Appears in: "Proceedings of the First International Symposium on HPCA", January 1995.

Thread Prioritization: A Thread Scheduling Mechanism for Multiple-Context Parallel Processors*

Stuart Fiske and William J. Dally
stuart@ai.mit.edu, billd@ai.mit.edu
Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Multiple-context processors provide register resources that allow rapid context switching between several threads as a means of tolerating long communication and synchronization latencies. When scheduling threads on such a processor, we must first decide which threads should have their state loaded into the multiple contexts, and second, which loaded thread is to execute instructions at any given time. In this paper we show that both decisions are important, and that incorrect choices can lead to serious performance degradation. We propose **thread prioritization** *as a means of guiding both levels of scheduling. Each thread has a priority that can change dynamically, and that the scheduler uses to allocate as many computation resources as possible to critical threads. We briefly describe its implementation, and we show simulation performance results for a number of simple benchmarks in which synchronization performance is critical.*

1 Introduction

Parallel processor performance is critically tied to the mechanisms provided for tolerating long latencies that occur during remote memory accesses, and processor synchronization operations. Multiple-context processors [20, 3, 13, 15] provide multiple register sets to multiplex several threads over a processor pipeline in order to tolerate these communication and synchronization latencies. Multiple register sets, including multiple instruction pointers, allow the state of multiple threads to be loaded and ready to run at the same time. Each time the currently executing thread misses in the cache or fails a synchronization test, the processor can begin executing one of the other threads loaded in one of the other hardware contexts.

For a multiple-context processor as shown in Figure 1, there are both *loaded* and *unloaded* threads. A thread is loaded if its register state is in one of the hardware contexts, and unloaded otherwise. Unloaded threads wait to be loaded in a software scheduling queue. To allow a traditional RISC pipeline design, we assume a block multithreading model [23, 3], in which blocks of instructions are executed from each context in turn, rather than a cycle-by-cycle interleaving of instructions from the different contexts [20, 15, 13]. At any given time, the processor is executing one of the loaded threads. A *context switch* occurs when the processor switches from executing one loaded thread, to executing another loaded thread, an operation that can be done in 1 to 20 cycles, depending on the processor design. A *thread swap* involves swapping a loaded thread with an unloaded thread from the software queue. The cost of a thread swap is one to two orders of magnitude greater than a context switch, because it involves saving and restoring register state, and manipulating the thread scheduling queue.

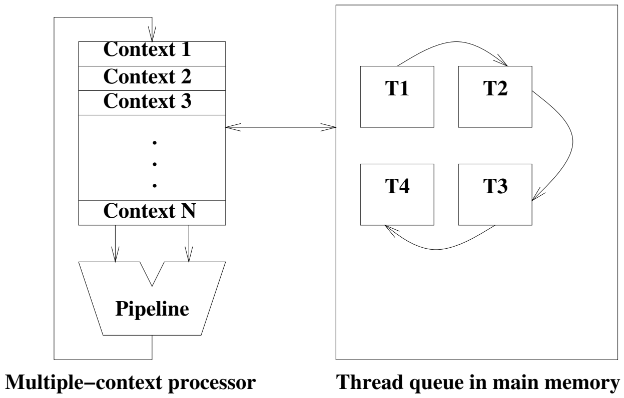

Figure 1: Multiple-context processor with N contexts.

The scheduling problem on a multiple-context processor involves two components. First, we must decide which threads should be loaded in the contexts. Second, in the case that there are multiple loaded threads, we must decide which one is executing at any given time. In this paper we

*The research described in this paper was supported by the Advanced Research Projects Agency under ARPA order number 8272, and monitored by the Air Force Electronic Systems Division under contract number F19628-92-C-0045.

show that it is important to correctly make both types of scheduling decisions. If a critical thread is not loaded, then no progress can be made along the critical path, and runtime performance suffers. If the critical threads are loaded along with other non-critical threads and the scheduler treats all the loaded threads as equal, then runtime performance suffers. Time devoted to the non-critical loaded threads could potentially be devoted to the critical threads.

Thread prioritization is a simple means of guiding the scheduling of threads on a node. Each thread has a priority that indicates the importance of the thread in the overall problem. The software scheduler on each node chooses the highest priority threads as the loaded threads. On a context switch, the hardware scheduler chooses the loaded thread with the highest priority as the next thread using simple hardware, in order to minimize context switch overhead. The goal is to devote as many of the processor resources as possible to the tasks that are known to be critical to overall performance.

This paper examines a number of benchmarks that show the effects of prioritizing at both levels of scheduling. These benchmarks evaluate the performance of barrier synchronization, queue locks, and fine-grain synchronization. Our experiments vary the number of threads and contexts per processor. When threads are prioritized, the performance of the barrier benchmark improves by up to a factor of 2, and the performance of the queue lock benchmark improves by up to a factor of 7. For the fine-grain synchronization benchmark, performance was improved by up to 26%.

This paper is organized as follows. Section 2 describes thread prioritization and outlines some of the implementation details and costs. Section 3 briefly outlines the simulation environment and assumptions, and Section 4 presents the results from a number of simple scheduling experiments. Section 5 describes related work, while Section 6 concludes the paper and discusses future work.

2 Thread Prioritization

Thread prioritization involves assigning a priority to all the different threads in an application, and then using this priority to make thread scheduling decisions. The priority reflects the importance of a single thread to the completion of a single application. The thread scheduler uses the thread priority in a very different way than process scheduling in UNIX for instance, where the goal is to achieve good interactive performance and time sharing between competing processes [16]. In our case, the goal is to identify as exactly as possible a relative order in which threads should be run, and devote as many resources as possible to the most important threads. Also, the granularity of scheduling is much different: in our case the priority is used to make scheduling decisions on every hardware context switch in a multiple-context processor.

2.1 Priority Thread Scheduling

Consider an application that consists of a set T of threads on each processor, where each processor has C contexts. Each thread $t_i \in T$ has a priority P_i, with a higher value of P_i indicating a higher thread priority. The hardware and software schedulers use the priority to do the scheduling.

First, the software scheduler uses the priority to decide which threads are loaded. Specifically, it chooses a set T_L of threads to load into the C contexts, and a set T_U of unloaded threads to remain in a software scheduling queue. The scheduler chooses the loaded threads such that $P_l \geq P_u$ for all $t_l \in T_L$ and $t_u \in T_U$. Threads of equal priority are scheduled in round-robin fashion.

Second, at each context switch the hardware scheduler uses the priority to determine which loaded thread to execute. The scheduler chooses a thread $t_x \in T_L$ such that $P_x = max\{P_l\}$ for all $t_l \in T_L$. If a thread is waiting for a memory reference to be satisfied then it is *stalled* and is not considered for scheduling until the memory reference is satisfied. If several loaded threads have the same priority, then these threads are chosen in round-robin fashion. A context switch can occur on a cache miss, on a failed synchronization test, or on a change of priority of one of the threads on the processor. Each change in priority results in a re-evaluation of T_U, T_L, and t_x. In this sense, the scheduling is preemptive.

Note as well that thread scheduling as defined here is purely a local operation. Each processor has its own set of threads, and schedules only these. We do not consider dynamic load balancing issues in this paper.

2.2 Assigning Thread Priorities

In our benchmarks the user explicitly assigns a priority to each thread, and changes this priority as the algorithm requires. Although initially the use of thread prioritization is likely to be limited to special runtime libraries (e.g. synchronization primitives) and user-available program directives, we expect that it will eventually be possible to have a compiler assign priorities to threads automatically. Automatic thread prioritization is particularly straightforward when the program can be described as a well defined DAG (Directed Acyclic Graph) that can be analyzed and used to assign the priorities.

Prioritizing threads incorrectly can lead to a number of deadlock situations. Specifically, if thread A is waiting for another thread B to complete some operation, and thread B has a low priority that does not allow it to be loaded, then deadlock results. Thus the priorities assigned to threads must respect the dependencies of the computation.

One way of avoiding deadlock is to guarantee some sort of fairness in the scheduling. If all threads are guaranteed to run some amount of time despite their priorities, then we can guarantee that deadlock will not result. However, as will be shown in the examples, doing fair scheduling without regard to priority, or not specifying the priority of threads as exactly as they could be, can lead to a serious performance penalty. Thus, both the hardware and the software schedulers assume that the prioritizing of the threads is correct and deadlock free. Between threads of the same priority scheduling is fair.

2.3 Hardware Support for Context Switching

The context switch time is the time spent in switching between two active contexts, and is an important parameter in determining the efficiency of context switching for tolerating latency. In order to effectively tolerate latency, the total context switch time in a multiple context processor must be small [1, 23].

The context switch time consists of a number of components. For instance, in the APRIL processor [3] the time required is 11 cycles, which is used to drain the processor pipeline (5 cycles), and execute a trap handler which saves state, and chooses the next context to execute using a round robin scheme (6 cycles). Duplicating instruction pointers and the processor status word for each context would reduce this time to just the cost of draining the pipeline, and more complicated processor designs could reduce this time further, possibly to as little as a single cycle [13, 15]. In a more software oriented approach, Waldspurger's flexible register relocation scheme [22] minimizes software context switching cost by allocating registers in each context to maintain an active thread data structure. Choosing the next context takes 4 to 6 instructions for round-robin scheduling.

When priority scheduling the active threads, choosing the next context on a context switch becomes more complicated, and if done in software can potentially increase the context switch time well beyond the minimum cost of draining the pipeline. For our simulations, we assume hardware support for choosing the next context, as shown in Figure 2. Each context has a special register, the priority register, into which the thread priority is loaded. The priorities of all the contexts feed into the context selection circuit that selects the next context on a context switch. When several threads have the same priority, the hardware scheduler chooses them in round robin fashion. Stalling can be incorporated into the active thread scheduling scheme by having a *stall* bit for each context indicating if it is waiting for a memory reference.

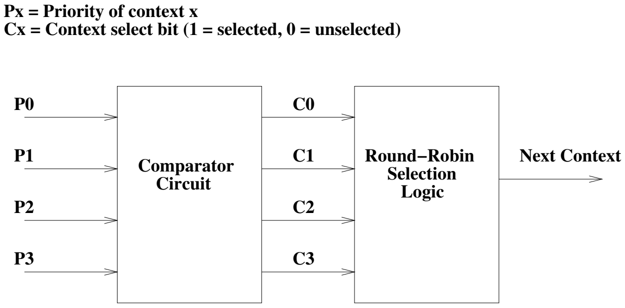

Figure 2: Priority context selection logic.

3 Simulation Parameters

3.1 Simulation Environment

Simulation experiments were run using Proteus [5], a simulator for MIMD computer architectures. It allows architectural features and parameters associated with the network, the memory system, and the processor to be varied. Programs are written in C with language extensions for concurrency. Simulator function calls support non-local interactions between processors such as shared-memory operations (including a complete set of atomic read-modify-write operations), inter-processor interrupts, and message passing. Proteus also provides a basic runtime system written in C.

One important assumption made by Proteus in order to make simulation tractable is that only the references to addresses that have been explicitly declared as shared are simulated in detail. That is, it is assumed that all local instruction and data references (to a thread's stack for instance) hit in the cache. This assumption has been found to be a reasonable approximation of the case where every single memory reference is simulated through the cache [5, 9] since the hit rates for instructions and local data are typically very high.

3.2 System Parameters

The basic system consists of a collection of multiple-context processing nodes connected by a high speed interconnection network. Each processing node has a cache and some portion of the global memory. We use a 2-dimensional torus type network that uses wormhole routing. Each processor has both a shared memory interface, as well as an efficient message passing interface with high priority interrupts that is used in a number of the benchmarks. A variation of the shared-memory, directory-based cache coherence protocol described by Chaiken [6] is used to maintain a consistent view of memory.

The system parameters that are kept constant across the different simulations are shown in Table 1. They were chosen to represent a processor similar to the MIT Alewife machine [2, 3], with modifications that reflect the increasing ratio of processor speed to memory speed, and that allow faster hardware context switching.

The local memory latency is assumed to be 20 cycles. This corresponds to the time to fill a cache line from the local memory when there is a miss in the cache, the value is in the local node memory, and no coherency protocol messages have to be sent before returning the data. If the data is not in local memory, then the time for the response is

Parameter	Value
Cache Latency	1 cycle
Local Memory Latency	20 cycles
Memory Bandwidth	1 word/cycle
Hardware Context Switch Time	5 cycles
Time to Unload Registers	32 cycles
Time to Reload Registers	32 cycles
Network Data Transfer Size	0.5 word
Network Wire Delay	1 cycle
Network Switch Delay	1 cycle
Network Input Bandwidth	0.5 word/cycle
Network Output Bandwidth	0.5 word/cycle

Table 1: Important system parameters.

variable and depends on factors such as the network traffic, and the number of messages that have to be sent to satisfy the protocol. The memory bandwidth available is 1 word/cycle.

The 5 cycle hardware context switch time is the cost of draining the pipeline on a context switch. We assume that this is the only cost of context switching, and that no additional instructions have to be executed.

The cost of a thread swap is the time to save the registers of the first thread, perform various operations on the scheduling data structures, and restore the registers of the next thread [17]. We assume that there are 32 registers in each context, and that there is a single cycle cache read and write hit time. The cost of the scheduling and descheduling is modeled directly by the cost of the runtime scheduler. The total swapping cost is on the order of 115 to 150 cycles, which assumes all references hit in the cache. This swap time is significant in comparison to the hardware context switch time.

Finally, the network transfers data one half word at a time, with a switch delay of 1 cycle, and a point-to-point wire delay of 1 cycle. The network input and output bandwidths are each 0.5 word/cycle.

4 Experimental Results

In this section we present the results from three simple benchmarks. These experiments concentrate on improving synchronization performance, which is crucial to the performance of many parallel applications. The first two benchmarks are synthetic benchmarks which look at the performance of a combining tree barrier, and of a mutual-exclusion queue lock. The third benchmark is an implementation of Lower-Upper Decomposition (LUD) that uses fine-grain synchronization in the form of a Full/Empty tag bit associated with each memory location.

For each benchmark we consider three different scenarios:

1. **SINGLE:** There are several threads, but there is only one context so that only a single thread is loaded at a time.

2. **ALL:** There are sufficient contexts so that all threads created can be loaded. We use 16 contexts in our simulations.

3. **LIMITED:** There are several contexts, but there are potentially more threads than contexts so that only a limited number of the available threads are loaded. We use 4 contexts in our simulations.

The **SINGLE** and **LIMITED** cases represent situations in which not all threads can be loaded at the same time. These cases can arise in the context of data-dependent thread spawning, runtime dynamic partitioning, or in a multiprogramming environment. The **ALL** case is balanced in the sense that all threads can be loaded at once. The first case emphasizes the type of scheduling which is required for threads that are not currently loaded. The second case emphasizes the scheduling required for loaded threads. The last case combines the two problems to study what must be done to schedule both loaded and unloaded threads.

4.1 Barrier Synchronization

The first benchmark is a barrier benchmark using a shared memory combining tree [25, 18]. In this benchmark, a number of threads are spawned on each processor, and these threads repeatedly perform a barrier synchronization. The first level of the combining tree has a fan-in equal to the number of threads on each processor[1]. The threads on each processor first perform a local combine, and then the last thread to combine on each local processor participates in a global barrier using a radix-4 combining tree. The simulation uses 64 processors, with a large fully associative cache, so that only cache invalidation traffic affects performance.

Prioritization is simple, and is done as follows. When a thread arrives at the barrier and it is not the first thread in the leaf group, it decreases its priority in preparation for the next phase of the computation, and begins to spin. The last thread to arrive at a leaf node maintains its priority, and proceeds up the combining tree. Thus on each node, only the thread that is participating in the non-local barrier tree is using any cycles — it can either be spinning at an intermediate node of the combining tree, or it can be proceeding up or down the combining tree. Once a thread going back down the combining tree reaches the leaves of the tree, it decreases its priority to the priority of the other spinning leaf threads, and they can all proceed to the next phase of the computation. Note that the prioritization required for other tree-like barriers including tournament barriers and MCS

[1] If there is only a single thread per processor, then this first level is eliminated.

tree barriers [18], is qualitatively similar to the prioritization of the combining tree barrier.

4.1.1 Results

Figure 3 shows the average barrier wait times for the three different scenarios, where the barrier wait time is the time spent by each thread waiting at the barrier. Each case is discussed below.

SINGLE: With unprioritized threads, performance of the barrier decreases as the number of threads increases due to two factors. First, each thread that participates in the barrier must be swapped into the context in order to reach the barrier. Second, when a thread that is spinning at an intermediate node of the combining tree does an unsuccessful poll, the scheduler swaps out this thread, and successively load in all the other spinning threads on the local node. It does this because it does not differentiate between the locally spinning threads and thus treats them all fairly. In the prioritized case, the time to perform the barrier increases due to the larger number of threads, but once the local barrier has been completed and one thread has been chosen to represent the node in the global barrier, this thread is never swapped out regardless of how often the polling is unsuccessful. As a result, the second component which contributed to poor performance in the unprioritized case is eliminated. For 4 threads per processor performance improves by 11%, and for 16 threads per processor performance improves 41%.

ALL: The unprioritized **ALL** scenario suffers from a similar problem to the unprioritized **SINGLE** scenario, except that no thread swapping is necessary since all threads are loaded, only context switching. Although a context switch is much cheaper than a full thread swap, the context switchs happen more often in the **ALL** case than thread swaps in the **SINGLE** case because they occur not only on failed synchronization tests, but also on cache misses. Each time the thread participating in the global barrier misses in the cache or does an unsuccessful polling operation, the processor runs through all the other contexts before returning to the critical context. It is important to note that the time to switch between the contexts is more than simply the number of cycles to switch between hardware contexts, in this case 5 cycles. This is because once the actual context switch takes place, the new thread issues some number of instructions, until it either misses in the cache, or tests its flag unsuccessfully and context switches. The prioritized scheduling eliminates unnecessary context switching during the global barrier with performance improving by 23% for 4 threads, and by 47% for 16 threads.

LIMITED: The case of having more threads than contexts with multiple contexts can potentially suffer from the worst of both the **SINGLE**, and the **ALL** scenarios. With unprioritized threads, each time a non-critical spin-

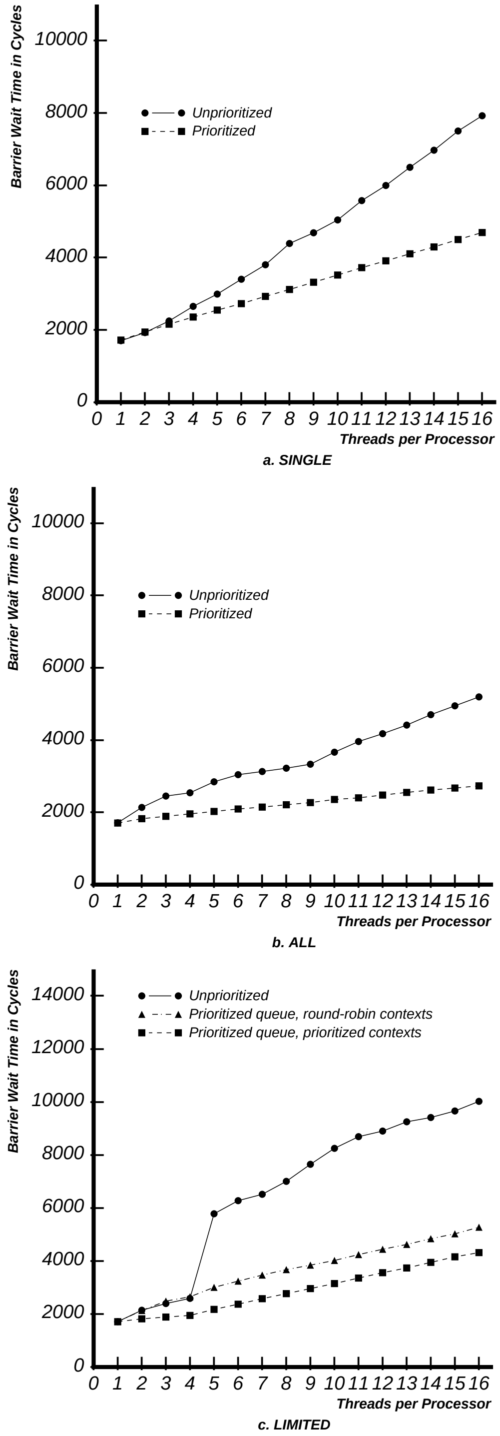

Figure 3: Average barrier wait time for 64 Processors. **SINGLE**, **ALL**, and **LIMITED** scenarios.

ning thread runs, it not only checks its flag but also does a thread swap if there are other threads on the scheduling queue. Thus the time between when the critical thread context switches to the time it is again the executing thread is increased by the time to run through all the other loaded threads, where each is checking its flag and then swapping itself with some other spinning thread on the software scheduling queue. This in effect represents a worst case scenario in terms of the amount of thread swapping that is done. Figure 3c also shows the case when the thread priorities are used only by the software scheduler, and not the hardware scheduler. In this case the hardware scheduler does round-robin scheduling of the loaded threads rather than priority scheduling and thus does some amount of unnecessary context switching. With 16 threads, software thread prioritization without hardware thread prioritization reduces the barrier wait time by 47%, whereas doing both software and hardware thread prioritization reduces the wait time by 57%.

These three scenarios show that the prioritizing of both loaded and unloaded threads is important for the performance of the barrier synchronization. Prioritizing unloaded threads in the thread queue is important because it guarantees that threads that still have to participate in the barrier are loaded, and it eliminates unnecessary thread swapping. Prioritizing the loaded threads themselves guarantees that lower priority spinning threads do not steal cycles from the higher priority threads, so that a critical thread can proceed as soon as it is able.

4.2 Queue Locks

A queue lock is a mutual exclusion mechanism that is appropriate for high contention locks [18, 4]. Each thread inserts itself onto a queue, and then spins on its own flag so as to not generate the hot spots and excess network traffic that can be generated by simpler Test-and-Set style locks. Our implementation of queue locks is inspired by the MCS lock of Mellor-Crummey and Scott [18]. The MCS lock has the advantage that the flag on which each thread spins while waiting in the queue is locally allocated and generates no global traffic while spinning.

Scheduling threads waiting for a lock raises a number of issues. It is clear that once a lock is acquired the thread owning the lock should not be swapped out [26, 17]. This is because all other threads waiting for the lock will be unable to make progress until the lock is released, and so performance can be seriously degraded. In the case of the queue lock there is an additional factor to be considered: the order in which threads are going to acquire the lock is determined by the order in which threads are inserted into the queue. The priority of a spinning thread should be determined by the position of the thread in the queue, so that when there are multiple spinning threads, the one earlier in the queue will be given priority.

The synthetic queue lock benchmark consists of an equal number of threads on each processor, each trying to obtain a lock. Each thread repeatedly obtains the lock, runs a critical section, releases the lock, and then runs a non-critical section. Both a high contention and a low contention case are run. In the high contention case, the critical section is about 100 cycles, and the non-critical section varies between 50 and 150 cycles, with a uniform distribution. In the low contention case, the critical section is the same, but the non-critical section varies between 5000 and 15000 cycles with a uniform distribution. When executing the non-critical section, a context switch is forced about every 40 cycles to simulate a cache miss. The total number of locks acquired over a sample period of 10^6 cycles was taken as the figure of merit. It should be noted that the latency tolerance properties of having multiple threads, as well as possible cache interference between threads are not measured in this benchmark, but rather only the effects of the thread scheduling. The simulation uses 16 processors, with a large fully associative cache so that only invalidation traffic occurs.

The following variations on the queue lock benchmark were run:

- **Priority1**: the threads repeatedly context switch when they poll their local variable and determine that they are not next in line in the queue. If there are more threads than contexts, then there is also a thread swap with an unloaded thread. A thread that acquires the lock has high priority so that it is not swapped out, and when it releases the lock it decreases its priority.

- **Priority2**: If a thread owns the lock or is trying to determine its position in the queue, it has the highest priority. If the thread is trying to insert itself into the queue, it has the next highest priority[2]. Threads that are spinning in the queue have a priority based on their position in the queue[3]. Thus whenever a processor has several threads waiting in the queue, it will give priority to the thread that will next acquire the queue. Note that this requires the addition of a count field to the data structure in order to keep track of the position in the queue, and slightly more complicated lock acquisition code.

- **Signalling1**: In this version, the benchmark has been modified so that after inserting itself into the queue a thread suspends itself. A suspended thread is put into a

[2]Their is a subtle issue here having to do with a thread trying to release the lock while the next element into the queue is in the midst of inserting itself in the queue. The thread owning the lock has to drop its priority temporarily to allow the insertion to take place before it can release the lock.

[3]Note that the priority only needs to be calculated once during insertion into the queue, and does not have to be recomputed each time the lock is released.

suspended thread data structure, and is put back in the scheduling queue once it is explicitly woken up. When the suspended thread is next in the queue, a message is sent by the previous thread in the queue to wake it up. This requires that the location and the ID of the next thread in the queue be stored in the queue data structure. A thread owning the lock has high priority.

- **Signalling2**: This version combines signalling and spinning. When a thread first acquires the lock, it sends a message to the next thread in the queue to increase its priority. It releases the lock by writing the shared memory location on which the next thread in the queue is spinning. Note that this scheme can only improve performance if the critical section is long enough so that the next thread has a chance to be loaded before the lock is released.

4.2.1 Results

The results of the simulations for the three different scenarios, **SINGLE**, **ALL**, and **LIMITED** are shown in Figure 4 and are discussed below.

SINGLE: Figures 4a and 4b show the results for the **SINGLE** scenario. The main problem is making sure that the next thread to acquire the lock does so in a timely fashion. The **Priority1** scheme does not solve this problem because the next thread in the queue is still likely to be in the software scheduling queue of another processor and the next acquisition of the lock will be delayed until this thread is loaded. In both the high and low contention cases the performance drops significantly as the number of threads per processor goes from 1 to 16, by factors of 5.7 and 3.3 respectively. The **Priority2** scheme deals with this problem by prioritizing spinning threads such that their priority depends on their position in the queue. On any given processor, the next thread that is to acquire the lock is loaded and spins waiting for the lock to be released, which results in much better performance. The **Signalling1** scheme also performs consistently. Passing the lock requires a message send and the loading of the receiving thread's state into a hardware context. This can require a full thread swap if the receiving processor is already busy. It is never the case that the next thread is ready and waiting to acquire the lock. Because of this, **Signalling1** performs 8% worse than **Priority2** in the high contention case, and 11% worse in the low contention case when there are 16 threads per processor and passing the lock nearly always requires a full thread swap. **Signalling2** performance drops off as the number of threads increases because the overhead of passing a lock has now increased. The length of the critical section is increased because the thread owning the lock must both send a message and change the shared memory variable on which the remote thread is spinning. On the remote node, expensive re-scheduling operations must take place based on the changed priorities. In the low contention case the performance of **Signalling1**, and **Signalling2** become similar because both typically require a thread swap operation when the next thread is signalled to acquire the lock.

ALL: Figures 4c and 4d show the results for the **ALL** scenario. In the high contention case, **Priority1** performance suffers because of unnecessary context switching when it should wait for a critical reference to be satisfied. Performance drops by a factor of 2.4 in going from 1 to 16 threads per processor. The jog in the curve at 5 threads is caused by one particular critical reference in the acquisition of the lock. This reference misses in the cache and then the processor context switches 5 times before it returns to the original context, which is often just enough time for the reference to be satisfied. **Priority2** performs best because it keeps all threads loaded and executes them in the correct order. **Signalling2** performs better than **Signalling1** because the it never suspends a thread. In the low contention case and a low number of threads, performance is no longer dominated by the performance of the lock, and all 4 scenarios perform similarly up to 9 threads per processor. However, once the number of loaded threads per processor reaches 9 the contention on the lock again becomes important for **Priority1** and its performance of falls off because of the extra context switching when trying to acquire the lock.

LIMITED: Figures 4e and 4f show the results for the **LIMITED** scenario. In the high contention case, **Priority1** performance drops by a factor of 7.4 in going from 1 to 16 threads per processor, with a dramatic drop occurring once there are more threads than contexts due to all the thread swapping done on failed polling operations. The performance of the other three cases is much the same as in the **ALL** scenario when there are 4 or less threads per processor, and much the same as in the **SINGLE** scenario when there are more than 4 threads. Having more contexts helps the **Signalling1** scenario in the low contention case, because it is more likely that a context will be free when a thread's state has to be loaded in order for it to acquire the lock.

Several general observations can be drawn from this queue lock study. First, some sort of prioritization is helpful to make sure that threads acquire the lock in reasonable time, without having to resort to polling threads by continuously swapping them in and out of the loaded set to determine the next thread in the queue. Associating a priority with each thread based on its position in the queue is one approach to solving the problem. It allows the next thread that is to acquire the lock to be loaded and ready to accept the lock. Note however that determining a prioritization of threads that is both correct and performs well is trickier than one might expect. Using a signalling mechanism to wake up the next thread in the queue is also a reasonable approach,

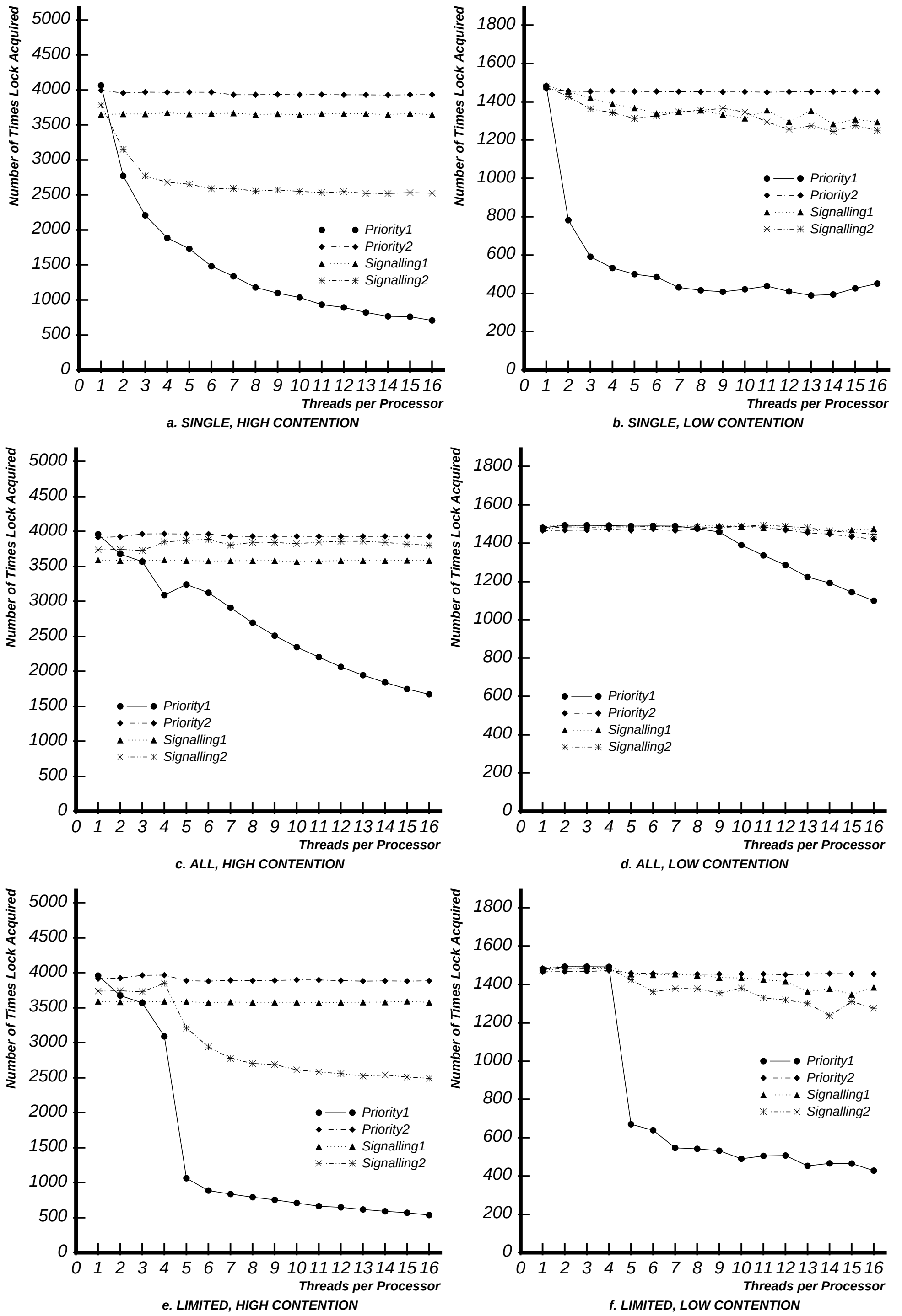

Figure 4: Queue Lock acquisitions. **SINGLE**, **ALL**, and **LIMITED** scenarios with high and low lock contention.

but incurs the extra overhead of swapping threads in and out of contexts. Also, the relative performance of the three cases that prioritize effectively, **Priority2**, **Signalling1**, and **Signalling2**, depends very much on the assumptions made about thread swap time, the length of the critical section, and the cost of message sends. In particular, if the cost of waking up a suspended thread is assumed to be higher than just the minimum time to load the context from the cache (due to cache misses for instance), the relative advantage of **Priority2** over **Signalling1** will increase. Also, if the cost of loading a thread is sufficiently high and the length of the critical section is sufficiently long, then **Signalling2** will have an advantage over **Signalling1**.

4.3 Fine-Grain Synchronization LUD

Barrier synchronization provides a coarse level of synchronization whereby different stages of a computation are clearly separated. Data dependencies between stages are guaranteed to be satisfied because all the processors finish one stage before any processors begin the next stage. Fine grain synchronization is another means of enforcing producer/consumer synchronization, but it does so at the level of individual data items. For the purposes of producer/consumer synchronization, fine grain synchronization provides two main benefits [24]: first, it eliminates the need for the global communication required in a barrier synchronization and second, each thread only waits for the data it needs so no thread must wait unnecessarily.

For our purposes, fine-grain synchronization is implemented with J-Structures [14] using Full/Empty bits that are associated with each memory location. When a thread tries to read a location which is empty or tries to write a location which is already full, a synchronization fault occurs. In some implementations this can result in a synchronization fault handler enqueueing the thread in a queue of threads to be woken up once the synchronization condition is met. In our case, a synchronization fault results in a context switch (and potentially a thread swap operation) and a retry of the reference at a later time.

4.3.1 LUD

One way of solving a system of linear equations $Ax = b$ is to first find the LU decomposition of the A matrix, followed by forward and backward substitution steps to find the vector x [12]. The decomposition phase of the algorithm is $O(N^3)$ and represents the major portion of the computation for large problems. This phase of the algorithm uses Gaussian elimination to find the lower triangular matrix L and upper triangular matrix U such that $A = LU$. The algorithm with partial pivoting is shown in Figure 5. At each iteration one row and one column of the final LU matrix is determined. Each iteration requires the following 4 steps to be taken:

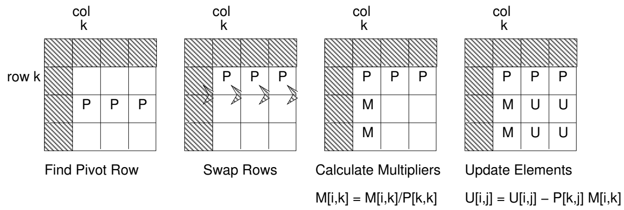

Figure 5: LUD with partial pivoting.

1. Search all elements in the leftmost column of the current submatrix for the element with the largest absolute value. This element is the pivot and its row is the pivot row.
2. Switch all the elements of the pivot row and the topmost row of the current submatrix.
3. Calculate the multiplier column by dividing all the elements below the pivot by the pivot.
4. Update all elements in the new submatrix which excludes the topmost row and leftmost column of the current submatrix, by subtracting the product of the multiplier corresponding to the element's row and the element in the pivot row from the same column.

The benchmark does a 64X64 LUD with matrices distributed in a column interleaved fashion across the processors. Processors are responsible for calculating all values related to the columns that they own. Note that there is some inherent load imbalance both in the problem itself, and in the way of dividing up the work across the processors. The load imbalance within the problem comes from the fact that there is a different amount of work required for calculating the final value of each column, and from the fact that the work for each column is statically allocated across the processors. The benchmark uses 16 processors. Each processor has a small 1Kbyte, 4-way set-associative, 16 bytes per line (2 64-bit words) cache. This small cache allows relatively little reuse until the very end of the computation, and highlights the latency tolerance provided by the multiple contexts. Two different versions of the benchmark were used[4]:

1. Fine-Grain with No Prioritization (**FG_NP**): The fine-grain synchronization version of the program uses the

[4]Though a simple LUD example has been chosen here for illustration purposes, it should be noted that there is generally enough easily exploitable parallelism in LUD to achieve good performance without resorting to fine-grain synchronization. The real benefits of fine-grain synchronization are only obvious in more complex wavefront computations such as the preconditioned conjugate gradient computation discussed by Yeung and Agarwal [24].

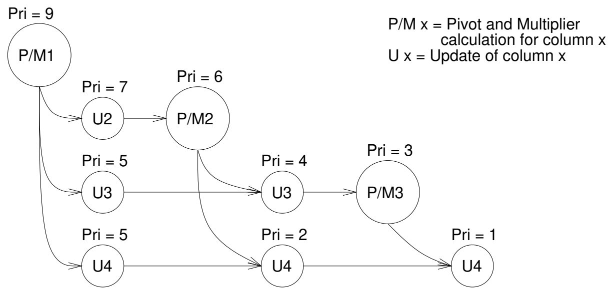

Figure 6: Critical path prioritization of LUD tasks for a 4 column problem.

fact that it is not necessary to wait for all the processors to finish before calculating the next pivot and multiplier column. At each stage, each processor generates one thread to update each column it owns. Also, the processor responsible for generating the next multiplier column generates a thread to do so. Thus stages of the computation related to different multiplier columns can proceed at the same time. Threads are run in FIFO manner, and if a synchronization fault occurs, a new thread is swapped in if one is available.

2. Fine-Grain with Prioritization (**FG_P**): This version is the same as **FG_NP**, but the threads are prioritized. At any given stage the thread that is updating the first column is given higher priority than the threads calculating the other columns, as is the thread that is responsible for generating the next multiplier column for use in the next stage. In this way, two stages of the computation are nearly always active, and the calculation of the next multiplier column is effectively overlapped with the updating of the submatrix. An example of this prioritization is shown in Figure 6 for a simple 4 column problem.

4.3.2 Results

Figure 7 shows the running time of the benchmark for a varying number of contexts. Note that this is different from the previous synthetic benchmarks, in which the number of contexts was held fixed, and the number of threads were varied.

The **FG_P** version of the benchmark is from 19% to 26% better than the **FG_NP** version. **FG_NP** performs worse because it does not give special treatment to the threads responsible for generating the next multiplier column. When it spawns threads to update the submatrix it begins to execute these threads, and they occupy all the contexts. The critical thread responsible for finding the pivot and the multiplier

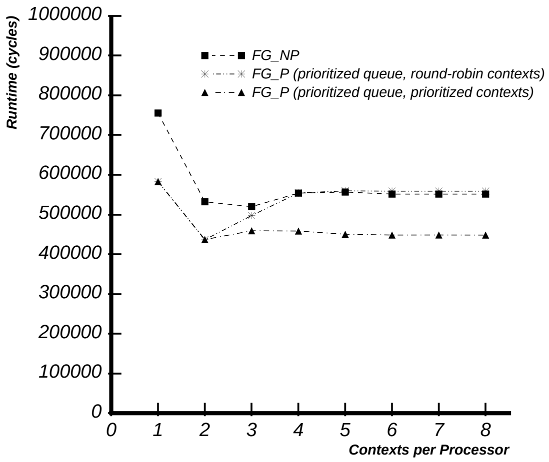

Figure 7: 64x64 LUD benchmark on 16 processors, with a variable number of contexts.

column is typically sitting in the thread queue, and waits until a context is free before it executes. This delays the calculation of the next pivot column to the point that it cannot be completely overlapped with the update phase. The **FG_P** version prioritizes the threads in such a way that generating the next multiplier column is given priority over updating the current submatrix. As a result, the update threads for the next stage are generated before the current update stage has completed.

It is interesting to note the effect of the hardware priority scheduling. A **FG_P** curve which uses round-robin scheduling for the contexts is shown in Figure 7. For a small number of contexts, the round-robin scheduling performs approximately the same as the **FG_P** with prioritized contexts. However, performance becomes similar to the **FG_NP** when the number of contexts increases to 4. This is expected since there are no threads in the thread queue when there are this many contexts, and it reflects the importance of prioritized context scheduling in the case that there are many contexts. It should be noted that for such a small system with relatively short latencies, 2 or 3 contexts are all that is necessary to get most of the latency tolerance benefits of multi-threading. Using more contexts can lead to different thread's working sets interfering with each other in the cache without providing any additional benefits [1, 19, 21]. This is the reason that the running time of both **FG_NP** and **FG_P** increase slightly with more contexts. Prioritizing threads helps with this problem as well, by favoring execution of the critical threads and hence favoring their working sets being in the cache. We are doing more study to quantify this effect.

5 Related Work

Numerous studies have been done on implementations of synchronization primitives such as locks and barriers [4, 10, 8, 18]. These studies do not address the issues that arise when there are multiple contexts, or more threads than available contexts. Two-phase algorithms have been studied as a method for deciding whether a thread should spin or block on a synchronization failure [11, 17]. A two-phase algorithm first spins for a determined amount of time in the hope that the synchronization condition will be satisfied and the blocking overhead will be avoided, and then blocks if this is not the case. In particular, Lim and Agarwal [17] study two-phase algorithms in the context of a multiple context shared memory multiprocessor. In contrast, our approach uses exact information about the priority of the thread in the global computation to decide whether a thread should spin or be swapped out. These two-phase algorithms could also be used in conjunction with our approach, in the case that it is impossible to determine an absolute prioritization of threads.

Perhaps closest in spirit to our work is the graph-level priority scheduling done by Evripidou and Gaudiot [7], in which they augment the data driven scheduling of a parallel data flow machine with priority information. In the context of iteratively solving linear systems, they used the priority so that operations from earlier iterations are given higher execution priority, which improved performance and resource utilization. Our work is different in that it concentrates on a more conventional multithreaded architecture that runs much coarser grained threads. As a result both the implementation, and the way the priority is used are different.

6 Conclusions and Future work

In this paper we present thread prioritization as a fundamental mechanism for scheduling threads on a multiple-context parallel processor. The main goal of thread prioritization is to schedule threads so that processor resources are devoted to the most critical threads. Each thread has a priority and this priority is used by both the software and hardware schedulers. The software thread scheduler uses the thread priority to keep critical threads loaded. The hardware thread scheduler uses the priority to choose the most critical loaded unstalled thread for execution, using simple hardware to minimize context switch time.

Two synthetic benchmarks and one simple application are studied. When threads are carefully prioritized these benchmarks show best case performance improvements which range from 26% to 700%. The experiments show that the software thread prioritization which keeps critical threads loaded is more important than the hardware thread prioritization which chooses the highest priority loaded thread for execution, because it avoids unnecessary and expensive thread swap operations. The hardware prioritization is not so important when there are few contexts (less than 4), but becomes increasingly significant as the number of contexts increases above 4. Performance improves the most in cases that use spinning as a synchronization mechanism, as is the case for the barrier and queue lock benchmarks. This is because the spinning threads consume processor cycles without making any forward progress, delaying threads that can make progress. The performance of the fine-grain synchronization LUD benchmark also improves, due to a re-ordering of thread execution based on the thread priorities which allows the scheduler to recognize the critical threads and devote more processing resources to them.

At a more general level, prioritization as described in this paper really refers to the use of extra information about the problem to make intelligent scheduling decisions. This information is encapsulated into a simple form, which can be easily and efficiently used by the software and hardware schedulers. Although we restricted ourselves in this paper to simple synchronization benchmarks in which the priority of each thread was based on some obvious critical path, it is also possible to use other criteria for assigning priorities. For instance, we are actively investigating the use of thread prioritization to improve cache performance. We also envision thread prioritization as being useful in making load balancing decisions, and operating system scheduling decisions.

Acknowledgements

We would like to thank Richard Lethin, Steve Keckler, and Kathy Knobe for their helpful comments and their careful reading of various versions of this paper. We also thank the anonymous referees for their comments.

References

[1] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, October 1992.

[2] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

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

[4] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] Eric A. Brewer, Chrysanthos N. Dellarocas, et al. Proteus: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1991.

[6] David L. Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. LCS/TR 489, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1990.

[7] Paraskevas Evripidou and Jean-Luc Gaudiot. Block Scheduling of Iterative Algorithms and Graph-Level Priority Scheduling in a Simulated Data-Flow Multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):398–413, April 1993.

[8] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75. ACM Press, April 1989.

[9] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263. ACM, May 1991.

[10] Debra Hensgen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[11] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 41–55, 1991.

[12] A. H. Karp. Programming for Parallelism. *Computer*, 20(5):43–57, May 1987.

[13] Stephen W. Keckler and William J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202–213, Queensland, Australia, May 1992. ACM.

[14] David Kranz, Beng-Hong Lim, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. Technical Report MIT/LCS/TR-470, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, June 1992.

[15] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318. ACM Press, October 1994.

[16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of 4.3BSD UNIX Operating System*. Addison Wesley, 1990.

[17] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. Technical Report 342, University of Rochester, Computer Science, Rochester NY, April 1990.

[19] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *ACM Symposium on Parallel Algorithms and Architecture*, pages 169–178. ACM, July 1990.

[20] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, Col, 1981.

[21] Radhika Thekkah and Susan J Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337. ACM Press, October 1994.

[22] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 120–130, San Diago, May 1993. ACM.

[23] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 1989. ACM.

[24] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Principles and Practices of Parallel Programming, 1993*, pages 187–197, San Diego, CA, May 1993. IEEE. Also as MIT/LCS-TM 479, October 1992.

[25] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[26] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.