Low-Cost Support for Fine-Grain Synchronization in Multiprocessors

David Kranz, Beng-Hong Lim, Donald Yeung and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

As multiprocessors scale beyond the limits of a few tens of processors, they must look beyond traditional methods of synchronization to minimize serialization and achieve the high degrees of parallelism required to utilize large machines. By allowing synchronization at the level of the smallest unit of memory, fine-grain synchronization achieves these goals. Unfortunately, supporting efficient fine-grain synchronization without inordinate amounts of hardware has remained a challenge.

This paper describes the support for fine-grain synchronization provided by the Alewife system. The premise underlying Alewife's implementation is that successful synchronization attempts are the common case when serialization is minimized through word-level synchronization. For our applications, the failure rates were less than 7%. Efficiency at low hardware cost is achieved by providing hardware support to streamline successful synchronization attempts and relegating other non-critical operations to software. Alewife provides a large synchronization name space by associating full/empty bits with each memory word. Successful synchronization attempts execute at normal load-store speeds, while attempts that fail invoke appropriate software trap handlers through a fast trap mechanism. The software handlers deal with the issues of retrying versus blocking, queueing, and rescheduling. The efficiency of Alewife's mechanisms is analyzed by comparing the costs of various synchronization operations and parallel application execution time. In several applications we studied, our hardware support improved performance by 35%–50%.

1 Introduction

To execute on a MIMD multiprocessor, a program must be partitioned into threads that communicate either by reading and writing to shared memory or sending messages. This communication must be synchronized to ensure correctness. Under the shared-memory abstraction, synchronization between threads is used to enforce one of the following conditions:

1. *Read-after-write data-dependency.* This ensures that a thread that uses data computed by another thread reads the data only *after* the producing thread has written the data. Producer-consumer and barrier synchronization are examples of synchronization that enforce this data-dependency.

2. *Mutual exclusion.* Mutual exclusion allows different threads to modify shared data atomically. It allows threads to have exclusive read/write access to shared data if threads are forced to acquire a mutual-exclusion lock before modifying the data and release the lock after modifying the data.

Synchronization incurs an overhead because of a loss of parallelism and the cost of the synchronization operation itself. In this paper, we are concerned with providing support for enforcing these conditions efficiently without requiring inordinate amounts of hardware.

For a program to execute efficiently on a multiprocessor, the serialization imposed by the synchronization structure of the program must be reduced as much as possible and the overhead of the synchronization operations must be small compared to real computation time. Multiprocessors have traditionally supported only *coarse-grain* synchronization (*e.g.* barriers and mutual exclusion locks). It is known that *fine-grain* synchronization is an effective way to enhance the performance of many applications, *provided it can be implemented efficiently.* By efficiently, we mean that good performance is obtained for programs that benefit from fine-grain synchronization without affecting the performance of other programs adversely.

In the case of data-dependence, fine-grain synchronization allows the amount of data transferred from one thread to other threads in one synchronization operation to be small, *e.g.*, a word or small cache block. In the case of mutual exclusion, it allows memory words to be individually locked with minimal overhead. The above definition of fine-grain synchronization does not make any assumptions on the granularity of threads. However, it does imply that the frequency of synchronization will be high because each synchronization operation signals the production or release of a small amount of data.

Although barrier synchronization can result in unnecessary serialization, it has the advantage that the shared data protected by a barrier can be accessed with normal reads and writes – synchronization doesn't take place until all of the data have been accessed. In contrast, with fine-grain synchronization, each reference to that shared data must be a synchronizing memory operation (as opposed to a normal load or store). It is thus important that the cost of a synchronizing memory operation be made as close as possible to that of an ordinary memory operation.

An added advantage of cheap synchronizing reads and writes is that they allow one to dynamically partition data even if the compiler does not know whether a particular reference needs to be synchronized. In many cases better load balance can be obtained through dynamic partitioning. If the application has sufficient parallelism, this will result in a large number of memory operations being potentially synchronizing. However, in many of these cases the data will be produced and consumed by the same processor such that no waiting will be required for the synchronizing memory operations.

We argue that the issue of fine-grain synchronization must be addressed at all levels of a computer system. The programming language must allow parallelism and synchronization at the level of a data word to be expressed easily, and fine-grain synchronization must be implemented efficiently at the system level, in both hardware and software.

This paper focuses on the support for fine-grain synchroninzation in the Alwife machine [2], a shared-address space, distributed-memory multiprocessor being developed at MIT. At the system level, Alewife provides hardware-support for fine-grain synchronization in the form of full/empty bits (as in the HEP [18]) and efficient traps.

The processor implements load and store instructions that trap on various states of the full/empty bit. Little or no overhead is incurred if the referenced data is available. If the data required by a read operation is not yet available, the processor traps. We avoid providing extensive hardware support by handling these traps entirely in software. We have implemented various software strategies that use synchronization type information provided by the compiler/programmer to choose suitable waiting algorithms when the trap is taken.

The contributions of this work are:

- We propose an architecture that implements fine-grain synchronization using a single hardware full/empty bit, an atomic swap instruction and software scheduling of threads. We observe that synchronized data are usually available when needed, so we simplify the hardware by implementing thread waiting or blocking in software. We provide some evidence that this observation is indeed true for real applications executing on Alewife–for the applications we studied, the failure rates were less than 7%.
- The required hardware support is simple enough that our processor, Sparcle [1], was implemented by making minor modifications to LSI Logic's existing SPARC processor [19], without affecting the speed of the processor. This processor has been running in our laboratory since March 1992.
- We evaluate the efficacy of our implementation by showing the performance of several applications running on a detailed Alewife simulator. We compare coarse-grained (barrier) and fine-grained implementations of these applications. In addition, we evaluate the advantage of the Alewife hardware support as compared to a pure software implementation. In several applications we studied, the hardware support provided a performance gain of 30%–50% over the pure software approach.
- To our knowledge, this is the first paper to empirically quantify the performance of a system that makes a careful tradeoff between hardware and software support for fine-grain synchronization.

The rest of this paper is organized as follows. Section 2 describes our general approach for implementing fine-grain synchronization. Section 3 discusses programming language support. Section 4 describes the Alewife implementation. Section 5 contains some preliminary experimental data supporting the proposition that fine-grain synchronization can be implemented reasonably at low cost. Section 6 describes related work and Section 7 concludes the paper.

2 A Low-Cost Approach to Fine-Grain Synchronization

We argue that an efficient implementation of fine-grain synchronization must cover all levels of a computer system:

- It must be possible to express fine-grain parallelism and synchronization conveniently at the language level.

- Synchronization at the word level implies that there is at least one bit of extra synchronization information associated with each word, the full/empty bit. This memory cost would ideally be only one bit per word.
- The cost of a synchronizing operation that does not need to wait should be small; ideally no more expensive than an ordinary memory operation. Full/empty bits have the added advantage that both data and synchronization information can be accessed simultaneously.
- On synchronization failure, a decision must be made to either busy-wait until the data is available, or to block and switch to another task. This process must either occur infrequently, or not take very long.

If a synchronizing read or write to a data word does not have to wait to access or modify the data, we say that the operation succeeds. If it fails, the thread issuing the request cannot continue until the data becomes available or modifiable. The major premise behind our implementation approach is that most synchronizing reads will succeed. There are several reasons why we expect this to be the case:

1. The compiler or runtime system can often schedule consumers after producers.
2. A common use of producer/consumer arrays is to have a loop that operates on each element in turn. In this case, if a synchronization failure occurs the consumer will have to wait. Thus the consumer will fall behind the producer and future failures are less likely to occur.
3. In the case of dynamic partitioning, synchronizing reads and writes often occur on the same processor and so will succeed.
4. And, of course, with fine-grain synchronization a thread has to wait only for data on which it *actually* depends.

In Section 5 we provide some empirical evidence that synchronization failures are relatively infrequent. For the applications we studied, the failure rates were less than 7%. In Section 4 we will show how this assumption of success allows us to implement fine-grain synchronization efficiently with little hardware support. First, however, we discuss some of the language issues in using fine-grain synchronization.

3 Programming Language Issues

It is desirable that fine-grain parallelism and synchronization be expressible at the language level. We take the position that the programmer will specify which parts of a program *may* be executed in parallel. This does not preclude a compilation phase that converts sequential to parallel code. It is up to the system to decide which parts to actually execute in parallel and to handle proper synchronization. There are two ways a programmer may express parallelism in a program: *control parallelism* and *data-level parallelism*. It is natural to associate synchronization with each style.

3.1 Data-Level Parallelism

Data-level parallelism expresses the application of some function to all or some elements of an aggregate data object, such as an array. Data-level parallelism is often expressed using parallel do-loops. Synchronization within a parallel do-loop can be either coarse or fine-grain. For producer-consumer synchronization, coarse-grain synchronization involves placing a *barrier* at the end of the loop. Elements of the aggregate are written by threads using ordinary stores. At the end, each thread waits for all to complete. The values can then be accessed with ordinary reads. Alternatively, in the case of mutual-exclusion locks, coarse-grain synchronization will associate a lock with a large chunk of data.

Fine-grain data-level synchronization is expressed using data structures with accessors that implicitly synchronize. We call these structures J-structure and L-structure arrays. A J-structure is a data structure for producer-consumer style synchronization inspired by I-structures [4]. A J-structure is like an array, but each element has additional state: *full* or *empty*. The initial state of a J-structure element is empty. A reader of an element waits until the element's state is full before returning the value. A writer of a J-structure element writes a value, sets the state to full, and releases any waiting readers. An error is signalled if a write is attempted on a full element. The difference between J-structures and I-structures is that, to enable efficient memory allocation and good cache performance, J-structure elements can be reset to an empty state.

L-structures are arrays of "lock-able" elements that support three operations: a locking read, a non-locking peek, and a synchronizing write. A locking read waits until an element is full before emptying it (i.e., locking it) and returning the value. A peek also waits until the element is full, but then returns the value *without* emptying the element. A synchronizing write stores a value to an empty element, and sets it to full, releasing any waiters. As for J-structures, an error is signalled if the location is already full. An L-structure therefore allows mutually exclusive access to each of its elements. The synchronizing L-structure reads and writes can be used to implement M-structures [5]. However, L-structures are different from M-structures in that they allow multiple non-locking readers, and a store to a full element signals an error[1].

3.2 Control Parallelism

Using control parallelism, a programmer specifies that a given expression X may be executed in parallel with the current thread. In our system this behavior is specified by wrapping `future` around an expression or statement X. Synchronization between these threads is implicit and occurs when the current thread demands, or *touches*, the value of X. The programmer does not have to explicitly specify each point in the program where a value is being touched. A touch implicitly occurs anytime a value is used in an ALU operation or as a pointer to be dereferenced, but not when a value is returned from a procedure or passed as an argument to a procedure. Storing a value into a data structure also does not touch the value.

[1] As with M-structures, it is possible to implement pairwise producer-consumer synchronization using L-structures: producers use synchronizing writes and consumers use locking reads. This can be viewed as an optimization of J-structures to avoid having to reset them when there is only a single consumer for each value produced.

In Alewife, the `future` keyword does not necessarily cause a new runtime thread to be created, together with the consequent overhead. The system must, however, ensure that the current thread and X can be executed concurrently if necessary (*e.g.*, to avoid deadlock). We call this behavior, where a new thread is created at runtime only for deadlock avoidance or load-balancing purposes, *lazy task creation* [15].

Using `future` provides a form of fine-grain synchronization because synchronization can occur between the producer and consumers of an arbitrary expression, *e.g.*, a procedure call can start executing while some of its arguments are still being computed.

The Alewife system currently supports two programming languages: Mul-T [13], a parallel Lisp language, and Semi-C. Semi-C [10] is a parallel C-like language with extensions for expressing parallel execution. Both Mul-T and Semi-C support control-level and data-level parallelism as described in this section.

4 Alewife Implementation

Existing multiprocessor systems have supported fine-grain synchronization in different ways ranging from a complete hardware implementation to doing it all in software. Good engineering practice suggests that it is cost-effective to provide hardware support for the common case and handle other cases in software. We argued in Section 2 that the common case is a successful synchronization. Accordingly, we provide hardware for automatic detection of failure, leaving the actual handling of the failure case to software. Doing so allows us to leave an efficient CPU pipeline and register set in place, thus retaining good single-thread performance.

We outline the hardware and software mechanisms necessary to support our language primitives for fine-grain synchronization. To implement data-level parallelism, we need to synchronize at the level of individual memory words. References to L-structures and J-structures are examples of fine-grain synchronizing loads and stores. Such an operation reads or writes a data word while testing and/or setting a synchronization condition. In the event of success this operation should to take no longer than a normal load or store. In the event of failure, the processor traps.

To implement control parallelism we need to know when a value produced by a `future` expression is being touched. This might happen anytime a value is used as an argument to an ALU operation or dereferenced as a pointer. Since these operations are very frequent, the touching operation should take no longer than an ordinary ALU operation, load, or store. The processor traps if the value being touched is not ready.

4.1 Hardware Support

Full/empty bits are used to represent the state of synchronized data in J- and L-structures. A full/empty bit is referenced and/or modified by a set of special load and store instructions. References to values that are being computed in parallel as a result of `future` are detected through tagged add and subtract instructions as well as misaligned memory reference traps. In this paper, we will concentrate on the implementation of J- and L-structures for fine-grain, data-level parallelism.

4.1.1 Hardware Support for J- and L-Structures

References and assignments to J- and L-structures use the following special load, store, and swap instructions, depending on whether detection through traps is desired or not:

LDN	Read location.
LDEN	Read location and set to empty.
LDT	Read location if full, else trap.
LDET	Read location and set to empty if full, else trap.
STN	Write location.
STFN	Write location and set to full.
STT	Write location if empty, else trap.
STFT	Write location and set to full if empty, else trap.
SWAPN	Swap location and register.
SWAPEN	Swap location with register and set to full.
SWAPT	Swap location with register if empty, else trap.
SWAPET	Swap location with register if full, else trap.

In addition to possible trapping behavior, each of these instructions sets a condition code to the state of the full/empty bit at the time the instruction starts execution. The compiler has a choice to use traps or tests of this condition code. When a trap occurs, the trap handling software decides what action to take as described later in this section.

4.1.2 Hardware Support for Futures

In order to implement futures efficiently, "full/empty bits" are needed in registers as well as in memory. These are implemented by treating the low bit of a pointer as a one-bit tag. When a new thread is created to compute the value of a future object, a pointer to a *placeholder* is returned immediately. This pointer has the low bit set while all other data objects have the low bit clear. When a touching operation is performed on a register with the low bit set, a trap occurs. The trap handling is much like that for J- and L-structures. When the new thread has computed its value, that value is stored in the placeholder object. In addition, the trap handler alters the contents of the register that caused the trap to contain the new value so that the next reference to that value will not trap.

This mechanism eliminates the cost of checking for placeholders with almost no additional hardware cost. If the touching action is the dereferencing of a pointer, a misaligned address trap occurs just as in SPARC. If the touching operation is an arithmetic operation a tagged add or subtract is used. We made only two small modifications to support this placeholder checking in Sparcle:

1. The SPARC tagged add and subtract instructions were supplemented with versions that only look at the low bit, instead of the low two bits.

2. When these new instructions trap, they vector to a handler specific to the register containing the placeholder.[2] The trap vector dispatch for misaligned address traps was modified in the same way.

[2] motivated by [9]

4.2 Implementation of J- and L-Structures

We now describe in more detail the implementation of J- and L-structures, and present machine code for synchronously reading and writing these structures. These synchronization structures provide for data-dependency and mutual-exclusion, and are primitives upon which other synchronization operations can be built. The cycle counts presented below represent the cost of the actual code, assuming cache hits.

Failed synchronizations are handled entirely in software. As previously described, failure is detected in hardware, and the trap dispatch mechanism passes control to the appropriate handler. Handling failure completely in software is our biggest saving in complexity because blocking a thread is a complex operation. In contrast, machines that support blocking in hardware pay an enormous hardware cost. For example, Monsoon [16] implements special I-structure boards to queue failed synchronizations in hardware.

Even though we expect successful synchronizations to be the common case, we would like to handle failed synchronizations as efficiently as possible. In Section 4.3 we describe how we use several methods to reduce the overhead of handling failed synchronizations in software. Both the compiler and runtime system collaborate to make handling failed synchronizations as efficient as possible.

4.2.1 J-structures

J-structures were described in Section 3. Recall that each J-structure element has a full or empty state associated with it. It is natural to use full/empty bits to represent that state. Allocating a J-structure is implemented by allocating a block of memory with the full/empty bit for each word set to empty. Resetting a J-structure element involves setting the full/empty bit for that element to empty.

Implementing a J-structure read is also straightforward: it is a memory read which traps if the full/empty bit is empty. It is implemented with a single instruction:

```
ldt (r1),r2 ; r1 points to J-structure location
```

If the full/empty bit is empty, the reading thread may need to suspend execution and queue itself on a wait queue associated with the empty element. Where should this queue be stored? A possible implementation is to represent each J-structure element with two memory locations, one for the value of the element and the other for a queue of waiters. However, this would double the memory requirement of J-structures.

An alternative is to use a single memory location for both the value and the wait queue, since they never need to be present at the same time. A problem with this approach is that we need to associate two bits of synchronization state with each J-structure element: whether the element is full or empty, and whether the wait queue is locked or not. Other architectures have solved this problem by having multiple state bits per memory location [3, 16]. Instead of providing additional hardware support, we take a different approach.

Like SPARC, Sparcle supports an atomic register-memory swap operation. Since the writer of a J-structure element knows that the element is empty before it does the write, it can use the atomic swap to synchronize access to the wait queue. With this approach, a single full/empty

```
    move   $0,r3    ; set up swap register
    swapt  r3,(r1)  ; swap zero with J-structure location,
                      trap if full
    cmp   $-1,r3    ; check if queue is empty.
    beq,a %done     ; branch if no waiters to wake up.
    stft   r2,(r1)  ; write value and set to full
                      (in delay slot).
         :
    <wake up waiters and store value>
         :
  %done
```

Figure 1: Machine code implementing a J-structure write. `r1` contains the address of the J-structure location to be written to, and `r2` contains the value to be written. `-1` is the end of queue marker and `0` in an empty location means that the queue is locked.

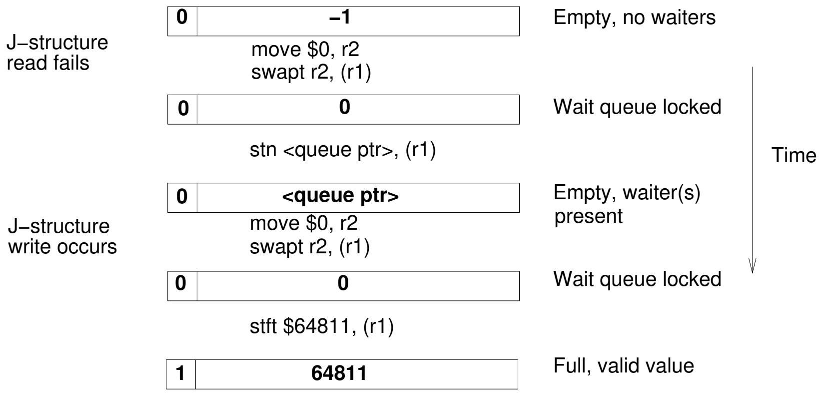

Figure 2: Reading and writing a J-structure slot. `r1` contains a pointer to the J-structure slot. The possible states of a J-structure slot are illustrated here.

bit is sufficient for each J-structure element. A writer needs to check explicitly for waiters before doing the write.

Using atomic swap and full/empty bits, the machine code in Figure 1 implements a J-structure write. Compared with the hardware approach, this implementation costs an extra `move`, `swap`, `compare` and `branch` to check for waiters. However, we believe that the reduction in hardware complexity is worth the extra instructions. Figure 2 gives a scenario of accesses to a J-structure location under this implementation and illustrates the possible states of a J-structure slot.

4.2.2 L-structures

The implementation of L-structures is similar to that of J-structures. The main differences are that L-structure elements are initialized to full with some initial value, and an L-structure read

```
                  ; r1 points to L-structure location
move   $-1,r2     ; empty queue
swapet r2,(r1)    ; store null queue and get value if full,
                    else trap
```

Figure 3: Machine code implementing an L-structure read.

	Action	Instructions	Cycles
Array	read	1	2
	write	1	3
J-structure	read	1	2
	write	5	10
	reset	1	3
L-structure	read	2	5
	write	5	10
	peek	1	2

Table 1: Summary of fast-path costs of J-structure and L-structure operations, compared with normal array operations.

of an element sets the associated full/empty bit to empty and the element to the null queue. An L-structure read is therefore implemented as in Figure 3. In Sparcle, this takes three extra cycles compared to a normal read. An L-structure peek, which is non-locking, is implemented in the same way as a J-structure read.

On an L-structure write, there may be multiple readers requesting mutually exclusive access to the L-structure slot. Therefore, it might make sense to release only one reader instead of all readers. On the other hand, a potential problem arises if the released reader remains descheduled for some significant length of time after being released. It is not clear what method of releasing waiters is best, and our current implementation releases all waiters.

Table 1 summarizes the instruction and cycle counts of J-structure and L-structure operations for the case where no waiting is needed on reads and no waiters are present on writes. In Sparcle, as in the LSI Logic SPARC, normal reads take two cycles and normal writes take three cycles, assuming cache hits. A locking read is considered a write and thus takes three cycles.

4.3 Handling Failed Synchronizations in Software

Due to full/empty bits and signalling failures via traps, successful synchronizations incur very little overhead, as described in the previous section. For failed synchronizations, we provide just enough hardware support to rapidly dispatch processor execution to a trap handler. We describe here how trap handler software handles failed synchronizations with efficiency comparable to a hardware implementation.

A failed synchronization implies that the synchronizing thread has to wait until the synchronization condition is satisfied. There are two fundamental ways for a thread to wait: polling and blocking. Polling involves repeatedly checking the value of a memory location, returning control to the waiting thread when the location changes to the desired value. No special hardware support is needed to aid in polling. Once the trap handler has determined the memory location to poll, it can poll on behalf of the synchronizing thread by using non-trapping memory instructions, and return control to the thread when the synchronization condition is satisfied.

Blocking is more expensive because of the need to save and restore registers. The scheduler may also need to be invoked. Saving and restoring registers is particularly expensive in Sparcle because loads take two cycles and stores three. If all user registers need to be saved and restored, the cost of blocking can be several hundred cycles, more or less, depending on cache hits.

We reduce the blocking cost in two ways. First, if the thread actually needs to be saved into memory, the compiler communicates the number of live registers to save to the trap handler via otherwise unused bits in the trapping instruction. This information can significantly reduce the blocking overhead by reducing the number of registers that need to be saved. [14] describes how the cost of blocking can be reduced to less than 100 cycles on a processor with single-cycle loads and stores and with information on live registers. Second, since Sparcle has multiple hardware contexts, we can block a thread without saving and restoring registers by disabling the context on which the thread is executing. Sparcle provides instructions (NEXTF and PREVF) to switch to the next enabled context directly.

A common hardware approach to efficient blocking of threads is to minimize the processor-resident state of a thread. This is done by restricting a thread to a very small number of registers (one or two) so that hardware can save the state of a thread and queue it on a wait queue in a small number of cycles. We reject this approach because minimizing processor resident state of a thread has adverse effects on single-thread performance. We are willing to sacrifice some cycles when blocking a thread for higher single-thread performance.

We do not always need to block a waiting thread. On a failed synchronization, the trap handler is responsible for implementing the waiting algorithm that decides whether to poll or to block the thread. Karlin *et al.* [11] and Lim and Agarwal [14] investigate the performance of various waiting algorithms. They show polling for some length of time before blocking can lead to better performance, and investigate various methods for determining how long to poll before blocking.

Lim and Agarwal also demonstrate the performance benefits of choosing a waiting algorithm tailored to the type of synchronization being performed. In our system, the compiler informs the synchronization trap handler which waiting algorithm to execute. If there are other threads to execute, the appropriate waiting algorithm is to block immediately for barrier synchronization, and to poll for a while before blocking for fine-grain producer-consumer synchronization. Since fine-grain synchronization leads to shorter wait times, this reduces the probability that a waiting thread gets blocked.

The compiler passes information on the synchronization type and the number of live registers to the trap handler in otherwise unused bits in the trapping machine instruction. The overhead to dispatch to an appropriate type-specific handler is about 11 cycles: recalling that failed synchronizations are signalled via traps in Sparcle, it takes 4 cycles from the time the trap is taken to the time the trap handler begins execution. To access the compiler-passed type information, the trap handler reads the trapping instruction, then masks and shifts the relevant bits to dispatch to the correct waiting routine directly. This dispatch can be done in 7 cycles.

Our current implementation takes 10 cycles because of an additional check for the case when the compiler neglects to specify this information.

To control hardware complexity, thread scheduling is also done entirely in software. Once a thread is blocked, it is placed on a software queue associated with the failed synchronization condition. When the condition is satisfied, the thread is placed on the queue of runnable tasks at the processor on which it last ran. A distributed thread scheduler that runs on all idle processors checks these queues to reschedule runnable tasks.

4.4 An Open Problem with Fine-Grain Synchronization

When the memory containing synchronized data must be re-used, several problems arise. There are two issues:

1. How do we know when the consumers have finished reading synchronized data so that its memory can be re-used?
2. What is the cost of resetting the state of the synchronization structure?

One way to determine when memory can be re-used is for the consumers to meet at a barrier when finished, just as the producers do. In fact, it can be arranged so that there is only one barrier shared by the producers of one structure and consumers of another. For coarse-grain synchronization there is very little state to reset so there is not much cost involved.

For fine-grain synchronization a barrier could be used to reset synchronization structures. In many cases it is possible to amortize the cost of the barrier by resetting many structures at once. In many cases the compiler can tell when a J-structure can be re-used, in others a garbage collector can be used to avoid doing barrier synchronizations. Although the cost of resetting a full/empty bit is small compared to the computation likely to be done on each element, and the resetting can be done in parallel, this issue needs to be addressed in future work on fine-grain synchronization.

5 Performance Results

To evaluate the performance of fine-grain synchronization in our system, we monitored the performance of several applications, each synchronized in coarse-grain and fine-grain styles. This section will discuss in detail two of these applications and refer to two others that provide supplementary data to our main results. The measurements were acquired on an accurate cycle-by-cycle simulator of the Alewife machine. The simulator was configured to simulate a 64-processor machine in an 8×8 mesh topology.

Our data offers three main results. First, we show that fine-grain data-level synchronization provided by J-structures results in improved end-application execution time over coarse-grain barriers. Next, we present data to support our earlier claim that in an application that uses fine-grain synchronization, successful synchronization operations are the common case. Finally, we investigate the benefits of the hardware support in Alewife for fine-grain synchronization by showing that application performance is significantly improved when full/empty bits are used to implement the fine-grain synchronization primitives.

5.1 Applications

Our performance results were acquired from an in-depth study of two applications used to numerically solve partial differential equations: the **SOR** (Jacobi with Successive Over-Relaxation) algorithm, and a variant of the preconditioned conjugate gradient algorithm known as **MICCG** (Modified Incomplete Cholesky Conjugate Gradient). In addition, we provide synchronization fault rates for two other applications: **Multigrid**, a solver using Jacobi iterations on grids of varying granularity to enhance convergence. The other is **Gamteb**, a photon transport simulation based on the Monte Carlo method from the Los Alamos National Laboratory. The **Gamteb** code was originally written in Id and was ported to Semi-C.

In **SOR**, the algorithm was used to solve Poisson's equation on a two-dimensional grid of size 32×32 (unless otherwise stated), and consists of a series of Jacobi iterations on fixed-size grids. At each iteration, the new value for each grid point is a function of the current value for that grid point and the values of its four nearest neighbors.

MICCG solved Laplace's equation on a three-dimensional grid of size $16 \times 16 \times 16$ (unless otherwise stated). Preconditioning adds to the basic conjugate gradient iteration a back substitution and forward substitution step which we refer to as the "solver operation." Traditionally, this operation has been difficult to parallelize. The difficulty lies in the solution of a 3-term recurrence expression, a computation that involves complex data-dependencies.

5.1.1 SOR

In **SOR**, the 2-D grid is block partitioned into subgrids in the obvious way, and a thread is assigned to each subgrid. The threads are mapped on to the processor mesh such that data communication is always to neighboring processors during each Jacobi iteration. This leads to load-balanced threads.

In the coarse-grain implementation, a barrier is placed between each Jacobi iteration. The barrier implementation is based on combining trees and is highly optimized for the Alewife machine. A barrier incurs a small latency[3] of 20 μsec on 64 processors [12]. By comparison, typical software implementations (e.g. Intel DELTA and iPSC/860, Kendall Square KSR1) take well over 400 μsec.

In the fine-grain implementation, borders of each subgrid are implemented as J-structures. Thus fine-grain synchronization occurs between nearest neighbors through the J-structures. At each iteration, each thread first writes its border elements to the border J-structures of its neighbors. It then proceeds to compute the solutions for all elements internal to its subgrid. After all internal elements have been computed, the border elements are computed using J-structures that contain the border values of its neighbors. This allows the communication of border elements between neighboring processors to be overlapped with computation of internal elements, reducing the probability of failed synchronizations.

[3]This is measured as the time between successive barriers with null computation in between barriers.

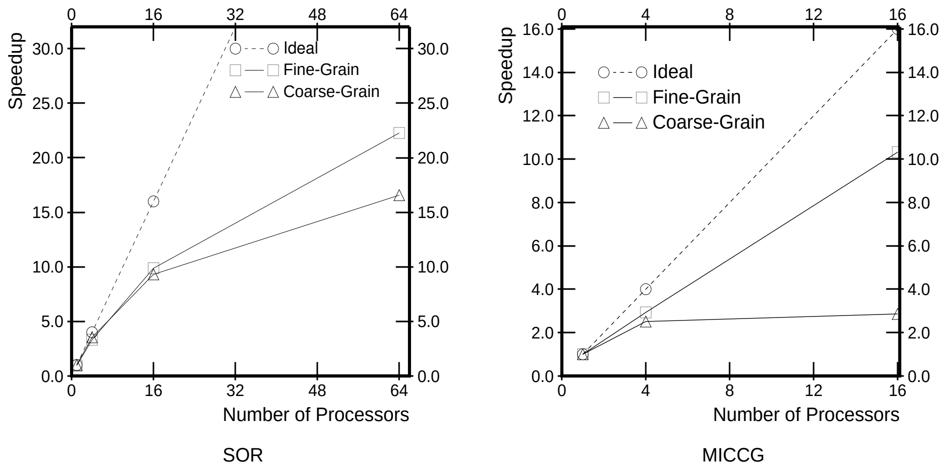

Figure 4: Speedup curves of the coarse-grain and fine-grain implementations of **SOR** and **MICCG**.

5.1.2 MICCG

In the coarse-grain implementation of **MICCG**, the data is block partitioned, and each partition is assigned to a single thread. The data blocks and threads are statically placed such that all communication is confined to nearest-neighbor processors. Wherever a phase of computation consumes the results from another phase, a barrier is inserted to enforce the dependency. Because of the complex data-dependencies in the solver operation, many barriers are needed to properly sequence the computation.

In the fine-grain implementation, the data is partitioned at a finer granularity in order to take advantage of the parallelism in the solver operation. While this reduces physical locality, it is still possible to confine communication to nearest neighbor processors. All elements in the global solution array are allocated as a three-dimensional J-structure. An implicit barrier, however, is still needed to implement two dot product operations that occur in each **MICCG** iteration. (For a detailed discussion of our study and implementation of **MICCG**, see [20]).

5.2 Measurements

A performance comparison of the coarse-grain versus fine-grain implementations appears in Figure 4, showing speedups for **SOR** and **MICCG** attained on 4, 16, and 64 processors. (The speedup for 64 processors for **MICCG** was not attainable due to simulation limits on the problem size). Performance is better in the fine-grain implementation for both applications.

To account for this performance difference, Table 2 shows cycle breakdowns for each data point in the speedup curves. In **SOR**, a cycle breakdown is shown for one iteration averaged over 20 iterations, and in **MICCG**, a cycle breakdown is shown for two iterations taken from a simulation of three iterations with the first iteration thrown away. Total execution time, barrier overhead, and J-structure reference overhead are tabulated. All times are in the units of cycles.

The data in the table show that the coarse-grain implementation does worse than the fine-grain implementation because it incurs a higher synchronization overhead as machine size grows.

Coarse-Grain **SOR**.			
P	Total	Barriers	J-Structs
1	40378	75	N/A
4	11377	385	N/A
16	4325	933	N/A
64	2429	1114	N/A

Coarse-Grain **MICCG**.			
P	Total	Barriers	J-Structs
1	6943004	17921	N/A
4	2769725	1020654	N/A
16	2428515	1680669	N/A

Fine-Grain **SOR**.			
P	Total	Barriers	J-Structs
1	42558	N/A	1600
4	12127	N/A	800
16	4076	N/A	572
64	1807	N/A	447

Fine-Grain **MICCG**.			
P	Total	Barriers	J-Structs
1	6831696	1270	196882
4	2328728	68924	160847
16	662230	81492	48606

Table 2: Cycle breakdowns for **SOR** and **MICCG**. "P" = number of processors, "Total" = cycles per iteration for **SOR**, and total cycles for 2 iterations in **MICCG**, "Barriers" = average overhead and waiting time incurred by a barrier call, "J-structs" = average overhead and waiting time incurred by J-structure references per processor.

Notice that as the number of processors is increased, barrier overhead in the coarse-grain implementation increases. With a larger number of processors, the cost of each barrier is higher. Moreover, since the problem size is fixed, the application becomes finer grained. In contrast with barriers, the cost of each fine-grained data-level synchronization operation remains fixed as machine size is increased. Moreover, the total synchronization overhead is parallelized along with useful computation. This trend is visible in both applications and accounts for the performance difference in Figure 4 for **SOR**.

MICCG has the added problem that the data-dependencies in the solver operation are complex. Enforcing these dependencies with barriers results in a large number of barriers (and in fact, the number of barriers necessarily increases with machine size; see [20]). Because of the flexibility that fine-grain J-structures provides in expressing these data-dependencies, a significant reduction in synchronization overhead is attained. This accounts for the dramatic performance difference in Figure 4 for **MICCG**.

5.2.1 Synchronization Success Rates

Another measurement of interest is the synchronization fault rate in the fine-grain implementation, *i.e*, the percentage of failed synchronizations. Figure 5 shows the number of synchronization faults as a percentage of the total number of synchronization operations for **SOR** and **MICCG**. This data is presented for 16 and 64 processors in **SOR** and 4 and 16 processors in **MICCG** across a range of problem sizes. Notice the low fault rates experienced.

Similar results were observed for both **Multigrid** and **Gamteb**. In **Multigrid**, we observed a synchronization fault rate of 2.8% for solving a 33×33 grid on 64 processors, and a fault rate of 1.3% for solving a 65×65 grid on 64 processors. In **Gamteb**, we observed a synchronization fault rate of 6.6% on 16 processors. This evidence supports our claim made earlier that fine-grain synchronization results in successful synchronizations being the common case.

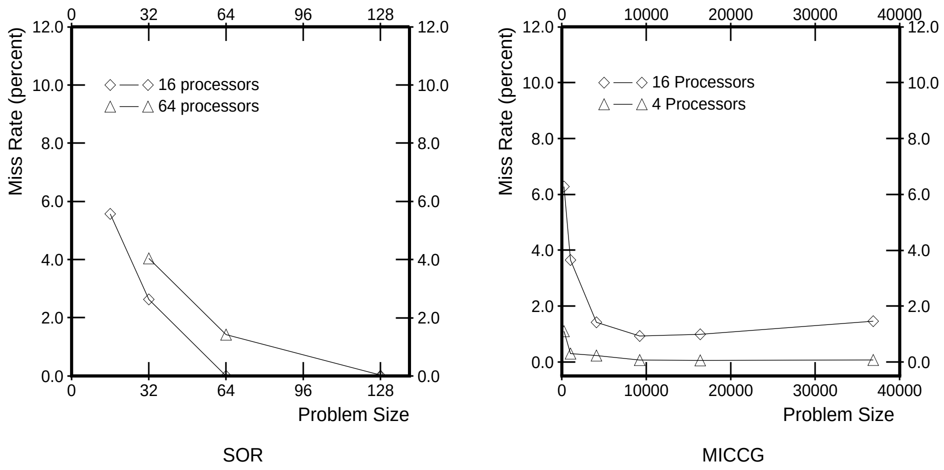

Figure 5: J-structure reference fault rates as a function of problem size for 16 and 64 processors in **SOR** and **MICCG**. For **SOR**, the problem size is the width of the grid to be relaxed. For **MICCG**, the problem size is the number of J-structure elements in the solution.

5.2.2 The Importance of Hardware Support

Finally, we wanted to address the question of the value of our hardware support for fine-grain synchronization. To do this, we ran our simulations using two different implementations of J-structures. The first uses the implementation provided by Alewife as described in this paper. The second uses an implementation without full/empty bits that explicitly allocates a word in memory for every J-structure element. This software approach has a high memory overhead. One way to reduce this overhead would be to pack multiple full/empty bits together (up to 32 per word in memory). However, schemes that pack full/empty bits introduce the difficulty of finding the word which holds the desired full/empty bit and extracting the bit with a bit mask and shift operation. Packing full/empty bits also introduces false sharing problems in cache-coherent multiprocessors. In the final version of this paper, we will explore the tradeoff of packing full/empty bits, but for our preliminary results, we use an additional word of memory for each synchronization variable.

The results of this study appear in Figure 6. The graphs in the figure show a group of three bars for each machine size simulated. The first bar, labeled “C” (for Coarse-grain), is the execution time using the coarse-grain barrier implementation. The next two bars show execution times for the fine-grain implementation. The “S” bar (for Software) corresponds to using an additional memory word as a synchronization variable for each J-structure element. The “H” bar (for Hardware) corresponds to having full/empty bit support for J-structures. All three bars in the group have been normalized against the execution time of the coarse-grain implementation; the normalizing factor in raw cycles appears directly above the “C” bar in each group (in units of thousands of cycles).

We see that the software implementation of J-structures runs slower than the hardware implementation in both applications (with the exception of **MICCG** on 16 processors; more on this shortly). This degradation in performance can be attributed to the extra communication necessary in the software implementation. With full/empty bits, both the data and synchronization variable are acquired in one memory transaction while in the software implementation, an access takes two memory transactions. Another factor contributing to this performance degradation of the software implementation is the need to explicitly check for failed synchronizations instead of automatically signalling them via traps.

There is a question of whether cache pollution contributes to the poorer performance of the software implementation. We verified in other simulations that the effect of cache pollution from increasing the working set size due to the extra word for synchronization contributes a negligible overhead of less than 5%. In these simulations, we also found that increasing the number of instructions for a J-structure access in the software implementation from 1 to 5 instructions incurs an overhead of only 10%.

In **MICCG**, the effect of extra communication when full/empty bits are not provided is consistently about 60%, while in **SOR**, this effect is a bit less and is especially small for small machine sizes. In **MICCG**, it is necessary to allocate all the data in J-structures while in **SOR**, it is possible to block the data and only use J-structures at the subgrid edges. Consequently, the frequency of synchronization operations is much higher in **MICCG**, and the full/empty bit optimization has a greater effect. Moreover, in **SOR**, the frequency of synchronization operations increases with the number of processors since the perimeter-area ratio of each subgrid grows with machine size. This explains why the full/empty bit optimization has the most effect in the 64 processor simulation of **SOR**.

Figure 6 underscores the importance of providing hardware support for efficient fine-grain synchronization. In most of the simulations, we find that the fine-grain implementation without hardware support performs worse than the coarse-grain implementation. We also notice that in **SOR**, the fine-grain implementation with hardware support performs better than the coarse-grain implementation only beyond 4 processors. This is due to the fact that in **SOR**, all the threads are extremely well balanced; the bulk of the synchronization overhead in the coarse-grain implementation of **SOR** comes from the cost of the barrier operation. In this case, going to fine-grain synchronization improves performance only when the cost of the barrier operation becomes significant, i.e. when the machine size is large. For **MICCG**, barriers are detrimental mainly because they introduce lots of false dependencies in the solver operation. In this type of application, the greatest gain of fine-grain synchronization comes from avoiding the false dependencies that barriers introduce. This gain can be attained regardless of how the fine-grain synchronization primitives are implemented. A dramatic demonstration of this is seen in the 16 processor data point of **MICCG** where the fine-grain implementation using software J-structures does significantly better than the coarse-grain implementation. Notice that this performance gain is even greater than the gain provided by having full/empty bit support. We expect that in applications like **MICCG**, the benefit of avoiding false dependencies will become increasingly pronounced for large machine sizes where false dependencies are particularly detrimental.

6 Related Work

The advantages of fine-grain synchronization have been noted before, and several machines that address this issue with varying degrees of synchronization granularity have been built or proposed, including HEP [18], Monsoon [16], Tera [3], MDP [6], Cedar [7], Multicube [8], and the KSR1 [17]. In fact, our language notation for fine-grain synchronization and hardware

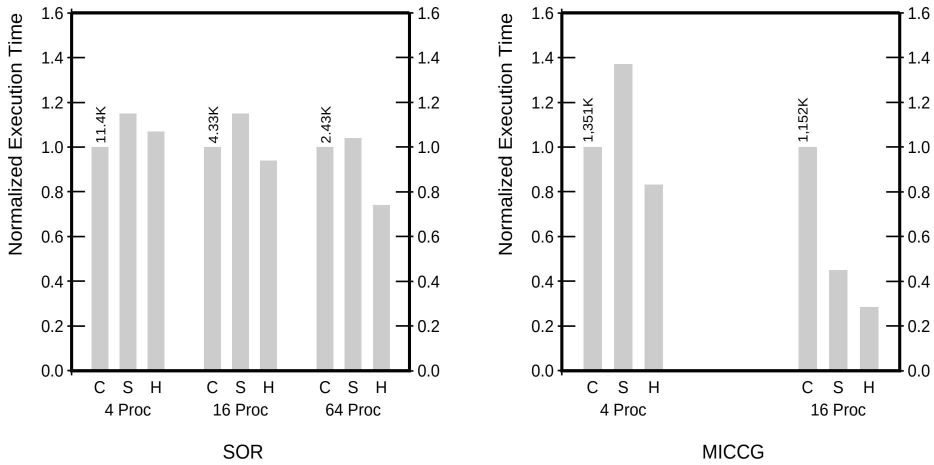

Figure 6: Normalized execution times in hardware versus software J-structures in **SOR** and **MICCG**. "C" = Coarse-grain barriers, "S" = Software J-structures, and "H" = Hardware J-structures.

full/empty bits were inspired by the HEP. The HEP, Monsoon, and Tera support word-level synchronization and bundle hardware support for fine-grain synchronization with instruction-level multithreading to hide latency. Tera provides data trap bits and a hardware retry. When a synchronization attempt fails, the hardware is responsible for storing the request and retrying it automatically. The hardware maintains a retry counter, which when exceeded causes a trap. In Monsoon, a request is issued over the network to the synchronization store. On a failed attempt, the hardware queues the request at the synchronization store. If other requests to the same location arrive, all but one is sent back to the respective requesting node, where they are guaranteed a waiting slot. A pointer is maintained by each waiting request to a waiter ahead in line. Generalizing the notion of full-empty bits, Cedar provides synchronization counters with each memory word, and supports operations on these counters in the memory hierarchy. The MDP provides 4 bits of tag with each memory word and associates synchronization types with certain combinations of the tag bits. The MDP traps into software on failed synchronization attempts. The Multicube and KSR1 associate a synchronization bit with every cache line. The implementation proposed for the Multicube used a hardware-supported mechanism for maintaining first-come first-served queues of waiting processors.

7 Conclusions

This paper discussed how fine-grain synchronization can be supported on a multiprocessor without inordinate amounts of hardware, and described a particular implementation in the context of the Alewife multiprocessor. We argued that efficient fine-grain synchronization requires support at all levels of a multiprocessor system. Driven by the desire to control hardware complexity, we identified successful synchronizations as the common case in fine-grain synchronization and provided hardware support for that case. In the common case of successful synchronizations, the system incurs very little overhead over normal non-synchronizing reads and writes. Experimental results indicate that a high percentage of fine-grain synchronization operations succeed.

Failed synchronizations that need to wait are detected via traps and handled entirely in software. The trap handlers implement sophisticated waiting algorithms that rely on useful information from the compiler to reduce the cost of waiting. The compiler communicates this information to the trap handlers using otherwise unused bits in the instruction.

For programmability, we provide language support in parallel versions of C and T to allow a user to specify fine-grain synchronization in a straightforward manner. We have implemented a number of parallel programs that utilize fine-grain synchronization specified with this language support.

The resulting system needs very little hardware support such that we were able to implement a processor that supports our design for fine-grain synchronization by making minor modifications to the LSI Logic SPARC, an existing off-the-shelf processor. The processor, called Sparcle, has been fabricated and is fully functional. We have run several programs that use fine-grain synchronization on a single-node Sparcle testbed. Sparcle is expected to clock at 40 MHz in the Alewife system. The memory system provides a bit for each memory word and some logic for trapping the processor on synchronization faults. The cost of one extra bit per memory word is small and will be even smaller in a 64-bit machine. Moreover only globally accessible memory need to include full/empty bits. The full/empty trap logic accounts for a mere 500 gates out of 75,000 gates in the Alewife memory controller chip.

The system was analyzed by running several applications using fine-grain and coarse-grain synchronization on an Alewife simulator. The measurements show that, compared to barrier synchronization, fine-grain synchronization leads to significant performance gain. In **SOR**, all of the performance gain is attributed to our modest hardware support, while in **MICCG**, a large fraction of the performance gain is due to the expression of fine-grain synchronization, and a smaller, but significant (50%) gain is due to the hardware support. The experiments also support our assumption that most potentially synchronizing operations will, in fact, not have to wait. To be sure, not all applications will benefit from fine-grain synchronization, but we stress that our design will not negatively affect such applications.

The preliminary evidence presented in this paper suggests that it is worth having modest hardware support for fine-grain synchronization. When the Alewife machine becomes available we will be able to run more extensive experiments and come to a stronger conclusion one way or the other.

References

[1] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, New York, June 1990.

[2] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] Gail Alverson, Robert Alverson, and David Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Workshop on Multithreaded Computers, Proceedings of Supercomputing '91*. ACM Sigraph & IEEE, November 1991.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, September/October 1986.

[5] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, August 1991.

[6] W. J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, Washington, D.C., June 1987. IEEE.

[7] 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.

[8] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, Hawaii, June 1988.

[9] Douglas Johnson. Trap Architectures for Lisp Systems. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[10] Kirk Johnson. Semi-C Reference Manual. ALEWIFE Memo No. 20, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1991.

[11] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[12] David Kranz, Beng-Hong Lim, Kirk Johnson, John Kubiatowicz, and Anant Agarwal. Integrating Message-Passing and Shared-Memory; Early Experience (Extended Abstract). In *Proceedings of the Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, October 1992.

[13] David A. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, June 1989.

[14] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. To appear in ACM Transactions on Computer Systems. Also available as MIT VLSI Memo 91-632, February 1991, 1991.

[15] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[16] G. M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[17] James B. Rothnie. Architecture of the KSR1 Computer System, March 1992. MIT LCS Seminar, Cambridge, MA.

[18] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[19] SPARC Architecture Manual, 1988. SUN Microsystems, Mountain View, California.

[20] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. MIT Lab for Computer Science Tech. Memo MIT-LCS-TM-479, MIT, Cambridge, MA 02139, October 1992.