A Study of Backoff Barrier Synchronization

by

Mathews Malieakkal Cherian

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

© Massachusetts Institute of Technology 1989

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 19, 1989

Certified by..
Thesis Supervisor (Academic)

Certified by..
Company Supervisor (Cooperating Company)

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A Study of Backoff Barrier Synchronization

by

Mathews Malieakkal Cherian

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 1989, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Shared-memory multiprocessors commonly use shared variables for synchronization. Simulations of real parallel applications show that large-scale cache-coherent multiprocessors suffer significant amounts of invalidation traffic due to synchronization. Large multiprocessors that do not cache synchronization variables are often more severely impacted. If this synchronization traffic is not reduced or managed adequately, synchronization references can cause severe congestion in the network. This thesis reports on data from trace-driven simulations of shared-memory multiprocessors, and proposes a class of adaptive backoff methods that do not use any extra hardware and can significantly reduce the memory traffic to synchronization variables. These methods use synchronization state to reduce polling of synchronization variables. Our simulations show that when the number of processors participating in a barrier synchronization is small compared to the time of arrival of the processors, reductions of 20 percent to over 95 percent in synchronization traffic can be achieved at no extra cost. In other situations adaptive backoff techniques result in a tradeoff between reduced network accesses and increased processor idle time.

Thesis Supervisor: Anant Agarwal
Title: Assistant Professor of Computer Science and Electrical Engineering

Thesis Supervisor: Harold Stone
Title: IBM

Acknowledgments

I would like to thank most sincerely my advisor, Anant Agarwal, for providing me with many ideas and much encouragement. His insight and enthusiasm made this experience highly productive and enjoyable.

I also have many people at IBM to thank. Many thanks to Kimming So who gave much of his time and enthusiasm to help move my work forward. I thank Kimming and Manoj Kumar for contributing many ideas for this work. Thanks also to Alan Norton, who has always been ready to help when necessary.

Of course, I'd also like to thank the members of Alewife, our research group: David Chaiken, Beng-Hong Lim, Gino Maa, Kiyoshi Kurihara, and Dann Nussbaum. Thanks to everyone for putting up with my ten million questions and making life in the lab fun. Special thanks to nmy office-mate Dann Nussbaum, not only for being such an off-the-wall character, but for always being willing to discuss anything at any time for any length of time.

Finally, I have my parents to thanks for the tremendous support and encouragement they have provided. Thanks be to God.

Contents

1 **Introduction** **9**

2 **Multiprocessor Simulation** **13**

2.1 Objectives 13

2.2 Trace-driven Simulation 14

2.2.1 The Hardware Model 16

2.2.2 The Programming Model 17

2.3 Simulation Methodology 24

2.3.1 Timing 25

2.3.2 Limitations 25

2.4 Post-mortem Scheduler Design Features 26

2.5 The Cache Simulator and Network Model 28

2.5.1 Multiprocessor Cache Simulator 28

2.5.2 The Network Model 29

2.6 Summary of methodology 31

3 **The Synchronization Problem** **33**

3.0.1 Background 33

3.0.2 Synchronization References and Scalability 39

3.1 Why do synchronization references hurt performance? 42

3.1.1 Disallowing Caching of Synchronization Variables 44

4 Analysis **47**

4.1 Adaptive Backoff Barrier Synchronization 47

4.1.1 Previous Work 48

4.1.2 Backoff on the barrier variable 48

4.1.3 Backoff on the barrier flag 49

4.2 The Network Model 50

4.3 A Barrier Model 51

4.3.1 Analytically Estimating Barrier Performance 53

4.3.2 Simulation Methodology 55

4.4 Evaluation 56

4.4.1 Estimating the Potential Reduction in Traffic 56

4.4.2 Simulation results 58

4.5 Discussion of Tradeoffs 62

4.5.1 Summary 64

4.6 Optimizations and Extensions 66

4.7 Conclusions 69

A Applications **70**

List of Figures

2-1 Multiprocessor Model. 14
2-2 Serial Section 20
2-3 Parallel Section 22
2-4 Barrier Synchronization. 23
2-5 Scheduler structure 27
2-6 Scheduler Input 29

3-1 Read-Modify-Write Synchronizations 35
3-2 Fetch&Add Synchronization 37
3-3 Cache invalidation statistics for SIMPLE with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages. 40
3-4 Cache invalidation statistics for WEATHER with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages. 41
3-5 Cache invalidation statistics for FFT with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages. 42

4-1 Intervals of execution and synchronization. 52
4-2 Comparing the predictions of the analytical model and predictions of barrier performance. 57

4-3 Performance of backoff algorithms for $A = 0$. 60

4-4 Performance of backoff algorithms for $A = 100$. 60

4-5 Performance of backoff algorithms for $A = 1000$. 61

4-6 Processor waiting times for backoff algorithms for $A = 1000$. 63

List of Tables

2.1 Memory references resulting in network transactions. 30

3.1 Percentage of non-synchronization (NS) and synchronization (S) references that cause invalidations in directory schemes with 2, 3, 4, 5, and 64 pointers in a 64 processor system. Synchronization references comprised 0.2%, 7.9%, and 5.3% of the data references in FFT, WEATHER, and SIMPLE respectively. 43

3.2 Synchronization traffic to main memory as a percentage of the total traffic when the synchronization variables are not cached. Block size is 16 bytes and cache size is 256KBytes. The non-synchronization blocks are cached and coherence is maintained using directory schemes with 2, 3, 4, 5, and 64 pointers. 45

4.1 Average number of cycles, A, between first and last arrivals at waits and barriers. E is the average number of cycles between the last arrival at the previous barrier (or wait) and the first arrival at the next barrier (or wait), i.e. it is the average time between barriers or waits. 52

Chapter 1

Introduction

Shared-memory multiprocessors have received much attention in recent times as a means to achieve cost-effective high performance computing. As designers consider building larger and larger Multiple-Instruction-Multiple-Data (MIMD) machines [1], *synchronization* has arisen as an important problem. These machines typically run a number of processes concurrently. The processes might be independent or they all might be working on a single application. The one characteristic common to every process, though, is that it must be *synchronized* to ensure correct sequencing among sister processes and also guarantee mutually exclusive access to shared mutable data. As greater numbers of processors must be coordinated, the more crucial it is that synchronization operations be implemented efficiently.

This thesis considers synchronization in self-scheduling shared-memory multiprocessors. These systems commonly use shared variables to synchronize activities among processors [8, 33, 21] often leading to widespread sharing among processors. Trace-driven simulations of parallel applications show that these widely shared synchronization variables adversely impact the performance of large-scale multiprocessors, cache-coherent or otherwise.

[1] In an MIMD machine, every processor executes its own code independently of the other processors. Thus, different processors can run different programs[10].

In systems without hardware support for cache coherence, such as the IBM RP3 [25], Ultracomputer [14], Cedar [11], these references to shared variables must traverse the interconnection network. Not only do synchronization references consume a significant fraction of the network bandwidth, but more important, a widely-shared synchronization variable (such as in a barrier synchronization) will result in heavy traffic to the same location in memory and cause hot-spot contention problems [26].

On the other hand, in systems that use directory schemes to maintain cache coherence, we show that synchronization variables result in excessive invalidation traffic when the number of pointers in the cache directory is limited. A potential solution for cache directories would be to implement software combining trees [35] for synchronization variables. As long as the degree of the nodes in the combining tree is less than the number of pointers in the cache-directory, then synchronization variables will not result in extra invalidation traffic. We are currently investigating this approach and will not address it here.

In this thesis we consider software schemes to reduce the number of synchronization spins in multiprocessors that do not cache their synchronization variables. We propose a set of adaptive backoff techniques which make use of available synchronization state information in order to "back off" and postpone polling a synchronization variable.

The general idea of backoff has been used in one form or another in a number of applications. The approach was first used in Aloha [1], a radio-based, packet-switching network. If a collision occurred in the network, each source would backoff for a random interval before attempting to retransmit. The Ethernet [22] went one step further and used a random retransmission interval in which collision history influenced the choice of the mean of the random intervals. Adaptive control schemes for multiple access communications networks have been analyzed in [19, 18, 20]. We discuss in section 4.1.1 how and why adaptive backoff synchronization methods differ from these adaptive control schemes.

We evaluate the performance of adaptive backoff synchronization techniques by applying them to the *barrier* synchronization. Barrier synchronizations are commonly used in applications to guarantee that all processors have reached a given point in a program before proceeding.

We focus on barriers implemented using two shared variables with busy waiting on synchronization variables [33] (described in detail in section 3.0.1). Alternate barrier implementations might use a scheme where processors arriving at a barrier are put to sleep until the last processor arrives. This method avoids the extra network traffic of polling a barrier flag, but incurs the potentially high overhead of enqueuing a process on a condition variable. Often, the choice of busy waiting or blocking cannot be made at compile time due to uncertainty in execution times of processes. In such cases, our adaptive methods can be used to decide when it might be best to put to sleep a busy-waiting process as explained in a later section.

Hardware support for barriers has also been proposed in several forms. The RP3 [25] proposed a combining network in which switches contain special hardware to combine simultaneous data accesses destined to the same memory location and forward one request. This would eliminate contention in the network and at the memory modules when a large number of processors reference the same synchronization variable simultaneously, but RP3 cost estimates predict that the switches are expensive and slow [26]. Furthermore, combining performance is eroded when synchronization references to a given variable do not arrive simultaneously at a switching node. Several cache-coherent multiprocessors allow simultaneous invalidates of all cached copies of a block. In such systems all repeat accesses of a synchronization variable can be satisfied by the cache. However, the need to rely on resources that can support broadcast invalidates, such as a shared bus, limits the scalability of such systems. The PAX computer [16] uses special global-synchronization logic implemented in hardware to allow low-latency, low-cost barrier synchronization. Issues which arise with this approach concern flexibility in allowing multiple numbers of barriers to execute simultaneously

with varying numbers of processors.

The results of this thesis show that backoff techniques applied to barriers yield reductions in synchronization traffic by 20 percent to over 95 percent in cases where the number of processors involved in the barrier is small compared to the time of arrival between processors. In other situations, these schemes provide a tradeoff between cost (in terms of processor idle time) and performance - a tradeoff that can be determined by the user's needs.

This thesis consists of two parts: 1) development of a trace-driven large-scale shared-memory multiprocessor simulator in order to study the effect of synchronization on multiprocessor performance; and 2) analysis and simulation of adaptive backoff synchronization techniques.

The rest of the thesis is organized as follows. Chapter 2 first describes the multiprocessor simulation methodology we use. Chapter 3 describes current approaches to barrier synchronization in shared-memory multiprocessors and presents results from trace-driven simulations describing how synchronization impacts large-scale multiprocessors. Chapter 4 presents adaptive backoff synchronization techniques as they apply to barriers. A barrier evaluation model and a simulation methodology is described. Finally, the adaptive backoff techniques are evaluated via an analytical model and simulations and the tradeoffs involved in their implementation are discussed. Chapter 5 suggests extensions to this work and summarizes the findings.

Chapter 2

Multiprocessor Simulation

2.1 Objectives

The objective of the multiprocessor simulations in this thesis is to understand the impact of synchronization on large-scale shared-memory multiprocessor performance. Given this objective, we are primarily interested in gaining an understanding of multiprocessor reference behavior. Specifically we would like to measure the amount of traffic to memory generated by synchronization events.

In simulating a multiprocessor system, we must simulate the behavior of each of the components depicted in Figure 2-1: processors, caches, interconnection network, and memory modules. Processor simulation can be slow and complex. We can capture the memory reference behavior of the processor in a couple of ways: 1) Do a functional multiprocessor simulation, simulating the arithmetic/logic capabilities of each processor; 2) Perform an instruction level simulation; or 3) Do trace-driven simulations, using an existing instrumentable multiprocessor to record the references made by each processor, thereby attaining a multiprocesssor reference trace.

The functional simulation approach entails onerous development and execution time because it attempts to simulate the instruction sequencing and arithmetic/logic actions of each processor in the system. Instruction level simulation are not quite

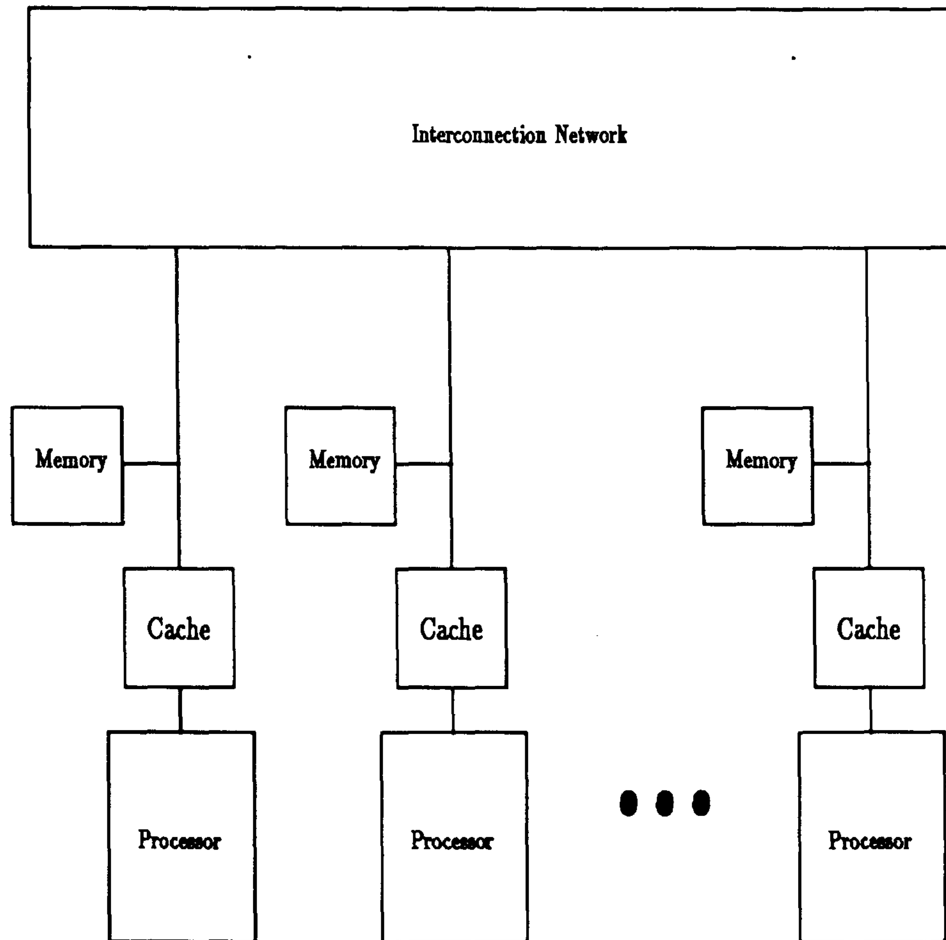

Figure 2-1: *Multiprocessor Model. The primary elements of the MIMD shared-memory machine are the processors, caches, interconnection network, and memory modules.*

as computation intensive, but both approaches become prohibitively expensive as we attempt to simulate large systems. So we look to the trace-driven approach for our simulations.

2.2 Trace-driven Simulation

Trace-driven simulation refers to the simulation of a computer system using a trace of an application's references to memory as the input driving the system. Trace-driven simulation has been of invaluable benefit in analyzing and designing numerous ele-

ments of high-performance computer systems: memory hierarchies, instruction sets, pipeline organization. Trace-driven simulation has become particularly important in the area of cache design [29, 13, 15, 30]. Previously cache designers relied on analytical models or simulation of proposed caches using rough assumptions of memory reference behavior. Cache behavior is highly sensitive to memory reference patterns, though, so the results of these earlier evaluations were highly dependent on the assumptions made about reference behavior. Often trace-driven simulation is used to experimentally validate analytical or intuitive models of systems as well as evaluate proposed architectures.

In a similar fashion, multiprocessor design can benefit from the trace-driven simulation of real parallel applications. Optimal design of the memory subsystem is crucial to multiprocessor performance. Many important questions bearing relationship to memory subsystem design remain open:

- Is there locality among references in a multiprocessor system? If there is, then the memory structure should exploit the locality to improve request service time, rather than making all memory locations a uniform distance away from processors - thereby enforcing a slower service time for all processors in the system.

- How frequently are variables shared among processors? When sharing does occur, how many processors are involved?

- Understanding the answers to the above two questions would help to answer the question of whether multiprocessors should utilize caches. The overhead of caching shared variables might outweigh the benefits.

- Is hotspot contention truly a problem in real application? Is it a bad enough a problem to merit building expensive combining network to resolve contention?

This list is by no means complete. A thorough understanding of multiprocessor reference behavior is needed in order to address these questions. In addition, refer-

ence traces allow testing out of new ideas such as cache-directory schemes for cache coherency. In this thesis, the multiprocessor traces generated are used to gauge the impact of synchronization on performance in shared-memory systems.

The design of large-scale multiprocessor systems has, as of yet, not benefitted from trace-driven simulations because of the difficulty in obtaining multiprocessor traces of large systems. Large-scale multiprocessors with tracing capability do not exist, forcing designers of these systems to rely on analytical models of multiprocessors and extrapolations of address patterns from uniprocessor systems (unless of course one wanted to actually build the proposed hardware to see how it performed!). Multiprocessor tracing techniques have been developed [28, 32] but none of these techniques produce data for systems of greater than sixteen processors because they rely on an existing parallel system to produce a multiprocessor trace.

In order to obtain multiprocessor traces of large-scale systems, this thesis makes use of a "post-mortem" scheduling technique [5] in which a multiprocessor trace of a real parallel application is generated from the application's uniprocessor trace with synchronization information imbedded in it. The uniprocessor trace with synchronization information essentially provides us with a task trace which we can then schedule into a multiple processor execution trace with the post-mortem scheduler. This methodology (described in detail below) allows simulation of any n-way system with constant overhead, ie. simulating a 1024-way system is as cheap as simulating an eight-way system, and only a uniprocessor is required to produce the trace. The multiprocessor trace can then be used to drive cache and network simulations.

2.2.1 The Hardware Model

Figure 2-1 describes the logical hardware model of the multiprocessor simulated in this thesis. Processors operating in a MIMD fashion are connected to shared-memory modules via an interconnection network. All data (private and shared) and instructions are cached, and all cache misses must be serviced through the network.

The primary hardware primitive used for synchronization in this model is a generic read-modify-write (RMW) primitive which allows an atomic update of a memory location. If more than one processor attempts a RMW on the same location, only one succeeds, while the others have to wait and try the next cycle. All synchronization events in the application code are based on this hardware synchronization primitive.

For our trace-driven simulations, the network and memory were not modeled. The caches were simulated using a cache-directory to maintain coherency of data among the processors, and we assumed a network model (described in section 2.5.2) in order to measure the network load generated by the applications.

A few important assumptions are made in this model: 1) we assume a Risc-like architecture for the processor, ie. processors execute an instruction every cycle; and 2) we assume a canonical model of time in the simulations where the principal clock is a processor's reference. A processor makes a reference each network cycle unless it is waiting due to RMW contention (simulation timing issues will be discussed in more detail in section 2.3.1.

2.2.2 The Programming Model

The model of computation assumed for the parallel applications traced in this thesis is the *Single-Program-Multiple-Data* [8] (SPMD) programming paradigm. In this model all processors execute the same application code, but synchronization constructs dynamically assign portions of the code (tasks) to each of the processors. SPMD is an example of a self-scheduling computational model, differing from other computational models such as master-slave or fork-join which require that one process execute the code and spawn other processes when parallel execution is required. These other approaches have a much higher system overhead than SPMD because they spawn processes at the beginning of every parallel section and kill them at the end. With the SPMD paradigm processes are spawned once - at the beginning of the application.

The SPMD model used in this thesis was implemented on the EPEX [12] system,

which allows parallel execution on IBM S/370 machines. The Fortran applications used were all parallelized by imbedding into the code synchronization constructs which result in execution conforming to the SPMD paradigm.

In SPMD code, there exist three types of tasks: serial, parallel, or replicate. Synchronization code surrounding these tasks determines whether or not the section should be executed by an arriving processor. Each of these code sections is described in detail below along with the synchronization code used to implement SPMD self-scheduling.

The synchronization implementations described are those used in the post-mortem scheduler and differ slightly from the synchronization code described in [8]. The primary difference is that the synchronization code in [8] allows more flexibility in parallel execution. The synchronization simulated in the scheduler is optimized to the applications we traced. In these applications every serial and parallel section contains a wait, guaranteeing that all processors in the system are executing the same section at any given time. We do not have to worry about a variable number of processors executing different sections simultaneously or about stragglers arriving at a section long after everyone else has arrived and departed. Thus, our synchronization code is not as complicated as that of EPEX. The code described in [8] allows processors to overlap execution of consecutive sections if possible.

Serial Sections

A serial task is executed by only one processor. Serial sections in applications often correspond to initialization code or I/O. Synchronization code before and after the section ensure that only the first processor arriving at the section executes it while the others skip to the end. An option at the end of the serial section specifies whether processors must wait for the processor executing the section to finish or whether they can simply proceed. If a process arrives after the serial section has been executed, it simply skips to the end and proceeds. All the serial sections in the applications

traced for this thesis specify the wait option.

The synchronization code for the serial section (as simulated by the post-mortem scheduler) maintains a shared-variable *CNTR*, initialized to zero, which is incremented via a RMW operation to indicate the number of processors which have arrived at the section. If the value returned is greater than one the processor knows another processor is already executing the section, so it proceeds to the end of the section where it atomically decrements the counter. If the wait option is set and the counter value returned upon decrementing is not zero (indicating that not all the processors have arrived), the processor repeatedly polls, ie. *busy waits*, on a shared-variable wait *FLAG* until it is set. Figure 2-2 illustrates the serial section synchronization in terms of the references made to the synchronization variables involved. Every process increments and decrements the shared-variable *CNTR*, and all wait on the shared-variable *FLAG*. When the processor executing the serial section finishes (decrementing the shared counter to zero), it sets the wait flag and all processors proceed upon seeing the set flag. The general form of this wait, also used at the end of a parallel section, is illustrated in Figure 2-2.

Parallel Sections

Parallel sections are primarily parallel loops whose iterations are mutually independent. Since the iterations are mutually independent, processors can execute them in parallel. If dependencies do exist among the iterations, locks can be placed around the critical sections and the processors can still execute the sections in parallel. Synchronization code dynamically assigns iterations of the parallel loops to processors as they arrive at the beginning of the loop. Upon completing an iteration, the processor is assigned another iteration if more work is available. When no more iterations are available the processor proceeds to the end of the parallel section where, as with the serial section, the processor can proceed or wait depending on whether the wait option is specified. Once again, all the parallel sections in our applications have the

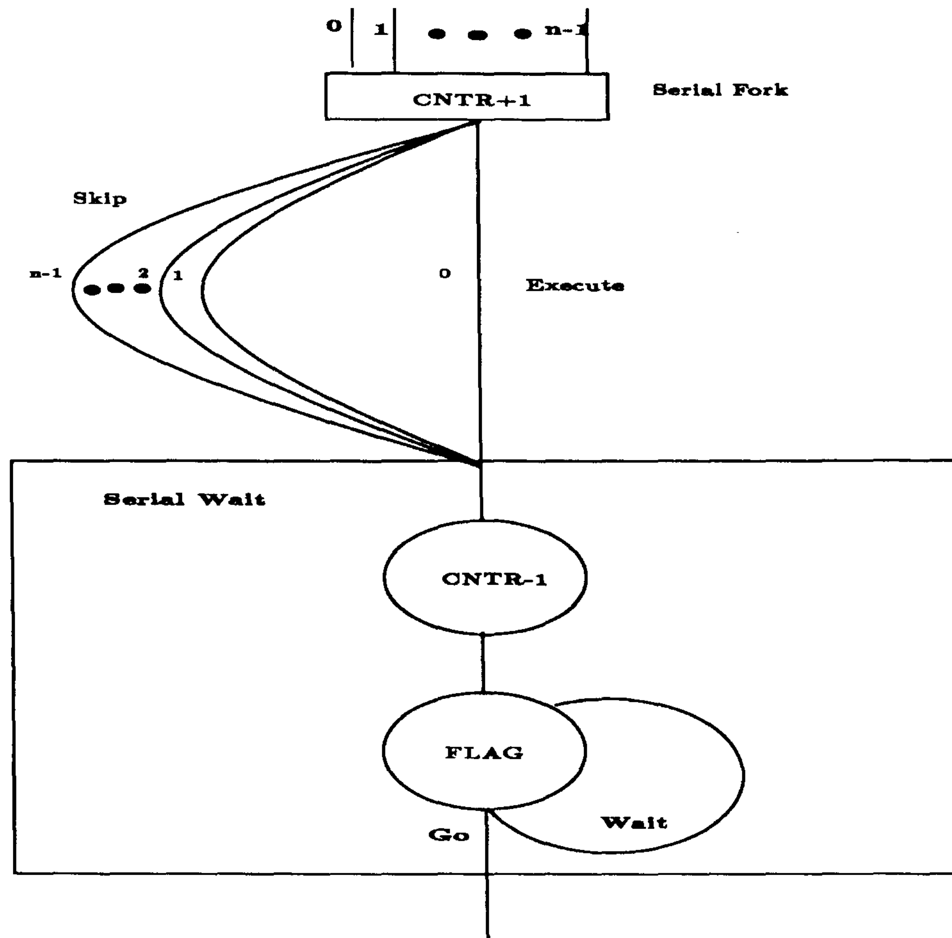

Figure 2-2: *Processors increment the shared variable CNTR upon arriving and decrement the variable upon finishing the section. Processors then access the shared variable Flag as they busy wait at the end of the loop.*

wait option specified.

In the SPMD paradigm, parallel iterations can also be *chunked*, ie more than one loop iteration can be assigned to a processor every time it comes around for more work. Thus, the time between synchronization is reduced. This technique is, naturally, often used when iterations are of short length. In the simulations we ran, loop iterations often ran into the tens of thousands of instructions, so we had no need to chunk. Table 4.1 displays the average length of the parallel sections in the applications we traced.

Figure 2-3 illustrates the synchronization and flow of the parallel loop section as implemented in the post-mortem scheduler. The synchronization code is similar to that of the serial section, except now we have a loop index variable, *LPINDEX*, which processors decrement upon picking up an iteration to execute. The index value is initialized to the number of available parallel loops. When the index reaches zero, indicating all the work in the section has been assigned, processors skip to the wait section at the end of the loop where they atomically decrement a shared variable,*CNTR*, initialized to one less than the number of processors executing the application. The processors busy-wait on a shared variable, *FLAG*, until the last processor that is returned a value of one upon decrementing *CNTR* then sets the flag. All proceed after reading the set flag.

This implementation is much simpler than the more flexible implementation described in [8]. In [8] additional shared synchronization variables are needed to keep track of which sections of the application have been executed. Each section maintains a *shared clock*, against which processors compare their *private clocks* to see whether they have arrived early, late, or on time. If a processor arrives early, it waits for the shared clock to catch up; if it arrives late, it increments its private clock and skips to the next section; if it arrives on time, it enters the section and looks for work (through loop index distribution). If there is no work left, the processor sets another shared variable, *gate*, which prevents any more processors from entering the section. When all processors in the section have finished, the processor which set the gate increments the shared clock and resets the synchronization variables used in the section. The important point to note with this implementation is that only those processors which have entered the section need to wait for the section to be finished. Others (late arrivals, for example) can continue to the next independent section or reexecute the current section.

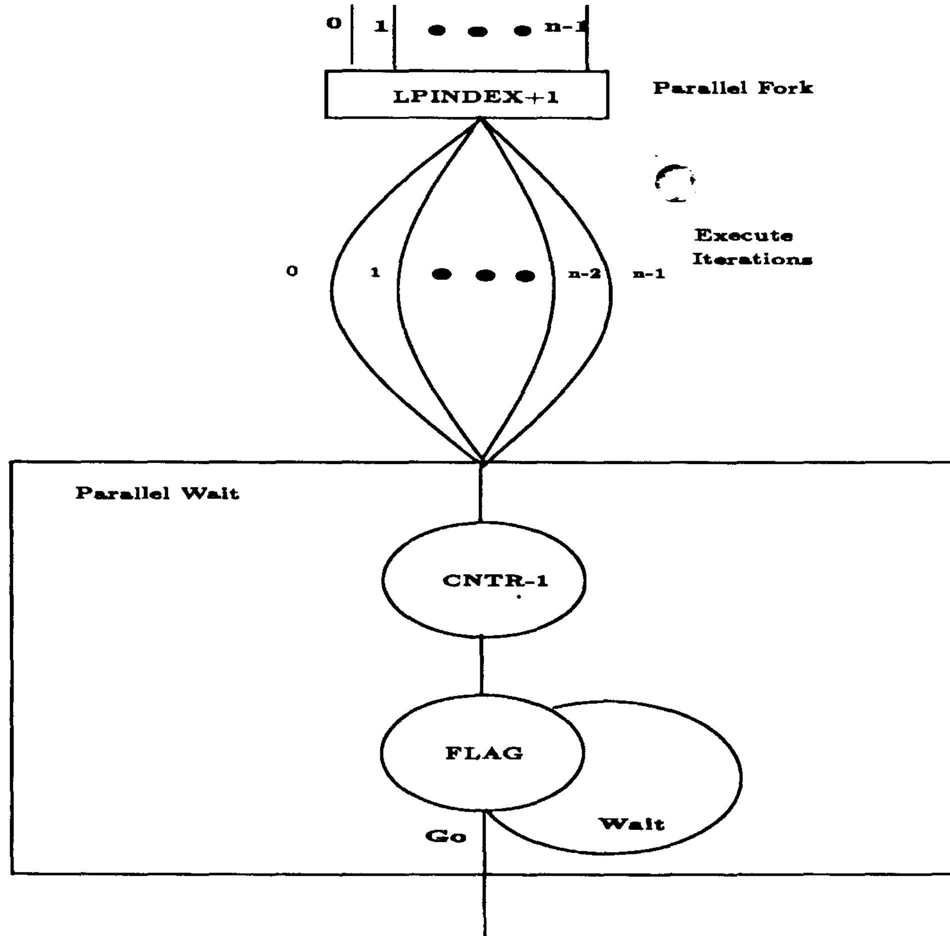

Figure 2-3: *The synchronization code of a parallel section is similar to that in the serial section, except now we have a LPINDEX shared variable which processors decrement every time they pick up a loop iteration.*

Replicate Sections

Replicate sections are sections which can be executed by all processors in the system. Typically these sections contain a small amount of code that could be executed serially, ie. put into a serial section. The amount of code is so small, however, that the synchronization overhead of a serial section is too high to merit enclosing the code in a serial section. Thus, as long as execution by all processors does not affect the correctness of the code, every processor *replicates* the computation. Since all processors execute this section, no synchronization is required.

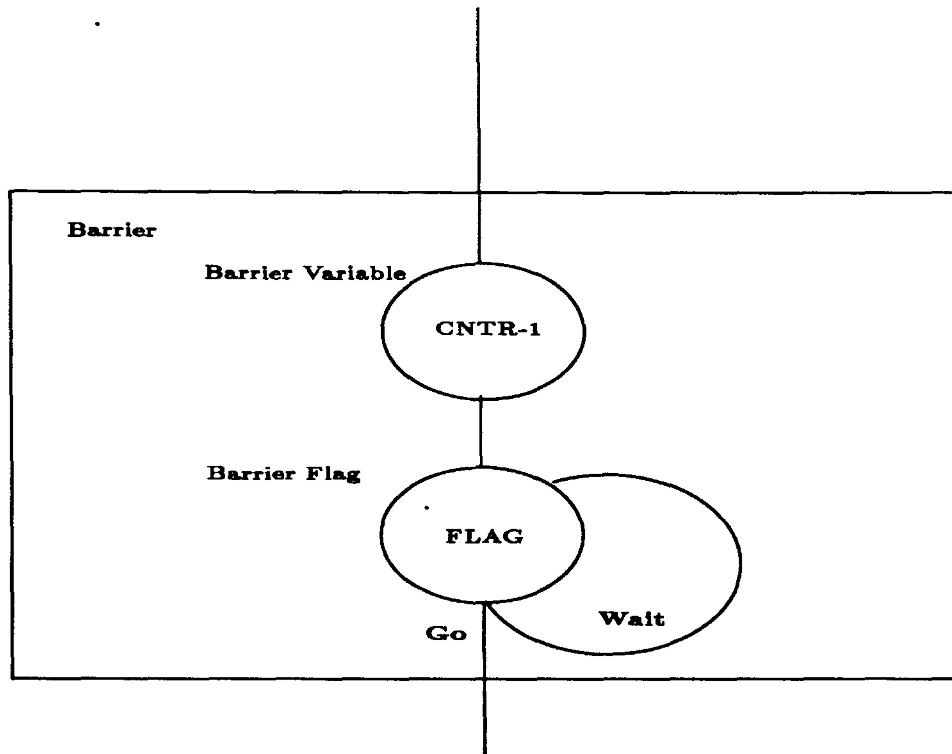

Figure 2-4: *A barrier synchronization is similar to the wait at the end of a parallel or serial section. Arriving processors increment a shared-variable CNTR and busy-wait on Flag.*

Barriers

A barrier is a synchronization event which requires that all processors executing an application reach the barrier before proceeding. Barriers are similar to the waits at the end of serial and parallel sections. Figure 2-4 illustrates the flow diagram of a barrier. Processors arriving at a barrier increment the barrier variable, a shared-variable *CNTR* initially set to one less than than the number processors executing the application. If the variable's value is greater than zero, the processor polls the barrier *FLAG* which is set by the last processor to reach the barrier. Note that the processors waiting are busy-waiting on the barrier flag.

One note about the SPMD computational model is that it classifies data as private or shared. Instructions and private data are in *private memory* whereas data shared by

all processors and synchronization variables are in *shared memory*. In our simulations both private and shared variables are cached, and miss requests are sent out over the network. Cache coherency is not maintained, however, for private data.

A final note to make is that our simulations assumed no process-switching.

2.3 Simulation Methodology

The multiprocessor simulation methodology used in this thesis makes use, as mentioned earlier, of a post-mortem scheduling technique. This technique generates a multiprocessor trace from the memory reference trace of a uniprocessor execution of an application parallelized using the SPMD computational model. Key to the scheme is that the uniprocessor execution trace include information about synchronization events in the code. Since in the SPMD model, synchronization events surround each task in the system, whether serial, parallel, or replicate, the uniprocessor trace with synchronization information can be viewed as a trace of all the tasks in a parallel application. Thus we have a description of the reference behavior of all the tasks, but we have no temporal ordering of tasks. The post-mortem scheduler produces a temporal ordering by simulating processors executing the parallel application described by the uniprocessor trace. The scheduler first makes a pass through the uniprocessor trace and constructs a task trace from the synchronization markers in the trace. The scheduler then simulates processors processors executing these tasks in parallel in an SPMD fashion and assigns tasks to processors. Execution of the tasks by processors is simulated in a round-robin fashion with each processor making one reference (unless waiting to do a RMW) each cycle from the task the processor is executing.

The scheduler also simulates processor behavior at the synchronization events in the program and outputs the appropriate synchronization references into the multiprocessor traces. Note that since the scheduler simulates the behavior of the processors at the synchronization events, we are free to choose whatever synchronization

implementation we desire, whether, for example, in the case of barrier synchronizations, busy-wait, semi-busy wait, adaptive backoff, or distributed barrier trees.

2.3.1 Timing

An important issue in multiprocessor simulation is timing. Synchronization determines the temporal behavior of references in a multiprocessor system. If we are studying synchronization traffic we must accurately simulate the temporal behavior of the processors. For example, the number of synchronization spins a processor makes while waiting at a barrier synchronization will be affected if timing is off.

In our simulations, we use a canonical cycle in which a processor makes a reference every cycle. Therefore cycles in our simulations are determined by processor references. For Vax-like machines, this a reasonable approximation. In our case, our reference trace is generated by PSIMUL [32], which runs on an IBM S/370. The approximation of one reference a cycle is not quite as accurate, but good enough for the purposes of our simulations.

2.3.2 Limitations

As just mentioned, the scheduler assumes a processor makes a reference every cycle - an approximation. Instructions of varying execution times all are assumed to execute in unit time. Moreover, time taken to execute register-to-register instructions is not reflected in the multiprocessor trace. The reference trace generated by the multiprocessor simulator, therefore, is an approximation to a true execution.

A related problem is that the post-mortem scheduler is not connected to the memory subsystem (caches, network and memory modules), so delays in any of these elements is not reflected in the execution trace. Delays, especially those due to hotspot contention, could seriously affect the synchronization reference behavior of processors. The simulator does take into account delays in one case (mentioned earlier): when processors perform simultaneous RMWs to the same location. Only one processor

can make a RMW to a location at any given time. All other processors executing a RMW to the same location at the same time execute a NOP.

2.4 Post-mortem Scheduler Design Features

Trace-driven simulation, whether for uniprocessors or multiprocessors, is an onerous process. The sheer amounts of data involved (in the hundreds of megabytes of trace data) demands that careful thought be given to the design of an efficient simulator. Following are some important features of the design of the post-mortem scheduler.

Perhaps the most important feature of the simulator's design is its use of direct access i/o when simulating the multiple processors' progress down the uniprocessor trace. As described in section 2.3, the scheduler first makes a pass through the entire uniprocessor trace and builds a data structure which points to the beginning of every task in the uniprocessor trace and contains a description of the task type. The data structure also contains entries for state information describing which processors have executed the section, etc. Please see Figure 2-5. Processors traverse this data structure, rather than the actual reference trace, to determine which tasks need to be executed. A task might contain over one hundred-thousand references. It is crucial that the scheduler use a higher level description of the trace in order to schedule processors. Once the processor has traversed the data structure and finds a task to execute, the scheduler uses direct access i/o[1] jump to the location in the uniprocessor trace pointed to by the data structure, and the processor can start picking up references, "executing" the section. This ability is especially useful for scheduling the execution of parallel loops. As figure 2-3 illustrates, processors are assigned iterations of a parallel loop. The iterations appear sequentially on the uniprocessor trace. Processors traverse the data structure until they find an unexecuted iteration (task). With direct access i/o they can jump to the location pointed to in the uniprocessor

[1]Both Unix and the VM operating systems have facilities to do direct access i/o. A C or Pascal application on these systems can use the *Lseek* function to jump to a desired location.

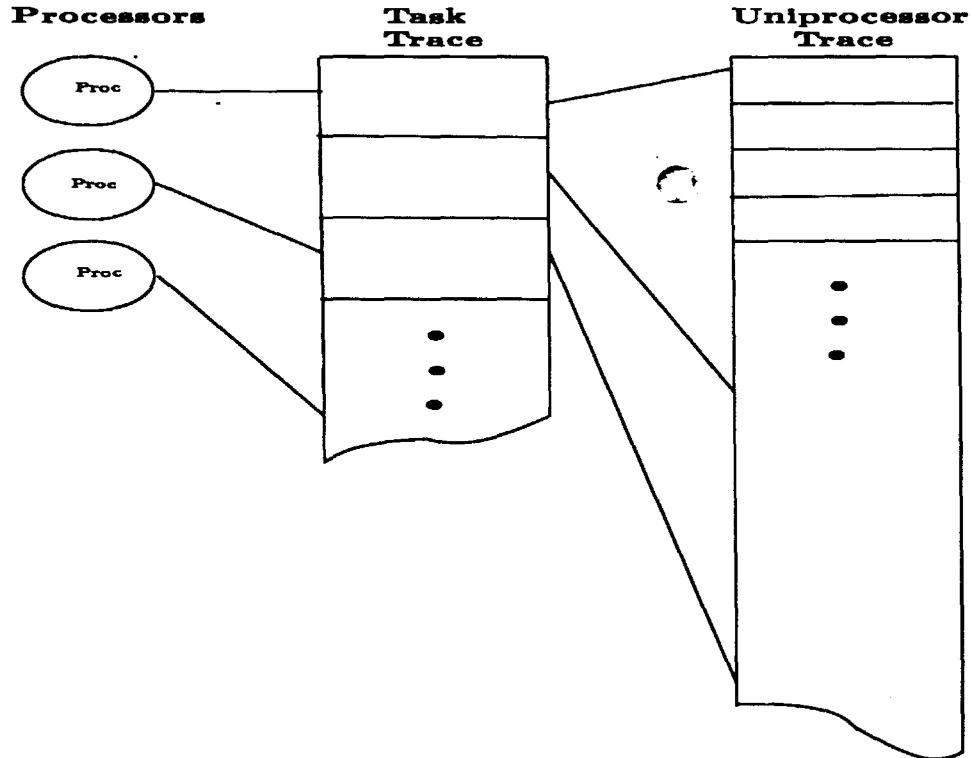

Figure 2-5: *Simulated processors traverse a task trace rather than the actual reference trace.*

trace and execute the task.

Another feature of the scheduler is that it can simulate different forms of synchronization at a barrier (wait). The scheduler contains a barrier routine which simulates the processor's behavior at a barrier (wait). If an implementation other than busy-wait is desired, this routine need simply be changed to reflect the desired behavior. The adaptive backoff synchronization techniques presented in this paper, for example, can be implemented in the waits.

Finally the scheduler was designed with a convenient interface to a network simulator. Rather than writing to disk the references made by each processor, and then having the network simulator read the trace, each processor in the scheduler has an output buffer from which the network simulator can directly read references.

2.5 The Cache Simulator and Network Model

Thus far, we have described only the simulation of the processor element in Figure 2-1. The multiprocessor traces produced by the post-mortem scheduler's simulation of processor can be fed into a cache simulator. We then make certain assumptions (described shortly) about an interconnection network in order to take statistics from the cache simulator and arrive at an approximate measure for network load - certainly an important metric in any evaluation of multiprocessor performance.

2.5.1 Multiprocessor Cache Simulator

The cache simulator implements n direct-mapped caches of a specified size and block-size which use a central directory to maintain cache-coherency in an n-processor system.

The cache takes input in the form of an interleaved multiprocessor trace which contains every processor's reference on a cycle-by-cycle basis. Figure 2-6 describes the input trace. Each 6-byte input trace entry contains a 8-bit processor id, and 8-bit tag describing each the kind of reference being made, and a 32-bit word for the actual reference. The cache simulator simply reads the input reference and takes the appropriate set of actions to update the cache and maintain coherency as mandated by the write-invalidate directory scheme.

The simulator's output is in the form of cache statistics, such as private and shared hits and misses, invalidations to clean objects, invalidations to read-only objects, etc. Parameters for cache size, line size, number of processors, and number of directory pointers can be chosen. If a variable is shared among more processors than available pointers, then the simulator uses a random method to invalidate a pointer and adds a pointer in its place to satisfy the new request. Processor NOPs result, of course, in no action taken for the cycle.

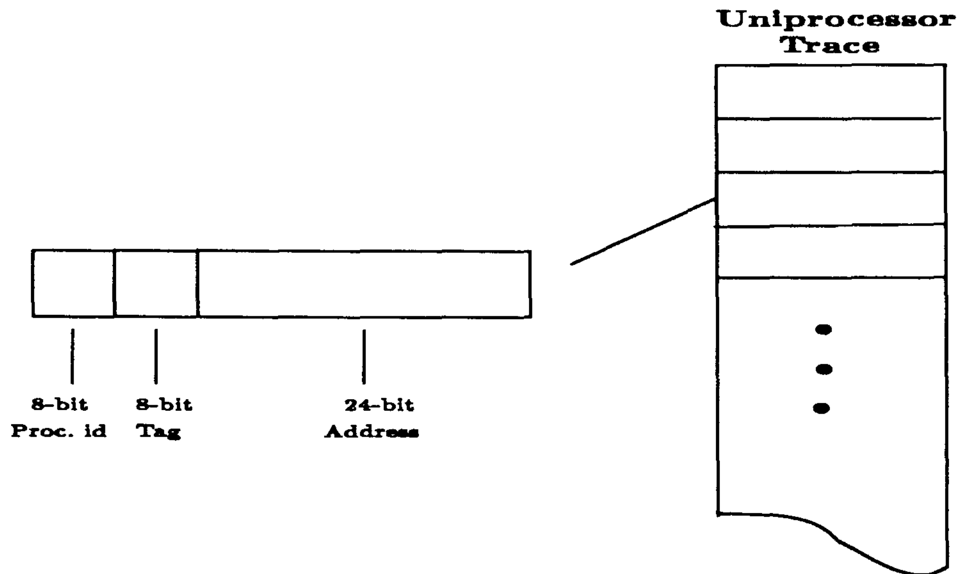

Figure 2-6: *The cache-directory simulator takes a 6-byte interleaved multiprocessor trace of the above form.*

2.5.2 The Network Model

In this thesis we do not actually simulate the interconnection network or memory modules of a multiprocessor system. We are, however, interested in the average request rates to memory and the percentage of it caused by synchronization references. Rather than running a detailed simulation which would take into account contention in the network, backpressure, and other factors (and would require further analysis), we simply count network transactions resulting from each reference in order to find an approximate figure for the network load. A network transaction is simply defined as a trip across a network. For example, in the event of a cache miss to an object not in any cache, two network transactions are generated: one to send the requested address to memory, and one to send the requested data from memory to the processor. In our model, all references, whether local or shared, traverse the network. Table 2.1 describes the possible reference events and their costs in terms of network transactions. We arrive at the average network load by multiplying the event frequencies (outputted from cache-directory simulator) by the network transaction cost per

Event Type	no. of Network Transactions	Blocks Transferred
dirninsm	2	mod+addr+mod+data
dirnrmwr	4	mod+addr+mod+addr+mod+data+mod+data
dirnrmnw	2	mod+addr+mod+data
dirnrmbn	2	mod+addr+mod+data
dirnwmwr	4	mod+addr+mod+addr+mod+data+mod+data
dirnwmnw	4	mod+addr+mod+inv+mod+mod+data
dirnwhnw	4	mod+addr+mod+inv+mod+mod
dirnwmbn	2	mod+addr+mod+data
dirnrmi1	2	mod+inv+mod
dirnwi01	2	mod+inv+mod
drtyrplc	1	mod+addr+data
syncsnc	2	mod+addr+mod+data

Table 2.1: Memory references resulting in network transactions.

event and dividing the sum of these weighted event frequencies by the total number of references (cycles) made. As described in [23], the network load can be used by a network model to compute processor utilization numbers.

LEGEND

Event type

```
dirninsm   Instruction miss
dirnrmwr   Read miss, block dirty in another cache
dirnrmnw   Read miss, block clean in another cache
dirnrmbn   Read miss, block not in any other cache
dirnwmwr   Write miss, block dirty in another cache
dirnwmnw   Write miss, block clean in another cache
dirnwmbn   Write miss, block not in any other cache
dirnwhnw   Write hit, block clean in another cache
dirnrmi1   Pointer invalidation due to too few pointers
dirnwi01   Pointer invalidation due to a write
drtyrplc   Memory update on replace of a dirty object
syncsnc    Uncached synchronization reference
```

```
Blocks
    Type        size            description
         (in 32-bit words)
    mod          .5             memory module number state info.
    addr          1             memory address
    data          1             cache blocksize
    inv      an invalidation
```

2.6 Summary of methodology

Listed below are the steps necessary to produce a multiprocessor trace using the post-mortem scheduler.

- Parallelize a Fortran application using the SPMD computational model. All independent loop iterations can be parallelized.
- Run the application on the EPEX/Fortran system under PSIMUL [32] to obtain a uniprocessor reference trace of the application. The memory reference trace PSIMUL outputs is arranged in blocks of 24000 bytes. Each trace entry is four bytes. The first entry in a block has the following format:

```
        FIELD    # of BITS      DESCRIPTION
---------------------------------------------------
        cpuid       8           cpuid (0 for uniprocessor run)
        blank      24      blank
```

  The subsequent 5999 entries have the following format:

```
        FIELD    # of BITS      DESCRIPTION
```

```
--------------------------------------------------
      tag          8          Fetch\&Add Reference     10ccc1xx
                              Synchronization Marker   01ccc000
                              Instr./Data Reference    00cccsxx
                              xx = 01        instr. fetch
                              xx = 10        data fetch
                              xx = 11        data store
                              s  =  1        shared ref.
                              s  =  0        private ref.
                              ccc            # of instructions
                                             between a ref.

      addr/marker 24          if tag is a marker, marker number
                              else, reference virtual address
```

For a description of the markers which surround the parallel, serial and replicate sections, please refer to [31].

- Simulate a multiprocessor execution using the post-mortem scheduler. Specify the number of processors desired.
- Use the multiprocessor trace to drive cache and network simulators.

Chapter 3

The Synchronization Problem

This chapter presents data from trace-driven multiprocessor simulations of the FFT [6], SIMPLE [7], and WEATHER [17] applications[1] and uses the example of a barrier synchronization to explain why synchronization is a problem in large-scale systems, cache-coherent or otherwise. First we describe the many different ways in which synchronization is implemented in machines and focus on the special case of barrier synchronizations.

3.0.1 Background

Several different approaches have been taken in addressing synchronization in shared-memory multiprocessors. Often machines have simple synchronization primitives in hardware which allow implementation of more sophisticated synchronization in software. This chapter first describes the hardware synchronization primitives machines use. Approaches to barrier synchronization are then described.

Figure 2-1 in chapter 2 describes the logical model of the system considered in this thesis: a large number of processors attached to memory modules via an interconnection network. Caching synchronization references in the system is optional. When

[1] See Appendix for a description of the applications.

caches are included, a mechanism for maintaining consistency of data in the caches must be maintained, often exacerbating the synchronization problem. This problem will be discussed in detail later. A more detailed description of the model actually simulated is described in Chapter 2.

Hardware Synchronization Primitives

Most multiprocessors have low-level synchronization mechanisms which provide some form of an atomic operation in order to allow exclusive access to shared variables. Higher-level synchronization mechanisms (such as a barrier or a fork) are then built on top of these hardware primitives in microcode or software. Synchronization can also be achieved via interprocessor interrupts, ie. a process can synchronize with a sister process by sending an interrupt to the processor on which the sister process is running. Interprocessor interrupts can be used in conjunction with lower-level hardware synchronization primitives as will be explained below.

Perhaps the most common class of hardware synchronization primitives used are so-called *read-modify-write* (RMW) primitives. Test-&-Set and Compare-&-Swap are examples of RMW mechanisms. See Figure 3-1. With both these primitives, a processor attempts to access a lock via an atomic read-modify-write to a shared location in memory. The higher-level synchronization microcode or software will typically repeat the operation if the lock is not obtained - a process called *busy-waiting* or *spinning* on a lock. In Test-&-Set the operation is successful, ie. the lock has been obtained, if the returned value is 0; in Compare-&-Swap success is indicated by the flag z being set.

Spin locks have two major drawbacks: 1) the processor is idle while it spins on the lock waiting for it to be released; and 2) spin locks generate a tremendous amount of traffic to memory which can result in bus or network contention. This spin traffic is especially deleterious in the case where more than one processor is spinning on the same lock (as is typically the case in barrier synchronizations). The adaptive backoff

```
Test_And_Set(lock)
{
 temp=lock;
 lock = 1;
 return(temp);
}
Reset(lock)
{
lock=0;                                                    10
}


Compare_And_Swap(r1, r2, w)
{
 temp = w;
 if(temp == r1) {
   w = r2;
   z = 1;                                                  20
 } else {
   r1 = temp;
   z = 0;
 }
}
```

Figure 3-1: *Test&Set and Compare&Swap are common examples of hardware synchronization primitives upon which more sophisticated synchronization events are built in software (eg, barriers).*

techniques proposed in this thesis can also be applied to these spin locks to reduce the number of fruitless synchronization spins performed as discussed in Chapter 5.

One alternative to spinning used in the C.mmp is to make use of interprocessor interrupts. If a processor unsuccessfully attempts to obtain a lock, it puts itself on a queue of processors waiting for the lock and disables all interrupts except for the interprocessor interrupt. When a process releases the lock it broadcasts an interprocessor interrupt, "awakening" the waiting processors. This type of lock is called a *sleep-lock* or *suspend-lock*. The advantage, of course, is that memory traffic is reduced. However, operating system overhead is incurred every time the processor goes

to sleep, and processor utilization is still expended.

Another hardware synchronization primitive similar to Test-&-Set and Compare-&-Swap is the Full/Empty bit synchronization primitive used in the HEP. Shared memory words in this system are tagged with an Full/Empty bit which essentially acts as a lock. A shared location can only be read when it has been updated and tagged as full. After a successful read, the tag is reset to empty. Conversely, writes to the location can only occur when the tag indicates empty. A successful write leaves the location tagged as full. This system also "spins" on a memory location when unsuccessful in obtaining it. Note that both the Test-&-Set and Full/Empty bit schemes can be implemented in hardware at the memory module.

The one common characteristic of the schemes described so far is that when a number of processors are trying access the same lock simultaneously, the processors are necessarily serialized because only one processor can own the lock at any given time. The Fetch&Add synchronization has been proposed to address this problem in the RP3 and NYU Ultracomputer. See Figure 3-2. The Fetch&Add synchronization in these machines makes use of a special combining interconnection network to combine *simultaneous* accesses to the same location in memory, thus updating the location only once. If N processors attempt to increment the same location in memory through a Fetch&Add, the location is updated once, with the sum of the N Fetch&Adds, and the processors are each returned values which correspond to an arbitrary serialization of the N increment requests. This feature is well-suited to applications such as accessing sequentially allocated queue structures or forking processes in a parallel loop whose iterations are independent. The SPMD paradigm of computation uses Fetch&Adds in parallel loop forks and barrier synchronizations.

Barrier Synchronizations

As mentioned earlier, a barrier synchronization requires that all processors executing the application arrive at the barrier before proceeding.

```
Fetch_And_Add(x,a)
{
 temp = x;
 x = temp + a;
 return(temp);
}
```

Figure 3-2: *Description of Fetch/Add Synchronization*

A typical implementation of a barrier might use a shared variable whose initial value is zero. Each processor arriving at the barrier increments the shared variable. If the variable attains the value N, implying that all N processors have reached the barrier, the processor can proceed. Otherwise, it repeatedly tests the barrier until the above condition is true. The increment operation on the barrier variable must be atomic. This implementation has the drawback that each processor attempting to increment the barrier variable must contend with all the others simply polling it to test for the proceed condition.

A better implementation, e.g., Tang and Yew's [33], splits the barrier into two shared variables: an incrementing variable (henceforth called the ***barrier variable***) initially set to zero, and a ***barrier flag*** variable also initially reset. An arriving processor increments the barrier variable. If the variable's value is less than N, the processor polls the barrier flag which is set by the last processor to reach the barrier. Even this scheme requires that the last processor to reach the barrier compete with the N-1 processors testing the barrier flag when it tries to set the flag. More importantly, however, in both implementations the shared variables involved are ***necessarily shared*** among all processors in the system. It is precisely this widespread sharing that impacts performance when scaling to large systems.

The shared variables, barrier variable and barrier flag, can be updated with any of the hardware synchronization primitives discussed above. In the RP3, for example, Fetch&Adds are used to increment the barrier variable and set the barrier flag. The RP3's combining network aids this barrier implementation because it combines all

simultaneous accesses to the same location (in this case, the barrier variable and barrier flag). Without this hardware feature, multiple processors spinning on the barrier flag would easily result in hotspot contention at the barrier flag's memory module. As shown in [26], this contention can result in tree saturation thereby reducing the utilization of all processors in the system. The only problem with combining networks is that they are expensive to build and are slow. An RP3 report [26] estimated that just a 2x2 combining switch would be from 6 to 32 times more expensive, in terms of switch size and/or cost, than a normal switch.

The problem of hotspot contention arising from processors spinning on the barrier flag can also be alleviated by caching synchronization variables. Processors spin on the barrier variable in their cache rather than making references across the network. Problems with this scheme arise when the barrier flag is set by the last arriving processor. The copies of the barrier flag in each cache must then be either updated or invalidated and read again depending on the system's cache-coherency policy. We will discuss the effects of barrier synchronization on cache-coherent multiprocessor performance in detail shortly.

One hardware scheme used in the PAX computer [16] completely avoids problems of memory contention and cache-coherency by implementing barrier synchronization with special hardware which allows for low-latency global synchronization. A high-level logical description of the hardware is as follows: upon reaching a barrier every processor sets a line connected to a large, central AND-gate. Once all processors have arrived, the AND-gate sets a global line which signals all processors to proceed. The primary considerations with this approach is the additional hardware complexity involved in having a centralized resource such as an AND-gate when trying to build a scalable architecture. Ideally we would like a scalable machine to be as modular (and distributable) as possible. Also the barrier will have to be flexible enough to deal with situations where multiple barriers are simultaneously executing among subsets of processors.

3.0.2 Synchronization References and Scalability

The widespread sharing that occurs with synchronization variables is not a problem when used in bus-based snoopy-cache multiprocessors. Because snoopy-cache-based protocols perform broadcast invalidates or updates, a variable shared among all processors generates no more traffic on the shared bus than a variable shared among only two processors. The limitation of snoopy-based schemes, however, is that they do not scale to large multiprocessor systems. Since these schemes require low latency broadcasts for cache coherence, as well as the ability to "watch" all bus transactions, they must use a shared bus for communication. A single bus cannot offer the bandwidth demanded by large-scale shared-memory multiprocessors.

Unfortunately, widespread sharing of synchronization variables can drastically impair performance in large-scale multiprocessors, cache-coherent or otherwise.

An Aside on Directory Schemes for Cache Coherency

First, let us consider multiprocessors with coherent caches, where a directory is used to keep track of cached copies of shared blocks. In general, for every memory block, a directory must store as many pointers as the number of processors (say N) in the system [4]. Such a scheme is termed $Dir_N NB$, for N-pointers-No-Broadcast in [3]. In practice, it is possible to maintain just i pointers ($i < N$) to yield the limited-directory scheme denoted $Dir_i NB$ [3]. Invalidations are forced to limit the cached copies of a block to i, or to gain exclusive ownership on a write. Results in [3] showed that during an invalidation situation, few invalidations were actually necessary. Results from our trace-driven simulations of 64-processor systems discussed below as well as the results in [34] corroborate the findings in [3].

Figure 3-3 shows an invalidation histogram for a 64-processor simulation of $Dir_N NB$ driven by a trace from the SIMPLE application. We also ran simulations on FFT and WEATHER application traces with 64 processors.[2] The simulations used

[2]See Section A in the Appendix for a description of the applications.

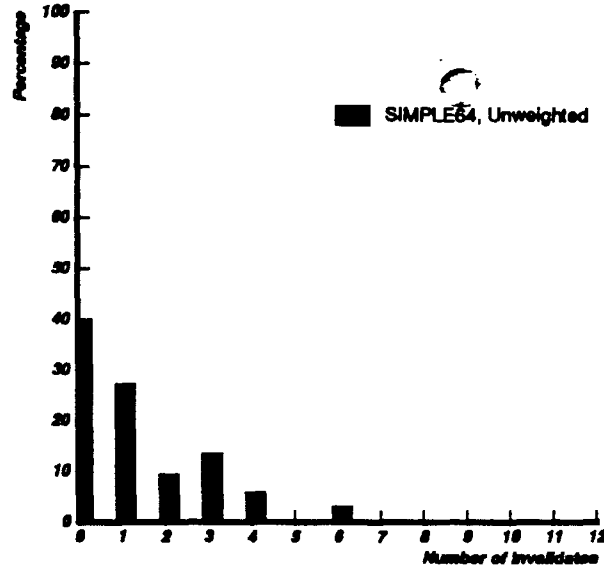

Figure 3-3: Cache invalidation statistics for SIMPLE with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages.

direct-mapped caches of size 256KBytes and block size 16 bytes. The graph shows the histogram of the number of invalidations required during a write to a previously clean block. We see that in over 95 percent of the times that an invalidation occurred (in both 16 and 64 processor simulations), a block had to be invalidated from no more than three caches. Figures 3-4 and 3-5 show the invalidation histograms for FFT and WEATHER; in both these applications the corresponding figure is over 99 percent. The graphs shows the percentage of writes which resulted in invalidations to up to 12 caches. Writes resulting in invalidations of greater numbers of caches were proportionately insignificant.

Figures 3-3, 3-4, and 3-5 essentially show the amount of sharing occurring among processors between writes. Since we are using a write invalidation scheme all sharing is ceased upon a write and processors start from scratch loading shared variables into their cache as needed. If we used a write-update scheme, sharing among processors

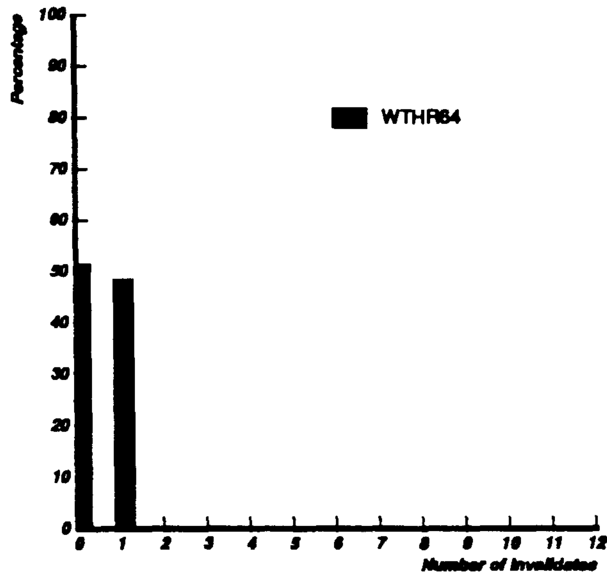

Figure 3-4: Cache invalidation statistics for WEATHER with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages.

would likely be greater than the above graphs indicate because shared variables never get invalidated unless replaced by another block.

Another important parameter which can affect the invalidation graphs is the cache block size. The larger the block size, the more the block can potentially be shared among processors. This statement is particularly true if an application exhibits fine grain sharing, ie. processors frequently contend for one or more words in the same block. In these cases, the overhead of write-invalidation schemes as described above can be very high if a small block size is not used. On the other hand, if an application has good per processor locality, then write-invalidate protocols perform better [9]. The effect of block size on the performance of cache-coherent multiprocessors is an important area of study. For the purposes of this thesis, we perform all our simulations with a fixed block size of 16 bytes.

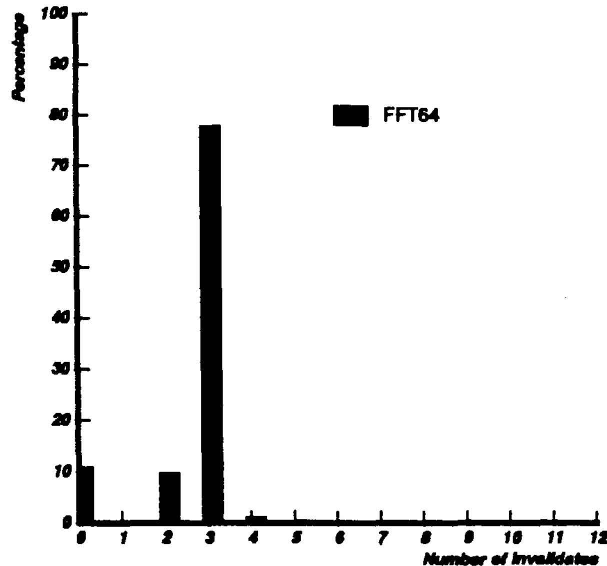

Figure 3-5: Cache invalidation statistics for FFT with 64 processors. The height of a bar at x reflects the fraction of write hits to previously clean blocks that resulted in x invalidation messages.

3.1 Why do synchronization references hurt performance?

Our simulations revealed that synchronization variables were largely responsible for the cases in which more than four caches were invalidated. Barrier synchronizations, for example, required sharing of the barrier flag among all processors, Thus, when the barrier flag was updated by the last processor, an invalidation to all caches in the system was required.

The true damaging impact of synchronization references, however, is observed only when the effect of simultaneous read sharing is considered. Recall that using i pointers limits simultaneous read sharing of a block to only i copies, and invalidations must occur to enforce this rule. For synchronizations like barriers, active sharing might occur among all processors involved, resulting in a high invalidation rate in

Ptrs	SIMPLE		WEATHER		FFT	
	NS	S	NS	S	NS	S
2	8.5	93.5	1.9	99.9	6.7	99.0
3	7.1	81.3	1.7	99.9	5.0	99.0
4	6.0	81.1	1.5	99.9	3.5	98.9
5	5.2	99.9	1.5	99.9	3.5	98.8
64	.53	1.2	1.2	.03	3.5	3.5

Table 3.1: **Percentage of non-synchronization (NS) and synchronization (S) references that cause invalidations in directory schemes with 2, 3, 4, 5, and 64 pointers in a 64 processor system. Synchronization references comprised 0.2%, 7.9%, and 5.3% of the data references in FFT, WEATHER, and SIMPLE respectively.**

directory-based schemes.

Table 3.1 shows the fraction of synchronization references out of the total number of synchronization references which resulted in an invalidation. The percentage is far higher than the corresponding fraction for non-synchronization data references. Note that the percentage of synchronization and non-synchronization references that cause invalidations decrease dramatically for SIMPLE and WEATHER in the 64 pointer case. This implies that most of the invalidation occurring is due to widespread read sharing. The FFT application clearly illustrates this fact. The percentage of non-synchronization references causing an invalidation stays constant at 3.5 percent after three pointers, indicating that non-synchronization variables get shared among no more than three pointers (between writes). The remaining 3.5 percent of references that cause invalidations in the four, five, and sixty-four pointer cases, are due to write invalidations. The percentage of synchronization references for FFT that cause invalidations, on the other hand, remains constant through the five pointer case and drops dramatically for sixty-four pointers. This behavior is directly due to the widespread sharing of the barrier synchronization variables in FFT. Since the barrier variable is shared among all the processors, it thrashes the pointers in the cases where only a few pointers are available - resulting in a high invalidation rate. In the sixty-four pointer case, however, there is no invalidation due to read-sharing.

It is clear that invalidation traffic due to synchronizations can be deleterious to

the performance of cache-coherent multiprocessors. One solution is to use software combining trees. Alternatively, one can disallow caching synchronization variables.

3.1.1 Disallowing Caching of Synchronization Variables

If most synchronization accesses cause invalidations that involve multiword transfers, then why cache synchronization variables? The problems with this approach are similar to those in multiprocessors that make all shared locations uncacheable: increased network traffic and potential hot-spot contention. Synchronization references are often to the same location in memory, and even a small fraction of all data accesses to the same "hot" module can cause tree saturation [26] in the interconnection network and a corresponding severe drop in the effective memory bandwidth.

Table 3.2 shows that the percentage of uncached synchronization traffic to memory out of the total data traffic can be large. Section 2.5.2 describes how we computed traffic to memory. Briefly, we compute traffic to memory by summing the total number of network transactions generated by references. For example, in the case of a cache miss, two network transactions are generated: one to send the requested address to memory and one to send the requested data from memory to the processor.

The reason SIMPLE and WEATHER generate far more synchronization traffic than FFT is that their load balancing is not as good as in FFT (see Section A for details), resulting in more synchronization accesses at loop barriers as processors wait for all processors to arrive. The slight relative increase of synchronization overhead in all cases when going from two to five pointers is because synchronization traffic remained constant while invalidation traffic (part of total memory traffic) decreased as more pointers were available for sharing of blocks. The fact that synchronization overhead increased for SIMPLE when going from 5 to 64 pointers indicates that SIMPLE has relatively more invalidation than WEATHER or FFT. The synchronization overhead for WEATHER was particularly bad because the number of processors running the application was often greater than the parallelism afforded in the application,

Ptrs	SIMPLE	WEATHER	FFT
	Traf. (%)	Traf. (%)	Traf. (%)
2	22.0	55.4	1.3
3	23.5	56.3	1.4
4	24.6	57.4	1.5
5	25.6	57.6	1.5
64	35.3	59.9	1.5

Table 3.2: Synchronization traffic to main memory as a percentage of the total traffic when the synchronization variables are not cached. Block size is 16 bytes and cache size is 256KBytes. The non-synchronization blocks are cached and coherence is maintained using directory schemes with 2, 3, 4, 5, and 64 pointers.

resulting in synchronization traffic as processors sat busy-waiting for their brethren to arrive. In this case, the WEATHER simulation is a good example of the unfortunate effects of poor load-balancing. FFT, on the other hand, represents the ideal case when synchronization overhead can be very low as Table 3.2 shows. Though the execution was load-balanced, synchronization traffic in FFT still accounted for 1.2 percent of all references. Considering that most of this traffic is to the same location in memory, we cannot ignore the potential for hotspot contention and the resulting tree saturation. The SIMPLE data is probably the most representative example of a typical parallel execution. SIMPLE contained a number of serial sections and parallel sections which did not contain n-way parallelism - typical of most applications.

Therefore, if we are to scale multiprocessors, network traffic due to synchronization must be rigorously minimized.

In large-scale shared-memory multiprocessors, such as the RP3, Ultracomputer, Cedar, all traffic to shared variables must go over the network[3], and the relative fraction of network accesses attributable to synchronization is slightly smaller. We measured memory traffic when shared variables were not cached and found that synchronization traffic accounted for 25.5%, 49.2%, and 1.47% of the total traffic in SIMPLE, WEATHER, and FFT, respectively, in 64 processor simulations. Our mo-

[3]Although temporary caching of shared locations with compiler inserted cache flush directives can help relieve network load.

tivation for reducing the network traffic, especially traffic that is partial to a specific memory location, still remains.

Chapter 4

Analysis

In this chapter we apply adaptive backoff techniques to barrier synchronizations and evaluate their performance. The evaluation is carried out via an analytical model and via simulation of a barrier synchronization model. First we describe adaptive backoff synchronization techniques as applied to a barrier evaluation model. We then describe the model of the network used. Finally we discuss the results of simulations and the tradeoffs involved in using adaptive backoff techniques.

4.1 Adaptive Backoff Barrier Synchronization

The basic idea behind adaptive backoff methods is simple. An adaptive backoff barrier technique makes use of available information in deciding how long to wait before trying to read a barrier flag rather than continuously polling the flag.

We will assume barriers implemented using a separate barrier variable and a barrier flag as described earlier. If the barrier variable and flag are one and the same object, the relative advantage of using adaptive backoff techniques will be even greater.

4.1.1 Previous Work

As mentioned in section 1, the fundamental idea of backoff has been applied in many situations previously. The majority of the work has been in developing adaptive control algorithms for transmission across communications networks. These schemes typically involve backing off by random intervals with means determined by collision history.

We do not use this method to determine backoff intervals for two primary reasons. First, we want the backoff to be as efficient as possible, ie. backoff decisions must cost just a few instructions. Calculating a random backoff interval could involve hundreds of instructions - too high an overhead for our liking.

More importantly, though, backing off by a random interval would result in increased contention at memory modules because it destroys serialization among the processors. When multiple processors initially attempt to update the barrier variable simultaneously, they are necessarily serialized (we do not use a Fetch&Add). This serialization remains for the duration of the barrier synchronization while processors poll the barrier flag and backoff. We desire serialization because it reduces contention, especially when the processors attempt to poll the barrier flag. If we used a backoff method with a random wait interval, all serialization is destroyed if processors come back to poll the barrier flag at the same time. The processors are serialized again as they contend to access the barrier flag, and the process repeats itself.

4.1.2 Backoff on the barrier variable

The first method, called ***backoff on the barrier variable***, is the simplest and attempts to make use of the state of the barrier variable to reduce unnecessary network accesses on the barrier flag. The barrier variable value reveals the number of processors waiting at the barrier. Let there be N processors that must arrive at the barrier, and let the average memory access time over the network be 1 cycle as mentioned earlier. If i processors have reached the barrier, then an arriving processor can start polling the

barrier flag at least (N-i) cycles after reaching the barrier variable A.

4.1.3 Backoff on the barrier flag

Backoff on the barrier flag is another method that tries to reduce the number of spins on the barrier flag. Processors can remember the number of times they have polled the barrier flag and correspondingly backoff by a linear or exponential amount the longer they have waited. This code can be part of the barrier implementation in software and needs no hardware support. In discussing the performance of these latter methods, we assume that backoff on the barrier variable is also applied.

In backoff on the barrier variable, if the interarrival times of processors are very large, then a processor might wait its N-i cycles and start polling the barrier flag long before the last processor arrives at the barrier. In these situations, we might wait longer before polling the flag, say (N-i)+C or (N-i)*C, where C is some positive integer. While this might reduce the number of unnecessary network accesses, it might also cause the processor to remain idle and miss accessing the barrier at the earliest it becomes available. We suggest some methods of choosing appropriate backoff parameters in Section 4.6.

In backoff on the barrier flag, there exists a danger of backing off much more than necessary. Clearly there is a tradeoff between network access reduction and cpu idle time. If only a small fraction of the processors are involved in a barrier synchronization, then to reduce the hot-spot contention problem, one might prefer to take the hit in cpu idle time for these contending processors so that the remaining processors in the system can perform unhindered. As mentioned before, even a small percentage of memory references to the same "hot" memory module can result in severe congestion in the interconnection network, thereby reducing all processors' utilization [26]. Of course, if all processors in a system are involved in a barrier synchronization, then the cpu idle time becomes an important consideration.

Backoff decisions are made only when a process has just updated the barrier

variable, and when the process has read the barrier flag and the flag is not set. So, once a processor initiates a barrier read request, the network controller for that processor attempts to read the barrier. If contention thwarts this attempt, the access is repeated until the flag is read. We propose some other schemes where the network controller can back off if network congestion is high.

For software-tree based implementations of barriers on non-cache-coherent multiprocessor as suggested by Yew, Tzeng, and Lawrie [35], our methods can still be used to reduce the spins on the intermediate nodes of the tree. Of course, in a cache-coherent multiprocessor with more pointers than the degree of the nodes, all intermediate nodes can be cached and backoff is not necessary.

We evaluate these ideas using a barrier model through analysis and simulations and discuss the tradeoffs between reduced synchronization accesses and wasted cpu cycles. First, though, we must describe the model of the network used for the simulations.

4.2 The Network Model

The network model we assume is the following: processors can access any memory over the network in one network cycle. We do not model network contention, but do model contention for the barrier variable and flag. We also assume that the barrier variable and flag are in different memory modules, so simultaneous requests to the two by different processors can be satisfied. We assume that in a network cycle only one processor can access the barrier variable or the barrier flag. If a processor is denied access to the variable in a network cycle it repeats the access to the variable in the next network cycle. This model might correspond to a crossbar switch where the only contention is for the end memory modules that have the barrier variable and flag. It also roughly approximates the performance of a circuit-switched multistage interconnection network, where the network cycle time can be the round-trip time

over the network. In the latter case the contention at intermediate network nodes is not included.

The network traffic rates computed using our barrier scheme might also be input into a more complex model of a multistage interconnection network such as that proposed by Patel [24] if network contention results are desired. Unfortunately Patel's model does not account for hot-spot contention. We are also using large parallel traces of real applications derived using various synchronization schemes to drive network models to obtain performance estimates in the presence of hot-spots caused by barrier traffic and when the barrier traffic is reduced using our techniques.

4.3 A Barrier Model

We first describe the model that we use to evaluate barriers. We use two metrics: (1) the number of network accesses per process in accessing the barrier variable and barrier flag; and (2) the number of cycles that an average process spends from the time it arrives at the barrier to the time it is allowed to proceed from the barrier.

Overall performance is impacted by the total network traffic, which includes the regular non-barrier traffic and the barrier traffic. Because we currently do not model hot-spot traffic contention in the network, we preferred to present the numbers for the barrier traffic alone, as average numbers for overall traffic might be misleading in terms of the adverse effect of the barrier traffic focused on one memory module.

Let us define A to be the time interval during which processes can arrive at the barrier. A is the time from the first processor's arrival at the barrier variable to the last processor's arrival. The complementary interval between these two events we call E, i.e., the time between barriers in an application. If we were to follow an application's execution through time, E and A would appear as shown in Figure 4-1.

Table 4.1 measures A for our applications. A is defined to be the number of cpu cycles from the time the first processor starts polling the barrier flag to the time

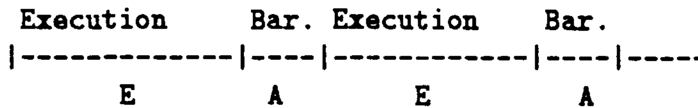

Figure 4-1: Intervals of execution and synchronization.

Application	Processors	A	E
SIMPLE	16	7021	42007
	64	7068	6195
WEATHER	16	82754	495298
	64	82787	82716
FFT	16	237	228073
	64	285	57997

Table 4.1: Average number of cycles, A, between first and last arrivals at waits and barriers. E is the average number of cycles between the last arrival at the previous barrier (or wait) and the first arrival at the next barrier (or wait), i.e. it is the average time between barriers or waits.

the last processor sets the barrier flag. Interestingly the average A for SIMPLE and WEATHER did not increase as greatly as for FFT when going from 16 to 64 processors. For highly uniform and load-balanced applications such as FFT the spread among arrivals is primarily due to the serialization which takes place at the loop index assignment. Thus, FFT was relatively more affected than the other applications when the number of processors increased.

E and A for SIMPLE and WEATHER with 64 processors are similarly sized because the applications were not perfectly load-balanced: Not all the parallel loops contained a nice multiple of iterations which could be distributed evenly among all processors. The few processors without work proceeded directly to the barrier.

The barrier model we use for our analysis and simulations is actually slightly different from the A we measured, and allows us to model a varying number of synchronizing processors for a given value of A. Our measurements of A from the applications were for a relatively large number of processors and this measurement yields an indication of the maximum time span between the first and last arrival at

a synchronization point in that application. It is likely that a smaller number of processors can have an actual value of A much smaller than this maximum span. Therefore, we now define A to be the interval during which processors *may* arrive at the barrier, and N to be the number of synchronizing processors. We further assume that each processor has a uniform probability of appearing at any time instant during the interval A. From the uniform probability of arrival during the interval A we must compute the average time span between the first and last arrivals out of a total of N arrivals. This span must tend to A as N becomes large.

To determine whether our assumption of uniform probability of arrival within A was reasonable we measured the arrival times in our applications. We found that the distribution is roughly uniform for FFT but is skewed towards the beginning and the end of the interval for SIMPLE. This skewing occurs because of uneven load-balancing. We observed, however, in the last peak that processor arrivals were still uniform over the last 200 references [2]. There seems to be no real pattern and our assumption of a uniform distribution is not expected to significantly change our results for minor variations in the arrivals. We also present additional validation of this model by comparing the predictions obtained through simulations using the model and through measurements using the actual traces in Section 4.5.1.

4.3.1 Analytically Estimating Barrier Performance

We first present some simple calculations for extreme cases of A to determine the bounds on the possible savings and to provide insight into our simulations.

When all processors arrive simultaneously ($A = 0$) and no backoff, a processor will make on average $N + N + N/2$ synchronization references. Each processor makes $N/2$ barrier variable references, polls the barrier flag $N/2$ times before the last processor gets through the barrier variable, continues polling the barrier flag N times until the last processor sets the flag, and finally leaves after $N/2$ references, on average. We denote this model that assumes simultaneous arrival as Model 1.

If $A >> N$, there is practically no contention to get the barrier variable. In this case we assume that processors appear at the barrier at a given time instant within the time interval A with uniform probability. Let us first compute the average time span τ between the first and the last arrival within the interval A given N processors. The average time from the beginning of the interval to the first arrival can be shown to be $A/(N+1)$, and the average time from the last arrival to the end of the interval to be $AN/(N+1)$. The required time span τ is the difference of the two, or

$$\tau = A\frac{N-1}{N+1} \tag{4.1}$$

Observe that τ approaches A as N becomes large. Thus, each processor make on average $\tau/2 + N + N/2$ network accesses during the synchronization phase. This is Model 2.

Let us now consider backoff on the barrier variable, where we backoff an amount proportional to the the barrier variable value. If i is the value of the barrier variable upon a processor's arrival, then the processor can wait $N - i$ cycles before beginning to poll the barrier flag. When $A = 0$, the average number of synchronization accesses becomes $N/2 + N + N/2$ cycles because the processor does not start polling the flag until the last processor gets through the barrier. A similar savings of N/2 is made for $A >> N$. With backoff only on the barrier variable, the potential savings get smaller as A gets larger because the savings is a constant $N/2$ no matter what A is.

Backoff on the barrier flag uses the number of times the flag has been polled. Rather than continuously polling the barrier flag until it is set, we backoff by some function of the number of times we have already read the shared variable. From Model 2 for $A >> N$, the potential savings in network accesses can be as large as $log_b(\tau/2)$ for exponential backoff, where b is the basis of the exponential backoff algorithm used. In addition we reduce interference with the final processor write request that will release the processes waiting on the flag. The backoff on the barrier flag can also incur a high penalty – we might backoff too far and waste cpu cycles. This idea is tested out in simulations discussed in the next section.

Finally, we present some network access rates for barriers on multiprocessors with hardware support for barrier synchronization to provide a basis for comparison with the backoff schemes. Examples of such hardware support are a bus to allow either global invalidations, or global updates, of cache entries, a directory with a full pointer map, and special logic to implement a global synchronization gate as suggested by Hoshino [16]. If there are N processors the invalidating bus incurs $3n+1$ accesses for a barrier: n fetches of the barrier variable, n invalidations for n writes of the barrier variable, n fetches of the flag, and the final global invalidation caused by the write into the barrier flag, yielding roughly 3 accesses per processor per barrier operation. The updating bus uses roughly 2 bus accesses per processor. The same number of accesses applies to an invalidating scheme that can detect a fetch with intent to write. Like the bus, the directory scheme incurs $3n$ on barrier variable accesses and invalidations, and flag accesses, but lacking a global broadcast must incur an additional n for the individual invalidates on the final write to the barrier flag, yielding 4 on average per processor per barrier operation. The Hoshino scheme uses n accesses to the global synchronization gate and the final single broadcast message to the participants to inform them to proceed, for a per-processor average of 1.

4.3.2 Simulation Methodology

We also use simulations to predict barrier performance with and without backoff. The barrier and network models are the same as described previously. Our simulation methodology is described here.

In our simulations, processors arrive with uniform probability during the interval of size A. Each processor first increments the barrier variable and then spins on the barrier flag until it is set by the last arriving processor. Our previous data in Table 4.1 showed that for three applications the value of E was between 6195 and 495298 cycles on average and the value of A was between 237 and 82787 cycles. While we simulated a wide range of A values, we show the results for $A = 0$, 100, and 1000 for brevity

because it is only the relative size of the interval with respect to the number of processors involved in the barrier that is important; larger values of A yielded no additional insight.

Each simulation run measured the average number of network accesses made by a process from the time it arrived at the barrier variable to the time it proceeded from the barrier flag after having successfully tested the flag and observing a true value. As mentioned before, the number of network accesses includes contention for the barrier. We also measured the average time each process spent from the time it arrived at the barrier to the time it left.

4.4 Evaluation

We evaluate the backoff methods using the models just described. This section first compares the predictions of the model with simulations. We then estimate the potential savings in network traffic using backoff techniques and discuss the tradeoffs involved in choosing the right parameters for the backoff algorithm.

4.4.1 Estimating the Potential Reduction in Traffic

We will first analyze the accuracy of our simple model in predicting the behavior of the barrier synchronization under various load conditions. The model will indicate the range of performance gains that we might expect using the backoff techniques and give insight into our simulation numbers.

Figure 4-2 compares model predictions of network accesses with simulation results for $A = 0$, $A = 100$, and $A = 1000$, without backoff. The model can be modified to predict the performance of the backoff schemes, but for certain cases it can get quite complicated. We will, however, mention what terms in the model equations get impacted by the various schemes.

The network accesses for $A = 0$, $A = 100$ do not differ much overall, but the way

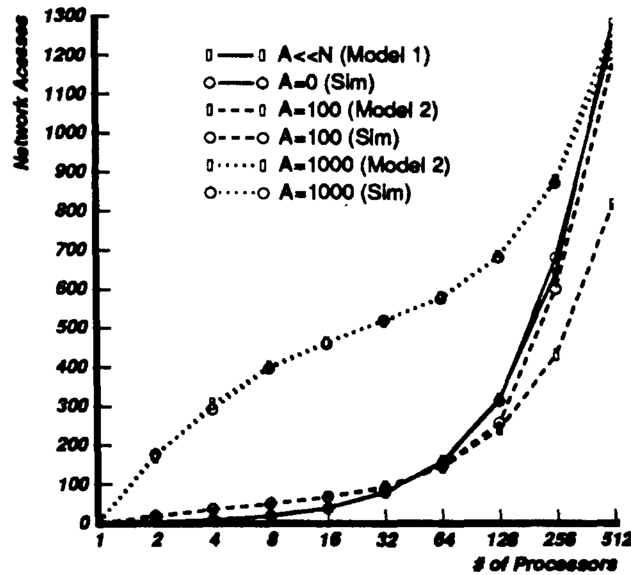

Figure 4-2: Comparing the predictions of the analytical model and predictions of barrier performance.

in which they differ is significant. For $N < 32$, $A = 0$ results in fewer accesses than $A = 100$ because when $A = 0$ processes do not have to wait for the last processor to arrive at the barrier. For larger N, however, $A = 100$ starts performing better because when the arrivals are spread out slightly, there is less contention in accessing the barrier. We observe a similar behavior for $A = 1000$ as N approaches A. When N is small, $A = 1000$ makes far more accesses than $A = 0$ or $A = 100$.

The model is accurate as the figure shows. Model 1, as expected, matches the curves for the $A << N$ cases. In particular, Model 1 closely approximates the $A = 0$ case, and yields a good match with the $A = 100$ curve for $N > 16$.

Model 2 matches all the cases where $A >> N$. Specifically, the Model 2 curve for $A = 1000$ provides a near perfect match with the corresponding simulation curve for all the values of N shown. The Model 2 curve for $A = 100$ matches the simulation $A = 100$ curve for $N < 128$. When N is greater than 128, the model begins to underestimate the contention in accessing the barrier variable. In general, the maximum of the predictions of the two models yields a good fit with simulation in all ranges.

Where $N > A$, the model implies that the potential reduction in network traffic is

20%. When $A > N$, the potential gains are much more significant. If an exponential backoff method is used with base b, then if the network accesses of the flag were M, with backoff these accesses can be reduced to the order of $log_b(M)$. Because the waiting processes are not busily accessing the flag, the final process that must set the flag can usually proceed to update the flag without interference.

4.4.2 Simulation results

We now present simulation results for barrier synchronization performance. Figure 4-3 shows the net accesses for N ranging from 2 through 512 when $A = 0$, i.e., when all processes arrive at the barrier at the same time. The curve follows the model as shown before, which means that the net accesses increase as $5N/2$, where N is the number of processors. The curves for backoff on the barrier variable alone, and backoff on the barrier flag with backoff constant 2, 4,and 8 are also shown (as mentioned before, all our simulated cases of backoff on the barrier flag include first backing-off on the barrier variable.)

Figure 4-3 corresponds fairly well with our model's prediction of the reduction in synchronization references due to backing off on the barrier variable. Backoff on the barrier variable reduced the number of network accesses from 160 to 132, a 15% reduction. Not surprisingly, backoff on the barrier flag made no difference because everyone reaches the barrier at the same time when $A = 0$.

Backoff with $A = 1000$ often has a savings greater than the log of the time interval of arrival at the barrier because of reduced interference with the final write request into the flag. This phenomenon also explains the fewer network accesses for backoff with base 8 at A=1000 than at A=0 for 32 processors. However, this savings often comes at the expense of increased processor waiting times.

Figures 4-4 and 4-5 correspond to the network accesses by a process for $A = 100$ and $A = 1000$ respectively. In Figure 4-4 for the backoff on the barrier variable we see similar savings as in Figure 4-3 with $A = 0$ because the interval A is still not

very big compared to the number of processors. Note, however, the big reductions that the exponential backoffs on the barrier flag gave. With $A = 100$, not everyone reaches the barrier flag simultaneously, so the ones who arrive early backoff by a large value. For example, with 16 processors and a base 4 backoff on the barrier flag, we see a savings of over 90% in network accesses. For 64 processors, base 8 backoff yields savings of about 60%.

The proportional benefit due to backoff decreases as N increases because contention in the network to access the barrier flag becomes a sizable portion of the network accesses. Recall that an unsuccessful network access in accessing the barrier flag is still counted as a network access. (To reduce these unsuccessful accesses one might use backoff techniques in the network accessing as discussed later.) For example, when $A = 100$ and $N = 512$, base 8 backoff yields less than a 30% reduction in network accesses.

Backoff on the barrier variable alone, for $A = 1000$, offers only modest savings. Interestingly this scheme offers virtually no savings for up to 32 processors, because few processors contend for the barrier flag, but the savings become more significant as the number of processors increases. For 256 processors, for example, backoff on the barrier variable yields about a 15% improvement.

The savings due to exponential backoff on the barrier flag with $A = 1000$, however, are quite dramatic. Since the processors potentially have a large interval to poll the barrier flag before everyone arrives, over 95% savings in network accesses results with binary backoff on the barrier flag with 16 processors. The 64 processor case offers a similar improvement.

The small number of network accesses with backoff on the barrier flag for the cases $A = 0$ and $N < 8$, $A = 100$ and $N < 32$, and $A = 1000$ and $N < 128$, compares reasonably with the network accesses in the bus-based schemes, the broadcast based schemes, or the Hoshino scheme, with no extra hardware or the broadcast requirement. However, when A is smaller or N is larger, the backoff schemes tend to do much

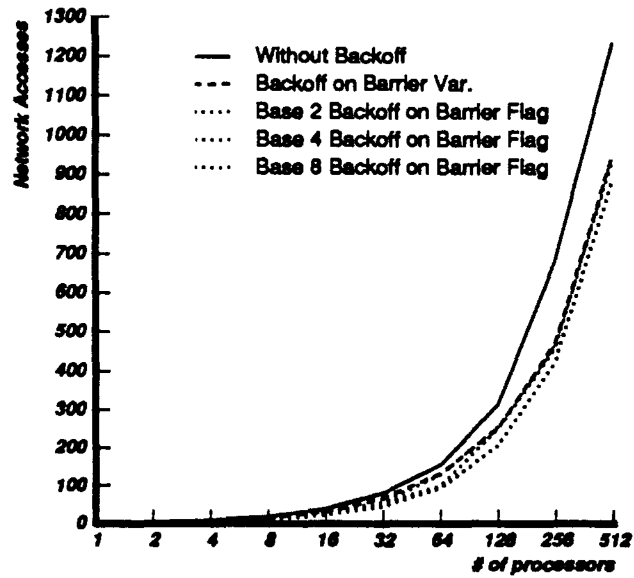

Figure 4-3: Performance of backoff algorithms for $A = 0$.

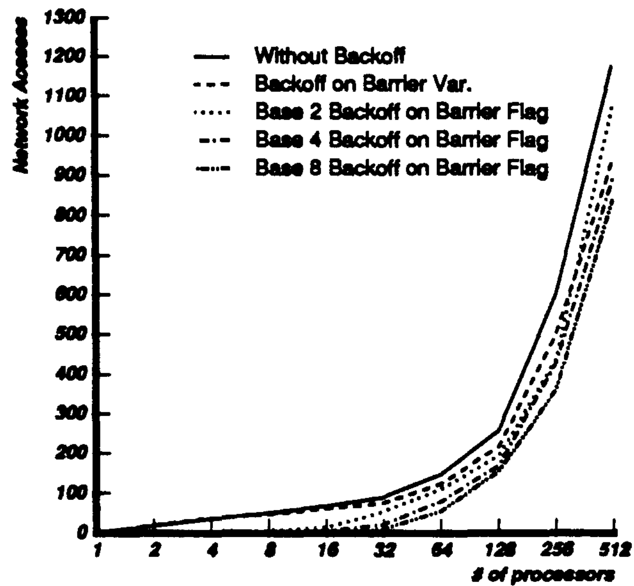

Figure 4-4: Performance of backoff algorithms for $A = 100$.

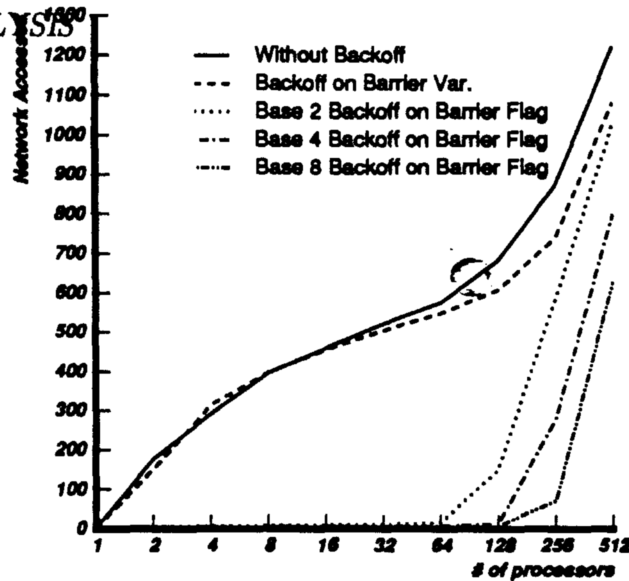

Figure 4-5: Performance of backoff algorithms for $A = 1000$.

worse than the schemes that have special hardware support for synchronization.

It is clear that backoff on the barrier flag is potentially much more beneficial for large A because most of the network accesses that happen while the processes await the remaining processes to arrive at the barrier can be obviated. These accesses correspond to the first term in the Model 2 equation. Backoff on the barrier variable alone does not impact performance significantly when N is small compared to A, but can yield up to a 20% improvement when N is large.

It is interesting to see that the network accesses increase dramatically for $N = 128$ ($A = 1000$). It seems that the backoff techniques are not as useful in this case (improvement is less than about 30% for $N = 256$ and backoff with base 2), although for these cases barrier synchronization is probably inappropriate anyway without some form of distributed software combining [35]. Our backoff methods can still be used on the intermediate nodes of the combining tree. The reason for the sharp increase can be described as follows: When the number of processors is small compared to A, a process can get access to the barrier flag usually within one network access. However, when the number of processors is not small compared to A, then a process will suffer contention in trying to access the barrier flag, and contention shows up as repeated network accesses.

In both cases the network accesses can be dramatically reduced for $N < 128$. For larger N, when the contention due to multiple processors simultaneously accessing the barrier increases, the relative benefit decreases. Recall, we do nothing about these contention accesses. A method described in the next section can help reduce this problem.

Our simulations show that using a backoff method on both the barrier variable and the barrier flag can yield savings from 20% to over 95% of the network accesses. However, the reduction in network traffic using the backoff methods does not always come for free. Because a backoff method can cause unnecessary processor idle time, we must carefully analyze the delays that these techniques can introduce. The occurrence of delays alone might not be a major cause for alarm, because these delays correspond to the delays suffered by the synchronizing processes alone, and do not affect other processes. The next section addresses these issues.

4.5 Discussion of Tradeoffs

An appropriate backoff constant must be determined by trading off the reduction in network accesses with the potential increase in the number of cycles the cpu spends idling during backoff.

We determined from our simulations the average waiting times per processor when $A = 0$, 100 and 1000 for four cases: without backoff, with backoff on the barrier variable and with exponential backoff on the barrier flag with bases 2, 4 and 8. The waiting time for a process is computed as the number of cycles between first arriving at the barrier to when the process finds the barrier flag set.

For $A = 0$, and $A = 100$, we found that the waiting times for all the four curves are similar because the opportunity for a large backoff time is rare given that all the processes arrive within a 100 cycles of each other. The waiting time in these cases is proportional to the number of network accesses, as it is precisely these network

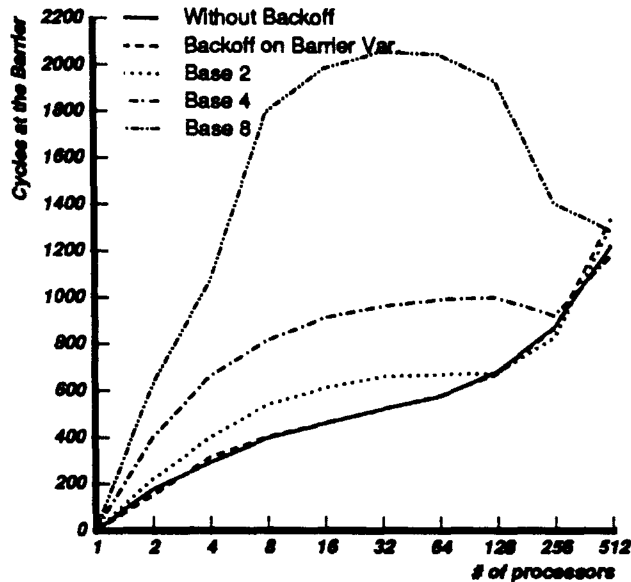

Figure 4-6: Processor waiting times for backoff algorithms for $A = 1000$.

accesses that give rise to the delays at the barrier.

We found that in all cases binary backoff provides a favorable tradeoff between large reductions in synchronization references and contained increases in wasted cpu cycles. Consider, Figure 4-6, which shows average processor waiting times for $A = 1000$. For 64 processors, binary backoff decreased synchronization accesses by 97% while increasing the time spent at the barrier by only 16%.

However, the average time spent idling can increase dramatically when both A and the base of the backoff algorithm are large because of the possibility of large backoff times. As an example, for 64 processors and $A = 1000$, the waiting times without backoff and with base 8 exponential backoff on the flag are 576 and 2048 respectively – depicting an increase of over 350% due to backoff. Even in this case, one important benefit is that the barrier accesses are both reduced and spread out uniformly over time.

When the arrival interval A is much larger than the number of processors, and a high processor utilization is important, one can modify the backoff algorithm as follows. If the backoff amount crosses some preset threshold, then it might be worth-

while to place the process on a queue pending the arrival of the last process. The enqueuing operation incurs a constant overhead that might be unnecessary should the processes arrive within a small interval. Because A cannot often be determined a priori, such a method of deciding when to put a process to sleep seems promising.

Interestingly, for $A = 1000$, the average waiting times per processor reach a maximum around 64 processors and then actually decline as N increases. When the number of processors is small compared to A, the processors can test the flag without excessive contention with other processors. After each unsuccessful test, they back off, and the backoff time is exponentially related to the number of unsuccessful tests. Because the number of such accesses can be quite large when contention is low and A is large, the potential for overshooting the point where the flag is set arises. Conversely, when the number of processors is comparable to A (or greater than A), the number of times a process manages to access the barrier flag is small due to contention with other processes. In such cases, the network access count increases, but the average waiting time per processor decreases. Referring to Figures 4-5 and 4-6 the decrease in the waiting time for the backoff curves closely corresponds to the increase in network accesses.

4.5.1 Summary

A few general observations can be made at this point. When the number of processors participating in the barrier synchronization is small compared to the time of arrival of the processors, significant reduction in network accesses can be achieved without compromising processor utilization due to backoff waiting for a small backoff base. In such cases, the number of synchronization network accesses is similar to those made in schemes that use special hardware support such as synchronization buses, broadcasts, or global synchronization logic. When the number of processors is large, and if they arrive within a relatively small interval of time, a penalty in either network accesses or processor idle time must be paid. However, depending on the situation, one can

be traded for the other.

A good example of a situation in which backoff barriers are useful would be in cases where an application is poorly load balanced. As we saw with WEATHER, non-optimal load-balancing can result in a large amount of synchronization traffic to memory as processors busy-wait at the end of a parallel section waiting for everyone to arrive. Since this is a situation where the arrival-time spread of processors can be large compared to the number of processors in the application, backoff barriers can potentially significantly reduce the number of synchronization accesses made at the barrier without hurting processor utilization. We can verify this by simulating backoff barrier synchronization in the post-mortem scheduler and measuring the resulting synchronization traffic in the multiprocessor traces.

Our discussion thus far focused on the traffic and the waiting time *during* the execution of the barrier. We can also look at the effect on average traffic with the caveat that such smoothing might tend to make barrier accesses seem less disruptive. We measured the average network data traffic per processor in FFT (assuming separate packet-switched networks for the request and response), excluding synchronization references, to be 0.133 network accesses per cycle. Using results from our simulations of the barriers with $A = 100$ (roughly approximating the barrier interval A in FFT with 64 processors) we compute the extra traffic due to barriers when the barrier variable and the barrier flag are not cached. Adding these synchronization references to our base network traffic, the average traffic increases to 0.136 network accesses per cycle (assuming that the base traffic in A is also 0.133). Now, with a base 8 exponential backoff we find that the average network traffic drops to 0.134. This decrease is significant considering that these savings come from reductions in synchronization references which are effectively hot-spot references. Moreover, we observe in this case that the base 8 exponential backoff also results in a 10 percent decrease in waiting time at the barrier. Both average network traffic and waiting time at synchronizations are reduced using backoff methods for our FFT application.

As a validation of our barrier simulation model, we also compared the average network traffic in FFT when synchronization references are not cached with the average network traffic predicted by our barrier model simulations. The numbers correlated well, with barrier simulations predicting 0.136 net accesses per cycle per processor, while measurements from FFT yielded 0.135.

We analyzed the tradeoff between network accesses and processor idle time due to backoff. In general, reducing the number of network accesses might be more important than reducing the processor idle time because reducing the number of network accesses also reduces the processor idle time because of the reduced contention in the network, and because of decreased competition with the regular network activities of the other processors not involved in the barrier.

4.6 Optimizations and Extensions

Adaptive backoff techniques have several other applications. For example, this technique can be applied to processors waiting on a resource that requires mutually exclusive access. Instead of spinning on the resource lock, the processors can backoff testing the lock by an amount proportional to the number of waiting processors. Adaptive techniques are more suited to this situation than barrier synchronization because the waiting time is proportional to the number of processors (the constant of the proportion is the average time the resource is held by each processor).

An adaptive backoff method can be used to reduce contention in unbuffered circuit-switched networks. If a network access suffers a collision, instead of resubmitting the request immediately, one can backoff some amount first. We are investigating such schemes in a large-scale multiprocessor project called ALEWIFE at MIT. The backoff amount can be determined in one of several ways:

- The backoff amount can be proportional to the network depth traversed by the message, which is determined by a network supplied status byte indicating the

stage at which the collision occurred. The rationale for this choice is that the deeper a message travels, the greater the network resource that it ties up in its unsuccessful attempt. Conversely, if a collision occurs within a few stages of travel into the network, the access can be resubmitted sooner as the network resources tied up will be smaller.

- An argument for making the backoff amount *inversely* proportional to the network depth traversed can also be made. The deeper a message travels before colliding, the less congested the network is expected to be, and so the access can be retried sooner. Simulations can be used to study the tradeoffs involved in these two opposing arguments and suggest a practical backoff algorithm.
- A colliding network access might wait some constant time proportional to the average round trip time through the network before retrying.
- The number of previous unsuccessful tries can be used as a parameter to an exponential backoff algorithm.
- In a packet-switched network, Scott and Sohi [27] make use of state information in the memory module queues to signal processors to hold back requests in congested situations. This state information could also be used to have the processors backoff sending requests by some time proportional to the length of the queue.

As we mentioned before, the adaptive backoff techniques that we evaluated do not require special hardware support. The synchronization software that determines which backoff method is used can be designed in one of several ways. One can be conservative and use a simple adaptive backoff on the barrier variable and a binary backoff on the barrier flag. The programmer can write the algorithms into the synchronization macros or routines from a knowledge of the application. The compiler can determine appropriate code sequences for the barrier synchronizations based on

expected behavior of loops and the amount of visible parallelism. A more venturesome method might use profiling to determine the temporal behavior of the application and the number of processors participating in the synchronization and pass this information on to the compiler for further optimization. One case where such information might be useful is in determining when to (or whether to) queue a process to await a signal when the barrier flag is set rather than spinning on the network.

4.7 Conclusions

Network bandwidth is a precious resource in large-scale shared memory multiprocessors. This thesis presents a group of adaptive backoff techniques aimed at reducing the number of network accesses due to synchronizations. We model adaptive techniques for barrier synchronizations and show that in many cases these techniques can achieve dramatic savings at minimal extra cost. In other situations, however, network accesses can only be reduced while trading-off utilization of synchronizing processors. These techniques are implemented in software, and they can be optimized for varying applications.

The technique is simple and incurs little overhead (in implementation and run-time expense) unlike hardware combining schemes or specialized global synchronization logic. However, as mentioned above, it only provides exceptional advantage in certain situations such as when the number of processors participating in a barrier synchronization is small compared to the time of arrival of the processors. An example of a situation where backoff would help greatly is for applications where load balancing is poor. As we saw with the WEATHER application, poor load balancing can result in a tremendous amount of synchronization traffic while processors busy-wait at the barrier. Adaptive backoff barriers can significantly reduce this traffic.

The central idea behind an adaptive synchronization technique is to use synchronization state information and past history to reduce the number of idle synchronization spins. The general technique has many applications, such as reducing network accesses in barrier synchronizations and minimizing spin-lock accesses of processors waiting on a shared resource. The application of backoff techniques in unbuffered circuit-switched networks is especially interesting. These techniques can be applied in many ways to offer a simple, low-cost method to reduce contention in the network. Further study is needed to determine the optimal backoff algorithm.

Appendix A

Applications

The three applications which drive our trace-driven simulations are from the class of scientific/numeric applications. SIMPLE and WEATHER are representative of a large group of scientific/numeric applications which model physical systems over time by breaking the domain into a mesh or grid of regions whose state is computed and updated every time step in the simulation. This approach is naturally suited to parallel execution because all the gridpoints are working in parallel. Often each gridpoint in the system performs the same calculations each time step - only the data is different.

The Fast Fourier Transform (FFT) application, written at IBM, is a parallelized version of a Radix-2 FFT computation in two variables on a random array of complex numbers. Since we used a problem size of 128, the parallel loops working on the 128x128 matrix contained 128-way parallelism, providing for an even distribution of work to the 64 processors in our simulations.

The SIMPLE code models hydrodynamic and thermal behavior of fluids in two dimensions. Finite difference methods are used to solve the equations of inviscid compressible hydrodynamics and simple heat conduction on an N x N mesh. Once again, we used a problem size of 128, but several parallel sections do not contain fully 128-way parallelism resulting in an uneven distribution of work among the 64

processors in our simulations. SIMPLE is representative of an application with neither worst-case, nor best-case performance in the SPMD computational model.

The WEATHER code forecasts the weather by modeling the state of the atmosphere. The algorithm breaks the atmosphere down into a three-dimensional grid (108 x 72 X 9 in our case) encircling the globe and computes the value of several inter-related state variables using finite difference methods. WEATHER was the most poorly load-balanced application of the three we traced. Fifty-four processors (we used 64) would be the preferred number of processors to execute this application for the problem size used. Thus the load balancing in our three applications showed a wide range.

Bibliography

[1] Norman Abramson. The ALOHA System – Another alternative for computer communications. In *Proc. of the 1977 Fall Joint Computer Conf.*, pages 281–285, 1977.

[2] Anant Agarwal and Mathews Cherian. *Adaptive Backoff Synchronization Techniques.* Technical Report, MIT VLSI Memo, April 1989.

[3] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[4] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[5] M. Cherian, M. Kumar, and K. So. *A Trace-Driven Methodology for Simulation of Multiprocessors.* Technical Report to appear, IBM T. J. Watson Research Center, Yorktown Heights, 1989.

[6] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, April 1965.

[7] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The Simple Code.* Technical Report, Lawrence Livermore Laboratory, February 1978.

[8] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN.* Technical Report RC 11552 (55212), IBM T. J. Watson Research Center, Yorktown Heights, November 1986.

[9] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of ASPLOS III*, pages 257–270, April 1989.

[10] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[11] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Saleh. Cedar – A Large Scale Multiprocessor. In *International Conference on Parallel Processing*, pages 524–529, August 1983.

[12] D. A. George. *EPEX - Environment for Parallel Execution.* Technical Report RC 13158 (58851), IBM T. J. Watson Research Center, Yorktown Heights, September 1987.

[13] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, IEEE, New York, June 1983.

[14] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[15] Mark Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 158–166, IEEE, New York, June 1984.

[16] Tsutomu Hoshino. *PAX Computer. High-Speed Parallel Processing and Scientific Computing.* Addison Wesley, Reading Mass., 1989. Editor Harold S. Stone.

[17] Eugenia Kalnay-Rivas and David Hoitsma. *Documentation of the Fourth Order Band Model.* Technical Report, NASA Modeling and Simulation Facility Laboratory for Atmospheric Science, NASA/Goddard Space Flight Center, Greenbelt, MD, 1979.

[18] L. Kleinrock and Y. Yemini. An Optimal Adaptive Scheme for Multiple Access Broadcast Communication. In *Proceedings ICC*, June 1978.

[19] S. S. Lam. A Carrier Sense Multiple Access Protocol for Local Networks. *Computer Networks*, 4(1), January 1980.

[20] S. S. Lam and L. Kleinrock. Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures. *IEEE Transactions on Computers*, C-23, Sept. 1975.

[21] E. L. Lusk and R. A. Overbeek. *Implementation of Monitors with Macros: A Programming Aid for the HEP and other Parallel Processors.* Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois, December 1983.

[22] R. Metcalfe and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7), July 1976.

[23] Susan Owicki and Anant Agarwal. Evaluating the Performance of Software Cache Coherence. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[24] Janak H. Patel. Analysis of Multiprocessors with Private Cache Memories. *IEEE Transactions on Computers*, C-31(4):296–304, April 1982.

[25] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764–771, August 1985.

[26] G. F. Pfister and V. A. Norton. 'Hotspot' Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10), October 1985.

[27] Steven Scott and Gurindar Sohi. Using Feedback to Control Tree Saturation In Multistage Interconnection Networks. In *Proceedings 16th Annual Int'l Symp. on Computer Architecture*, June 1989.

[28] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, IEEE, New York, June 1988.

[29] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[30] James E. Smith and James R. Goodman. A Study of Instruction Cache Organizations and Replacement Policies. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 132–137, IEEE, New York, June 1983.

[31] K. So, A.S. Bolmarcich, F. Darema-Rogers, and V. A. Norton. *SPAN - A Speedup Analyzer for Parallel Programs.* Technical Report RC 12186 (54776), IBM T. J. Watson Research Center, Yorktown Heights, September 1986.

[32] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *PSIMUL - A System for Parallel Simulation of Parallel Systems.* Technical Report RC 11674 (58502), IBM T. J. Watson Research Center, Yorktown Heights, November 1987.

[33] Peiyi Tang and Pen-Chung Yew. Processor Self-scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[34] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[35] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributed Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(14):388–395, April 1987.