IEEE
Advancing Technology for Humanity

DECLARATION OF GORDON MACPHERSON

I, Gordon MacPherson, am over twenty-one (21) years of age. I have never been convicted of a felony, and I am fully competent to make this declaration. I declare the following to be true to the best of my knowledge, information and belief:

1. I am Director, Board Governance & Policy Development of The Institute of Electrical and Electronics Engineers, Incorporated ("IEEE").

2. IEEE is a neutral third party in this dispute.

3. I am not being compensated for this declaration and IEEE is only being reimbursed for the cost of the article I am certifying.

4. Among my responsibilities as Director, Board Governance & Policy Development, I act as a custodian of certain records for IEEE.

5. I make this declaration based on my personal knowledge and information contained in the business records of IEEE.

6. As part of its ordinary course of business, IEEE publishes and makes available technical articles and standards. These publications are made available for public download through the IEEE digital library, IEEE Xplore.

7. It is the regular practice of IEEE to publish articles and other writings including article abstracts and make them available to the public through IEEE Xplore. IEEE maintains copies of publications in the ordinary course of its regularly conducted activities.

8. The article below has been attached as Exhibit A to this declaration:

A.	S. Fiske, et al.; "Thread prioritization: a thread scheduling mechanism for multiple-context parallel processors", published in Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture, date of conference January 22-25, 1995.

9. I obtained a copy of Exhibit A through IEEE Xplore, where it is maintained in the ordinary course of IEEE's business. Exhibit A is a true and correct copy of the Exhibit, as it existed on or about May 11, 2023.

10. The article and abstract from IEEE Xplore shows the date of publication. IEEE Xplore populates this information using the metadata associated with the publication.

11. S. Fiske, et al.; "Thread prioritization: a thread scheduling mechanism for multiple-context parallel processors", published in Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture, date of conference January 22-25, 1995. Copies of the conference proceedings were made available no later than the last day of the conference. The article is currently available for public download from the IEEE digital library, IEEE Xplore.

12. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with the knowledge that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. § 1001.

I declare under penalty of perjury that the foregoing statements are true and correct.

Executed on: 5/11/2023

DocuSigned by:
Gordon MacPherson
E768DB210F4E4EF...

EXHIBIT A

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.

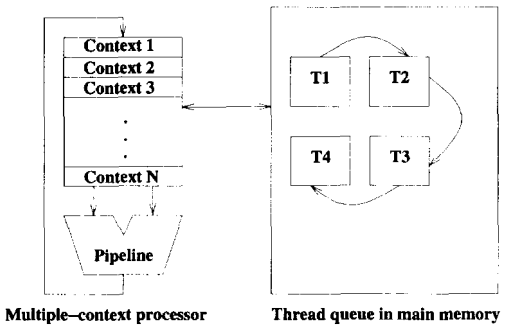

Figure 1: Multiple-context processor with N contexts.

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.

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

DOCKET ALARM

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.