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.*

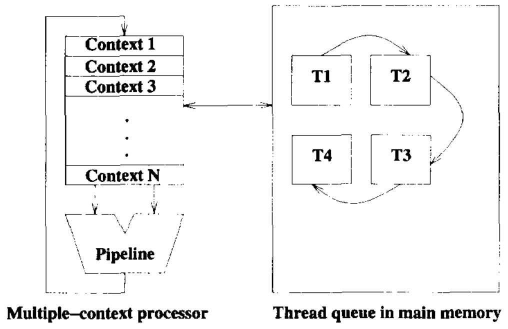


Figure 1: Multiple-context processor with N contexts.

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.

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.

0-8186-6445-2/95 $04.00 © 1995 IEEE

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 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].

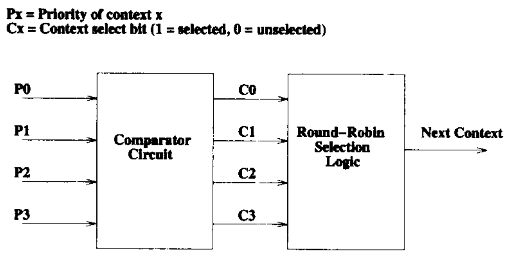
The context switch time consists of a number of components. For instance, in the APRIL processor [3] the time required is 11 cycles, which is used to drain the processor pipeline (5 cycles), and execute a trap handler which saves state, and chooses the next context to execute using a round robin scheme (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.

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.



Px = Priority of context x
Cx = Context select bit (1 = selected, 0 = unselected)
P0
P1
P2
P3
Comparator Circuit
C0
C1
C2
C3
Round-Robin Selection Logic
Next Context

Figure 2: Priority context selection logic.

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-

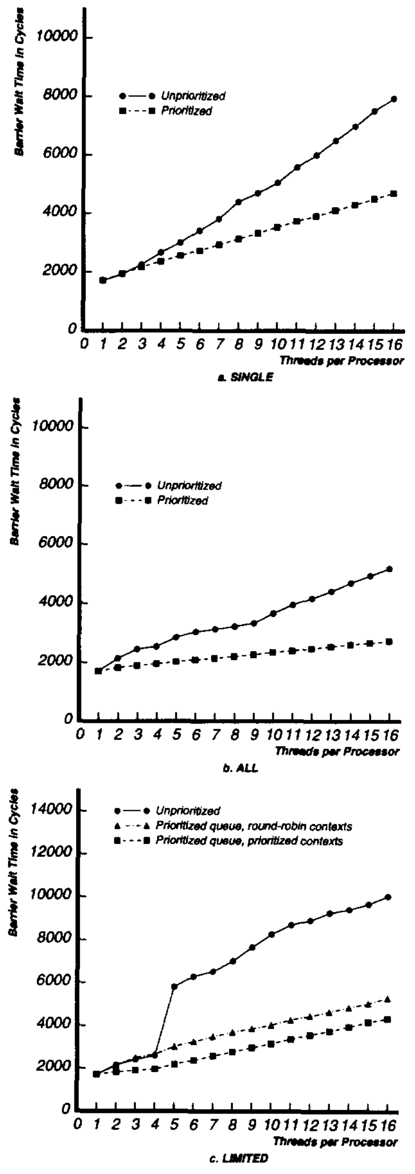


Figure 3: Average barrier wait time for 64 Processors. **SINGLE**, **ALL**, and **LIMITED** scenarios.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.