A Preliminary Performance Study of Architectural Support for Multithreading

Daniel Ortiz and Ben Lee

Department of Electrical and Computer Engineering
Oregon State University
Corvallis, OR 97331
{dortiz, benl}@ece.orst.edu

Suk-Han Yoon and Kee-Wook Rim

Computer Division
Electronics and Telecommunications Research Institute
Taejon, Korea
{shyoon, kwrim}@etri.re.kr

Abstract[1]

This paper discusses the preliminary performance study of hybrid multithreaded execution model that combines software-controlled multithreaded system with hardware support for efficient context switching and threads scheduling. The hardware support for multithreading is augmented with a software thread scheduling technique called *set scheduling*, and their benefit to the overall performance is discussed. Set scheduling schedules multiple threads onto the hardware scheduler to minimize the software scheduling and context switching costs. An analytical model of the proposed multithreaded model is discussed and simulation results of processor utilization based on the proposed model are presented. Through simulation, we find that the hybrid multithreaded execution model results in high processor utilization than traditional software-controlled multithreading.

1. Introduction

Multithreading has been proposed as a promising technique to improve the performance of shared-memory multiprocessor systems. In a multithreaded system, high processor utilization is achieved by interleaving the execution of a number of computational threads through a processor pipeline. To achieve maximum efficiency, a context-switch occurs when a thread execution blocks due to long latency operations, such as a cache miss or a thread synchronization. In the case of a cache miss, the requested data may be obtained from either the local memory—in the case of Uniform Memory Access (UMA) machines—or a remote memory access will be issued—in the case of Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines. The memory latency is then hidden by overlapping it with the execution of a new thread.

Traditionally, support for multithreading has been provided either in software or hardware. The hardware support for multithreading is done by providing fast context switching capabilities and multiple hardware context in the processor. The degree of hardware support provided can vary greatly. For example, it can be as simple as SPARC register windows for supporting multiple hardware contexts [1] or as complex as Tera multiprocessor where each processor supports up to 128 processor states and can context-switch on a cycle-by-cycle basis [2].

One approach to implementing multithreading in software is by using special languages and compilers that automatically generate multiple threads for execution. An example that follows this approach is TAM [6]. TAM relies on an appropriate compilation strategy and program representation rather that elaborate hardware. However, this approach requires languages with functional semantics and complex compiler analysis to generate threads [13].

An alternative method is to use traditional languages extended with software system support at various levels (herein refer to as *software-controlled multithreading*) [4, 9, 10]. For example, user-level multithreading support is provided by a collection of library function calls to create, synchronize, and schedule threads. At the system-level, the kernel manages all thread activities. There is also an approach where thread management is implemented entirely as a user-library [9]. One such is the POSIX 1003.4a threads extension [8], or *Pthreads* for short. Pthreads provides various functions, such as thread creation and synchronization, mutual exclusion, conditional variables, etc., to support multithreaded programming. Pthreads is widely available and runs on numerous commercial platforms including SGI-IRIX, Alpha-OSF, SPARC-SunOS, HPPA-HPUX, R2000-Utrix, etc.

In light of the aforementioned discussion, this paper presents the hybrid multithreaded model, which is a

[1] This research was supported in part by the Electronics and Telecommunications Research Institute, Taejon, Korea.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

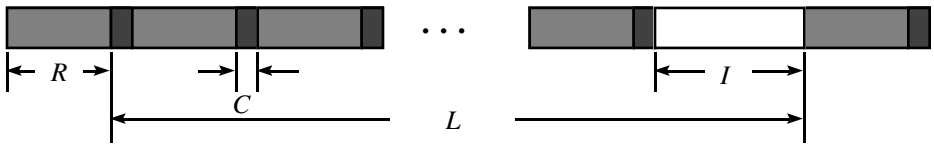

Figure 2.1. Software-controlled multithreading.

software-controlled multithreaded system extended with hardware support for efficient context switching and thread scheduling. The idea behind the hybrid approach is to utilize all the existing features in a software-controlled multithreaded system and at the same time migrate some of the responsibilities of thread management to hardware. This is achieved using a technique called *set scheduling* that acts as an interface between hardware and software and yet provides a transparent view to the programmer. The main advantage of the hybrid method is that expensive software context switching and thread scheduling costs occur only when threads are initially scheduled onto the processor and any subsequent context switching and thread scheduling are implemented in hardware. Over time, this leads to considerable reduction in the overhead cost thereby resulting in high processor utilization.

The organization of the paper is as follows: In Section 2, hybrid multithreaded execution model is described and a simple analytical model is presented. Experimental results obtained by simulation are described in Section 3. Section 4 provides a brief conclusion and possible future work.

2. The Hybrid Multithreaded Model

In order to illustrate the advantage of having hardware support for multithreading, consider the software-controlled multithreaded execution model illustrated in Figure 2.1. Each thread issues a remote reference at an interval of R cycles, i.e., run-length, and becomes blocked for L cycles waiting for the response to return before resuming execution. L depends on the memory access time and the delay through the interconnection network to and from memory. Between run-lengths, a context-switch occurs at a cost of C cycles. For the software-controlled multithreaded model, the cost of thread scheduling is included in the context switch overhead. The processor utilization U_{SC} based on this execution model is given by [12]

$$U_{SC} = \begin{cases} \dfrac{NR}{R+L} & \text{if } N < 1 + \dfrac{L-C}{R+C} \\ \dfrac{R}{R+C} & \text{otherwise} \end{cases} \qquad (1)$$

If the number of contexts supported is not sufficient, the processor will not be able to completely hide the memory latency L and will cause the processor to idle for I cycles (as in the case of Figure 2.1). On the other hand, if there is sufficient number of contexts, the processor utilization U_{SC} depends only on R and C.

As can be seen by Equation (1), the processor utilization is directly affected by the context switching and thread scheduling costs. For software-controlled multithreading, each thread is associated with a context that contains a thread ID, a set of registers including a PC, a thread priority, and a pointer to the stack. Whenever, a context switch occurs, a new thread has to be selected (i.e., scheduled) from a pool of ready threads, all the registers associated with the current thread must be flushed onto the stack before registers are loaded with the top frame of the new thread. This is done automatically by the Thread Management System, which is expensive. To reduce this cost, the objective of the proposed hybrid multithreaded model is to provide part of these features in hardware to make multithreading as efficient as possible, and yet provide a transparent view to the programmer.

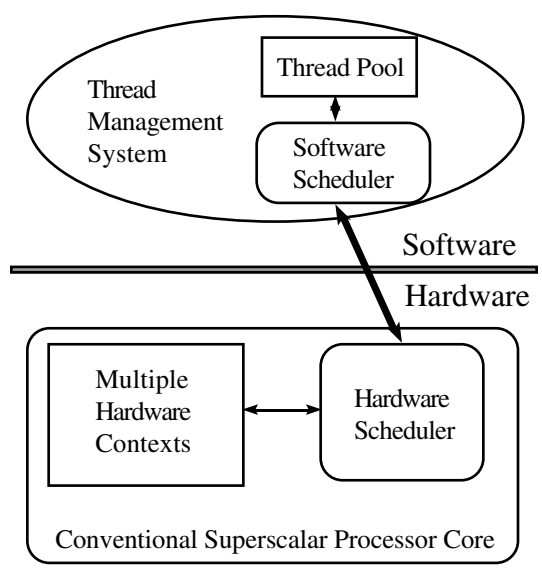

Figure 2.2. Coordination between hardware and software schedulers.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

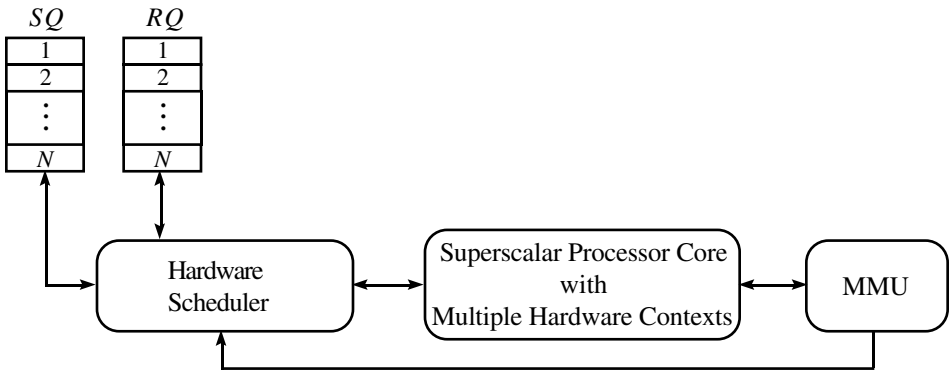

Figure 2.3. Hardware support for multithreading.

Although hardware support for thread scheduling and context switching would benefit any processor design, the challenge is to incorporate these features with minimal modifications to the operating system and the compiler, and at the same time to work within the constraints established by the base processor architecture.

Figure 2.2 shows the hardware and software schedulers that coordinate the thread selection and execution in the proposed hybrid multithreaded model. The software side of our model basically consists of an existing Thread Management System augmented with Software Scheduler that manages the Thread Pool. In most systems, the Thread Pool is implemented as a multi-level priority queue. In these systems, a thread has a priority assigned by either the Thread Management System or the user.

The responsibility of the Software Scheduler is to select a set of threads from the Thread Pool and schedule them onto the Hardware Scheduler of the processor that supports multiple hardware contexts. Threads are grouped into sets by the Software Scheduler with the objective of maximizing processor utilization. There are a number of possible policies that can be used to schedule thread sets onto the Hardware Scheduler. One straight forward approach is to schedule the next set of threads only after the previous selected threads have completed their execution. This approach is the most appropriate if thread run-lengths are about the same. However, if the thread run-lengths vary other possible scheduling policies exist. We explore these possibilities in Section 3.

Hardware support for our model consists of a conventional superscalar processor core augmented with the Hardware Scheduler and multiple hardware contexts. Once a thread has been scheduled onto the processor, it can be in one of the following three states: *running*, *ready*, or *sleeping*. The responsibility of the Hardware Scheduler is basically to maintain the control of thread states that have been scheduled onto the processor by the Software Scheduler. This is done by using the *Ready-thread Queue* (*RQ*) and the *Sleeping-thread Queue* (*SQ*). Figure 2.3 shows the hardware support needed for our hybrid model. A long latency operation detected by the memory management unit (MMU) causes a thread to context-switch. This is accomplished by the Hardware Scheduler where the blocked thread is placed in *SQ* and a new thread is scheduled from *RQ*.

In addition to the hardware support shown in Figure 2.3, a processor also needs multiple hardware contexts. This can be implemented in a number of ways. One possible method is to provide a separate, fixed-size contexts using a hardware managed register file (in the form of either register windows or duplicated register sets). However, this fixed and inflexible partitioning of the register file results in a waste of scarce high-speed registers. Since the number of registers required by thread contexts vary, a more flexible approach, called Register Relocation, has been proposed [16]. This method relies on the compiler or run-time system to mange the allocation and use of contexts. Instruction operands specify context-relative register numbers, which are numbered consecutively starting with register 0. These context-relative register numbers are dynamically combined (using an OR operation) with a special register relocation mask to form absolute register numbers that are used during instruction execution. We are currently investigating which approach is more suitable for the proposed hybrid model.

In order to manage multiple contexts, each context inside the processor is represented by a tag *T*, containing a thread ID, a PC, and a pointer to the thread stack. When threads are scheduled by the Hardware Scheduler, the tags of the threads are down loaded onto the *RQ*. A thread then can be scheduled by simply dequeuing its tag from *RQ*, updating the stack register and fetching the first instruction pointed to by PC. When a thread is blocked, its tag is placed in the *SQ* and a context-switch occurs to the next thread in *RQ*. Later, when the block thread changes its state to ready, it is enqueued onto *RQ*. When all the threads from *RQ* (i.e., within a processor) have completed their execution, the Software Scheduler schedules a new set of threads.

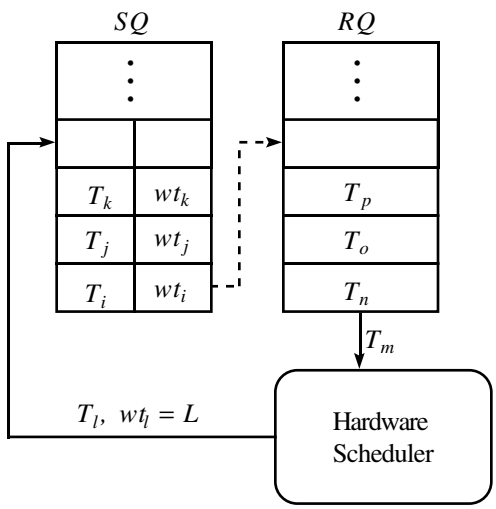

Figure 2.4. Queue management by the Thread Scheduler.

In order to keep track of the transition between sleeping and ready threads, each T in SQ is associated with a timer, wt. This is shown in Figure 2.4. When a context switch occurs during the execution of the thread T_l, it is sent to SQ with wt set to L and a new thread T_m, is selected for execution from RQ. T_l will remain in SQ for L cycles waiting for the memory to respond to its request. Eventually, when L cycles have elapsed, T_l will be placed into RQ by the Hardware Scheduler and its state will be changed to ready.

When R and L are constant, SQ will behave as a FIFO queue and thus each thread will be retired from SQ in order. However, this is not a realistic assumption because in UMA machines bus contention will cause L to vary. Moreover, in CC-NUMA machines, the network contention and routing algorithm will affect L. Variation in memory latency can be handled by mapping cache line tags to wt. The Hardware Scheduler then simply identifies threads whose request has been served by enqueuing it on to RQ.

A simple analytical model for our hybrid multithreaded system is obtained by considering the effects set scheduling operations have on processor utilization. Figure 2.5 shows the proposed multithreaded execution model through a series of set scheduling operations. During each set scheduling operation, the Software Scheduler of the Thread Management System schedules N threads onto the RQ at a cost of S cycles, i.e., $S=NC$. Between set scheduling operations, there is a total of G hardware context switches, each with a cost of c cycles, among the N contexts scheduled onto the processor.

Assuming that R, L, c, and C are constant, we can express processor utilization for two separate cases. In the first case, the number of contexts supported by the processor is not enough to hide the memory latency, and therefore the processor utilization U_H increases linearly as a function of N, i.e.,

$$U_H = \frac{NR}{R + L + \frac{NC}{G}} \tag{2}$$

where G represents the total number of context switches for all the threads and therefore G/N represents the average number of context switches in a thread. In the second case, the number of contexts is sufficient to hide the latency, thus performance loss comes from the context switching overhead and the set scheduling cost (as in the case of Figure 2.5), i.e.,

$$U_H = \frac{R}{R + c + \frac{NC}{G}} \tag{3}$$

Equation (3) shows the software scheduling and context switching cost C in Equation (1) has been replace by the hardware context switch cost c plus the amortized software context switching cost over the average number of context switches in a thread NC/G. This means even in the saturation region G/N has some effect on processor utilization. However, if G/N is sufficiently large, the processor utilization improves by a factor of $(R+C)/(R+c)$.

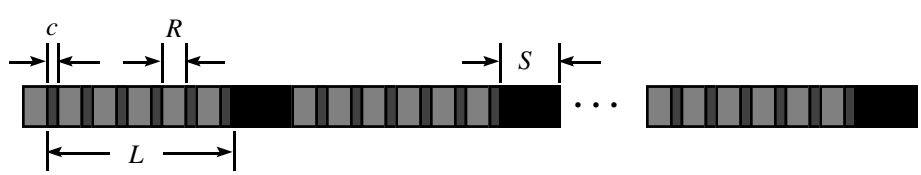

Figure 2.5. Hybrid multithreaded execution model.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

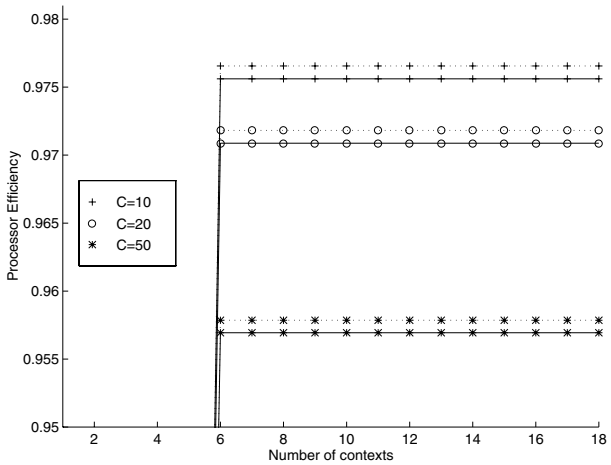

Figure 3.1. Comparison between theoretical (solid lines) and simulation (dotted lines) results.

3 . Simulation Results

In order to evaluate the performance of the hybrid multithreaded system described in the previous section, a simulation study was conducted. Figure 3.1 compares the theoretical results and the results obtained from our simulation for the hybrid multithreaded model on processor utilization when R and L are constant for various values of C. Plots were obtained by running 1,000 threads with R=100 cycles, c=2 cycles, and L=500 cycles. The comparison shows that the simulations results were comparable to the theoretical results from equations (2) and (3). More important, as C increases from 10 to 50, the processor utilization decreases only by approximately 2%. The primary reason for this is set scheduling cost is incurred only once and all subsequent context switches are done in hardware. Therefore, the hybrid method is more immune to variations in C.

To obtain a more realistic evaluation of our hybrid model, probability distributions were considered for R and L. Figures 3.2a and 3.2b shows the effects for both the hybrid and software-controlled models when R was modeled by a geometric distribution and L by a negative exponential distribution. Again, our simulation results were obtained by running 1000 threads with an overall execution time of approximately 500,000 cycles.

Figure 3.2a compares the performance when R has a mean value of 100 cycles, L has a mean value of 500 cycles, and c = 2 cycles. Results show that not only does the hybrid model outperform its software-controlled counterpart, but because it is more immune to variations in C the performance (i.e., processor efficiency) gap widens as C increases. Our findings also indicate the performance of the software-controlled execution model is strongly affected by granularity of threads. This can be seen in Figure 3.2b where R has a mean value of 20 cycles, L has a mean value of 100 cycles, and c = 1 cycle. When C/R is large, the performance of the software-controlled is severely affected by the software scheduling and context switching costs.

Another interesting observation is when thread run-lengths vary the utilization goes down (see Figure 3.1 and 3.2a-b). This is because when thread run-lengths are the same, all threads complete their execution about the same time. Therefore, scheduling the next set of threads only after the previous set of threads have completed execution will cause minimal idling. However, when thread run-lengths vary, some threads will complete first reducing the number of threads from which to context switch.

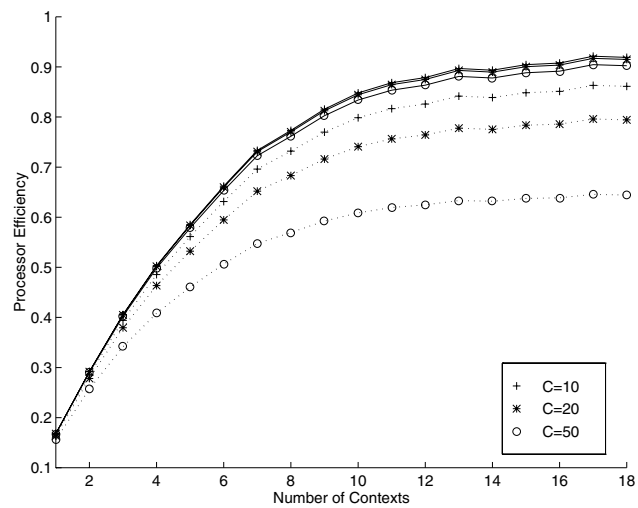

Figure 3.2a. Comparison between hybrid (solid lines) and software-controlled (dotted lines) execution models: R has a mean value of 100, L has a mean value of 500, and c=2.

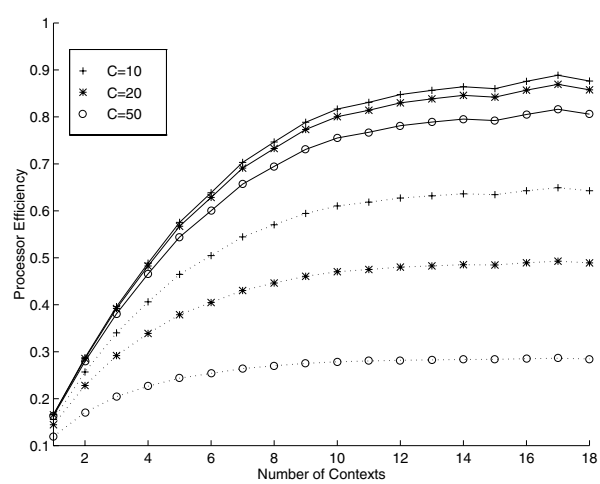

Figure 3.2b. Comparison between hybrid (solid lines) and software-controlled (dotted lines) execution models: R has a mean value of 20, L has a mean value of 100, and c=1.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

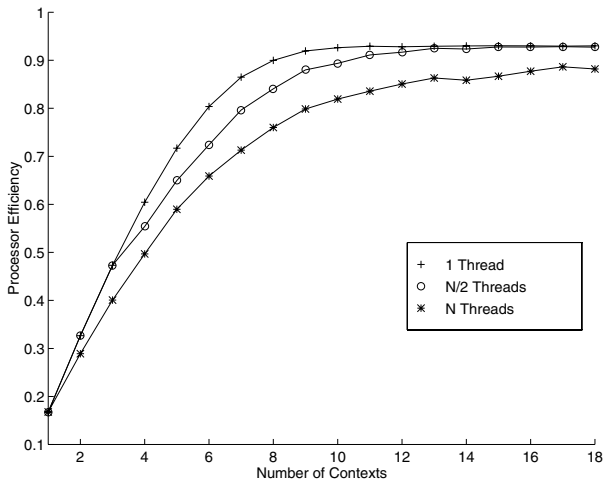

Figure 3.3a. Effects of scheduling policies when C=10, R=20, L=100, and c=1.

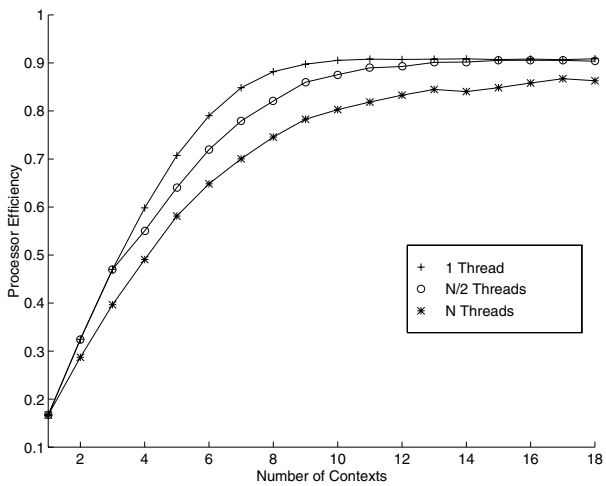

Figure 3.3b. Effects of scheduling policies when C=20, R=20, L=100, and c=1.

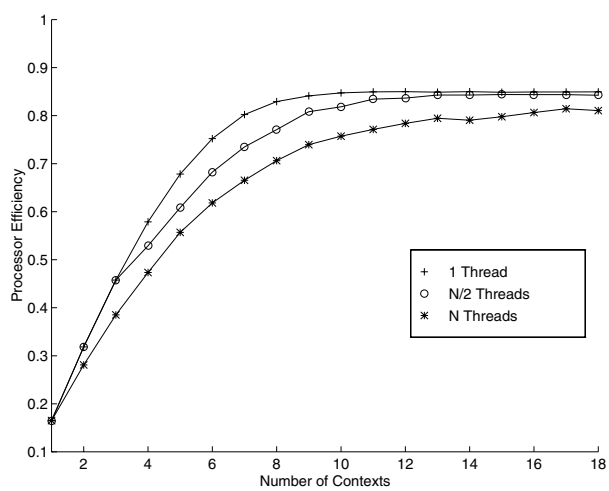

Figure 3.3c. Effects of scheduling policies when C=50, R=20, L=100, and c=1.

To overcome this deficiency, different scheduling policies were tried. They are (1) a new thread is scheduled immediately after a thread completes its execution and (2) schedule $N/2$ new threads when $N/2$ threads complete their execution. Results of these two scheduling policies were then compared against scheduling N new threads when N threads finish their execution. These are shown in Figures 3.3a-3.3c for various values of C. In these graphs, R was modeled by geometric distribution with a mean value of 20 cycles, L by a negative exponential distribution with mean value of 100 cycles, and c=1 cycle. These results show that for all three values of C it is always better to scheduled a new thread immediately after a thread completes its execution. Thus, scheduling one at a time will minimize idling due to lack of threads from which to context switch.

4. Conclusion and Future Work

Our preliminary performance study indicates that the proposed hybrid multithreaded model results in improved processor utilization over software-controlled multithreading. Higher processor utilization is achieved by having the Software Scheduler set schedule threads onto the Hardware Scheduler. The effects of various set scheduling techniques on the overall performance of the hybrid multithreaded system were studied. Set scheduling technique basically acts as an interface between existing software-controlled multithreaded system and the hardware support for multithreading. We found that set scheduling technique together with hardware support for multithreading has considerable performance advantage over traditional software-controlled multithreaded systems.

Although our performance results are encouraging they are based on a simple execution model and therefore quite preliminary. The future plan is to develop a detailed simulator for the hybrid multithreaded model. We are currently working on such a simulator that integrates the user-library Pthreads package developed by Chris Provenzano at MIT[2] with MIPS-based generic superscalar simulator, called SimpleScalar, developed at University of Wisconsin [5].

Using the hybrid multithreaded processor simulator, we plan to pursue a number of design issues. First, it is not clear at present what kind of hardware context representation is the most appropriate for our hybrid multithreaded processor. Multiple hardware contexts can be implemented either by duplicating register sets or using register relocation. Register relocation is more flexible but requires modification to the compiler. As a first-cut design, the plan is to use multiple-registers sets and use thread tags to map onto register sets.

[2] For more information on Pthreads package see http://www.mit.edu:8001/people/proven/pthreads.html.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

Another issue is the design of the instruction window. Currently, SimpleScalar implements centralized instruction window, where data hazards are resolved and ready instructions are issued to functional units. Once an instruction from a thread is issued to a functional unit, any subsequent blocking of that thread will not affect the execution of that instruction. However, this is not the case for instructions from the blocked thread waiting in the instruction window to be issued. These unissued instructions will continue to occupy valuable resources and impede the execution of other ready threads. There are two ways to resolve this problem. One method is to implement multiple instruction windows and multiplex the thread issuing among them. The other method is to simply buffer the blocked thread. Thus, there will be one instruction window and *N*-1 thread buffers. The latter method would be much cheaper but will result in higher hardware context switching cost since threads have to be move back and forth between the instruction window and thread buffers.

Another possibility we plan to explore is simultaneous multithreading (SMT) [15]. Simultaneous multithreading is a technique where multiple independent threads issue instructions to a wide-issue superscalar processor's functional units in a single cycle. The advantage of SMT is that both instruction-level parallelism and thread-level parallelism can be explored to achieve high performance. Implementing SMT will require relatively minor changes to the hybrid multithreaded processor—the instruction fetch mechanism can be implemented as multiple instruction window. However, since SMT proposed in [15] uses independent threads from different programs and SMT in hybrid multithreaded processor occurs among threads from the same program, we plan to study what effect interaction among the threads within a program will have on the design of the microarchitecture.

5. Bibliography

[1] Agarwal, A. *et al.,* "April: A Processor Architecture for Multiprocessing," *Proc. 17th Annual Int'l. Symposium on Computer Architecture*, May 1990, pp. 104-114.

[2] Alverson, R. *et al.*, "The Tera Computer System," *International Conference on Supercomputing*, Sept. 1990, pp. 1-6.

[3] Ang, B. S. *et al.*, "Star-T the Next Generation: Integrating Global Caches and Dataflow Architectures," Technical Report CSG Memo 354, LCS MIT, Feb. 25, 1994.

[4] Blumofe, R. D. *et al.*, "Cilk: An Efficient Multithreaded Runtime System," *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[5] Burger, D. *et al.*, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," UW Computer Sciences Technical Report #1308, July, 1996.

[6] Culler, D. E. *et al.*, "TAM-A Compiler Controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing,* Vol. 18, No. 3, July 1993, pp. 347-370.

[7] Chiou D. *et al.*, "*T-NG: Delivering Seemless Parallel Computing," *Proceedings of Euro-Par 95*, 1995.

[8] IEEE, Threads Extension for Portable Operating Systems (Draft 6). Feb. 1992. P1003.4a/D6.

[9] Mueller, F., "A Library Implementation of POSIX Threads under UNIX," *Proc. 1993 USENIX Winter Conference*, San Diego, CA, pp. 29-41.

[10] Nikhil, R. S., "Cid: A Parallel, 'Shared-memory' C for Distributed-memory Machines," *Proc. 7th Annual Wkshp. on Languages and Compilers for Parallel Computing*, Ithaca NY August 1994, Springer Verlag LNCS.

[11] Papadopoulos, G. M. *et al.*, "*T: Integrated Building Blocks for Parallel Computing," *Supercomputing93*, Portland, Oregon, Nov. 19, 1993.

[12] Saavedra, R. H. *et al.*,"Analysis of Multithreaded Architectures for Parallel Computing," *2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* July 1990, pp. 169-178,

[13] Schauser, K. E. *et al.*, "Compiler-Controlled Multithreading for Lenient Parallel Languages," *5th ACM Conference on Functional Programming Languages and Computer Architecture,* Aug. 1991, pp. 50-72.

[14] Thekkath, R. and Eggers, S. J., "The Effectiveness of Multiple Hardware Contexts," *6th Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 328-337.

[15] Tullsen, T. M. *et al.*, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proc. 22th Annual Int'l. Symposium on Computer Architecture*, Jun. 1995.

[16] Waldspuger, C. A. and Weihl, W. E., "Register Relocation: A Flexible Contexts for Multithreading", *Proc. 20th Annual Int'l Symposium on Computer Architecture*, 1989, pp. 273-280.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE