# A Case for the Multithreaded Processor Architecture

*Ghulam Chaudhry and Xuechang Li*

Computer Engineering Research Laboratory
Department of Electrical and Computer Engineering
The University of Missouri at Columbia
600 West Mechanic
Independence, MO 64050
Chaudhry @ miran.cuep.umkc.edu

## ABSTRACT:

As cache based shared-memory multiprocessors are scaled (the number of processors are increased), there will be an increase in the network latency, due to cache coherence and synchronization delays. These parameters can cause a significant drop in processor utilization. Once a process has remote procedure call or need data from remote side, the processor has to be idle or make context switch to load the next process in which the CPU (Central Processing Unit) time is wasted.

By maintaining the multiple threads in the hardware, and switching among them, the multithreaded processor can overlap the waiting time for the synchronization delay so that it decreases the processor idle time.

This correspondence describes the multithreaded processor architecture, in which there are a number of hardware contexts per processor. It uses coarse-grain method to schedule threads and two-phase waiting algorithm to synchronize the waiting threads. From this architecture, we can study how many hardware contexts are needed, so the processor utilization can be increased. Moreover, we can study the effect on the utilization by changing the cache miss ratio.

**Keywords:** shared-memory multiprocessor, interconnection networks, synchronization, multithreading.

## 1. BASIC CONCEPTS:

In the multithreaded processor, there are some concepts different from the usual machine such as the management of the threads, context switch time, waiting algorithms for the synchronization.

**Thread:** A thread is a process with its own processor state but without a separate address space. It has its own set of registers and shares the resources with other threads. Scheduling a thread is done in user space and does not require operating system intervention. A parallel program usually consists of a set of inter-communicating threads. All threads have exactly the same address space, which means they also share the same global variables.

**Context Switch:** The concept of the context switch in the threaded machine is different from the traditional view of context switching. In the traditional machine, the context switch involves saving all the states of the processor and contents in the registers into memory, and loading the states of new process into the processor. In the multithreaded processor, it is a transfer of processor control from one processor resident thread to another processor resident thread. Only a few global states need to be saved out into memory so that the overhead is much less than usual.

**Thread Management:** There are two multithreading methods which are fine-grain multithreading and coarse-grain multithreading. The former method is to cycle-by-cycle interleaving of threads. It can hide memory latency and also achieve high pipeline utilization. The latter method is that the processor executes a single thread until a memory operation involving a remote request or an unsuccessful synchronization attempt are encountered. The controller forces the processor to switch to another thread, while it serves the request.

**Waiting Algorithm:** The waiting algorithms are important for the synchronization. Once a process has a synchronization failure, different waiting algorithms give totally different results. Existing systems provide either spinning or blocking as waiting algorithms. The main problem with these algorithms is that their performance is very sensitive to waiting times. Spinning performs badly if waiting times are very long, while blocking is expensive if waiting times are short because the overhead of blocking is paid each time a wait is encountered, regardless of the wait time.

Under certain wait time distributions, a competitive two-phase algorithm never perform worse than both spinning and blocking[4].

**Switch-blocking:** It is another waiting mechanism which disables the context associated with the waiting thread and switch execution to another context. The disabled context is ignored by further context switches until it is reenabled when the waiting thread is allowed to process. It is a signalling mechanism. Contrast this with blocking, which requires unloading a thread. It is as processor-efficient as blocking and has a low-execution cost because threads are not unloaded. This assumes that there are enough hardware contexts to house the waiting threads. If we run out of contexts, unloading a thread will be necessary to provide the no-starvation guarantee of signalling mechanisms[4].

## 2. INTRODUCTION:

As we build larger and larger parallel machines, the proportion of processor time spent in useful work keeps decreasing. As we strive for great speedup in the applications through the parallelism, the communication latency, cache-coherence and synchronization delays, will have serious impact on the processor utilization. The fundamental problem that any scalable multiprocessor system must address is the ability to tolerate high communication latency or synchronization delays. Consequently, there will always be times when a processor sits idle, waiting for some remote operation to complete or due to the resources nonavailability.

One solution to solve this problem is to change single hardware context into multiple hardware contexts per processor so that it can help to mitigate the negative effect of high latency and synchronization delay. If more than one hardware contexts reside on a processor and context switch overhead is low, this idle time can be used for additional contexts. In other words, whenever a processor has to be idle due to the communication problem or synchronization delay, the processor can execute another process in other thread, which was waiting for the resources available. This approach employs the two-phase waiting algorithm (*switch-blocking*) as its waiting algorithm.

## 3. BACKGROUND:

The idea of multiple hardware contexts per processor in itself is not new. The HEP multiprocessor system from "Deneleor" also provided multiple hardware contexts per processor. Unlike the Alto, the contexts were available to arbitrary processes. The processes shared the large set of registers and at each cycle one instruction from a different process was executed. A minimum of 8 active processes (those processes that are not waiting for a memory reference to complete) were needed to keep the execution pipeline full. In order to keep the pipeline full, a large number of processes were needed. This is in stark contract to modern pipeline processors where a single process almost fully utilizes the pipeline processor.

Later, Iannucci has proposed using multiple contexts for his Hybrid Data Flow/Von Neumann Machine. Each processor consists of a hardware queue of enabled continuations. The continuations are very small in size (containing just the program counter and the frame base-register), and the hardware can switch between them in a single cycle. The registers are not saved on a context switch. The software is structured so that it does not rely on registers being valid between potential context switch points. The switch points are synchronizing references, where the section needed is not available[1].

Several years ago, the MASA architecture was proposed by Bert Halstead[2]. In this architecture, each processor has a fixed number of hardware task frames. Each task frame is capable of storing a complete process context and consists of a set of auxiliary registers (like the program counter) and a set of general purpose registers. Since the number of processes may exceed the number of task frames, the process contexts are allowed to overflow into memory. On each cycle, a context in the enabled or ready state may issue an instruction. However, once a process issues an instruction, it can not issue another instruction until the previous instruction has completed. Thus, in its current form, a process on MASA can get only 1/4 (inverse of pipeline depth) of the pipeline processor's performance which is a major drawback.

Recently, Alewife machine, designed at MIT which is a large-scale, cache-coherent, distributed-memory multiprocessor and supports a shared-memory programming abstraction. Cache-coherence is maintained using a directory-based protocol over a two-dimensional mesh network. The directory is distributed with the processing nodes and the LIMITLESS coherence scheme is used to reduce the hardware requirements. Each node consists of a processor, a cache, a portion of globally-shared distributed memory, a cache-memory-network controller, a floating-point coprocessor, and a network switch. The cache-memory-network controller is responsible for synthesizing a globally shared address space from the distributed memory nodes. The machine is also designed to support synchronization because of the increased synchronization rates expected in a large-scale multiprocessor[3].

## 4. PROPOSED ARCHITECTURE:

The architecture presented here is the model for a multithreaded processor which represents the tradeoff between increased processor utilization due to overlapping network access or synchronization delay with useful computation and the higher cache miss rates. For this purpose, assume that the processes resident on a processor have the same average cache miss rates, and processor time spent in context switching are considered wasted.

If p is the number of processor resident on a processor (or the number of hardware context), let the time between misses for each process be t(p), the time to satisfy a miss be T(p), and the time wasted in contests switching be C. A process executes useful instructions for t(p) cycles, suffers a cache miss, and waits T(p) cycles for the miss request to be satisfied before it can proceed. Context switches happen only on cache misses. t(p) represents the context-switching interval. Time wasted in context switching is called the context-switching overhead.

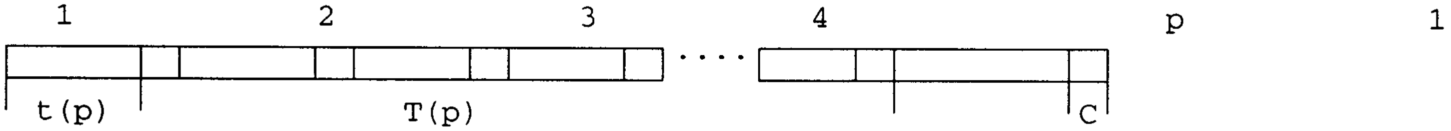

Fig. 1 Hiding wasted time by multithreading the processor.

As depicted in Fig.1, some of the time expended to satisfy a cache miss can be overlapped with useful processor execution. With p available processes and no context-switching overhead, effective processor utilization is

$$U(p) = \frac{p\ t(p)}{t(p)+T(p)} \quad \text{or} \quad U(p) = \frac{t(p)}{t(p)+C}$$

with a maximum utilization of 1[5].

(Where $p$ is the number of processes resident on a processor;

$T(p)$:the number of processor cycles to satisfy a cache miss;

$t(p)$:the number of processor cycles between misses, or the inter-cache-miss time;

$C$:the context switch overhead of the global states.)

The simulator model starts with creating the random jobs which have their own resource needs and mean processor burst time. There are two queues in the model: ready queue and block queue. A process can be in one of the queues. The random job created will be in the ready queue to wait for the turn for execution issued by the dispatching controller. Once it has a chance to be out of the ready queue, the resource needs will be checked. If the resource is available, the process will be loaded into register set and running until finish and exit according to the requirements of the process. If the process needs more network access, the resource will be checked again. If it is available, it continues execution; otherwise, it is forced to be disable and waiting for the next turn. If the resource is not available, we disable the thread associated with the waiting status and switch execution to another context. The disabled thread is ignored by further context switch until it is reenabled when the waiting thread is allowed to proceed. The schedule algorithm is using the Round Robin Method. If the process burst time is greater than the time slice, it has to wait until next turn until it can finish the all the burst time.

In this model, we are primarily concerned with the large latencies associated with cache misses that require a network access. Good single thread performance is also important. Therefore, the model continues executing a single thread until a memory operation involving a remote request or an unsuccessful synchronization attempt is encountered. The controller forces the processor to switch to another thread in which a new job will be loaded and executed. If all the threads are occupied by the waiting processes, the first thread has to be unloaded and the new job will be loaded. In this way, it can prevent the starvation and every process gets a fair chance to be executed.

## 5. CONCLUSION:

From the proposed model, we can make several experiments. When increasing the threads per processor, we can observe that the processor utilization will not increase after certain contexts per processor. This is because the more threads the processor has, the more resource competitions will appear. From the simulation model, we have seen that the high number of threads per processor can yield high utilization of the processor.

We also studied the effect of the cache miss rates on the processor utilization. Comparing the utilization of the single thread machine, the utilization of the multithreaded processor will not drop as much as that of the single thread machine. In other words, we can find the relation between the high price of the hardware contexts and improvement of the utilization the multithreaded processor can get.

## 6. REFERENCES:

[1] W. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results." In Proc. of 16th ISCA, pages 273-280, June 1989.

[2] R. H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing." In 15th International Symposium on Computer Architecture, pages 443-451, 1988.

[3] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL:A Processor Architecture for Multiprocessing," In *Proc. 17th Annual International Symposium on Computer Architecture*, June 1990.

[4] B.H. Lim and A. Agarwal, "Waiting algorithms for synchronization in Large-scale Multiprocessors," M.I.T. VLSI Memo 91-632, Feb. 1991.

[5] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Trans. Parallel Distributed Syst.*, vol. 3, no. 5, pp.525-539, Sept. 1992.