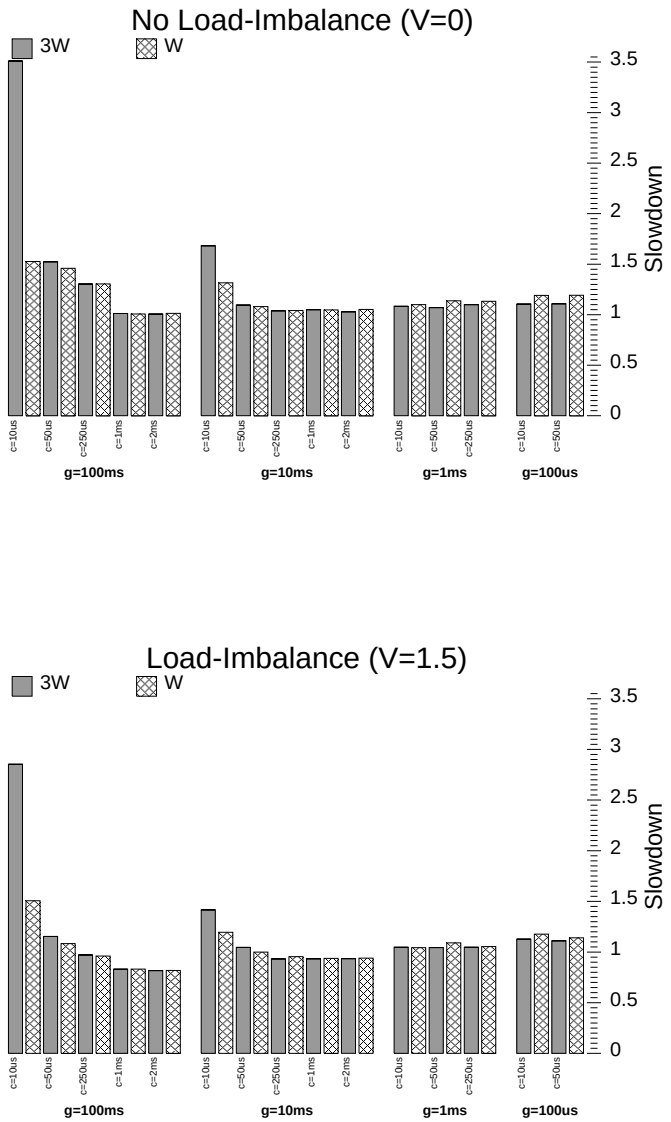


Figure 10.7: **Performance of Continuous-Communication Programs with One-Way Requests.** *Each application communicates with one-way request (store) messages in the* `NEWS` *pattern. The first bar in each set shows the slowdown when processes perform conditional spinning with* $S^O_{Cond} = 3W$*; the second bar in each set assumes* $S^O_{Cond} = W$*. The workload consists of three jobs running on 16 workstations. The time between barriers is varied between* $g = 100\mu s$ *and* $g = 100ms$ *and the time between reads is also varied between* $c = 10\mu s$ *and* $c < g$*. The applications in the first graph have no load-imbalance (*$V = 0$*); in the second graph, the load-imbalance is* $V = 1.5g$*.*

a one-way request message must arrive every W time units to justify having the receiving process spin longer than the baseline amount. However, a request-response message must only arrive every $3W$ units to justify spinning longer, due to the additional cost paid by the requesting process for such operations. Therefore, our previous experiments used $S^O_{Cond} = 3W$.

As Figure 10.7 indicates, when a process conditionally spins for an interval of $3W$ after receiving a one-way request, the performance of frequently communicating workloads suffers dramatically. If the destination process spins for $3W$ after receiving a message, then the process wastes too many cycles when messages are arriving frequently, yet the scheduling of processes is not fully coordinated. For example, with $g = 100ms$ and $c = 10\mu s$, the workload completes 3.5 times slower than with explicit coscheduling.

If a process conditionally spins for only W after receiving a one-way message, then the process correctly trades off the cost of spinning with the cost of being scheduled later to receive a one-way message. Thus, with $S^O_{Cond} = W$, this workload completes only 50% slower than with explicit coscheduling. Once again, workloads that perform more frequent barriers or communicate slightly are not sensitive to the amount of conditional spinning; these workloads perform within 20% of explicit coscheduling regardless of the conditional spin-time.

While these experiments reveal that workloads with one-way request messages perform best with $S^O_{Cond} = W$, the measurements in Section 10.1.2 showed that workloads with request-response messages perform best with $S^R_{Cond} = 3W$. This discrepancy in the best conditional spin-time for different message types is an issue for our implementation because the receiving process does not know the type of the message that it has received and handled; as described in Section 9.3.2, our current interface to the AM-II polling function simply returns the number of messages handled in that invocation. Since one-way messages are an application concept and are not primitives within the AM-II layer, distinguishing between message types must be done at the application layer – perhaps with additional accounting in the Active Message handlers. Therefore, in our prototype implementation, we use the single value of $S_{Cond} = 3W$ for our remaining experiments. However, a complete implementation should distinguish between incoming message types for the best performance on all workloads.

Bulk Messages

Our next experiments briefly evaluate the performance of implicit coscheduling on applications that communicate with bulk messages. These workloads contain three continuous-communication applications that are identical to those in the previous experiments, with the exception that 16KB of data are read from remote processes instead of a single word.

Our implementation reads bulk data from remote nodes by building on the support for medium-sized messages in AM-II. Each medium message can return 8KB of data. To read larger amounts of data, the sending process initiates multiple requests for 8KB, and then waits for all responses before continuing its computation.

Figure 10.8 shows that most workloads requesting bulk messages exhibit a speedup

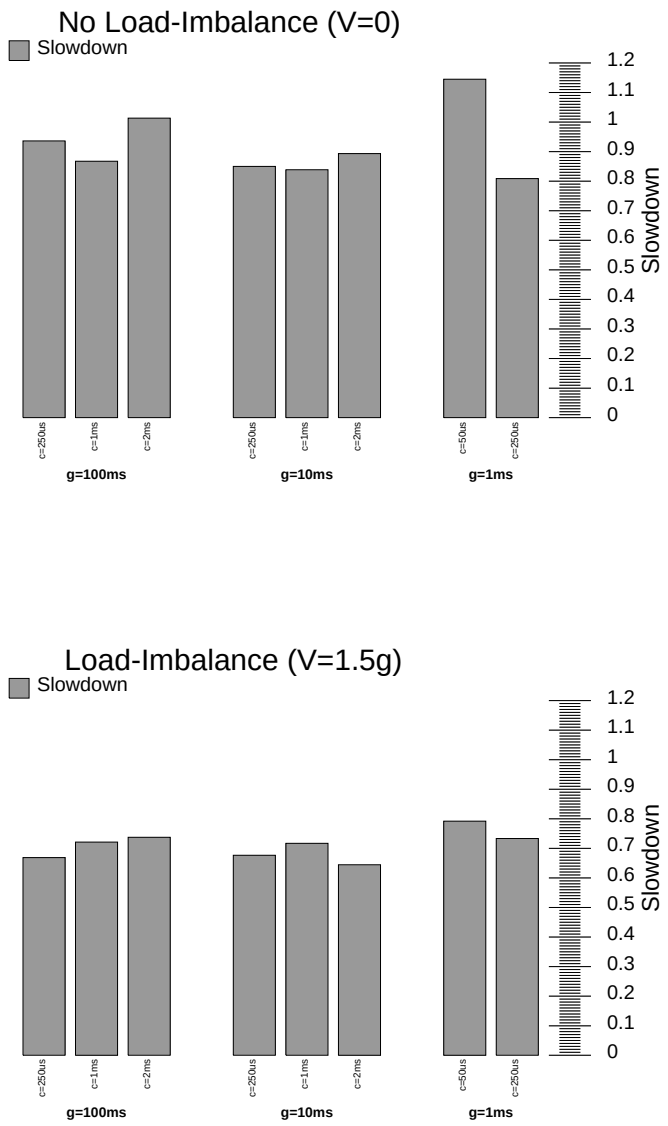


Figure 10.8: **Performance of Continuous-Communication with Bulk Messages.** *Each application repeatedly requests 16KB messages (i.e.,* `reads`*) from neighbors in the* `NEWS` *pattern. The workload consists of three jobs running on 16 workstations. Not as many workloads are shown as for the previous experiments: the time between barriers is varied between $g = 1ms$ and $g = 100ms$ and the time between reads is varied between $c = 50\mu s$ or $c = 250\mu s$ and $c < g$. The applications in the first graph have no load-imbalance ($V = 0$); in the second graph, the load-imbalance is $V = 1.5g$.*

with implicit coscheduling relative to explicit coscheduling. In these experiments, the baseline amount that processes spin while waiting for the bulk response is identical to that for short request-response messages (*i.e.*, $S^R_{Base} = 130\mu s$). Because bulk messages require more time than short messages to complete, processes relinquish the processor before the bulk read is completed.

Our measurements show that implicit coscheduling can achieve superior performance to explicit coscheduling for applications with bulk messages of 16KB. By waiting only a small baseline amount before relinquishing the processor, even workloads with no load-imbalance can finish in 80% of the time required with explicit coscheduling. Just as processes should relinquish the processor in the presence of high network latency or load-imbalance, processes should also relinquish the processor when waiting for bulk messages. These preliminary measurements indicate that processes should not wait for messages longer than 16KB.

For workloads with bulk messages shorter than 16KB or with a mix of short and long messages, it is likely that processes should remain coordinated. Remaining coordinated through the bulk request-response message requires an adjustment of the baseline spin amount to reflect the expected completion time of the longer transfer. We leave the investigation of the message size at which processes should wait for the bulk transfer to complete rather than relinquish the processor as another area for future work.

10.2.2 Real Applications

Analyzing synthetic applications exposes some of the strengths and weaknesses of our implementation in a controlled environment. However, the synthetic programs do not change behavior over time and do not mix different communication styles. To evaluate a more varied set of applications, we examine the real Split-C applications described in Section 9.4.

The performance of these seven Split-C applications with baseline and conditional spinning is shown in Figure 10.9. In these experiments, three copies of the specified application are run on 16 workstations. For all applications, performance is within 30% of ideal explicit coscheduling. A few of the applications exhibit significant speedups relative to explicit coscheduling due to their use of bulk messages: `mm` and `fft`.

The two applications, `radix:small` and `emd3d:small`, that perform the worst with implicit coscheduling rarely synchronize yet communicate frequently with random destinations, matching the worst-case continuous-communication synthetic applications. For example, as shown in Figure 9.7, `radix:small` has almost no barriers and usually computes for less than $c = 30\mu s$ between store operations; as shown in Figure 9.8, `em3d:small` has about 2.5 seconds between barriers, yet computes only about $c = 10\mu s$ between reads. We note that both of these applications were specifically written to stress communication performance and are known to not scale as well as their optimized counterparts with bulk messages in a dedicated environment [3, 39]. As expected for continuous-communication applications, conditional spinning significantly improves performance relative to baseline spinning. For example, with baseline spinning, `em3d:small` runs nearly 80% slower than explicit coscheduling; with conditional spinning it improves to within 30%.

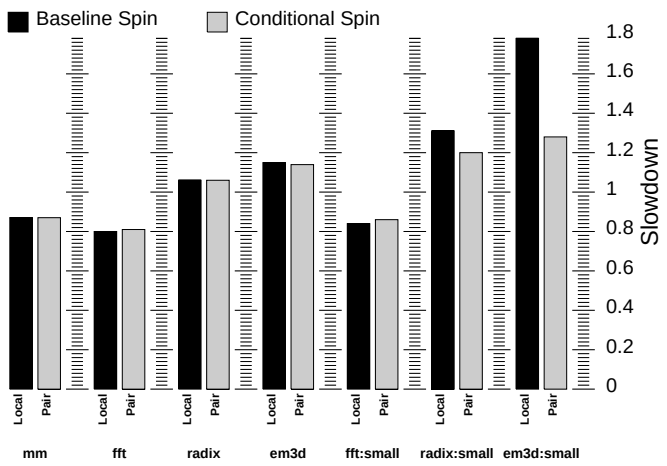


Figure 10.9: **Performance of Real Split-C Applications.** *Three copies of each Split-C application are run on 16 workstations. The first four applications use bulk messages, while the last three use small messages.* `mm` *is matrix multiply;* `fft` *is a fast Fourier transform;* `radix` *is a radix sort;* `em3d` *models the propagation of electro-magnetic waves in three dimensions.*

We believe that the performance of `radix:small` and `em3d:small` could both be improved in two ways. First, both applications perform both read and store operations. However, both applications use the same conditional spin amount as the applications built exclusively on reads (*i.e.*, $S_{Cond} = 3W$). Using the desired conditional spin-time of $S_{Cond} = W$ could improve these applications, as demonstrated in Section 10.2.1.

Second, our implementation of Split-C can slightly degrade the performance of applications sending many one-way messages to random destinations in a dedicated environment. As described in Section 9.3.1, to avoid spinning in AM-II, the Split-C run-time layer sends messages in bursts of no more than sixteen before waiting for all to return. Thus, store operations must sometimes wait on an acknowledgment with implicit coscheduling. As a result, running just a single copy of `em3d:small` with this constraint results in slowdowns of 10% compared to the unmodified version of Split-C. Leveraging the new non-blocking interfaces to AM-II layer is expected to improve these workloads.

Since it appears that implicit coscheduling can more than adequately handle real applications with a distribution of message sizes, communication patterns, and communication intervals, in our remaining experiments we return to synthetic applications to more thoroughly stress the performance of implicit coscheduling in a controlled manner.

10.2.3 Job Scalability

Our next experiments investigate the performance of implicit coscheduling as the number of competing jobs in the system is increased. We examine two bulk-synchronous and two continuous-communication workloads as we increase the number of competing jobs from one to seven. We stop at seven jobs because this is the number of communicating processes

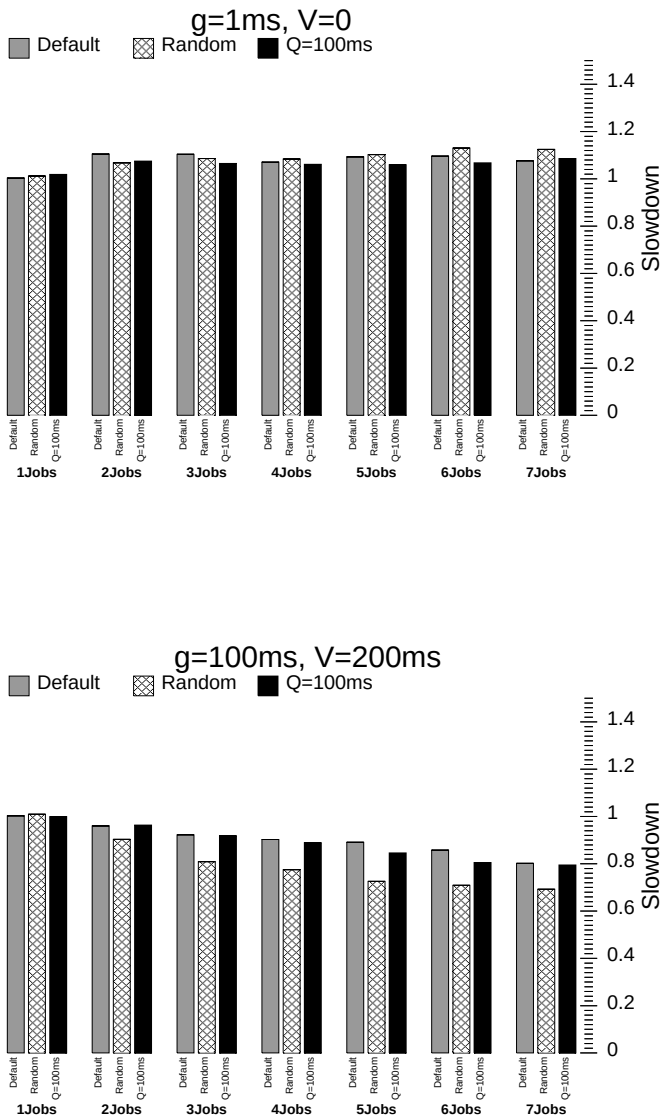


Figure 10.10: **Job Scalability with Bulk-Synchronous Programs.** *The number of jobs in the system is increased from one to seven, each running on 16 workstations. Within each graph we investigate three slight variations on the placement of jobs and on the local operating system scheduler. In the* `Default` *case, processes are placed with an identical layout across nodes. In the* `Random` *case, processes are randomly placed on different nodes. Finally, with* `Q=100ms`*, processes are placed identically across workstations as in the* `Default` *case, but the local scheduler is modified to use $Q = 100ms$ time-slices at all priority levels. Each workload reads from remote processors in the* `NEWS` *pattern. The first bulk-synchronous workload represents medium-grain applications ($g = 1ms$) with no load-imbalance ($V = 0$); the second bulk-synchronous workload represents coarse-grain applications ($g = 100ms$) with a large amount of load-imbalance ($V = 200ms$).*

the AM-II layer can simultaneously support without paging communication endpoints to and from the network interface card.

Bulk-Synchronous Workloads

Figure 10.10 shows the performance of implicit coscheduling with conditional spinning on bulk-synchronous workloads. The first bulk-synchronous workload represents medium-grain applications ($g = 1ms$) with no load-imbalance ($V = 0$); the second bulk-synchronous workload represents coarse-grain applications ($g = 100ms$) with a large amount of load-imbalance ($V = 200ms$). All applications communicate in the `NEWS` pattern.

Within each graph we investigate three slight variations on the placement of jobs and on the local operating system scheduler. In the `Default` case, processes are placed with an identical layout across nodes; that is, process 0 from each job is placed on node 0, and so on through process 15 and node 15. In the `Random` case, processes are randomly placed on different nodes. Finally, with `Q=100ms`, processes are placed identically across workstations as in the `Default` case, but the local scheduler is modified slightly to use $Q = 100ms$ time-slices at all priority levels.

As shown in the first graph, the medium-grain bulk-synchronous workload with no load-imbalance has very stable performance as the number of competing jobs is increased. As predicted by the simulation results across Figures 8.18 and 8.19, when the layout of processes is identical across jobs, performance is slightly better than when processes are placed randomly, due to the tendency of the root process to direct the scheduling of the entire job. However, this effect is much smaller in our implementation. In addition, this workload has little sensitivity to the time-slice used by the local scheduler.

However, as shown in the second graph, the performance of the coarse-grain workload with high load-imbalance actually increases relative to explicit coscheduling as the number of jobs is increased. For workloads with high load-imbalance, better system throughput can be achieved when processes relinquish the processor when waiting at barriers and allow a competing process to perform useful work, than when processes wastefully spin-wait with coordinated explicit coscheduling. 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.

Perhaps more surprisingly, performance for the highly load-imbalanced workload improves when processes are randomly placed across workstations. When the root processes from every job are colocated on the same workstation, this workstation performs more work than the other workstations and becomes a system bottleneck. Thus, when the root processes are allocated to different workstations, workloads with high load-imbalance achieve better performance. Finally, we note that this workload also has little sensitivity to the time-slice used by the local scheduler.

Continuous-Communication Workloads

We also investigate the scalability of two continuous-communication workloads. In both workloads, barriers are rarely performed (*i.e.*, $g = 100ms$), there is no load-imbalance,

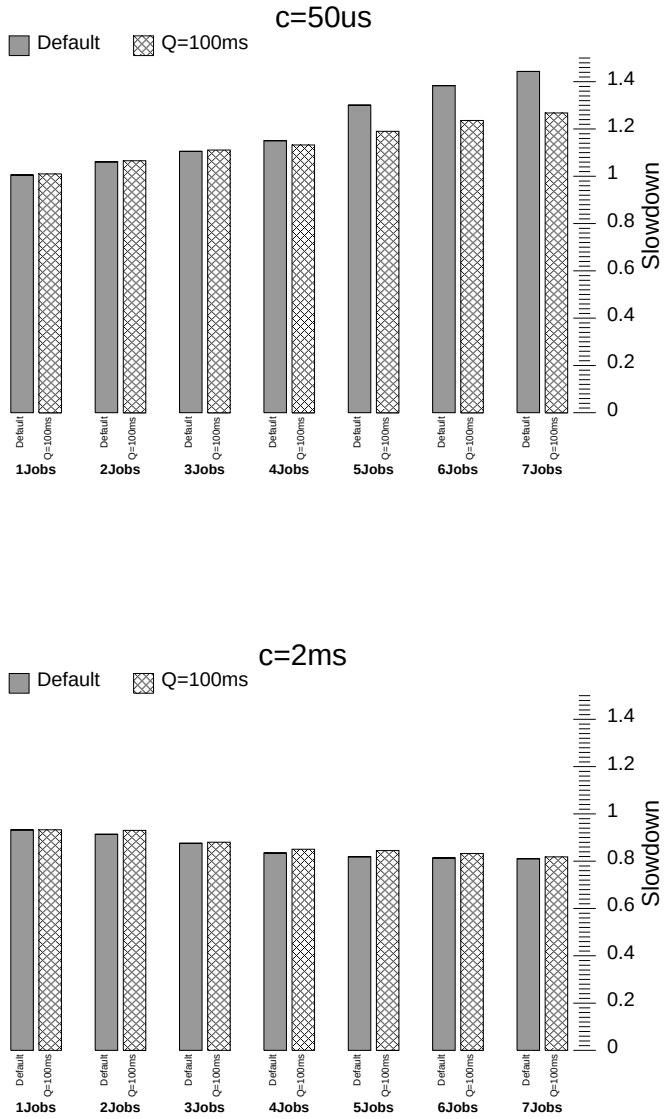


Figure 10.11: **Job Scalability with Continuous-Communication Programs.** *The number of jobs in the system is increased from one to seven. We investigate only the case where processes are allocated identically across jobs. We compare the performance of the* `Default` *Solaris Time-sharing scheduler to one modified to use* `Q=100`ms *time-slices for all priorities. In both workloads, barriers are rarely performed (i.e.,* $g = 100ms$*), there is no load-imbalance, and processes read from remote processes in the* `Random` *pattern. In the fine-grain workload on the left, processes communicate every* $c = 50\mu s$*; in the medium-grain workload on the right, processes communicate every* $c = 2ms$*.*

and processes read from remote processes in the `Random` pattern. In the fine-grain workload, processes communicate every $c = 50\mu s$; in the medium-grain workload, processes communicate every $c = 2ms$. Since these workloads perform few barriers, performance is insensitive to the layout of processes across workstations and we investigate only the case where processes are allocated identically across jobs. However, we do compare the performance of the default Solaris Time-Sharing (TS) scheduler (marked `Default`) to one modified to use $Q = 100ms$ time-slices for all priorities (marked `Q=100`ms).

The first graph in Figure 10.11 shows that the performance of the fine-grain continuous-communication workload is sensitive to the number of competing jobs and the scheduling policy. As the number of jobs is increased from three to seven, performance degrades from 10% to 45% worse than our ideal model of explicit coscheduling. Due to the sensitivity of this workload to whether communicating processes are coordinated, performance is sensitive to the lengths of the time-slices. Since time-slices in the Solaris TS scheduler vary from $Q = 20ms$ at the high priorities to $Q = 200ms$, the more time a process spends at high priorities, the shorter its time-slice and the more time is wasted achieving coordination. Because the priority of every process is raised once a second and lowered only when the process completes a time-slice, if a process is in competition with more processes it will complete fewer time-slices in the one second interval. Therefore, with more jobs in the system, processes spend more time executing at high priorities and receive shorter time-slices.

Setting the length of all time-slices to $Q = 100ms$ improves the performance of five or more competing jobs. With this small modification, implicit coscheduling performs within 30% of explicit coscheduling for up to seven jobs. Changing the local scheduler to use the SSC scheduler with time-slices of $Q = 100ms$ is expected to have similar performance.

The second graph which shows the behavior of the medium-grain continuous-communication workload is much more surprising. In Figure 10.4 we noted that a workload containing three applications performing infrequent barriers and infrequent communication (*i.e.*, $c = 2ms$) exhibits a speedup relative to explicit coscheduling. Since cooperating processes do not communicate with one another at precisely the same time and messages are only handled when the destination process touches the network, a sending process that is slightly slower than its destination will have to wait until the next time the destination process communicates for the request to be serviced. With such high waiting times at read operations, processes achieve better performance by relinquishing the processor after the baseline spin amount. With three competing jobs, the improvement was small relative to explicit coscheduling. However, as the number of jobs is increased, this effect increases in magnitude.

10.2.4 Workstation Scalability

The previous implementation measurements considered a cluster of 16 workstations. 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: two continuous-communication applications (*i.e.*, $g = 100ms$ and $V = 0$ with the `Random` communication pattern and

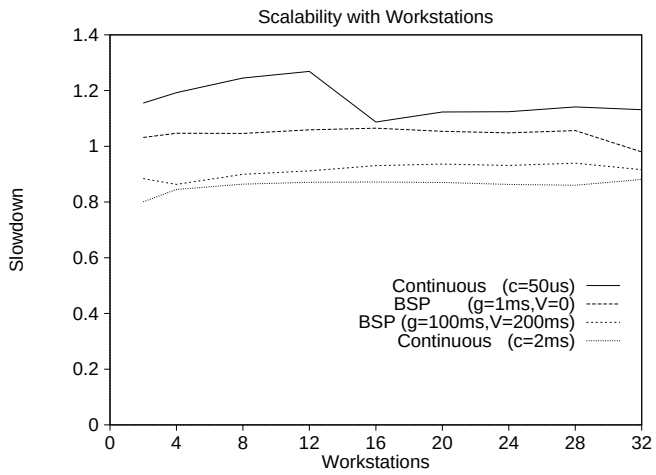


Figure 10.12: **Workstation Scalability.** *Three competing jobs are allocated to an increasing number of workstations. We evaluate two continuous-communication and two bulk-synchronous workloads. In both continuous-communication workloads, barriers are rarely performed (i.e., $g = 100ms$), there is no load-imbalance, and processes read from remote processes in the* `Random` *pattern. In the fine-grain workload, processes communicate every $c = 50\mu s$; in the medium-grain workload, processes communicate every $c = 2ms$. The bulk-synchronous workloads read from remote processors in the* `NEWS` *pattern. The first bulk-synchronous workload represents medium-grain applications ($g = 1ms$) with no load-imbalance ($V = 0$); the second bulk-synchronous workload represents coarse-grain applications ($g = 100ms$) with a large amount of load-imbalance ($V = 200ms$).*

either $c = 50\mu s$ or $c = 2ms$) and two bulk-synchronous applications (*i.e.*, $g = 1ms$ with $V = 0$ and $g = 100ms$ with $V = 200ms$, both with the `NEWS` pattern).

Figure 10.12 shows the performance of these four workloads as a function of the number of workstations in the cluster. As the size of the cluster increases, the baseline spin for barriers increases as well, as shown in Figure 9.4. 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 explicit coscheduling. 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 more beneficial. All in all, these results are very promising for larger configurations.

The main caveat with our workstation scalability results is that, as described in Section 9.3.2, barriers are implemented in a linear fashion. In a dedicated environment, hierarchical-tree barriers become more efficient than linear barriers for more than eight workstations. While we have shown that implicit coscheduling performs very well with linear barriers, we have not yet evaluated the impact of tree barriers. Tree barriers may not be as beneficial in our environment relative to explicit coscheduling if context-switches are triggered on all intermediate nodes. As a result, a thorough study of the scalability of implicit coscheduling should investigate the impact of tree barriers.

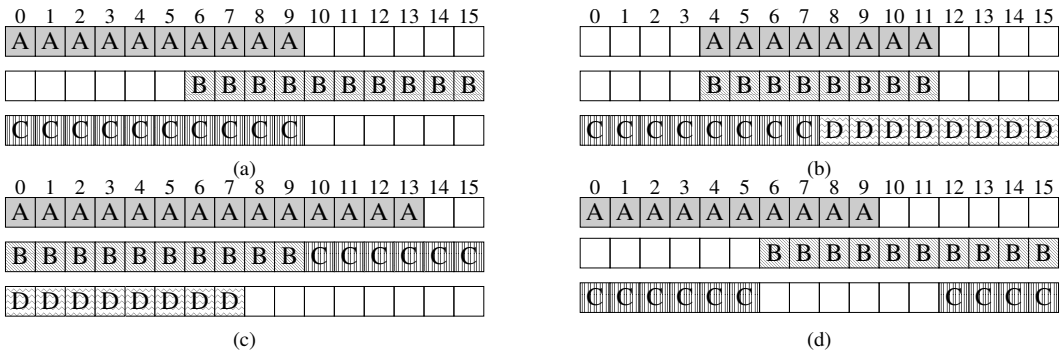
10.2.5 Job Placement

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

The three workloads that we examine (coarse, medium, and fine) consist entirely of bulk-synchronous applications. The coarse-grain workload consists of applications synchronizing every $g = 100ms$ with $V = 200ms$; the medium-grain workload contains applications synchronizing every $g = 10ms$ with $V = 5ms$; finally, the fine-grain workload contains applications synchronizing every $g = 100\mu s$ with $V = 0$. All workloads read in a `NEWS` pattern with $c = 8\mu s$ of intervening computation.

Figure 10.13 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* are constructed to 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 less jobs are idle with implicit coscheduling for the same time as an explicit coscheduling implementation. Thus, medium and fine-grain jobs perform similarly to explicit coscheduling, and the



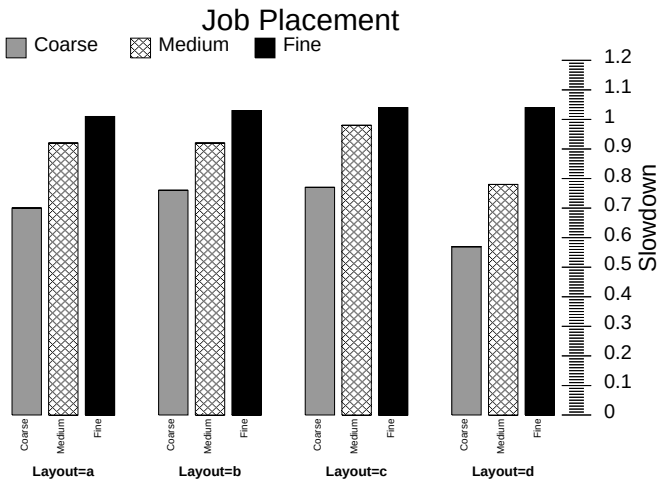


Figure 10.13: **Sensitivity to Job Placement.** *Three or four jobs, A, B, C, and, optionally, D are placed across 16 workstations. The figures designate how the processes would be scheduled over time with explicit coscheduling. For each of the four layouts, we investigate three workloads: coarse, medium, and fine. The coarse-grain workload consists of applications synchronizing every* $g = 100ms$ *with* $V = 200ms$*; the medium-grain workload contains applications synchronizing every* $g = 10ms$ *with* $V = 5ms$*; finally, the fine-grain workload contains applications synchronizing every* $g = 100\mu s$ *with* $V = 0$*. All workloads read in a* `NEWS` *pattern with* $c = 8\mu s$ *of intervening computation.*

coarse-grain jobs achieve a speedup, as expected.

Layout d is designed to show that implicit coscheduling automatically adjusts the execution of jobs to fill available time on each workstation. In this layout, three coscheduling rows are required for each of the parallel jobs to be scheduled simultaneously, yet the load on each machine is at most two. For jobs with coarse-grain communication dependencies, implicit coscheduling achieves an additional speedup relative to explicit coscheduling because processes can run at any time, not only in the predefined slots of the scheduling matrix. Obtaining this benefit with explicit coscheduling is only possible if the appropriate alternate job is carefully chosen to run when a machine is idle. Thus, with layout d, the coarse-grain workload with implicit coscheduling requires only 55% of the execution time as with explicit coscheduling.

10.3 Summary

In this chapter, we have measured the performance of our implementation of implicit coscheduling on a range of applications. From our measurements we make two contributions beyond that of simply understanding the absolute performance of our implementation. First, we can compare our results to those predicted by the simulations and understand the importance of the factors that we did not model in the simulations. Second, by analyzing which workloads perform well with implicit coscheduling and which do not, we can recommend a programming style to application developers. Therefore, to conclude this chapter, we first summarize our measured performance, then compare this performance to that predicted by the simulations, and finally, advise programmers on how to achieve the best performance with implicit coscheduling.

10.3.1 Implementation Performance

In most of the experiments in this chapter, we examined workloads containing three competing parallel applications, all with the same communication characteristics, running on 16 workstations with the default Solaris Time-Sharing (TS) scheduler. On synthetic applications, we found that spinning the baseline and conditional amounts specified in Table 9.2 resulted in the best implicit coscheduling performance. Bulk-synchronous programs proved to perform very well in our environment: performance on our measured workloads is always within 15% of ideal explicit coscheduling and sometimes better than explicit coscheduling. Continuous-communication applications can be more problematic. Although most perform within 30% of explicit coscheduling, a few extreme cases of very frequent communication ($c = 10\mu s$) coupled with large load-imbalance ($V > 150ms$) perform almost 80% worse than explicit coscheduling; however, we do not believe such workloads will occur often in practice.

We also investigated several types of applications that we did not examine in simulations. We showed that applications containing one-way request messages perform well in our environment. We found that for workloads with very frequent communication and infrequent barriers, conditional spinning should differentiate between message types (*i.e.*, spin only $S^O_{Cond} = W$ for a one-way message arrival, rather than $S^R_{Cond} = 3W$). We also

measured the performance of synthetic applications communicating with bulk messages and saw that processes requesting 16KB of data achieve speedups relative to explicit coscheduling. We also measured seven Split-C applications with a variety of characteristics and saw that the worst-case performance is within 20% of explicit coscheduling.

In other experiments, we studied the scalability of implicit coscheduling and its ability to handle workloads where some workstations are allocated more processes than others. We found that as the number of competing jobs increases up to seven per workstation, the performance of workloads that are difficult to coordinate can be improved by using a constant time-slice of $100ms$ in the TS scheduler for all priority levels. Our measurements showed no degradation as the size of the cluster was increased to 32 workstations. Finally, performance is robust even when the number of processes allocated to each workstation is not identical throughout the system.

Finally, our experiments show that our implementation of stride scheduling with system credit (SSC) is fair for a wide range of workloads. Even when a fine-grain job competes against three other jobs communicating several orders of magnitude less frequently, the fine-grain job exhibits only a 55% slowdown. This is a substantial improvement compared to the default TS scheduler which exhibits a slowdown of 2.1. The modified version of SSC in our implementation is fair even when some jobs compute between barriers for a greater interval than the scheduling time-slice, unlike the version in the simulator.

10.3.2 Comparison to Simulation Predictions

In general, our implementation results closely match the performance predicted by the simulations with the context-switch cost on message arrival set to $W = 50\mu s$. The primary differences in the two environments are: the mechanism for message-arrival notification and the presence of communication overhead and gap. We examine these two differences in turn.

In the simulations, a process is notified of a message arrival with an asynchronous interrupt. In our implementation, a process is notified only when it touches the network: either by sending a message or by explicitly polling the network with `AM_Poll`. Thus, to a remote process waiting for a response, a process that is ignoring the network appears as if it were not scheduled. Given a dedicated environment, an exclusive dependency on polling is known to harm the performance of applications that ignore the network for prolonged periods of time [23, 94, 113]. Interestingly, implicit coscheduling with two-phase waiting performs better for such programs than explicit coscheduling with spin-waiting. When a sending processes must wait a significant amount of time for the destination process to touch the network and return the response message, the process can achieve better performance by relinquishing the processor and allowing another process to be scheduled.

The presence of communication overhead, o, and gap, g, in the implementation make barriers costly. Given our implementation of linear barriers and the relatively low context-switch cost, the need to stay coordinated while waiting at a barrier is less than predicted. As a result, bulk-synchronous applications should only spin the minimum baseline amount at barriers before relinquishing the processor; for such workloads, there is no need to approximate load-imbalance. However, for applications that continuously-communicate

even while some processes wait at barriers, approximating load-imbalance may still be worthwhile. Finally, for clusters with more than 16 workstations, a tree barrier is superior and should be used; investigating the performance of implicit coscheduling with a different barrier implementation is reserved for future work. In retrospect, modeling o in our simulator would have been worth the additional programming complexity and increase in execution time.

10.3.3 Advice for Programmers

Our experience with a wide range of workloads has led us to understand which types of applications behave well with implicit coscheduling. To achieve the best performance with implicit coscheduling, applications should be written with the following characteristics.

1. **Infrequent Communication:** If processes communicate infrequently with one another, then their scheduling can proceed relatively independently of one another across nodes. Independent scheduling is especially advantageous in situations where the load across workstations is uneven, whether due to load-imbalance internal to the application or to irregular placement of processes across workstations.

2. **Frequent Barriers:** If processes must communicate frequently, then frequent barriers should be used to keep the scheduling of all processes coordinated. During an intense communication phase, there should be little load-imbalance. Ideally, barriers should precede all communication phases, as in our bulk-synchronous workloads.

3. **All-to-all Communication:** Applications in which all processes communicate with one another (*e.g.*, the `Transpose` and `Random` communication patterns) stay coordinated as a unit more effectively than processes which communicate in subgroups (*e.g.*, the `NEWS` pattern).

4. **Bulk Messages:** Most applications that communicate with bulk messages exhibit speedups relative to explicit coscheduling. Because processes relinquish the processor, rather than wait for slow responses to arrive, other processes can perform useful computation.

Many of these qualities match those that programmers strive to obtain to achieve good speedups in a dedicated environment. For example, barriers before communication phases are known to be beneficial [24, 48] as are bulk messages to amortize communication costs [3]. Therefore, many applications are expected to adhere to these principles even when not specifically targeted for an implicit coscheduling environment. These well-behaved applications can expect to see slowdowns with implicit coscheduling within 15% of explicit coscheduling.

Chapter 11

Conclusions

In this thesis, we have shown that implicit information can greatly simplify the task of building cooperative system services in a distributed environment. In an implicitly-controlled system, components infer remote state by observing naturally-occurring local events, rather than explicitly contacting other components to obtain information. Implicit information is particularly useful when explicit interfaces do not provide the desired information.

We have shown that implicit coscheduling is an effective method for scheduling communicating processes in a coordinated manner. In this chapter, we summarize the implicit coscheduling approach and our simulation and implementation results; we then describe extensions to implicit coscheduling that we did not have the opportunity to investigate.

11.1 Summary

This section summarizes implicit coscheduling, our approach to scheduling communicating processes in a distributed system, such as a network of workstations. With implicit coscheduling, no explicit messages are required to organize the coordination of processes across machines. 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 waiting algorithm processes employ when dependent on remote processes and the behavior of the local operating system scheduler. Implicit coscheduling requires that processes employ a conditional two-phase waiting algorithm with the correct baseline and conditional spin-times, and a preemptive, fair local scheduler. We discuss each of these two components in more detail.

11.1.1 Conditional Two-Phase Waiting

In this thesis, we have introduced conditional two-phase waiting, a generalization of two-phase waiting. With basic two-phase waiting, processes spin-wait for some amount of time while waiting for the desired event to occur; if the event does not occur, then

the process voluntarily relinquishes the processor and blocks. The amount of spin-time may vary dynamically from operation to operation, but is a predefined amount before each waiting operation begins. In conditional two-phase waiting, processes begin by spin-waiting a predefined baseline amount, but may then conditionally spin longer depending upon events that are observed while spin-waiting.

Picking the baseline and conditional spin amounts is critical to achieving the best implicit coscheduling 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. Determining the correct baseline and conditional spin amounts is relatively simple given a few heuristics.

Successful implicit coscheduling requires that all levels of an application (*e.g.*, the user-level code, the run-time library of the parallel language, and the message-layer) apply conditional two-phase waiting whenever dependent on a result generated from a remote process. The parallel language run-time library is the appropriate layer to implement the waiting algorithm because its performance determines the time for communication operations and it contains semantic information about the relationship between requests and responses within higher-level primitives, such as reads and barriers.

11.1.2 Local Operating System Scheduler

By choosing whether to spin-wait or relinquish the processor 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 and accurate 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. In this thesis, we described an extension to a stride scheduler that gives compensation tickets to processes that relinquish the processor.

11.1.3 Performance

In this dissertation, 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 most experiments, we examined workloads containing three competing synthetic applications: both bulk-synchronous and continuously-communicating jobs. Across workloads, we varied three important application parameters: the rate of communication, the rate of synchronization, and the amount of load-imbalance

internal to the application. In the simulations, we also evaluated the impact of network latency and the cost of a context-switch in the local operating system scheduler.

The primary differences across the simulation and implementation environments are the mechanism for message-arrival notification and the presence of communication overhead and gap. In the simulations, a process is notified of a message arrival with an asynchronous interrupt, while in our implementation, a process is notified only when it explicitly interacts with the network. The presence of communication overhead and gap in the implementation increased the cost of barriers and reduced the benefit of coordinated scheduling.

Our simulation and implementation 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. However, 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 matched the spin times derived in our analysis.

Bulk-synchronous programs perform well in our environment: performance on our measured workloads is always within 15% of ideal explicit coscheduling and sometimes better than explicit coscheduling. Continuously-communicating applications can be more problematic. Although most perform within 30% of explicit coscheduling, a few extreme cases of very frequent communication coupled with large internal load-imbalance perform almost 80% worse than desired; however, we do not believe such workloads will occur often in practice.

In our implementation, we demonstrated that applications communicating with bulk messages (16KB) exhibit modest speedups relative to explicit coscheduling. We also measured seven Split-C applications with a wide variety of characteristics and saw that the worst-case performance is within 20% of ideal explicit coscheduling. In other experiments, we studied the scalability of implicit coscheduling and its ability to handle workloads where some workstations are allocated more processes that others. We observed no degradation as the size of the cluster was increased to 32 workstations. Performance is robust even when the number of processes allocated to each workstation is unbalanced.

Finally, our simulation and implementation experiments show that our extension of stride scheduling with system credit is fair for a wide range of workloads. Even when a fine-grain job competes against other jobs communicating several orders of magnitude less frequently, the fine-grain job exhibits a worst-case slowdown within 55% of explicit coscheduling. This is a substantial improvement compared to the default Solaris Time-Sharing scheduler which exhibits a slowdown of 210%.

The types of applications that perform well with implicit coscheduling are the same that achieve good speedups in a dedicated environment; for example, barriers before communication phases are known to be advantageous [24, 48], as are bulk messages to amortize communication costs [3]. Therefore, many applications are expected to behave well with implicit coscheduling, even when not specifically targeted for our environment. In summary, these well-behaved applications can expect to see slowdowns with implicit coscheduling within 15% of ideal explicit coscheduling.

11.2 Future Work

This thesis has raised a number of new questions for scheduling workloads in a distributed environment. In this section, we describe some of the extensions to implicit coscheduling and the additional programming models and workloads we feel would be particularly interesting to evaluate.

11.2.1 Implementation Issues

We begin by briefly discussing the areas within our implementation that could be developed further.

Approximating Load-Imbalance

Measurements of our implementation revealed that bulk-synchronous applications can ignore load-imbalance within the application: acceptable performance is achieved when processes block after spinning only a small time at the barrier. However, in continuously-communicating applications, processes that arrive at the barrier early should remain scheduled to handle incoming messages; while conditional spinning greatly improves performance, it is not entirely adequate.

In our initial simulations and implementation measurements, we did not investigate continuously-communicating applications and, therefore, did not fully realize the importance of predicting load-imbalance. As a result, our implementation does not contain sufficient techniques for predicting load-imbalance from past measurements. We expect that spinning for the expected load-imbalance of the barrier would help continuously-communicating programs to stay coordinated and would improve performance.

Hierarchical Barriers

As discussed in Section 9.3.2, the barriers in our version of the Split-C run-time layer are constructed with a linear notification of processes. In a dedicated environment, hierarchical-tree barriers are more efficient than linear barriers for more than eight workstations. However, if context-switches occur on all of the intermediate nodes of the tree with implicit coscheduling, then a linear barrier may be advantageous. A thorough study of the scalability of implicit coscheduling should investigate the impact of tree barriers.

One-way Messages

As described in Section 10.2.1, our implementation does not deal with one-way messages appropriately in two circumstances. First, our conditional waiting algorithm in Split-C does not distinguish between types of arriving message (*i.e.*, request-response messages or one-way messages). As a result, the waiting process cannot accurately calculate the necessary arrival rate to justify the cost of spin-waiting. Second, the Split-C run-time layer does not leverage the non-blocking interface of AM-II; thus, the number of messages that the application can have outstanding may be artificially limited, reducing performance.

11.2.2 Programming Model

In this section we discuss possible extensions to implicit coscheduling to handle applications communicating with bulk messages and those written in a message-passing language.

Bulk Messages

Our measurements of implicit coscheduling on applications communicating with bulk messages showed that transferring a 16KB message took long enough that it was beneficial to relinquish the processor before the operation completed; as a result, implicit coscheduling exhibits a modest speedup relative to explicit coscheduling. Nevertheless, coordinated scheduling is still desired for somewhat smaller bulk messages.

Our analysis of the appropriate spin-time in the conditional two-phase waiting algorithm focused exclusively on short messages. Extending our analysis to consider bulk messages entails predicting the expected round-trip time of the longer message when the destination process is scheduled. This calculation is not difficult given a model of network bandwidth ($1/g$) and knowledge of the size of the message, N. Since the length of the message is known before it is sent, this calculation is more straight-forward than determining spin-time at barriers, which requires the load-imbalance to be predicted from past behavior.

Just as processes should prematurely relinquish the processor when waiting time is high due to network latency or internal load-imbalance, processes should relinquish the processor when waiting for long messages. The largest message size that processes should wait for, N_{Block}, depends on the bandwidth of the network and the cost of losing scheduling coordination for future communication events.

Message-Passing Applications

Applications written with a message-passing library, such as MPI [161] or PVM [157], represent a substantial segment of parallel programs. Supporting such applications is an important requirement of scheduling and allocation policies in clusters. In some ways, such applications are easier to support than the Split-C applications we evaluated, and in some ways are more difficult.

Most message-passing applications are optimized to transfer data in large messages, thus amortizing the relatively high overheads and startup costs 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 operation must wait for a corresponding receive operation to be posted on the remote process, then the time for the send includes not only the time to transfer the data, but also the time spent waiting for the remote process to post the receive. Both of these components must be incorporated into the baseline spin amount of the waiting algorithm, complicating the analysis. Predicting

the synchronization time is similar to predicting the amount of load-imbalance at barrier operations and is subject to errors. Adapting our prediction algorithms to send-receive operations would be an interesting and non-trivial area to investigate.

11.2.3 Workloads

In this work, we have measured the performance of implicit coscheduling on only traditional parallel applications, yet we have often argued that our approach provides better support for general-purpose workloads than explicit coscheduling. We now discuss our reasoning for a few of the specific job classes.

Client-Server Applications

Client-server applications are not supported by explicit coscheduling because it is not known *a priori* which processes communicate with each other. However, implicit coscheduling should be able to adapt without difficulty to client-server applications, where both the set of clients contacting a server as well as the servers contacted by a particular client change over time; with implicit coscheduling, communicating processes dynamically adapt to each other's scheduling behavior.

We expect there to be two new issues with a client-server workload. 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); this was described in Section 6.3.1. 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.

I/O-intensive Applications

Explicit coscheduling requires that communicating processes remain scheduled for their time-slice even when waiting for I/O to complete. The problem with this approach is that, unless the waiting process is actively handling messages from other processes, no useful work is being performed in this interval. For I/O-intensive applications, the amount of wasted resources may be significant [95, 144].

Conversely, the natural behavior of implicit coscheduling would be to perform conditional two-phase waiting when processes are dependent on I/O results. With this approach, processes would remain scheduled during I/O activity when receiving messages from communicating processes, but could 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.

Interactive Applications

Explicit coscheduling cannot simultaneously handle parallel and interactive jobs. If the interactive job is given a slot in the coscheduling matrix, parallel jobs are not severely impacted, but the interactive job receives poor response time. If the interactive job is instead allowed to interrupt the parallel job in the matrix when necessary, then the interactive job performs well, but the parallel job suffers [7].

Mixing interactive applications and compute-bound jobs is difficult enough in the sequential world; compromising between response-time and throughput while maintaining fairness is still an active area of research [60, 69, 74, 83, 87, 131, 155, 158, 170]. Since sharing the cluster dynamically between parallel and interactive applications is likely to give the best utilization [45, 128], it is important to solve this problem in the presence of parallel jobs, even though this further complicates the issues.

From the perspective of the interactive job, the ideal approach is to promptly interrupt the communicating job whenever the interactive job has work to perform. From the perspective of the parallel job, interactive jobs should only run when the parallel job is uncoordinated or not communicating. Whether or not implicit coscheduling can handle such a problem is an open question: the current fairness mechanisms in our local scheduler may allow jobs that have accumulated little CPU time to preempt the scheduled (communicating) job too readily, disrupting its coordination.

A fair and efficient implementation may require more sophisticated interactions between the local operating system scheduler and the two-phase waiting algorithm. There are two interesting approaches that we think are worth investigating. First, the scheduling of interactive jobs could be postponed by an amount small enough to not noticeably impact user response time; 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 only preempt a parallel job immediately if computing for only a very short time (on the order of a few context-switches); the baseline spin-time in the two-phase waiting algorithm would then be always increased by this amount. In this way, other cooperating processes that are communicating with this machine would not lose coordination when the interactive job briefly interrupts.

11.2.4 Job Allocation and Placement

In this dissertation, we focused on the problem of dispatching processes after those processes have been placed on the nodes of the cluster. While we addressed the fair allocation of resources across jobs with different communication characteristics, we did not thoroughly investigate fair allocation across users regardless of the number and type of jobs those users run. In Section 5.3.5 we outlined two sufficient conditions for ensuring that each user receives a fair share of the cluster: first, tickets are balanced across nodes and, second, each local proportional-share scheduler is informed of the number of tickets issued in each currency across the cluster. We also described our implementation of a parallel ticket server for informing each local scheduler of this currency information. Throughout our measurements, we assumed that the system automatically places jobs according to these constraints.

A more pleasant environment in which to operate differs in two main ways. First, users should have the option of specifying the workstation on which to execute a process (*e.g.*, to compute on machines near file data, to use special devices, to run diagnostics, or to explicitly avoid faulty machines). Second, users should be able to specify the distribution of tickets across their jobs, in order to prioritize more important tasks. These capabilities, especially in conjunction with the presence of parallel jobs, greatly undermines the ability of the system to provide fair allocations across users [8]. Measurements of implicit coscheduling in such environments, as well as in systems supporting migration, would be a worthwhile endeavor.

11.2.5 Theoretical Analysis

The work in this dissertation is highly empirical. In our development of this research, we began by developing intuition for the problem and proposing heuristics for its solution, and then performed a wide set of simulations and implementation measurements to validate our ideas. Ideally, we would like to prove that the independent schedulers converge on a common job in a certain amount of time, given certain characteristics. In addition to this "holy grail", we feel that a more theoretical treatment of several smaller issues would be enlightening.

Distribution of Costs

Our analysis of the baseline spin-time in the conditional two-phase waiting algorithm assumes that processes should wait for the *expected worst-case* time when the requesting message triggers the scheduling of the remote process. In the simulation environment, where there is a fixed cost for transmitting messages and context-switches, the worst-case time is easy to predict and model.

In our implementation, the measured round-trip time with a context-switch follows a distribution. Empirically, we found that given a significant tail on the waiting times, implicit coscheduling suffers dramatically. Therefore, it is important for the underlying message layer and operating system to be tuned well and to give predictable performance. Given such a system, the heuristic that the expected worst-case completion time is the amount of time required for approximately 97% of the benchmark measurements gives good performance. However, we believe that the distribution function should be incorporated more formally into the cost models for spin-time.

Penalty for Loosing Coordination

Determining the amount of load-imbalance at which a process should relinquish the processor rather than spin idly requires a model of the cost of losing coordination. In Section 6.4.2, we argued that the highest penalty a process may expect to pay is $W \cdot P$ (the cost of a context-switch times the number of cooperating processes). While this number roughly matches experimental results, understanding this cost as a function of the communication pattern would significantly improve our calculations for how long a process

should wait for barriers. Such analysis would complement a run-time system that records and analyzes the current communication pattern.

11.2.6 Implicit Systems

Finally, we would like to study the use of implicit control and implicit information in more system services. We believe that our approach has the potential to not only simplify the design and implementation of many distributed services, but also improve performance. We predict that implicit information will become more valuable as distributed systems increase in scale and complexity. In systems with many autonomous components, it is likely that cooperating components will only conform to a minimally defined interface. With only explicit information, the capability of the system as a whole would degenerate to that of the "least common denominator"; however, by leveraging the implicit information that naturally emanates from the interactions in the system, smart components may be able to infer additional knowledge to improve their performance.

Bibliography

[1] Anurag Acharya, Guy Edjlali, and Joel Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings of 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Seattle, June 1997.

[2] Rakesh Agrawal and Ahmed K. Ezzat. Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[3] Albert Alexandrov, Mihai Ionescu, Klaus E. Schauser, and Chris 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.

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

[5] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[6] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve Steinberg, and Kathy Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[7] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David 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.

[8] Andrea Arpaci-Dusseau and David 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.

[9] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, pages 243–254, 1997.

[10] Mikhail J. Atallah, Christina Lock Black, Dan C. Marinescu, Howard Jay Siegel, and Thomas L. Casavant. Models and Algorithms for Co-scheduling Compute-Intensive Tasks on a Network of Workstations. *Journal of Parallel and Distrbuted Computing*, 16:319–327, 1992.

[11] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

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

[13] Edward A. Billard and Joseph .C. Pasquale. Adaptive Coordination in Distributed Systems with Delayed Communication. *IEEE Transactions on Systems, Man and Cybernetics*, 25(4):546–54, April 1995.

[14] Andrew P. Black. Supporting Distributed Applications: Experience with Eden. In *10th ACM Symposium on Operating System Principles*, pages 181–193, December 1985.

[15] D. Blackwell and M. A. Girshick. *Theory of Games and Statistical Decisions*. John Wiley & Sons, Inc., New York, 1954.

[16] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kusmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[18] Robert D. Blumofe and David S. Park. Scheduling Large-Scale Parallel Computations on Networks of Workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, August 1994.

[19] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, April 1994.

[20] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Sietz, J. Seizovic, and W. Su. Myrinet – A Gigabit-perSecond Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[21] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[22] D. Boggs, J. Modgul, and C. Kent. Measured capacity of an Ethernet. In *Proceedings of the SIGCOMM'88 Symposium*, pages 222–234, August 1988.

[23] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, California, July 1995.

[24] Eric A. Brewer and Bradley 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, April 1993.

[25] Matt Buchanan and Andrew Chien. Coordinated Thread Scheduling for Workstation Clusters Under Windows NT. In *Proceedings of USENIX Windows NT Workshop*, August 1997.

[26] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[27] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Supercomputing'94*, November 1994.

[28] Clemens H. Cap and Volker Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, 19:1221–1234, 1993.

[29] Thomas L. Casavant and Jon G. Kuhl. A Formal Model of Distributed Decision-Making and Its Application to Distributed Load Balancing. *6th International Conference on Distributed Computing Systems*, page 232, May 1986.

[30] L.M Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEEE Transactions on Computers*, C-27(10):866–872, December 1978.

[31] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and Page Migration for Multiprocessor Computer Servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, October 1994.

[32] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. *ACM Operating Systems Review*, 23(5):147–158, December 1989.

[33] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 33–44, February 1994.

[34] Reuven Cohen and Adrian Segall. An Efficient Priority Mechanism for Token-ring Networks. *IEEE Transactions on Communications*, 42:1769–77, February 1994.

[35] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on Multiprocessors. Technical Report 385, University of Rochester, Computer Science Department, February 1991.

[36] D. Culler, A. Dusseau, R. Martin, and K. Schauser. *Portability and Performance for Parallel Processing*, chapter 4: Fast Parallel Sorting under LogP: from Theory to Practice, pages 71–98. John Wiley & Sons Ltd., 1994.

[37] David Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Rich Martin, Chad Yoshikawa, and Frederick Wong. Parallel Computing on the Berkeley NOW. In *Ninth Joint Symposium on Parallel Processing*, Kobe, Japan, May 1997.

[38] David Culler, Lok Tin Liu, Rich Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

[39] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[40] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Eicken von. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–273, May 1993.

[41] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the clouds distributed operating system. *Computing Systems*, 3(1), 1990.

[42] Dorothy E. Denning and Peter J. Denning. Data Security. *Computing Surveys*, 11(2):227–249, September 1979.

[43] David DeWitt and Jim Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[44] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, August 1991.

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

[46] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Re-examining Scheduling and Communication in Parallel Programs. Computer Science UCB//CSD-95-881, University of California, Berkley, December 1994.

[47] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1996.

[48] Andrea C. Dusseau, David E. Culler, Klaus E. Schauser, and Richard 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.

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

[50] Steve Evans, Kevin Clarke, Dave Singleton, and Bart Smaalders. Optimizing Unix Resource Scheduling for User Interaction. In *1993 Summer Usenix*, pages 205–218. USENIX, June 1993.

[51] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Jim Voll, Roger Faulkner, Anil Shivalingiah, Mark Smith, , Dan Stein, Mary Weeks, and Dock 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.

[52] Dror 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.

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

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

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

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

[57] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *International Conference on Distributed Computer Systems*, pages 491–499, 1988.

[58] R. A. Finkel, M. L. Scott, Y. Artsy, and H.-Y. Chang. Experience with charlotte: Simplicity and function in a distributed operating system. *IEEE Transactions on Software Engineering*, 15(6):676–685, June 1989.

[59] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, August 1993. implicit, network.

[60] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[61] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 78–91, New York, October5–8 1997. ACM Press.

[62] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computing*, pages 1–9, October 1996.

[63] G. A. Geist and Vaidy Sunderam. The Evolution of the PVM Concurrent Computing System. In *COMPCON*, February 1993.

[64] David Gelernter and David Kaminsky. Supercomputing Out of Recycled Garbage: Preliminary Experience with Pirhana. In *Proceedings of Supercomputing '92*, pages 417–427, July 1992.

[65] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a Network of Workstations. In *Software Practice and Experience*, 1989.

[66] Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[67] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[68] James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *International Conference on Computer Architecture*, pages 124–131, 1983.

[69] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

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

[71] Joseph Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

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

[73] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1990.

[74] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

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

[76] Atsushi Hori, Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. A Scalable Time-Sharing Scheduling for Partionable, Distributed Memory Parallel Machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, 1995.

[77] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Global State Detection using Network Preemption. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[78] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, Noriyuki Soda, Hiroki Konaka, and Munenori Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In *Proceedings of the IPPS '96 Workshop on Job Scheduling Strategies for Parallel Processing*, 1996.

[79] Intel. *Paragon User's Manual*. Intel Corporation, 1992.

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

[81] Raj Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. Technical Report DEC-TR-566, Digital Equipment Corporation, April 1988.

[82] Morris Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Supercomputing'97*, 1997.

[83] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 198–211, New York, October 1997. ACM Press.

[84] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[85] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

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

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

[88] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data-Touching Overheads in TCP/IP. In *Proceedings of the 1993 SIGCOMM*, pages 259–268, San Francisco, CA, September 1993.

[89] Kimberly Keeton, David A. Patterson, and Thomas E. Anderson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, Stanford University, Stanford, CA, August 1995.

[90] R. Kent Koeninger, Mark Furtney, and Martin Walker. A shared memory MPP from Cray Research. *Digital Technical Journal of Digital Equipment Corporation*, 6(2):8–21, Spring 1994.

[91] Richard N. Lagerstrom and Stephan K. Gipp. PScheD: Political Scheduling on the CRAY T3E. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[92] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[93] Butler W. Lampson. Hints for Computer System Design. *Operating Systems Review*, 17(5):33–48, October 1983.

[94] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating polling, interrupts, and thread management. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 13–22, Annapolis, Maryland, October 1996. IEEE Computer Society.

[95] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[96] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1990.

[97] Tom Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[98] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Atchitecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms*, pages 272–285, June 1992.

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

[100] Scott T. Leutenegger and Xian-He Sun. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Proceedings of Supercomputing '93*, pages 143–152, 1993.

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

[102] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[103] Steven B. Lipner. A comment on the confinement problem. In *Proceedings of the 5th Symposium on Operating Systems Principles*, pages 192–196, November 1975.

[104] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[105] Shau-Ping Lo and Virgil D. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 356–363, September 1987.

[106] P. Lopez, J.M. Martinez, and J. Duato. A Very Efficient Distributed Deadlock Detection Mechanism for Wormhole Networks. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 57–66, February 1998.

[107] Steven S. Lumetta, Arvind Krishnamurthy, and David E. Culler. Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines. In *Proceedings of Supercomputing '95*, 1995.

[108] Ewing Lusk and Ralph Butler. Portable parallel programming with p4. In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[109] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, and M. Frans Kaashoek. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 231–242, February 1998.

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

[111] Shikharesh Majumdar and Yiu Ming Leung. Characterization of Applications with I/O for Processor Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the 1994 IEEE Symposium on Parallel and Distributed Processing*, pages 298–307, 1994.

[112] Thomas W. Malone, Richard E. Fikes, Kenneth R. Grant, and Michael T. Howard. Enterprise: A Market-like Task Scheduler for Distributed Computing Environments. In *The Ecology of Computation*, pages 177–205. North-Holland, 1988.

[113] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin Theobald, and Xinmin Tian. Polling watchdog : Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 179–190, New York, May 1996.

[114] Jacob Marschak and Roy Radner. *Economic Theory of Teams*. Yale University Press, New Haven, 1972.

[115] Richard P. Martin. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of the 2nd Hot Interconnects Conference*, July 1994.

[116] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas 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.

[117] Murray S. Mazer. Reasoning about knowledge to understand distributed AI systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1333–1346, November-December 1991.

[118] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993. Also as tech report 90-03-02, University of Washington, Dept. of Computer Science and Engineering, 1990.

[119] Cathy McCann and John Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 19–32, February 1994.

[120] Cathy McCann and John Zahorjan. Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 208–219, May 1995.

[121] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Winter USENIX*, January 1996. Available from http://reality.sgi.com/lm/lmbench/lmbench.html.

[122] R. Metcalf and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–403, July 1976.

[123] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[124] Matt M. Mutka and Miron Livny. Scheduling Remote Processing Capacity In A Workstation-Processor Bank Network. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 2–9, September 1987.

[125] Matt M. Mutka and Miron Livny. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[126] Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings of Supercomputing '93*, pages 824–833, November 1993.

[127] Nenad Nedeljkovic and Michael J. Quinn. Data Parallel Programming on a Network of Heterogeneous Workstations. In *Proceedings of High Performance Distributed Computing (HPDC)*, pages 28–36, September 1992.

[128] Randolph Nelson and Donald Towsley. A performance evaluation of several priority policies for parallel processing systems. *Journal of the ACM*, 40(3):714–740, July 1993.

[129] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, second edition, 1947.

[130] David Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, November 1987.

[131] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard. 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*, November 1993.

[132] Richard L. Norton and Jacob A. Abraham. Using Write Back Cache to Improve Performance of Multiuser Multiprocessors. In *1982 International Conference on Parallel Processing*, August 1982.

[133] David Oppenheimer. Gang scheduling for the SHRIMP Multicomputer. Senior Thesis, 1997.

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

[135] Vivek Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 1998.

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

[137] David Lorge Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[138] Joseph Carlo Pasquale. Intelligent decentralized control in large distributed computer systems. Technical Report UCB//CSD-88-422, University of California Berkeley, April 1988.

[139] Vinod G.J. Peris, Mark S. Squillante, and Vijay K. Naik. Analysis of the Impact of Memory in Distributed Parallel Processing Systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, pages 5–18, February 1994.

[140] G. J. Popek and B. J. Walker, editors. *The LOCUS Distributed System Architecture*, pages 73–89. Computer Systems Series. The MIT Press, 1985.

[141] Jim Pruyne and Miron Livny. Parallel Processing on Dynamic Resources with CARMI. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 165–177, April 1995.

[142] Richard F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 64–75. ACM, 1982.

[143] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of PCs. In *Proceedings of IEEE Areospace*, 1997.

[144] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S. Squillante. The Impact of I/O on Program Behavior and Parallel Scheduling. In *SIGMETRICS '98/PERFORMANCE'98.*, June 1998.

[145] Rafael H Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* PhD thesis, U.C. Berkeley, Computer Science Division, February 1992.

[146] Vikram Saletore, J. Jacob, and M. Padala. Parallel Computations on the CHARM Heterogeneous Workstation Cluster. In *Third International Symposium on High Performance Distributed Computing*, August 1994.

[147] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[148] Sanjeev Setia. Trace-driven Analysis of Migration-based Gang Scheduling Policies for Parallel Computers. In *International Conference on Parallel Processing*, August 1997.

[149] Kenneth C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[150] Alan J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

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

[152] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the IPPS '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[153] Patrick G. Sobalvarro and William 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.

[154] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranwake, and Charles V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *Proceedings of the International conference on Parallel Processing*, volume 1, pages 11–14, 1995.

[155] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time Systems Symposium*, December 1996.

[156] Ion Stoica, Hussein Abdel-Wahab, and Alex Pothen. A Microeconomic Scheduler for Parallel Computers. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–135, April 1995.

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

[158] Ahmed N. Tantawy, Asser N. Tantawi, and Dimitrios. N. Serpanos. An adaptive scheduling scheme for dynamic service time allocation on a shared resource. In *12th International Conference on Distributed Computing Systems*, pages 294–301, June 1992.

[159] Robert R. Tenney and Nils R. Sandell, Jr. Structures for distributed decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(8):517–527, August 1981.

[160] Jack A. Test. Multi-processor management in the Concentrix operating system. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–182, January 1986.

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

[162] Marvin M. Theimer, K. Landtz, and David Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[163] Marvin M. Theimer and Keith A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57, November 1989.

[164] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control in Distributed Operating Systems. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

[165] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. In *Proceedings of Hot Interconnects II*, Stanford, CA, August 1994.

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

[167] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik 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.

[168] Carl 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.

[169] Carl A. Waldspurger, Tad Hogg, Bernado Huberman, Jeffrey Kephart, and Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

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

[171] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, June 1995.

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

[173] Fang Wang, Marios Papaefthymiou, and Mark 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.

[174] Kuei Yu Wang and Dan C. Marinescu. Correlation of the paging activity of individual node programs in the SPMD execution mode. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 61–71, Los Alamitos, CA, USA, January 1995.

[175] Michael S. Warren, Donald J. Becker, M. Patrick Goda, John K. Salmon, and Thomas 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.

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

[177] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

[178] Sognian Zhou, Jingwen Wang, Xiaohn Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.