On the effect of Multithreading in the Performance of Superscalar Processors

Ortiz-Arroyo, Daniel

Publication date:
1997

Document Version
Early version, also known as pre-print

Study on the effect of Multithreading in the performance of Superscalar Processors

Daniel Ortiz-Arroyo
ECE Department
Oregon State University
Corvallis OR, 97331-3211, USA
dortiz@ece.orst.edu

Jorge Martinez-Carballido
INAOE
AP 51, Puebla, Pue.
72000 MEXICO
jmc@gisc1.inaoep.mx

Abstract

Modern programming languages and operating systems provide software mechanisms to support the execution of multithreaded applications. These multithreaded applications are executed, commonly, on high speed superscalar processors that in general do not provide special support to manage multiple contexts. In this paper the effect that multithreading has on the performance of modern superscalar processors is studied. A simulation environment, designed specifically, to trace the execution of multithreaded programs is described. Simulation results that show the dynamic behavior of software multithreading and its impact on the performance of a superscalar processor are presented.

1 Introduction

Multithreading is being used extensively as a software technique to improve the performance and response time of client-server applications. Additionally, modern programming languages like Java [Flanagan, 1997] have included support for multithreading as part of the language. In applications such as user interfaces, multithreading is being used to improve user response time. This is done by context switching when long I/O operations are issued.

However, despite all these advantageous aspects of software multithreading, it is important to note, that real speedup improvement will be obtained for a multithreaded application by executing multiple threads at the same time, in parallel, on multiple processors.

Support for software multithreading is provided at several levels: the *user* or the *system-level*. In the user-level approach to multithreading, a thread is a light weight process whose dynamic behavior is controlled by a specialized run-time system. At the system-level, the OS kernel provides support to assign multiple user-level threads to one or more system-level kernel threads. In general, context switching is less expensive at user-level, compared to context switching at system-level.

The standard POSIX-1c also known as *Pthreads* specifies a set of function calls to create, schedule and synchronize multiple threads at the user or system level. An example of a POSIX compliant multithreading library is the MIT version of Pthreads (*MIT-Pthreads*) [Provenzano, 1996]. This version of Pthreads is a user level library that provides the basic functionality contained in the standard through a runtime system that manage threads in a transparent way.

Another important aspect of multithreading is its support in hardware. Recent research on support for multithreading in hardware, has been focused in studying new architectural strategies [Ortiz *et al.*, 1997] targeted to improve the performance of shared memory computers. In such specialized multithreaded processors, a context switch operation is triggered by long latency operations such as remote memory references and synchronization. Context switching in this case, is used to hide latency at the cost of a small overhead, the context switching time. Shared memory computers are being designed using, general purpose, high speed, superscalar processors. Such processors, are designed to being able to execute multiple instructions per clock cycle. To achieve this goal, the processor relies on effective mechanisms to detect *instruction level parallelism* (ILP) at *run-time*. Once parallel instructions are detected, their execution is overlapped on highly parallel cores containing specialized, functional units.

The effect of multithreading on data cache has been modeled analytically in [Agarwal, 1992], but few studies are known about how instruction cache and fetch rate are affected by multithreading [Tullsen *et al.*, 1996].

Another important aspect for both, single and multithreaded architectures is the instruction fetch rate. Since more aggressive forms of ILP are foreseen to be used in modern superscalar architectures the fetch rate can become soon a performance bottleneck [Conte *et al.*, 1995]. Herein, its importance for single and multithreaded superscalar processor performance.

The research presented in this paper is part of the

research currently being developed at the Electrical and Computer Engineering Department of Oregon State University to study different aspects of multithreading. One goal of this research, is to propose novel architectural mechanisms in superscalar cores, targeted to provide, efficient support for multithreading [Ortiz *et al.*, 1997]. In this paper a preliminary study on the effect of software multithreading in the performance of superscalar processors is presented. This effect is studied by tracing multithreaded applications in a specially designed simulation environment built around the SimpleScalar simulator (*SS*) [Burger, 1996]. Emphasis is placed on studying the effect of multiple thread execution on branch prediction mechanisms that rely on the past history of branches to generate future predictions. Other important issues such as, the instruction cache locality, instruction fetch rate and total execution time of multithreaded applications are also studied in this paper. Section 2 discusses briefly the MIT-Pthreads library used in this research to create the multithreaded programs traced. Next, in section 3 the simulation environment is described. Section 4 discusses, briefly, the results obtained during simulation. Finally, in section 5 the future research on this area and the conclusions are presented.

2 The MIT-Pthreads library

MIT-Pthreads [Provenzano, 1996], is a user-level library of function calls that supports the execution of multiple threads. This library implements a subset of the POSIX standard on support for multithreading. Current MIT-Pthreads version 1.60β6, provides the basic functionality of Pthreads. That is to say, the synchronization primitives, thread-specific data, and thread attributes routines. To create a multithreaded application the user should first allocate space for threads and assign a function to each thread by calling the *pthread_create* function. Next, the programmer needs to start the threads running by calling the *pthread_join* function. Once called, Pthreads takes over the running of the threads until all are finished and returns control back to the main program. Pthread runs one thread at a time in a round-robin fashion and switches between threads in the following ways:

- When the current thread ends its execution, it calls the function *pthread_exit*. This function deletes the thread from operation and turns control to the Pthreads scheduler whereby either a different thread is started or control is returned to the main program if there exists no more threads to run.
- As each thread is executed, Pthreads starts an OS timer. If the timer activates before the thread finish its execution, an interrupt is generated. At this time the state of the thread is saved, the Pthreads scheduler called, and a different thread is executed (providing, of course, that there are more threads to run).
- Since Pthreads was created to hide I/O latency, it also switches threads when detects that a thread is making a long I/O request. The thread switching process works the same as a timer switch described above.

The implementation of MIT-Pthreads relies heavily on OS support. Therefore, in order to run on different operating systems MIT-Pthreads makes use of machine dependency files to configure and generate the library.

Using this library, multithreaded applications are generally designed following different models of execution. One of such models, is the so called peer-to-peer or Master-Slave model [Lewis and J.Berg, 1996]. Figure 1 shows the layout of the Master-Slave model. In this model, a master thread (the main procedure) is in charge of tasks such as data initialization, thread creation (of slave threads), and synchronization. On the other hand, the slave threads are responsible of performing the required computations and synchronizing with the master thread in a coordinated way. Thread access to shared data is controlled by mutual exclusion mechanisms provided by Pthreads. The simulation results presented in section 4, were obtained by writing and executing and tracing two multithreaded programs designed around the Master-Slave model.

MIT-Pthreads was not designed to run in a simulation environment. Neither, SS was designed with capabilities to run multithreaded applications. In general what SS required in order to run Pthreads applications, was the following modules:

- A OS stack to save and restore states of the threads.
- A OS timer to generate an interrupt to switch between threads.
- Precise timer interrupts to allow the correct interfacing between the hardware, OS, and Pthread.

By writing all these modules and including them in SS the integration of MIT-Pthreads with SS was possible. Both, MIT-Pthreads and SS were used to create the simulation environment described in next section.

3 Simulation Environment

The SimpleScalar simulator [Burger, 1996] is the basic module of the simulation environment[1] used in this research. This simulation environment provides capabilities to trace the execution of instruction in multithreaded applications created using the MIT-Pthreads user-level library. SS is a small collection of execution based simulators where each one models a different type or level of

[1] The simulation environment was designed by the MVP group [Lee *et al.*, 1997] at ECE-OSU

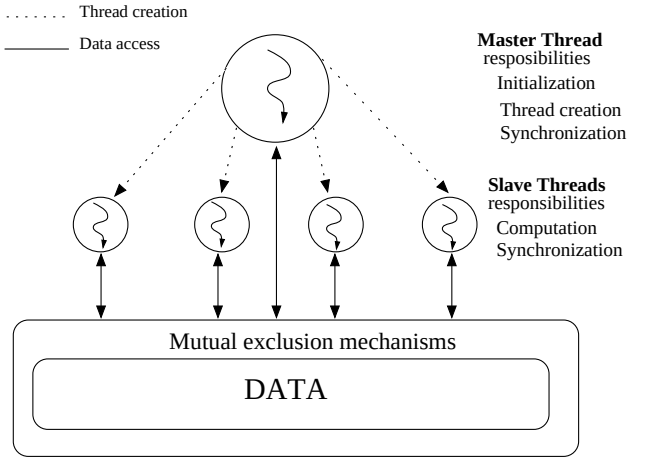

Figure 1: One basic programming model for multithreading

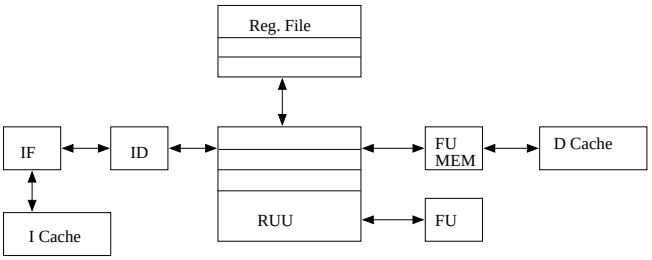

Figure 2: The SimpleScalar Simulator Architecture

detail within a processor. One of such simulators called sim-outorder, was used to generate results presented in section 4. Sim-outorder simulates an *out-of-order* processor based upon Gurindar Sohi's RUU design [Smith and Soh, 1995]. This sim-outorder processor model is shown in Figure 2.

However, SS is in fact an *in-order* execution based simulator followed by a trace-based out-of-order timing simulator. In other words, the instructions are decoded and instantaneously executed (the register files and main memory are modified). After that, the instructions are placed into the timing simulator where the instructions are moved into execution units and hazards are resolved similarly to execution in a super-scalar processor. It is important to note that no data is handled by the timing simulator, all data and *actual* execution is taken care of by the earlier execution simulator. Another way to look at this is that sim-outorder models an on the fly trace generator that runs the program and passes the instructions (as each is decoded and executed) to a trace-driven timing simulator. In order to correctly model wrong path execution, the in-order execution portion produces wrong path instructions called *spec_mode* instructions for the timing simulator that do not affect the state of the machine (i.e. except for cache valid/dirty bits, when the execution simulator enters *spec_mode* (*speculative mode*), the RF and memory are not modified). This *spec_mode* lasts until the timing simulator indicates that a branch instruction should have been executed by now, turns *off* *spec_mode*, and forces the execution simulator to restart execution down the correct branch. Since the Register File and memory have not been modified since the branch instruction (when *spec_mode* was initiated), the state of the simulated processor was preserved. This system allows sim-outorder to execute along speculative branch paths and have it not affect the correctness of the in-order simulation. Additionally to those issues, SS allows to change some architectural parameters by specifying them in the command line. Such parameters are for example: instruction buffer size and width, decode buffer size and width, instruction and data cache size and organization, type of branch prediction mechanisms etc.

MIT-Pthreads and the SS simulator were combined to trace the dynamic behavior of multithreaded programs. In order to do it, a number of features were added to SS like, Timers, support for Precise interrupts and a OS stack as was mentioned previously in section 2.

To implement timers in SS, the module syscall.c was modified to detect a *SETITIMER* syscall. Upon detection of the syscall the timer value is removed and stored in a global timer variable. To decrement the timer, code was modified in the *sim_main* function to decrement the timer every clock cycle.

Since Timer interrupts are those that are asynchronous to the currently executing instruction stream, it was easy to modify SS to handle them. The timer interrupts are checked at the beginning of *ruu_fetch* in module sim-outorder.c. This check is just a comparison with the global timer variable. It is important to note that interrupts cannot occur while the processor is in speculative mode because the entire interrupt handler would be run in speculative mode. To start executing the interrupt handler, *ruu_fetch* calls the SS interrupt handler (*interrupt_fetch*) which switches the current PC to that of the interrupt vector. This PC swapping method is acceptable since the RUU does not need to be drained before servicing the interrupt. This is mostly because the processor does not have a supervisor mode with multiple levels of memory protection.

To implement a OS stack into SS, a separate file, signal.c, was created. Signal.c creates and then maintains the OS stack. When *interrupt_fetch* is called, (to process a detected interrupt), it interfaces with the OS stack code to save the current state of the machine, increments the *stack pointer* and returns the interrupt handler PC. Upon a return from an interrupt, the code drops back into interrupt fetch, which decrements the stack pointer and returns the state of the machine. Last, the interrupt fetch routine returns the saved PC back into the normal SS program.

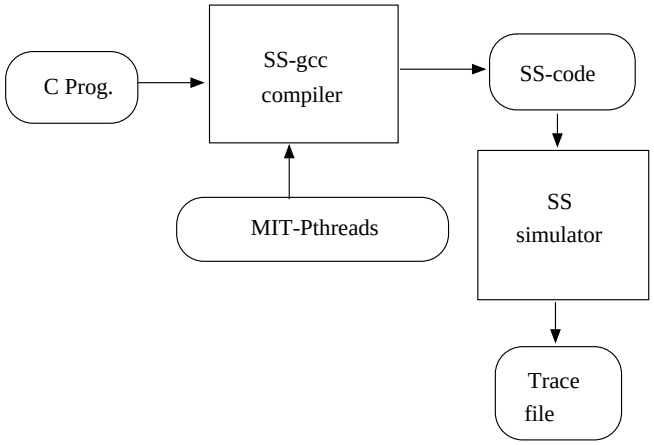

Figure 3: Simulation procedure

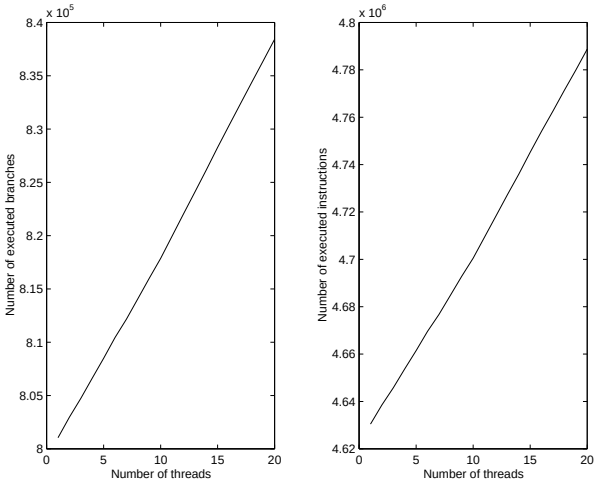

Figure 4: Trace of instructions executed in a multithreaded Matrix Multiplication

4 Simulation Results

Figure 3 shows the procedure to create a multithreaded application and execute it in SS. Two multithreaded benchmark programs were traced using the simulation environment: a matrix multiplication and a JPEG image decompressor. Both programs were executed in SS varying the number of threads from one to twenty. The number of instructions traced was in the range of 4,000,000 to 4,800,000 for both cases. The values taken by default in SS were used in the simulations. Some of those SS default parameters are shown in the next table.

IFB size	ID width	Type of BP	BTB size
4	4	Bimodal	4096

As can be seen in previous table the instruction fetch buffer size (IFB) and the instruction decode width have a size of four. This means that up to four instructions can be fetched, decoded and issued in one clock cycle. The type of branch prediction (BP) mechanism was bimodal, that is two bits are used to keep track of previous branch results. Finally a Branch Target Buffer (BTB) with size 4096 was used to store the addresses of next instructions to be fetched every cycle. These simulation parameters are commonly used in modern superscalar processors.

In the multithreaded matrix multiplication case, two matrices size 20×20 of floating point numbers were multiplied. The policy used to carry out such multiplication, was assigning to each thread, the task of calculating the elements of one block of rows in the resultant matrix. The size of the input matrices was divided between the number of threads and each thread assigned a block of rows in the resultant matrix. If the number of threads is less than the matrix size, then each thread will be in charge o calculating the values corresponding to several rows in the resultant matrix. In the case of the JPEG decompressor several images size 240×60 pixels in format JPEG were decompressed by the pool of threads. Each thread was in charge of decompressing one image at time. Notice that in the matrix multiplication case, the workload is fixed and the number of threads is varied. However, in the JPEG decompressor case, the workload varies according to the number of threads.

Figure 4, shows the effect of thread number on total number of executed instructions and branches in the multithreaded matrix multiplication. As is shown in this figure, the total number of instructions and branches increases linearly with the number of threads. This is due to the fact, that the number of context switch operations performed by the run-time system of MIT-Pthreads increases with an increase in the number of threads. This effect is caused mainly by the timer interrupt used by the library. With more threads available in the thread pool, the run-time system will allocate to each one an execution time slice. This is done by context switching in round-robin.

Figure 5 shows that the same behavior is obtained for the multithreaded JPEG image decompressor.

Branches limit the number of instructions that can be fetched from instruction cache without penalty. The average basic block (BB) size, that is the number of instructions between branches, for multithreaded matrix multiplication, is obtained using the equation:

$$BB\ size = \frac{Total\ Number\ of\ Instructions}{Total\ Number\ of\ Branches}$$

Using previous equation the following results were obtained for a variable number of threads.

1 Thread	5 Threads	10 Threads	20 Threads
5.87	5.87	5.85	5.79

The average basic block size for multithreaded JPEG decompressor is shown in the following table varying the number of threads.

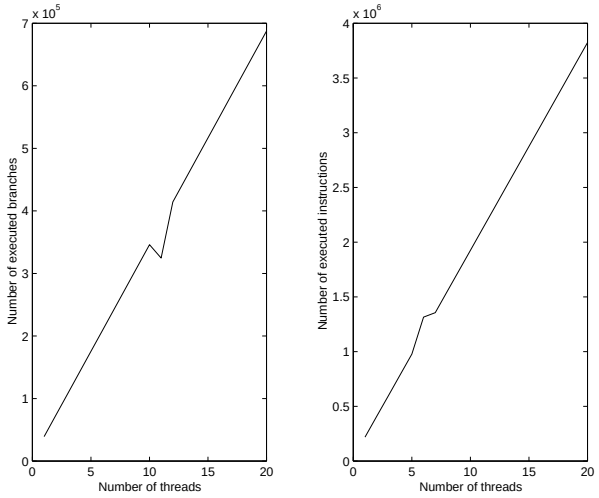

Figure 5: Trace of instructions executed in a multithreaded JPEG decompressor

1 Thread	5 Threads	10 Threads	20 Threads
5.60	5.56	5.56	5.56

Previous tables indicate that in both cases, an increase in the number of threads, produces a small decrement on the average size of the basic blocks available, on the average. Also, it is easy to see that the size of instruction fetch buffer in SS, should be increased up to six locations to allow fetching and storing a complete basic block in one clock cycle. This small change in SS along with a corresponding increment in instruction decode size and width will increase the instruction fetch rate.

Figure 6 shows the effect of the number of threads on instruction cache miss and branch prediction rates for multithreaded matrix multiplication. Cache miss rate is defined as:

$$Miss\ rate = \frac{Total\ Number\ of\ Misses}{Total\ Number\ of\ Cache\ References}$$

and branch prediction rate as:

$$BP\ rate = \frac{Total\ Number\ of\ Hits\ in\ BTB}{Total\ Number\ of\ BTB\ References}$$

Figure 6 for multithreaded matrix multiplication, shows that miss rate decreases with an increase in thread number in stepwise fashion. This behavior on miss rate is due to the fact that by decreasing the run-time length of a thread, (assigning it less workload) will allow to place more threads in the instruction cache, increasing therefore, thread cache locality. Since the pattern of branches is very similar for each thread the branch prediction rate increases accordingly. Looking at the results for multithreaded JPEG decompressor in figure 7 we can notice that the behavior is very different from the matrix multiplication case. In this case, miss rate increases with the number of threads because the workload is constant

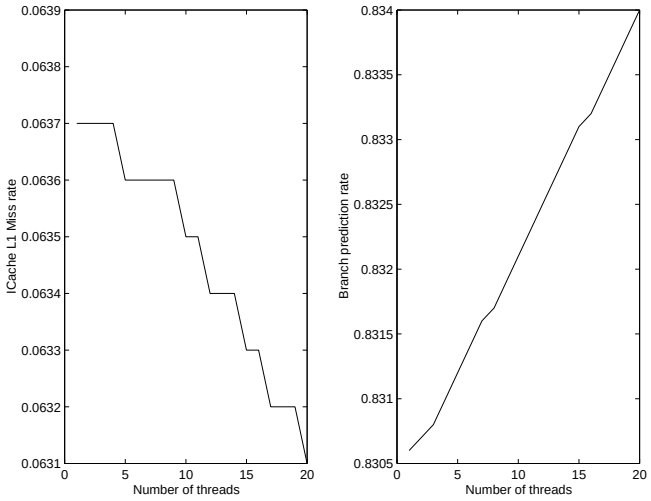

Figure 6: Trace of I-Cache miss and branch prediction rates for multithreaded Matrix Multiplication

and therefore, the run-length of a thread is also kept constant. Branch prediction rate improves with an increase in the number of threads because the same type of mathematical calculations are performed by all threads. Continuous repetition of same calculations increases the probability that the branch target address will be found in the BTB.

5 Summary and Conclusions

The effect of multithreaded applications on the performance of a superscalar processor was studied in this paper. For such purpose a simulation environment was built by the *Multithreaded Virtual Processor* (MVP) [Lee *et al.*, 1997] group at the ECE Department in Oregon State University.

The simulation environment was described in detail and the obtained experimental results were briefly explained.

Results showed that the number of threads impacts negatively the total execution time of multithreaded applications, increasing linearly the number of instructions to be executed. At the same time, a large number of threads tends to decrease in a small portion the size of the basic block available to the fetch unit. In this case, SS will benefit of including branch prediction mechanisms such as those proposed in [Chang *et al.*, 1994], to improve instruction branch prediction rate. On the other hand, results showed that a small number of threads (4-6) help to augment instruction cache locality and therefore instruction fetch rate. Moreover, the performance of the branch prediction mechanism is improved too. This is because each thread reproduces closely the same branch pattern, by performing similar calculations.

Additionally, from a simple analysis on the behavior of the multithreaded applications presented, it was con-

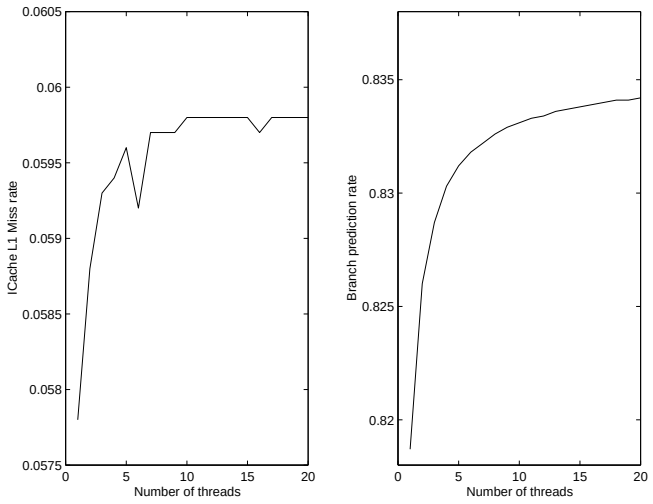

Figure 7: Trace of I-Cache miss and branch prediction rates for multithreaded JPEG decompressor

cluded, that the performance of SS will be improved if the instruction fetch buffer size is increased up to six locations. However, this small change will affect the design of the instruction cache.

In modern pipelined superscalar processors the fetch rate is becoming increasingly important. Being the fetch unit, the first stage in the pipeline, any bottleneck in this unit will drastically reduce the performance of the whole processor. A study on how instruction fetch bandwidth mechanisms such as those described in [Wallace and Bagherzadeh, 1996], [Rotenberg *et al.*, 1996] improve the performance of the fetch unit in the presence of multiple threads is considered in near future.

References

[Agarwal, 1992] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992. Also as Tech report MIT/LCS/TR-501, Massachusets Institute of Technology, Laboratory for Computer Science, April 1991.

[Burger, 1996] Doug Burger. Simple scalar tools. http://www.cs.wisc.edu/~mscalar/simplescalar.html, 1996.

[Chang *et al.*, 1994] P.-Y Chang, E. Hao, and Y. N. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th. ACM/IEEE International Symposium on Microarchitecture*, November 1994.

[Conte *et al.*, 1995] T. Conte, K. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. June 1995.

[Flanagan, 1997] David Flanagan. *Java in a Nutshell.* O'Reilly, 1997. 2nd edition.

[Lee *et al.*, 1997] B. Lee, H. Kawak, R. Carlson, M. Miller, and D. Ortiz. Mvpsim. http://www.ece.orst.edu/~benl/docs/mvp.html, 1997.

[Lewis and J.Berg, 1996] Bil Lewis and Daniel J.Berg. *Threads Primer: A Guide to Multithreaded Programming.* SunSoft Press, 1996.

[Ortiz *et al.*, 1997] Daniel Ortiz, Ben Lee, S. H Yoon, and K. W. Lim. A preliminary performance study of architectural support for multithreading. volume Software Track, Hawaii, January 1997.

[Provenzano, 1996] Christopher Provenzano. Pthreads. http://www.mit.edu/people/proven/pthreads.html, 1996.

[Rotenberg *et al.*, 1996] E. Rotenberg, S. Benett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 23rd. Annual International Symposium on Microarchitecture*, 1996.

[Smith and Soh, 1995] J. E. Smith and G. S. Soh. The microarchitecture of superscalar processors. December 1995.

[Tullsen *et al.*, 1996] Dean M. Tullsen, Susan Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebeca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the23th Annual International Symposium on Computer Architecture*, May 1996.

[Wallace and Bagherzadeh, 1996] S. Wallace and N. Bagherzadeh. Instruction fetching mechanisms for superscalar microprocessors. In *Proceedings of EURO-PAR'96*, 1996.