ABSTRACT

DEVELOPMENT AND EVALUATION OF A SIMULTANEOUS MULTITHREADING PROCESSOR SIMULATOR

by
Carla Verena S. Nuñez

Modern processors are designed to achieve greater amounts of instruction level parallelism (ILP) and thread level parallelism (TLP). Simultaneous multithreading (SMT) is an architecture that exploits both ILP and TLP. It improves the utilization of the processor resources by allowing multiple independent threads to reside in the pipeline and dynamically scheduling the available resources among the threads.

The first part of this thesis presents the development of a simultaneous multithreading processor simulator. The SMT simulator is derived from SimpleScalar, a superscalar processor simulator widely used in the computer architecture research field. The basic pipeline is expanded to allow multiple threads to be fetched, dispatched, issued, executed, and committed simultaneously. Benchmarks that were executed on the SMT simulator verified its functionality. The simulator produced the correct outputs and the performance levels achieved were similar to those produced by the original authors of the SMT architecture.

The second part of this thesis explores the register file for SMT processors. The register file size grows with increased issue widths, instruction window sizes, and number of thread contexts; as the register file size increases, so does its access latency. Solutions to the register file problem have been proposed but most of these were designed for and evaluated on superscalar processors. The use-based register cache is one such design and its effectiveness on an SMT architecture is evaluated in this thesis.

The SMT simulator is a useful tool for evaluating components designed for superscalar processors on a simultaneous multithreading environment and for testing future designs of SMT architectural elements.

DEVELOPMENT AND EVALUATION OF A SIMULTANEOUS MULTITHREADING PROCESSOR SIMULATOR

by
Carla Verena S. Nuñez

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 2007

APPROVAL PAGE

DEVELOPMENT AND EVALUATION OF A SIMULTANEOUS MULTITHREADING PROCESSOR SIMULATOR

Carla Verena S. Nuñez

Dr. Jie Hu, Thesis Advisor Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Sotirios G. Ziavras, Thesis Co-Advisor Date
Professor of Electrical and Computer Engineering, NJIT

Dr. John D. Carpinelli, Committee Member Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Carla Verena S. Nuñez
Degree:	Master of Science
Date:	May 2007

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2007
- Bachelor of Science in Computer Engineering,
  Ateneo de Manila University, Quezon City, Philippines, 2003
- Bachelor of Science in Physics,
  Ateneo de Manila University, Quezon City, Philippines, 2002

Major: Computer Engineering

This work is dedicated to the two people
responsible for my presence in this world.

Pa and Ma,
thank you for all the sacrifices you have made
so that your children could pursue their dreams.

ACKNOWLEDGMENT

I owe the timely completion of this thesis to my advisor, Dr. Jie Hu, who has guided me through much of my work. When the long hours of programming and debugging take their toll and cause me to lose sight of where I'm headed, I have been able to count on his input to put me back on track.

I am grateful to Dr. Sotirios Ziavras, who has co-advised this thesis and has given invaluable suggestions that aided me in my research. Dr. Hu and Dr. Ziavras have generously provided me with all the resources I needed and I greatly appreciate the constant support they have extended to me.

My appreciation goes to Dr. John Carpinelli for having been part of my thesis defense committee and for reviewing my work. I am also extending my heartfelt thanks to Hongyan Yang, Shuai Wang, and Donovan Jones, who assisted me during the hectic preparation for my thesis defense.

I would also like to acknowledge the funding provided by the U.S. Department of Energy under grant DE-FG02-03CH11171.

Finally, I would like to thank my aunt, uncle and cousin. Life in a foreign country has been made easier because of their constant love and support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Modern Processor Trends

Superscalar architectures have the ability to concurrently evaluate multiple instructions in the same clock cycle, achieving performances much greater than pipelined architectures that can issue only one instruction per cycle. The key architectural features that enable superscalar processors to dynamically issue instructions out of order [1] are:

- a fetching mechanism that retrieves several instructions from the instruction cache for every cycle;
- branch prediction for fetching beyond conditional branch instructions, allowing a continuous stream of instructions to be fetched and processed;
- decoding logic that handles multiple instructions simultaneously;
- register renaming that removes false dependences, which normally prevent instructions from executing in parallel;
- parallel initiation of instructions ready for execution;
- several functional units for removing structural hazards;
- mechanisms for reordering instruction results so the process state is updated in correct order; and
- mechanisms for recovering from incorrect branch predictions.

Ideally, a superscalar processor with an issue width of k can achieve a maximum speedup of k over its scalar counterpart. The real performance gain, however, is limited by the amount of parallelism that can be extracted from the instruction stream. David Wall published a study [2] that explored the limits of achievable instruction level parallelism (ILP). Results from his investigation revealed that, even with an ambitious platform, ILP was rarely greater than eight instructions per cycle. The results for realistic models indicated that improving the key architectural structures (e.g., having better branch prediction logic, a larger register file, etc.) would yield higher performance but such investments would have diminishing returns. A different architectural paradigm was required to significantly ramp up the processor performance.

A single program or thread usually does not have enough ILP to fully utilize the issue bandwidth of the superscalar processor so issue slots are wasted for some cycles. This case of resource under-utilization is referred to as horizontal waste. There are also instances when no instructions are issued in a cycle, when instructions in the issue queue are dependent on a long latency instruction. The case where the full execution bandwidth is left idle is called vertical waste. To eliminate horizontal and vertical wastes, designers began looking at running several threads on the processor.

The idea of multithreading has been around since the 1960s. Operating systems use time sharing to allow several applications to utilize the resources of a single CPU. This method increases the throughput since the processor is never left idle as long as there are threads waiting to be executed. The same idea was incorporated by computer architects into the design of the hardware and different types of multithreaded architectures arose:

- Chip multiprocessors [3] have two or more superscalar processor cores integrated in one chip; each processor core independently executes one thread.
- Fine-grained multithreading [4]-[7] allows two or more thread contexts to reside on the chip. The processor switches from one thread to another on a fixed, fine-grained schedule.
- Course-grained multithreading [8]-[10] provides multiple thread contexts on chip but context switching only occurs when the currently active thread stalls on a long-latency event.
- Simultaneous multithreading (SMT) [11], [12] provides multiple thread contexts on chip and issues multiple instructions from multiple threads in one cycle.

Fine-grained and course-grained multithreading both reduce vertical waste but, because only one thread issues per cycle, they are still likely to experience horizontal waste. Simultaneous multithreading has the ability to fill all issue slots by allowing different threads to compete for them and can thus be more effective in eliminating both vertical and horizontal wastes [11]. Exploiting thread level parallelism (TLP) has increased performance that used to be limited by instruction level parallelism.

Therefore, in addition to the trend of clocking at higher frequencies, modern processor design will continue in the direction of greater parallelism through increased ILP and TLP. We thus expect processors to have greater issue widths, larger instruction windows, and support for multiple thread contexts.

1.2 Simultaneous Multithreading

A 1995 paper by Tullsen, Eggers and Levy introduced the technique of simultaneous multithreading [11], which allowed several independent threads to reside in the processor pipeline and issue to multiple functional units in the same cycle. There is a better utilization of available resources through the dynamic scheduling of functional units among multiple threads.

In a follow up paper, the authors showed that the basic superscalar pipeline did not require a major overhaul to accommodate simultaneous multithreading [12]. The following basic changes are required to enable support for multiple threads on a superscalar pipeline: multiple program counters, mechanisms for choosing a thread to fetch from per cycle, separate stacks for each thread, per-thread instruction retirement, instruction queue flush and trap handling, a thread ID per branch target buffer entry, and a larger register file. Figure 1.1 shows the base SMT hardware architecture proposed by the authors.

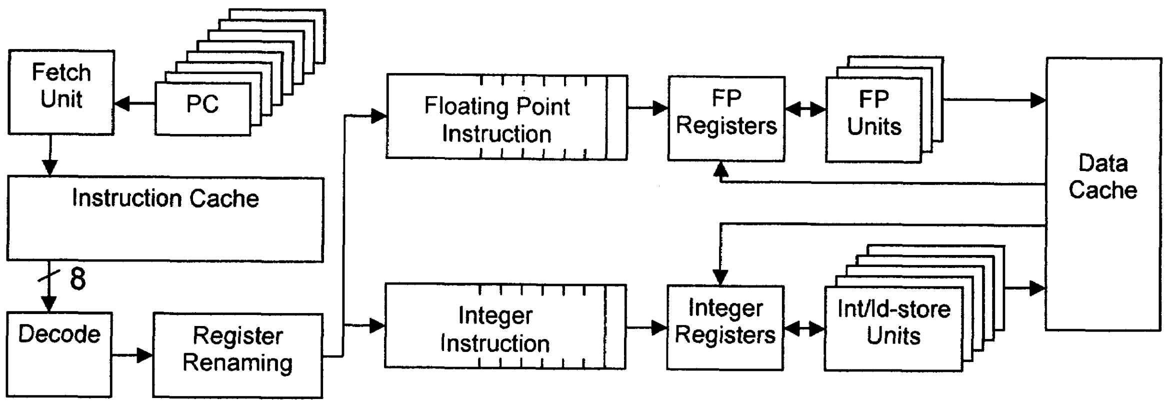


Figure 1.1 Base SMT architecture.
(Source: Tullsen, et al. [11])

1.3 The Register File: Future Bottleneck of SMT Processors

One major hardware modification required to support simultaneous multithreading is a larger physical register file. The register file holds the individual context of all threads as well as additional registers required for register renaming. This becomes a major design issue for reasons that will be discussed below.

The register file is the smallest and fastest component in the memory hierarchy and is used to supply operands to the processor's execution unit. The register file is usually implemented as a multi-ported SRAM (Figure 1.2); it is composed of address decoders, an array of bit cells, and sense amplifiers (Figure 1.3). During the read phase, the address decoders drive word lines that run across the bit cell array. When a word line is activated, it causes a row of bit cells connected to it to dump their data on the bit lines. Sense amplifiers connected to the bit lines detect bit-line changes and output the corresponding logic levels to the data bus. The access time of the register file depends on the time for address decoding, signal propagation delay through the word line wire, signal propagation delay through the bit line wire, and delay across the sense amplifiers.

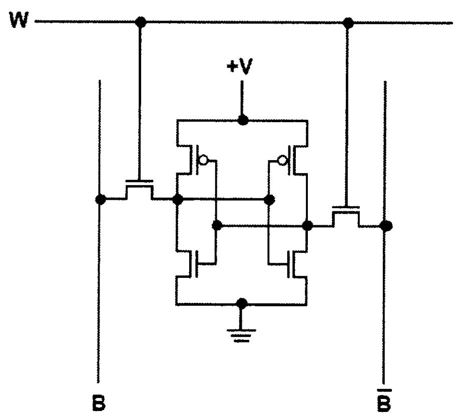


Figure 1.2 A single SRAM cell.

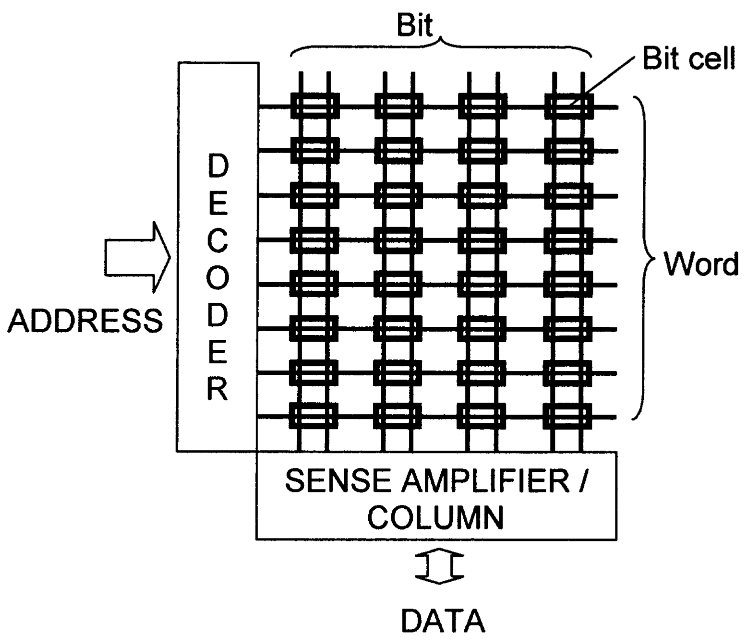


Figure 1.3 The SRAM organization.

When the issue width of the processor is increased, the number of ports that are connected to the register file also increases. There is usually one write port and two read ports for each execution unit. For every port that connects to the register file, one word line is required for a row of bit cells; for every write port, two bit lines are required. Increasing the number of ports to the register file therefore increases the size of the array exponentially (Figure 1.4). Palacharla, et al. indicated that the RAM structure's access time, due mostly to decoding and wire delays, increases linearly with the issue width [13].

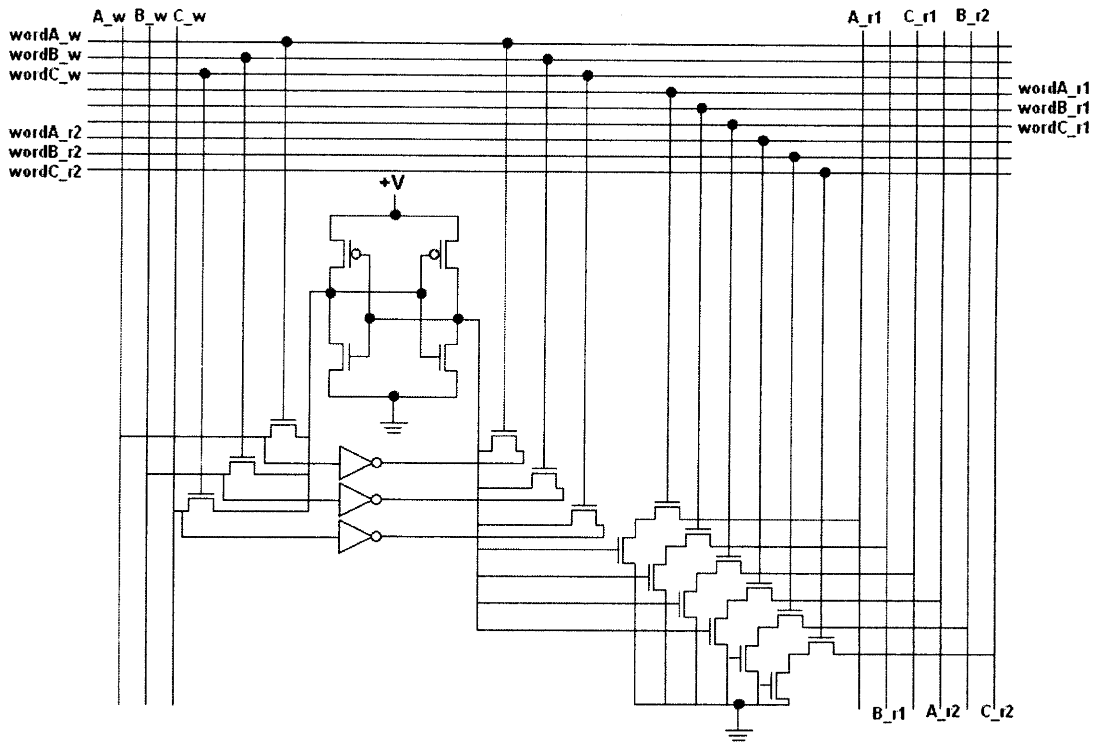


Figure 1.4 SRAM cell with six read ports and three write ports.

Another determinant of register file size is the instruction window size of the processor. Increasing the number of in-flight instructions will require a corresponding increase in the number of physical registers that are available for register renaming. This translates to an increase in the number of bit cells, an increase in the number of word lines, and an increase in the length of bit lines. The overall result is a larger register file with longer access latency. In some systems, the register file becomes so large that it exceeds the area of the cache. In the Alpha 21464 SMT processor, for example, the register file is five times the size of the L1 data cache [14].

The register access time is obviously exacerbated by modern processor trends of increased issue width (increases the number of read and write ports), instruction window size (increases the number of registers), and the number of threads (increases the number of registers). For example, in an SMT processor that can support four threads, with each thread described by 32 registers, a total of 128 registers are required simply to keep the thread contexts. Additional registers are also needed to be able to take advantage of register renaming. If register files become too big, there is a great possibility that in future processors, the register file's access latency becomes the major limiting factor of clock cycle time for the processor pipeline.

The register read and register write stages can be pipelined to allow higher clock frequencies to drive the processor but such a design choice has adverse effects on the processor performance. Adding more pipeline stages increases the branch resolution loop and the load resolution loop [15]; consequently, there is an increase in the branch misprediction penalty, which hurts the processor's performance. A two-stage write will also require a more complicated bypass logic. Therefore, it is more ideal to redesign the register file so that the access latency is reduced.

There have been different approaches to solving the register file problem. Some studies have focused on changing the register file organization so that even for a large number of registers, the access time remains fairly constant. Other studies have looked at ways of improving the utilization of a small register file.

1.3.1 Redesigned Register File Organization

Cruz, et al. introduced the concept of dividing the physical register file into several banks [16]. Banking enables register access time to be much smaller than that of a monolithic register file. The multi-banked register architecture can either be homogeneous or heterogeneous. In the former case, each bank has the same number of registers and the same number of ports so the access time to each bank is uniform. In the latter case, the number of registers and the number of ports can vary from bank to bank, giving rise to faster banks and slower banks. Multi-banked architectures can also be classified as either one-level or multi-level. Organizations that have only a single level have all banks providing operands to the functional units and a result is written to only one bank. Multi-level organizations have only upper level banks connecting to functional units and results are usually written to the lower level and optionally to the upper level.

Register caching is a form of the heterogeneous, multi-level, multi-banked architecture, which has a smaller, faster bank and a larger, slower bank. In the model proposed in [16] the smaller bank, which contains a subset of the values residing in the larger bank, provides source operands to the functional units. Results are always written to the larger bank and sometimes also to the smaller bank if the value is expected to be used soon. This model has an inclusive approach to data storage, similar to the cache model of memory hierarchies. A register caching model with an exclusive approach to data storage [17] was proposed by Balasubramonian, et al. Values have only one copy in the hierarchy, residing either in the upper or lower level.

Just like the memory hierarchy, register caching also requires management schemes to determine which values should be inserted into the register cache and which values in the upper level should be retired to the lower level. However, unlike a program's instruction and data streams, register values do not possess temporal or spatial locality properties. In [16], two types of caching policies for the multiple-banked register file were suggested: non-bypass caching, which wrote to the upper level only results that were not read from the bypass logic, and ready caching, which cached results that were source operands for instructions in the queue that had all operands ready. Other studies based the caching policy on the number of consumers. Balasubramonian, et al. tracked pending consumers for a value and transferred a register value from the upper level to the lower level once the pending consumer count reached zero [17]. Butts and Sohi [18] had the same idea when they proposed a register caching scheme that bases insertion and replacement policies on the number of consumers that a value has. Basically, values that have more consumers are maintained in the cache.

Although banked register files decrease access time, they normally also decrease the instructions per cycle (IPC) because of bank conflicts. Note that a register bank usually has a reduced set of register ports. If, for example, an N-banked register file has P ports per bank, then there can be as many as NxP values accessed per cycle as long as only P registers are accessed from any one bank [17]. Such register file architectures therefore require mechanisms to resolve bank conflicts. A select logic suggested in [17] takes into account the availability of the ports before granting the request. Tseng and Asanovic suggested speculatively issuing instructions and having a pipeline recovery scheme to repair the issue window in case of conflicts [19].

One proposed organization by Sangireddy, et al. splits the register file into two equal banks [20]. Each bank has sufficient read and write ports to support the instruction issue bandwidth. Since it is half the size of the monolithic register file, its access time is much shorter. Results are always written to the first bank but they are transferred to the second bank if a corresponding register is free. Because the whole issue bandwidth is supported, there is no resulting IPC degradation. This organization improves both performance and access latency.

1.3.2 Effective Utilization of Register Resources

Another camp of designers has looked at improving the utilization of the register file so that even with a reduced number of registers performance is unaffected. Improvements are introduced to various stages in the pipeline. Additional structures and logic circuitry are often required.

Designers have observed the following regarding the lifetime of a physical register:

- registers are allocated early but do not hold a value until the write-back stage;
- the register's active state is very small compared to its lifetime; and
- after the last consumption by a functional unit, there is a long latency before the register is freed up.

Virtual-physical registers [21] target the latency between register allocation and the write-back stage. As long as the value is not yet available, there is no need to tie up a physical register. So during the rename phase, a virtual register (which is just a tag) is assigned to the logical register. Only when the result is generated will a physical register be assigned. Other studies have focused on earlier deallocation of registers. The work by Ergin, et al. uses a register file with a checkpointing facility [22] to implement several techniques for early deallocation.

Another observation was that the value produced by an instruction is often the same as the value produced by some other recently executed instruction. The proposed architecture of physical register reuse [23], [24] keeps track of values that have been generated and are currently resident in the register file. When a new result matches a register value, the destination logical register is remapped to a physical register that already contains the value. A value cache is kept to maintain the physical register values.

The work by Lipasti, et al. [25] and other groups [26], [27] exploit narrow width values. The technique of physical register inlining [25] stores register values with few significant bits in the rename map and releases physical registers assigned to them.

1.3.3 The Register File in SMT Processors

Because of the large number of registers required to save thread contexts and support register renaming, some designers have proposed certain register file organizations for multithreaded architectures. One such architecture uses multiple physical banks of homogeneous structure that are dynamically allocated to threads [28]. When all threads are running, there is a minimum one-to-one correspondence between a thread and a bank. If fewer threads are active, then each thread may have more than one bank. The proposed architecture keeps an allocation decision table that quickly provides the bank assignments depending on which threads are currently active. Tseng and Asanovic [29] have also evaluated banked register files on a simultaneous multithreading architecture. They studied the performance of the register file with the banks shared by the different threads.

In addition to requiring a greater number of physical registers, simultaneous multithreading also places a stress on the register file that is unique from the stress it experiences for single-thread architectures. Proper resource allocation is critical in making simultaneous multithreading effective and, like the instruction queue and execution units, the register file is a resource for which threads must compete. If register renaming does not consider thread properties in allocating registers, then the overall performance of the processor can degrade. The technique of thread-sensitive register renaming [30] was proposed by Yang, et al. Their work is one of the few that have taken into account the inter-thread interference in simultaneous multithreaded processors that adversely affect register file utilization.

Designing a register file for SMT processors will have to be a blend of selecting the appropriate register file organization to support multiple thread contexts and implementing techniques that effectively utilize the register file in light of thread resource competition.

1.4 Objectives

This thesis has two main objectives:

The first objective is to develop a simultaneous multithreading processor simulator. Such a simulator would be a valuable tool for evaluating future designs targeted for SMT processors. The SMT simulator is based on an existing superscalar simulator, the SimpleScalar toolset [31], which models the functionality of an out-of-order superscalar processor. Since SimpleScalar has been widely used in evaluating computer architectural designs, developing the SMT simulator based on SimpleScalar will have a direct impact on the computer architecture research community in the current transition from superscalar to SMT architectures.

The second objective of this thesis is to use the SMT simulator developed in the first part to assess novel register file designs. Many papers have addressed the problem of the growing register file size but most of the solutions that have been proposed were designed for and evaluated on superscalar processors. The effectiveness of these designs in an SMT architecture remains unclear, since no study implementing them in an SMT processor has been conducted. This thesis exemplifies such a study by evaluating the use-based register cache proposed by Butts and Sohi [18].

CHAPTER 2

SMT SIMULATOR DEVELOPMENT

2.1 The SimpleScalar Simulation Tool

The SimpleScalar toolset [31] developed by Todd Austin and Doug Burger has been widely used in the computer architecture research field to simulate superscalar processor designs. The sim-outorder tool, in particular, models an out-of-order issue superscalar processor. It includes functional units, two-level cache, main memory, translation look-aside buffers, and virtual memory. The simulator has six stages as shown in Figure 2.1: a fetch stage, a dispatch stage, a scheduler stage, an execute stage, a writeback stage, and a commit stage.

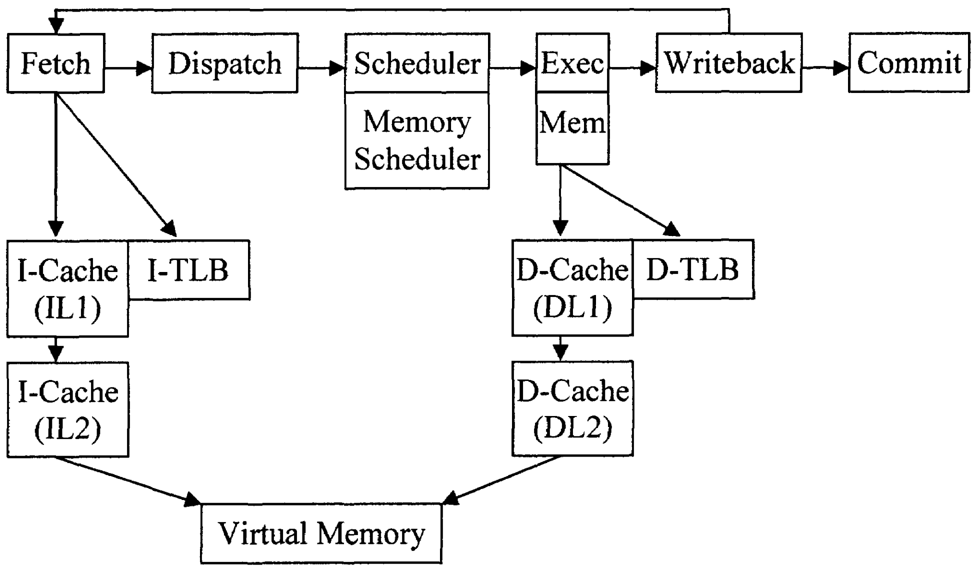


Figure 2.1 The sim-outorder superscalar processor model [31].

The fetch stage reads several consecutive instructions from the instruction cache and places them in an instruction fetch queue. The fetching of instructions stops when either the fetch bandwidth is reached or a conditional branch is encountered. Branch prediction allows the fetching of speculative instructions in the succeeding cycles.

The instructions in the fetch queue are processed in order during the dispatch stage and an entry is created for each instruction in the Register Update Unit (RUU). In the simulator, the reorder buffer and reservation stations are unified in the RUU, which is implemented as a circular queue of reservation stations. Each reservation station contains information about the instruction (e.g., address, decoded instruction opcode), the instruction's status (e.g., queued, issued, completed), and the input and output operands.

During dispatch, when a new RUU entry is created, a vector table is checked for each logical source register to determine the creator of the value. If the table reflects the value to reside in the logical register file, then the operand is marked as ready. Otherwise, the current instruction is added to the creator's dependency list. This list allows for faster instruction wakeup when the value becomes available after execution.

When all the operands are available, the RUU entry is marked as ready. The simulator issues ready instructions out-of-order, placing them in a ready queue from which they are sent to their respective functional units. If an instruction retrieved from the queue is unable to execute during that cycle (e.g., a functional unit is unavailable), then it is placed back into the ready queue and is processed in the next or later cycle.

The vector table is updated during the write-back stage but the logical register file does not contain the new values until after the producing instruction has been committed. The RUU entry assigned to an instruction during the dispatch stage is released once the instruction has committed. The commit stage graduates instructions in order.

2.2 Developing the SMT Simulator Sim-SMTP

2.2.1 Supporting Multiple Contexts

To support multiple thread contexts, several structures in the simulator have been replicated. Most of these structures have been redefined as arrays, with the thread ID used as the array index when accessing a particular context. Each thread has its own set of control registers and registers for integer and floating point data. The simulator also maintains separate vector tables to keep track of the instructions producing the latest result for a logical register and bit maps to track speculative execution.

2.2.2 Loading Multiple Binaries

The command to run a binary on the original Simplescalar simulator is of the form:

```
sim-outorder [-sim opt] program [-program opt]
```

For multiple threads, the option -threads has been added to the simulator's options database. The default value is 1 and executing multiple threads requires including this option in the command line followed by the number of threads (2 to 8). Following the simulator options are the program files and their arguments. Thread binaries and arguments are separated by the plus (+) sign:

```
sim-smtp [-sim opt] -threads N program_1 [-program_1 opt] + ... + program_N
[-program_N opt]
```

The revised simulator compares the number of threads specified by the -threads option with the number of binaries that follow. They must match in order for the simulation to proceed. The command line input is parsed to determine each thread's program name and options. Each thread will have its own stack segment with an initial set-up that is determined by the program's arguments.

Additional revisions were made to the simulator's program loader to accommodate multiple threads. Normally, the code segment is set to begin at 0x120000000 and data at 0x140000000 (see Figure 2.2). The stack segment starts at 0x11FFFFFFF and grows down to lower addresses. Addresses beyond the data segment up to 0x3FF7FFFFFFF are user-mappable.

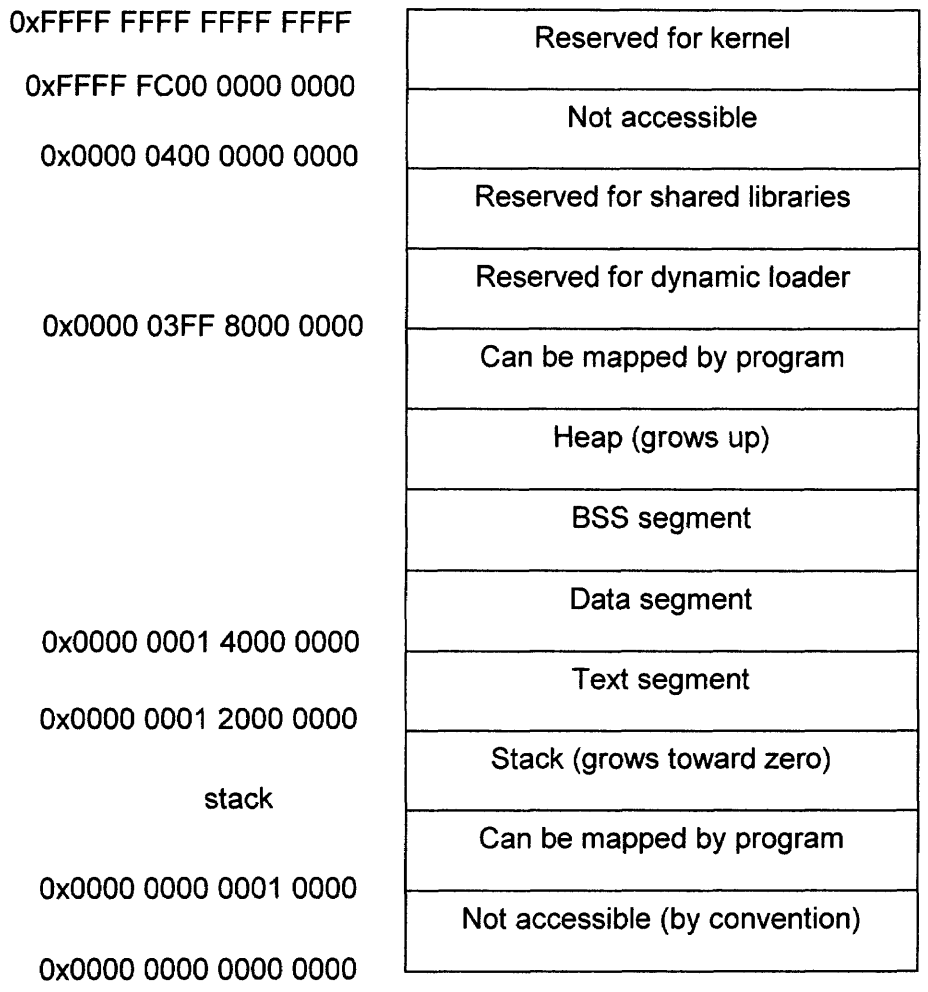


Figure 2.2 The Alpha memory space [32].

For eight threads to reside in the Alpha's memory space, the segments of each thread are assigned particular locations in memory. To simplify program loading, the segment sizes and addresses have been pre-assigned to eliminate size calculations at the start of the simulation.

The SPEC2000 benchmarks were used to determine the segments sizes that would be sufficient for the needs of typical applications. The code and data segment sizes can be extracted from the executable binary. The stack segment size, however, can only be estimated by running the application. The stack pointer register was monitored during execution to determine the lowest stack address and this address was used to calculate the stack's largest size during execution. The benchmarks were fast-forwarded for 200 million instructions and then simulated for 500 million instructions.

Table 2.1 Segment Locations

Thread	Text start	Data start	Stack start
1	0x120000000	0x140000000	0x11FFFFFFF
2	0x124000000	0x150000000	0x0FBFFFFFF
3	0x128000000	0x160000000	0x0D7FFFFFF
4	0x12C000000	0x170000000	0x0B3FFFFFF
5	0x130000000	0x180000000	0x08FFFFFFF
6	0x134000000	0x190000000	0x06BFFFFFF
7	0x138000000	0x1A0000000	0x047FFFFFF
8	0x13C000000	0x1B0000000	0x023FFFFFF
	64MB/thread	256MB/thread	512+MB/thread

Results revealed that the text segment is normally very small. For the SPEC2000 suite, code did not exceed 3MB while the data segment size was about a hundred times larger than the code at 256MB. The largest stack size during benchmark execution was only about 1MB. Table 2.1 shows the segment addresses and sizes assigned to the eight threads. The segments were intentionally made very large to ensure that the simulator had sufficient space when running unknown binaries. The first thread resides in the original segment locations while other threads have segments beginning at different offsets from the original starting address.

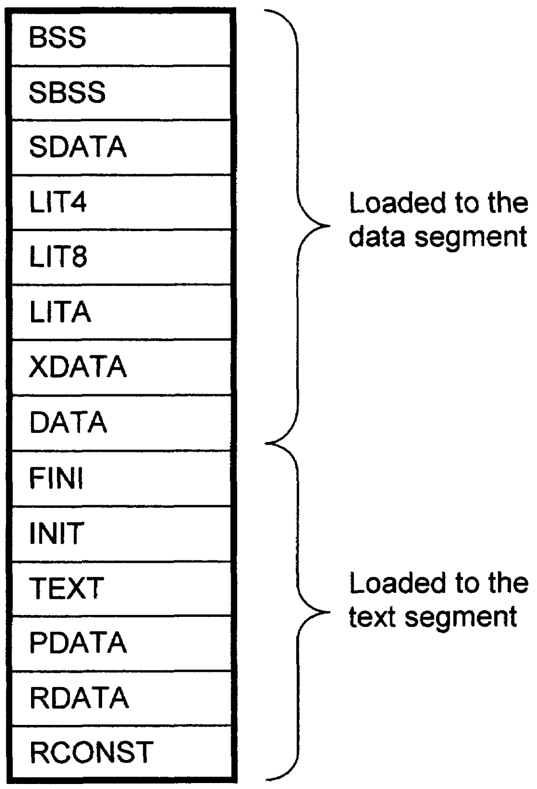


Figure 2.3 The object file sections.

The different sections of the binary file are loaded to either the text segment or the data segment (Figure 2.3). The data segment includes the sections BSS, SBSS, SDATA, LIT4, LIT8, LITA, XDATA, and DATA. The text segment includes the sections FINI, INIT, TEXT, PDATA, RDATA, and RCONST. The program loader parses through these sections and places them in the correct memory locations. The stack is also set-up during program loading using the arguments belonging to the thread as well as the environment parameters associated with the user's terminal shell (Figure 2.4).

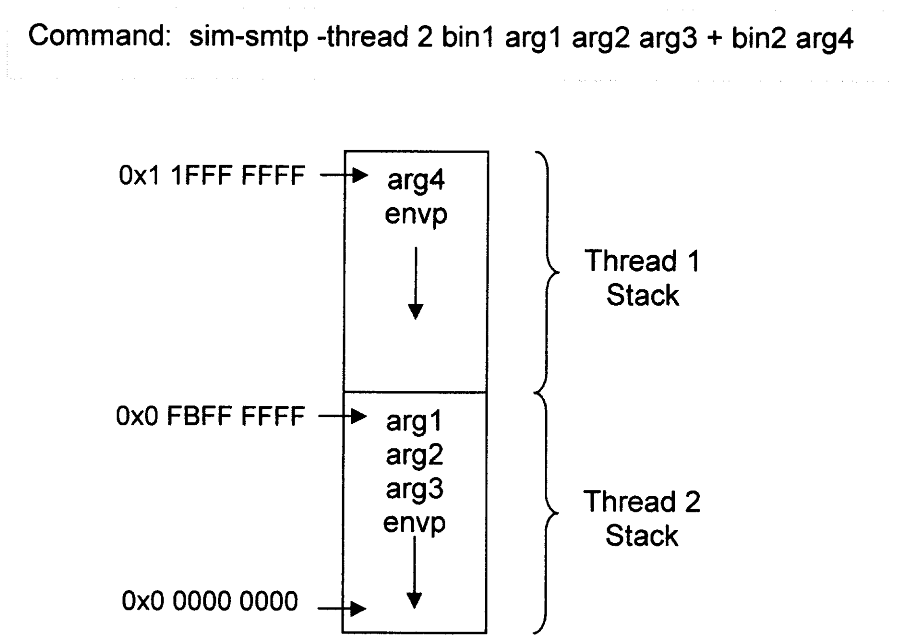


Figure 2.4 Initial set-up of the stack.

Initial simulations could not properly execute relocated code. Analysis of small Alpha binaries indicated that some programs used absolute addresses that were apparently stored in the text segment, the data segment, or both. These addresses target jumps to the original segment locations, causing invalid instructions to be executed. If these absolute addresses had been isolated to the text segment, then the problem would have been easily solved by filtering quad-word loads from this segment. However, it is near impossible to determine whether a quad-word load or store in the data segment involves data or an address to be used for jumps later on in the program. To confirm that absolute addresses are indeed embedded within the binary file, the file was scanned for quad words 0x12xxxxxxx and 0x14xxxxxxx. The offset of the relocated segment was added to each. After these changes were made, the binary file successfully finished execution.

However, pre-processing the binaries is not a viable option to running relocated code on the simulator. It is not guaranteed that all quad words replaced will actually be used as addresses. To circumvent this problem, the processor state reflected in the registers keep the original addresses of the code while an address translation stage has been added prior to the cache and memory units. It will therefore appear to the processor that all threads reside in the same segment, although they occupy different segment locations in memory. When a thread performs a read or write, its ID is used to determine the offset to be added to the effective address so that the address supplied to either the cache or main memory points to the relocated code or data.

To check the validity of the approach, the benchmarks were run from relocated segments on the superscalar processor. The simulation results were the same as the original ones, except for the page table statistics. Relocation can change the number of allocated pages to a program, causing a corresponding change in the number of page table misses. The impact of the page table misses on performance, however, is negligible.

2.2.3 Translating Addresses

The basic functions for reading and writing perform address translation prior to accessing the memory or the cache. The address is checked to determine the segment that is being accessed: stack segment addresses are less than 0x120000000; data segment addresses are equal to or greater than 0x140000000; and text segment addresses are in between. Once the segment is known, the corresponding segment offset for the thread making the access is added to the address (Figure 2.5). The result is the address in memory where the thread's code or data is stored. The cache module performs address translation prior to determining the cache index and tags to use.

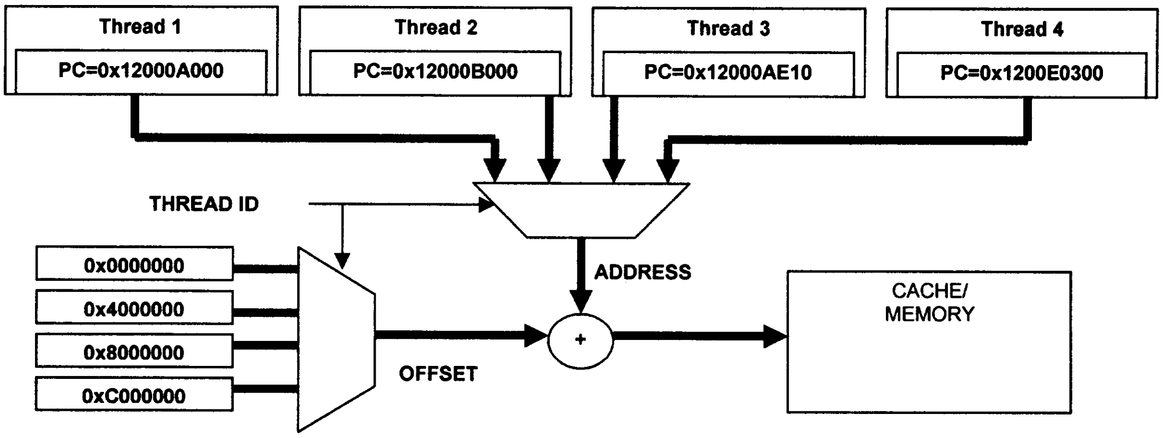


Figure 2.5 Instruction address translation.

Translated addresses are also used with the branch prediction unit, eliminating the need for a thread ID to be integrated into the branch target buffer entries. The simulator's branch prediction unit also maintains the return address stack. Call instructions push the return address into the return address stack while return instructions pop off the address at the top of the stack. Separate return address stacks have been created to correctly maintain return addresses for each thread.

2.2.4 Fetching from Multiple Threads

The fetch unit requires a non-blocking instruction cache that can supply a continuous stream of instructions to the SMT processor. Two basic assumptions have been made: first, only one cache replacement can occur at any time; second, other threads can still access the cache while a replacement is being serviced. Threads that experience a cache miss while a replacement is ongoing are placed in a queue.

The fetch unit now selects one program counter among the different threads for every fetch cycle initiated. The simplest fetch policy is round-robin selection, where the fetch unit cycles through the different threads in a pre-determined order. This policy has been adapted in the SMT simulator but with some modifications to take into account instruction cache misses. The implementation of this fetch policy involves maintaining a status register and a priority selection circuitry. Each bit in the status register corresponds to a thread's fetching status. The status bit is reset to 0 if the thread has experienced an instruction cache miss that has not yet completed the necessary block replacement. The status bit set to 1 if the thread has not experienced a miss or has just finished a block replacement in the instruction cache. The thread selection circuitry picks the thread with a set status bit that has the highest priority. The priority levels of the threads rotate for every fetch cycle.

For example, consider the case where, out of eight threads, only threads 4 and 6 have status bits set and the thread with the highest priority for the current fetch cycle is thread 5. Thread 6 is considered to have the second highest priority while thread 4 has the least priority. The circuitry will therefore fetch from thread 6.

Another fetch policy implemented in the simulator is similar to the ICOUNT policy proposed in [12] that gives priority to threads that have the fewest instructions in the decode, rename, and instruction queues. The instruction count policy adapted gives priority to threads that have the fewest entries in the RUU. The number of RUU entries reflects the number of instructions a thread has in the pipeline. This fetch policy ensures that the resources are better distributed among threads, preventing threads with long latency instructions from dominating the pipeline.

The fetch unit will get as many instructions from the thread as the instruction fetch bandwidth can accommodate or until the instruction fetch queue is full. Fetching for the thread also stops when a branch is predicted to be taken.

2.2.5 Sharing Pipeline Resources

Among the shared resources in the processor pipeline are the instruction fetch queue (IFQ), the load/store queue (LSQ), and the register update unit (RUU). The instruction fetch queue keeps fetched instructions that are yet to be dispatched. In the SMT simulator, instructions fetched from different threads are placed in the same queue so each IFQ entry maintains a thread ID. In the case of a branch misprediction, only instructions belonging to the mispredicted thread are purged from the IFQ.

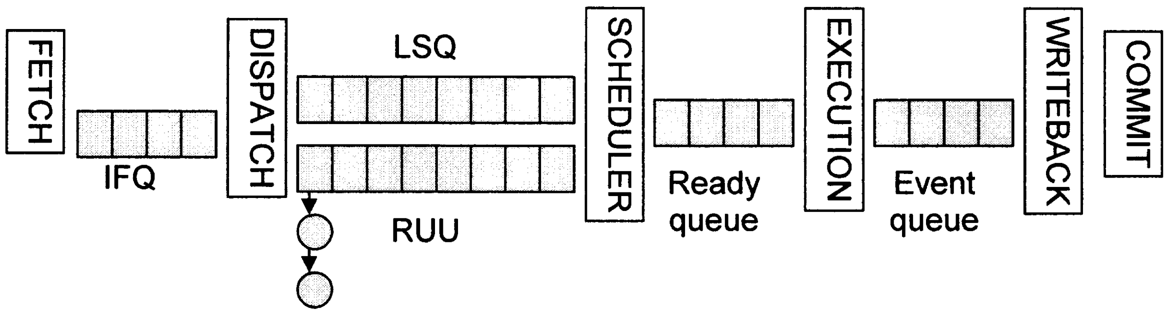


Figure 2.6 Shared resources in the pipeline.

The instructions in the IFQ are dispatched in the same order as they are fetched. The dispatch stage involves creating an RUU entry for each instruction retrieved from the IFQ. For load or store instructions, an LSQ entry is also created. RUU and LSQ entries maintain the thread ID, which is used to access and update the correct thread context. If the instruction produces a new value for a logical register, then the create vector table belonging to the instruction's thread is updated to reflect the logical register's latest value creator. If an instruction's operand has a value that is yet to be produced, it is added to the consumer list of the instruction that produces the latest value. During branch misprediction recovery, RUU and LSQ entries that have thread IDs matching the mispredicted thread are removed from the queues.

Once all instruction operands are available, the instruction is added to the ready queue. Ready queue entries are processed in order for each clock cycle. If the functional unit required by the instruction is available, then the instruction is issued and the corresponding resource is locked. Instructions that fail to issue due to unavailability of functional units or issue bandwidth restrictions are reinserted into the ready queue. Through the entire process, the order of instructions in the queue is always maintained. Instructions that have been invalidated due to branch misprediction are removed from the queue.

Issued instructions enter an event queue, which sorts entries according to their time of completion. During the writeback stage, instructions that have finished execution are claimed from the event queue. Results are broadcasted to consuming instructions, which are inserted into the ready queue if all their operands are already available. Branch mispredictions are also detected and misprediction recovery is initiated at this stage. The instruction's thread ID determines which entries in the IFQ, RUU, and LSQ are squashed.

Instructions commit in program order. In the original simulator, a completed instruction at the head of the RUU updates the processor state during the commit stage. All completed instructions accommodated by the commit bandwidth are processed during this stage. For the SMT simulator, each thread maintains its own head pointer in the RUU so that an earlier incomplete instruction from one thread does not prevent other threads from committing. The oldest thread, which has an instruction at the head of the RUU, is given first priority. If its completed instructions do not take up the entire commit bandwidth, then the next thread head is considered. The simulator simply cycles through the thread IDs when checking thread heads; for example, if the RUU head is occupied by thread 2 then the next thread head to be considered for commit is thread 3.

The queues were implemented as arrays in the original simulator. Having multiple threads in the pipeline leads to discontinuities in the array queue when a thread commits ahead of an older thread or if a misprediction leads to a thread's entries being squashed. Making the array compact by eliminating the invalidated elements within the queue takes additional processing time that is proportional to the size of the queue in the worst case. In order to make updates to the queue more efficient, the implementation has been changed from an array to a doubly linked list.

Pointers to the next and previous nodes in the queue were added to the definition of the queue node. Instead of head and tail indices, the different queues maintain pointers to the head and tail nodes. The head and tail pointers point to the same node when the queue is empty and when the queue is full. Otherwise, the head pointer points to the first valid node in the queue while the tail pointer points to the first empty node. The LSQ and RUU also maintain head pointers for each thread, making it easier to commit completed instructions from any thread during the commit stage. The queues keep track of the total number of entries as well as the number of entries per thread.

- When a node is invalidated, the different pointers are adjusted depending on the position of the node and the current size of the queue:
- If the node to be squashed is the head of the queue, then the queue head pointer is adjusted to point to the next node.
- If the node is not the head of the queue and the queue is not full, then the node is removed from its current position and inserted after the queue tail node.
- If the node is not the head of the queue and the queue is full, then the node is removed from its current position and inserted before the queue tail node. The queue tail pointer is then adjusted to point to the newly inserted node. For

threads with no entries in the queue, their head pointer is also adjusted to point to the new tail node.

- In the case where the invalidated node was the head of a thread and the thread still has entries in the queue, the thread head pointer traverses the queue until it points to the first node belonging to the thread. If the thread no longer has entries left in the queue, the thread head points to the new tail node.

Most of the operations run in constant time; only cases where the tail pointer or thread head pointers are adjusted require operations with running times dependent on the number of threads or the number of entries in the queue.

2.2.6 Redirecting Program Input and Output

The binary files running on the simulator often have system calls, which are executed by the simulator. Among the system calls are those that involve standard input and output. To keep input and output separate for each thread, the SMT simulator requires that input and output files for each thread be specified so the necessary redirection is performed.

Command line options -redir:inX, and -redir:outX, where X, stands for the thread number, have been added.

```
sim-smtp -threads 2 -redir:in1 in1.txt -redir:out1 out1.txt -redir:in2 in2.txt
-redir:out2 out2.txt prog1.bin + prog2.bin
```

The command above runs two program binaries, prog1.bin and prog2.bin, that do not require additional command line arguments. Standard inputs required for executing prog1 are stored in the file in1.txt while standard inputs for prog2 are stored in in2.txt. The outputs for prog1 will be written to out1.txt and outputs for prog2 in out2.txt.

Redirection of the program output has been adapted from the original simulator, which simply opens for the program an output file stream where output text is dumped. The SMT simulator uses the file stream corresponding to the calling thread when processing a system call specifying a write operation. File streams cannot be used for redirecting input, however, since many system calls implemented by the simulator involve the standard input and they always assume a file descriptor of 0. During a system call in the SMT simulator, the standard input's file descriptor is made a duplicate of the file descriptor of the calling thread's input file. This means that the standard input descriptor shares the locks, file position pointers, and flags of the input file. Whatever operations are performed on the standard input will also reflect on the input file's descriptor.

2.2.7 Tracking Thread Statistics

The results of the original simulator are from counters that track indicators of the overall performance of the simulator, such as instructions committed, cache misses, branch prediction hits, etc. For the SMT simulator, it is relevant to know how each thread performs so additional counters have been introduced. The following information is now available for each thread:

- total number of instructions, loads, stores, and branches committed
- total number of instructions, loads, stores, and branches executed
- total number of accesses, hits, misses, replacements, and invalidations for each cache and TLB specified
- miss rate, replacement rate, writeback rate, and invalidation rate for each cache and TLB specified

Individual thread results give an idea of how each thread performs during the simulator's run. These results can reveal, for example, how threads compete for the processor's resources.

CHAPTER 3

SMT SIMULATOR EVALUATION

3.1 Simulation Parameters

Two types of evaluation were performed on the SMT simulator. The first set of simulations involved verifying the functionality of the redesigned simulator. The second set of simulations served to determine the effectiveness of simultaneous multithreading over the original superscalar configuration.

For all simulations, the processor parameters listed in Table 3.1 were used.

Table 3.1 Simulation Parameters

Parameter	Value
Issue width	4 or 8 instructions
RUU size	256 or 512 entries
LSQ size	256 or 512 entries
Integer ALUs	6 units
Floating point ALUs	4 units
Integer multipliers	2 units
Floating point multipliers	2 units
Memory ports	4 ports
L1 Instruction Cache	64KB (64B/line, 2 associativity)
	LRU replacement, 1 clock cycle
L1 Data Cache	64KB (64B/line, 2 associativity)
	LRU replacement, 2 clock cycles
L2 Cache	4MB (128B/line, 8 associativity)
	LRU replacement, 12 clock cycles
Instruction TLB	1MB, 30 clock cycles
Data TLB	1MB, 30 clock cycles
Main memory	225 clock cycles
Branch predictor	Two-level, 4k-entry PHT
	10-bit history registers
BTB	1024 sets, 4 associativity

Table 3.2 Test Sets with Two Threads

Test	Benchmarks	Comment
1	wupwise, gzip	FP + INT
2	swim, vpr	
3	mgrid, gcc	
4	applu, mcf	
5	mesa, crafty	
6	galgel, art	All FP
7	equake, facerec	
8	lucas, fma3d	
9	sixtrack, apsi	
10	parser, eon	All INT
11	perlbmk, gap	
12	vortex, bzip2	
13	twolf, gzip	

Table 3.3 Test Sets with Four Threads

Test	Benchmarks	Comment
1	wupwise, swim, gzip, vpr	2 FP + 2 INT
2	mgrid, applu, gcc, mcf	
3	mesa, galgel, crafty, parser	
4	art, equake, eon, perlbmk	
5	facerec, lucas, fma3d, sixtrack	All FP
6	apsi, wupwise, swim, mgrid	
7	gap, vortex, bzip2, twolf	All INT
8	gzip, vpr, gcc, mcf	

Table 3.4 Test Sets with Eight Threads

Test	Benchmarks	Comment
1	wupwise, swim, mgrid, applu, mesa, galgel, art, equake	All FP
2	gzip, vpr, gcc, mcf, crafty, parser, eon, perlbmk	All INT
3	facerec, lucas, fma3d, sixtrack, gap, vortex, bzip2, twolf	4 FP + 4 INT
4	apsi, swim, applu, galgel, gzip, gcc, crafty, eon	
5	wupwise, mgrid, mesa, art, vpr, mcf, parser, perlbmk	

3.2 Verifying Functionality

To check whether the simulator runs correctly, all SPEC2000 integer and floating point benchmarks were executed on both the original simulator and the SMT simulator. For each benchmark, the first 200 million instructions were fast forwarded and then simulated for 500 million instructions.

The outputs of the programs were compared and no differences were found, suggesting that the SMT simulator functioned correctly. The simulation results were close but not exact in value. The differences can be attributed to the changes that have been made to the pipeline to process multiple threads.

3.3 Evaluating Performance

SMT performance evaluations were made using combinations of the SPEC2000 integer and floating point benchmarks. Tables 3.2, 3.3, and 3.4 show the combinations of benchmarks use for two, four, and eight threads, respectively. For simultaneous multithreading simulations, all threads are fast-forwarded for 200 million instructions then simulated for 300 million instructions multiplied by the number of threads being executed (600 million for two threads, 1200 million for four threads, and 2400 million for eight threads).

3.3.1 Fetch Policies

The round robin and instruction count fetch policies were first compared. The issue width was set to four and the RUU size to 256. Simulation results shown in Figures 3.1, 3.2 and 3.3 indicate that the instruction count policy is superior to the round robin policy for two, four, and eight threads. This supports the findings from previous studies on the performance of different fetch policies [12]. For the rest of the simulations, the instruction count fetch policy was used.

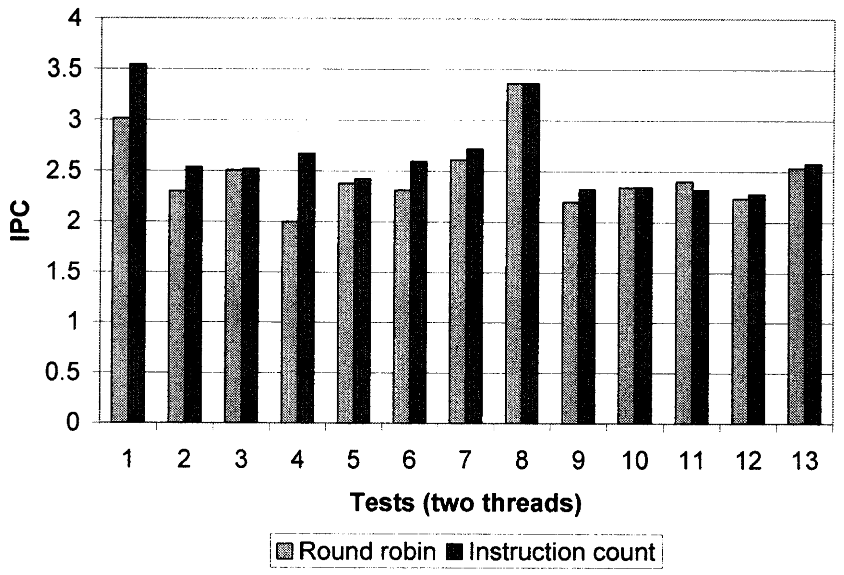


Figure 3.1 Comparison of round robin and instruction fetch policies for two threads.

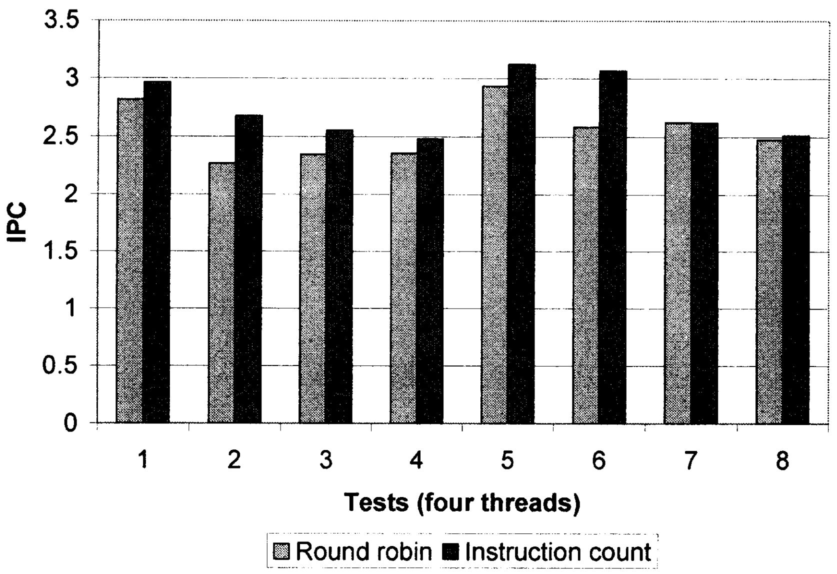


Figure 3.2 Comparison of round robin and instruction fetch policies for four threads.

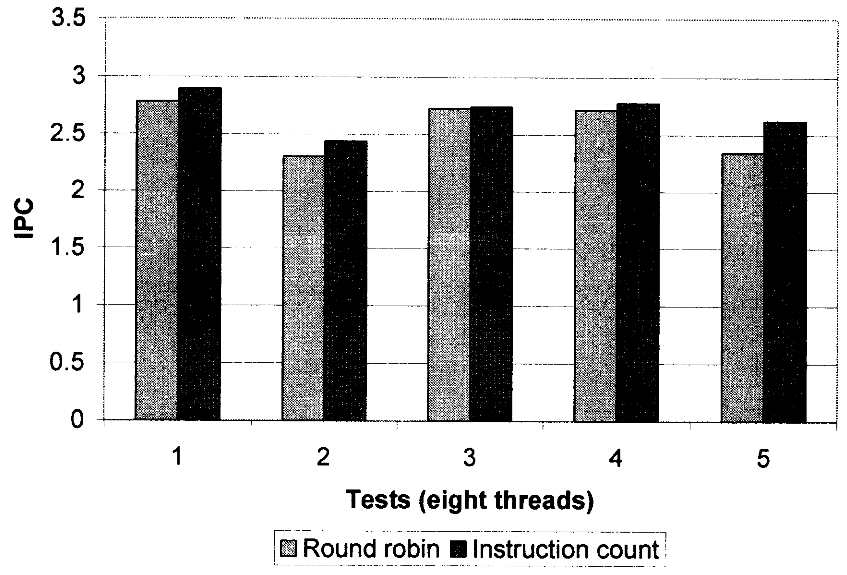


Figure 3.3 Comparison of round robin and instruction fetch policies for eight threads.

3.3.2 Sequential and Simultaneous Execution

To compare simultaneous multithreading with superscalar processing, the same threads run on the SMT are executed on the superscalar processor and calculation of IPC and other rates are determined from the sum of the results of individual threads. For sequential execution of threads, each thread is fast-forwarded for 200 million instructions and then simulation proceeds for 300 million instructions.

The first set of simulations were performed to compare sequential execution and simultaneous multithreading for an issue width of four instructions and an RUU size of 256. Figures 3.4, 3.5, and 3.6 show the results for two, four, and eight threads, respectively. The SMT results were consistently higher than the calculated performance for sequential execution. The simulations with four and eight threads showed greater IPC increases (34% and 32%, respectively) for simultaneous execution than the simulations with only two threads (25%).

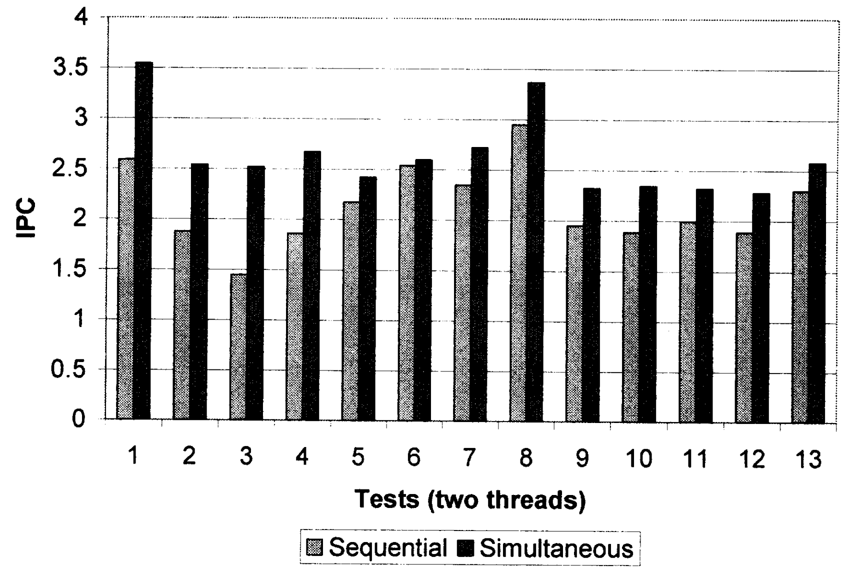


Figure 3.4 Comparison of sequential and SMT IPC for two threads (issue width 4, RUU size 256).

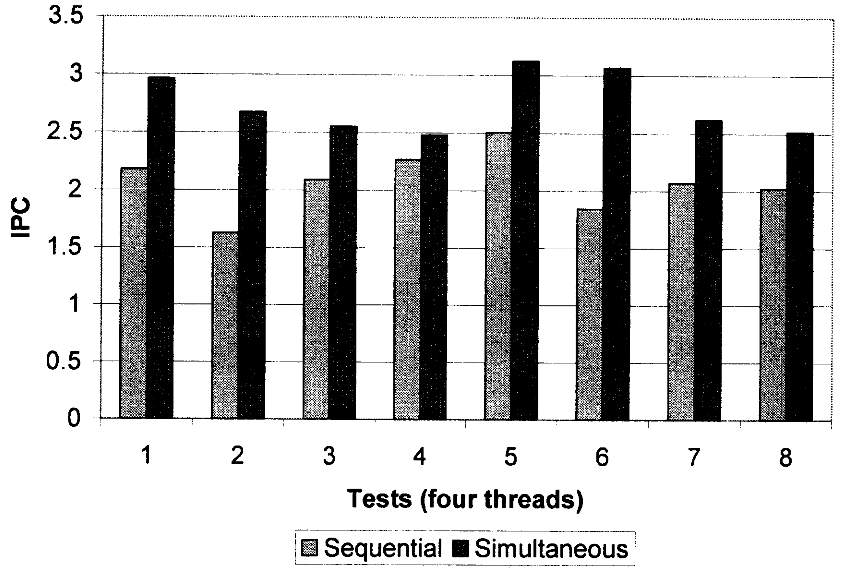


Figure 3.5 Comparison of sequential and SMT IPC for four threads (issue width 4, RUU size 256).

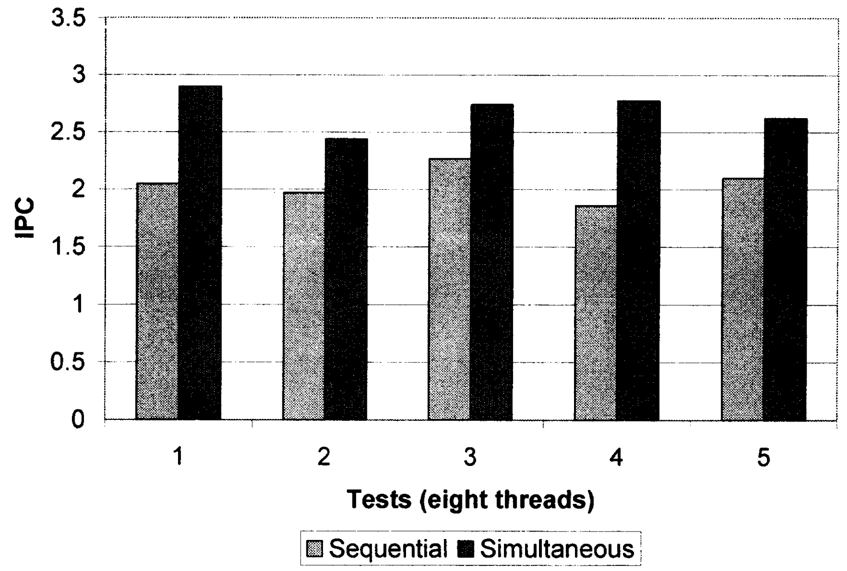


Figure 3.6 Comparison of sequential and SMT IPC for eight threads (issue width 4, RUU size 256).

The second set of simulations had the issue width increased to eight instructions while keeping the RUU size at 256. Figures 3.7, 3.8, and 3.9 show the results of this configuration for two, four, and eight threads, respectively.

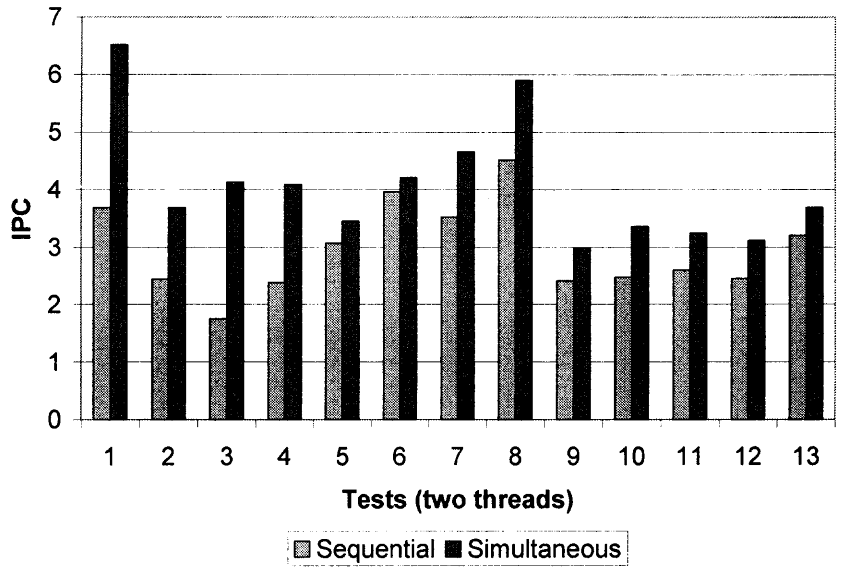


Figure 3.7 Comparison of IPC for sequential and simultaneous execution of two threads (issue width 8, RUU size 256).

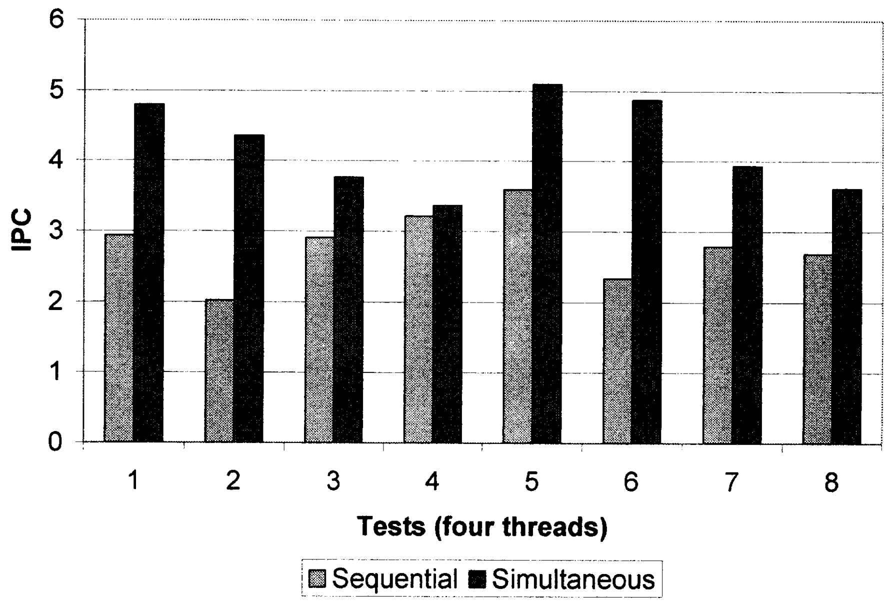


Figure 3.8 Comparison of IPC for sequential and simultaneous execution of four threads (issue width 8, RUU size 256).

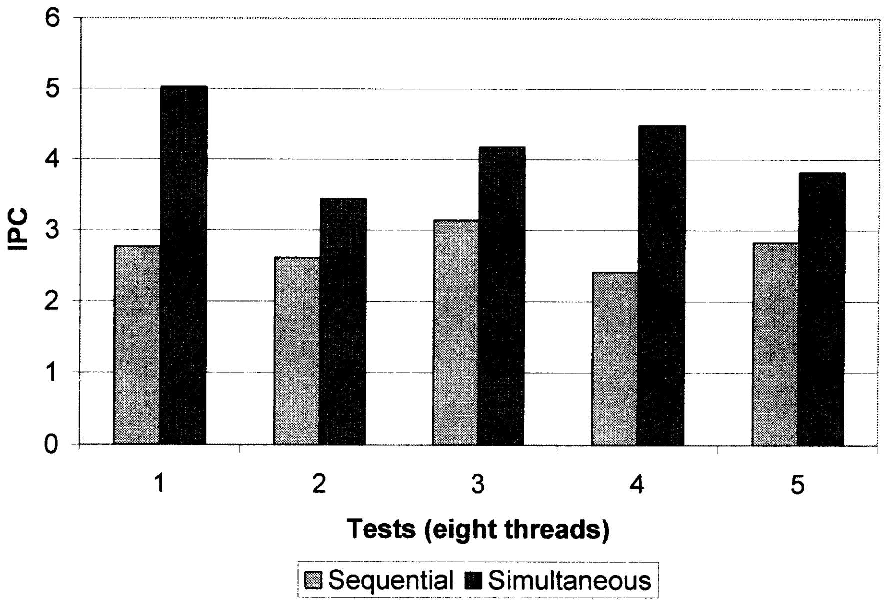


Figure 3.9 Comparison of IPC for sequential and simultaneous execution of eight threads (issue width 8, RUU size 256).

Similar to the first set, the results show the superiority of simultaneous multithreading to sequential execution of threads on a superscalar processor. The IPC gains in this set, however, are much greater because of the increased issue width. The average IPC increase is 41% for two threads, 54% for four threads, and 53% for eight threads. The average IPC for running eight threads is around 4.0, similar to the results obtained in [12].

Comparison of average execution bandwidth, which takes into account speculative instructions, shows that for the processor parameters given in Table 3.1 and an issue width of four instructions, about 2.8 instructions are issued per cycle. For an issue width of eight instructions, the average instructions issued per cycle is 4.4.

The third set of simulations had an issue width of eight instructions and an RUU size of 512. The results shown in Figures 3.10, 3.11 and 3.12 show that an RUU size of 256 is normally sufficient for the SMT processor. There is an increase in IPC but for many of the tests simulated, the gain was not very significant. When eight threads are being simultaneously executed, having an RUU of 512 entries offers no advantage over an RUU of 256 entries (see Figure 3.12).

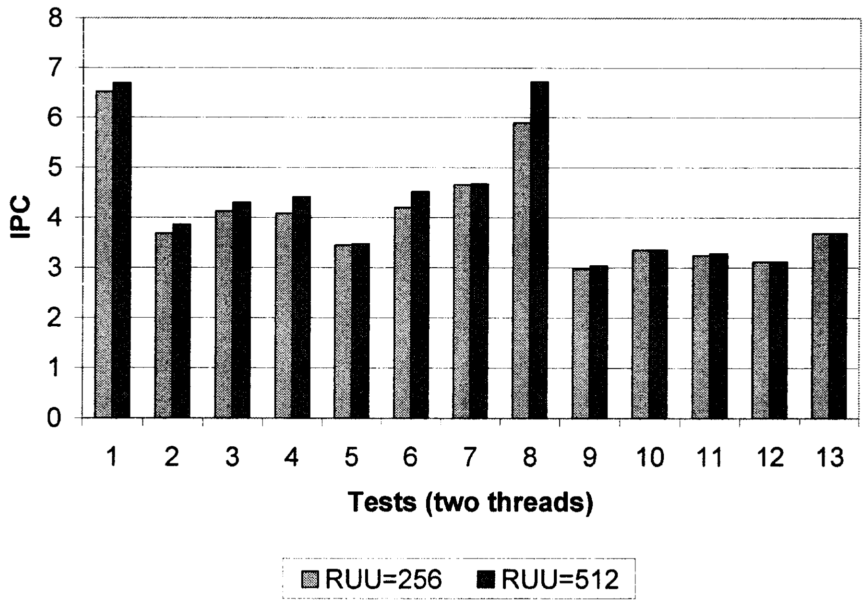


Figure 3.10 SMT IPC for two threads with RUU sizes 256 and 512.

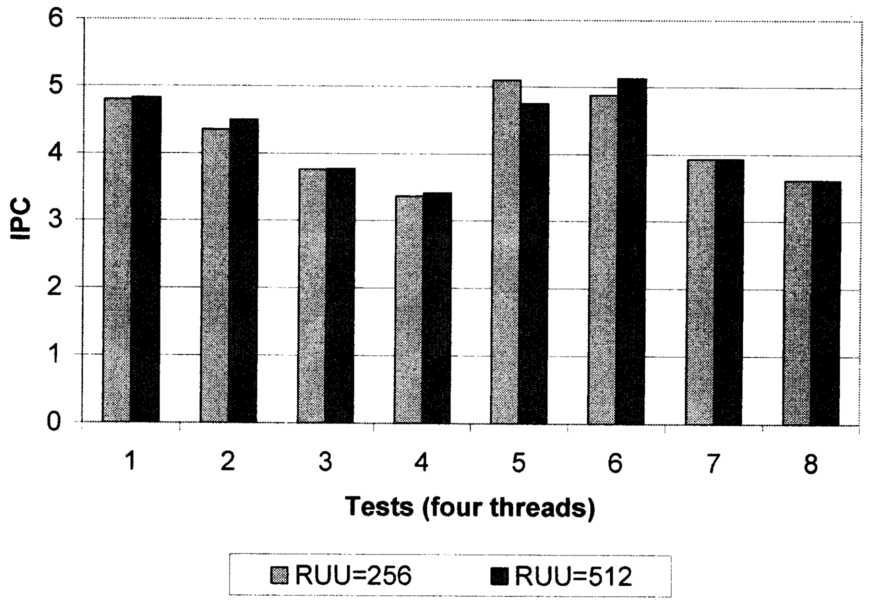


Figure 3.11 SMT IPC for four threads with RUU sizes 256 and 512.

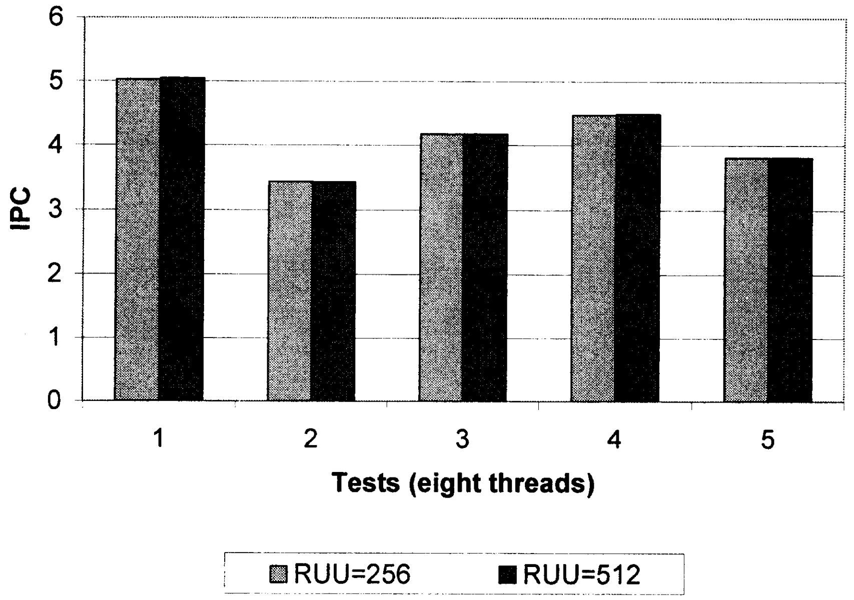


Figure 3.12 SMT IPC for eight threads with RUU sizes 256 and 512.

In summary, the simulation results show that simultaneous multithreading provides significant improvement over superscalar processing, especially when more independent threads are being executed.

CHAPTER 4

USE-BASED REGISTER CACHE EVALUATION

4.1 The Register Cache

The register cache model is similar in rationale to the memory hierarchy model, where a smaller but faster cache containing a subset of information from the main memory directly interacts with the processor. In the case of the register hierarchy, a register cache is a small bank of physical registers that directly provides the operands to functional units. The register cache contains only a subset of the values that are stored in a larger bank of physical registers called the backing file. Because of its small size, the register cache has an access latency that is much shorter compared to the access latency of the backing file.

Just like the memory hierarchy, the register hierarchy also requires management schemes to determine which values should be inserted into the cache and which values in the cache should be retired to the backing file. However, register values do not possess the same temporal and spatial locality properties observed by instructions and data in a program. The register cache management scheme, therefore, has to be designed especially for the properties exhibited by register values. Such characteristics may be highly program sensitive.

Early designs of the register hierarchy were dependent on compiler support to manage the register cache [33], [34]. Implementations of a hierarchical banked register file for dynamically scheduled processors were proposed in [16] and [17]. The latter's design was not strictly a cache, since the faster and smaller bank contained values that were not part of the slower and larger bank. A problem with maintaining exclusive banks is that the management of the register file hierarchy tends to be more complicated, requiring additional structures to maintain information and requiring modifications to other parts of the pipeline.

In 2004, Butts and Sohi [18] proposed a register caching scheme that bases insertion and replacement policies on the number of consumers that a value has. Values that have more consumers are maintained in the cache. Also, the authors propose decoupled indexing in assigning a set to a value. The basis of the index assignment is on the number of consumers; use-based algorithms avoid assigning a value to sets with high use registers. The use-based register cache with decoupled indexing was shown to effectively improve performance.

4.2 Adopting Register Caching in SMT

To support a register cache, several changes were made to the pipeline of the simulator.

- Physical registers corresponding to the backing file have been modeled. The simulator also maintains the list of physical registers that are free. A physical register is assigned to a logical register at the dispatch stage. The physical register is freed once another instruction producing a new value for the logical register commits or if the instruction is speculative and is squashed after a branch misprediction is discovered. The dispatch stage stalls when no physical register is available.

- A register mapping table is maintained to determine the most recent logical-to-physical register assignment. It also maintains the index that is associated to the physical register. The table is updated every time an instruction undergoes register renaming at the dispatch stage.

- A decoupled index is assigned to the instruction's destination register and maintained in the instruction's RUU entry. The RUU entry also maintains the physical register ID and index of each source and destination register.

- The register cache supports all the read and write ports required by the issue width. The backing file supports several write ports but only one read port.

When an instruction fails to issue because not all operands are in the register cache, it is reissued once the operand value has been added to the register cache.

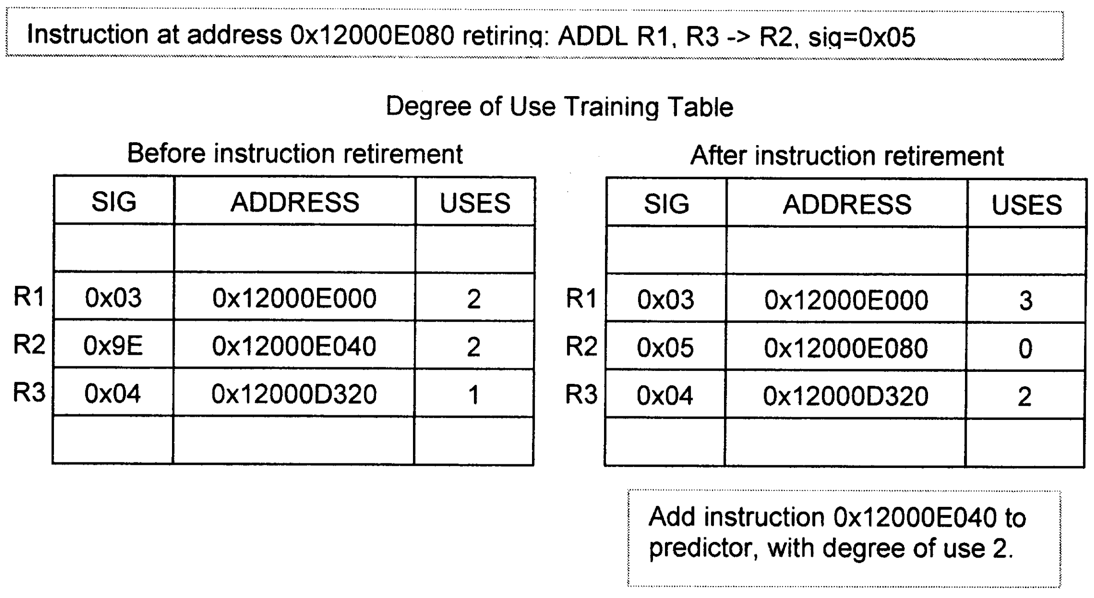
- The writeback stage updates the register cache if the produced value has at least one consumer. At the same time, the value is written to the physical register in the backing file.
- Several stages of bypass are required, the number of stages depending on the number of cycles needed to write to the backing file. If source operands are available through the bypass network, the issue stage skips the reading of the register file but decrements the uses of the corresponding physical registers.
- Separate training tables and future control flow information are maintained for each thread.

4.3 Use-Based Policies

4.3.1 Predicting the Number of Consumers

Butts and Sohi estimate the number of consumers for a register value through a degree of use predictor, which the authors developed in an earlier paper [35]. The degree of use predictor has two components: a degree of use training table and a set-associative predictor table. The training table (Figure 4.1) keeps track of the uses of a register value that has not yet been overwritten. When a register value produced by an instruction at address PC has been replaced by a newer instruction writing to the same register, it is retired to the predictor table (Figure 4.2) and is accessed the next time the instruction at PC is encountered.

The training table has entries for each logical register. Whenever an instruction commits, the use count of the source operands are incremented and the entry of the destination register is updated. The old entry of the training table is then added to the predictor table. Aside from the number of uses, the relevant information maintained in the training table is the address of the instruction producing the latest value of the register and a signature that contains future control flow information.



Instruction at address 0x12000E080 retiring: ADDL R1, R3 -> R2, sig=0x05

Degree of Use Training Table

Before instruction retirement

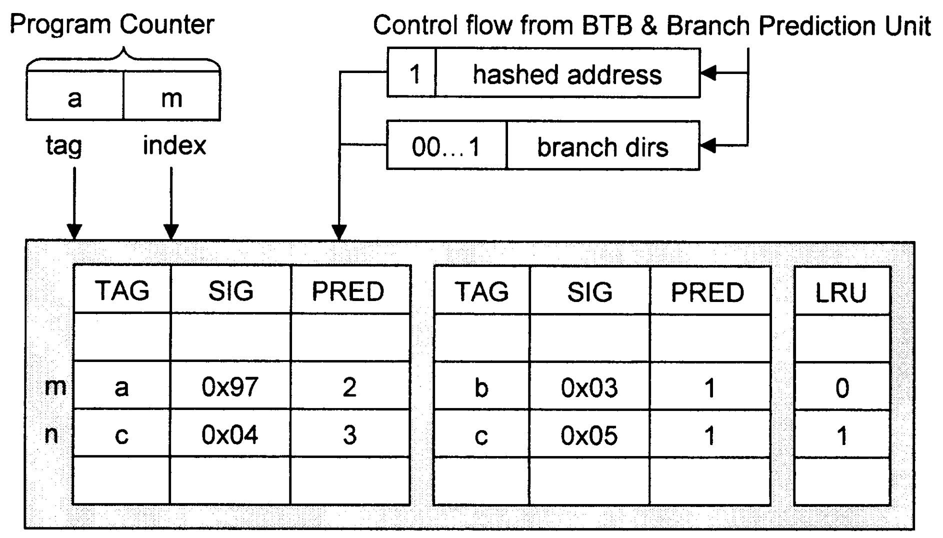
	SIG	ADDRESS	USES
R1	0x03	0x12000E000	2
R2	0x9E	0x12000E040	2
R3	0x04	0x12000D320	1

After instruction retirement

	SIG	ADDRESS	USES
R1	0x03	0x12000E000	3
R2	0x05	0x12000E080	0
R3	0x04	0x12000D320	2

Add instruction 0x12000E040 to predictor, with degree of use 2.

Figure 4.1 Degree of use training table.



	TAG	SIG	PRED	TAG	SIG	PRED	LRU
m	a	0x97	2	b	0x03	1	0
n	c	0x04	3	c	0x05	1	1

Figure 4.2 The degree of use predictor.

When a training table entry is forwarded to the predictor, the lower bits of the address of the value-producing instruction are used as an index to the predictor table. The higher bits are stored as a tag. The signature is copied to the table and the number of uses tracked by the training table becomes the predicted degree of use. The predictor table also maintains a bit to track the least recently used entry, which is used to select the table entry to be overwritten to accommodate new information.

Because the number of consumers of a register value is dependent on the future control flow, the signature makes the prediction more accurate by maintaining information regarding future branch directions. It is possible to sample the future control path since an instruction's predicted number of consumers is added to the degree of use predictor table only when the physical register containing the instruction's result has been freed. During the interval when the register value is considered to be live, a certain number of branches will have entered the pipeline.

The signature can have one of two formats, as shown in Figure 4.2. When an unconditional jump closely follows an instruction, the signature's most significant bit is set to 1 and the succeeding bits contain the lower bits of the target address. In the simulator, this format is used when an instruction is closely followed by a return instruction. In the case where an instruction is followed by conditional branch instructions, then the signature's most significant bit is reset and the rest of the bits encode the number of branches encountered and the predicted directions of these branches.

A prediction is only made if there is both a tag match and a signature match. If the predictor does not find a tag and signature match, the default degree of use is 1.

4.3.2 Register Cache Read and Write

The register cache (Figure 4.3) is set-associative and each entry maintains the physical register ID, the value, and the number of remaining uses. The register cache is accessed by using the index assigned to the physical register (saved in the instruction's RUU entry) to access a register cache line. If the register cache has multiple ways, then each way's register ID is compared with the ID of the physical register that is being looked up. A match will return the value stored in the register cache and decrement the corresponding number of uses. Otherwise, the backing file is accessed to bring the value into the register cache.

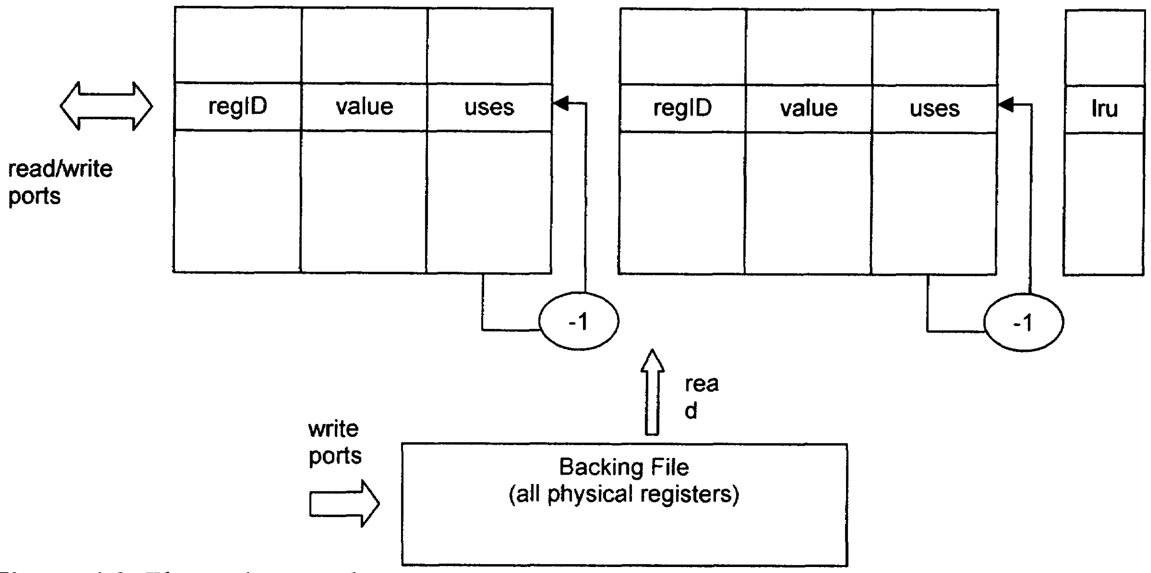


Figure 4.3 The register cache.

A register cache write can occur in two instances. The first case is when a register cache read misses. The second case is when a value produced by a functional unit has a predicted number of uses that is greater than zero. In both cases, the index assigned to the physical register is used to select the register cache line. The entry that is overwritten is the one with the least number of uses. In case of a tie, then the least recently used entry is selected for replacement.

4.3.3 Decoupled Indexing

Previous register hierarchy management schemes derived indices that were associated with the register ID, similar to how caches in memory hierarchies derive indices from memory addresses. However, register values do not observe the spatial and temporal localities exhibited by instructions so this kind of index assignment scheme is not effective.

Butts and Sohi proposed the use of decoupled indexing in assigning values to cache lines. The basic idea is to assign indices corresponding to cache lines that have minimal used. The best index assignment is to a cache line having the smallest number of consumers. Finding the cache line with minimum uses is not very easy to implement, though, so the authors proposed the use of a round robin scheme that filters out lines with high-use entries.

The simulator implements the filtered round robin scheme that skips cache lines when entries have high-use values. In the event that all of the cache lines have high-use entries, then all the index assignment proceeds in a round robin fashion without filtering.

4.4 Simulation Results

The use-based register cache was simulated for one, two, and four threads using the processor parameters listed in Table 3.1, with the RUU size set to 512 and the issue width at 8 instructions. The sets of benchmarks used for two and four threads are those appearing in Tables 3.2 and 3.3. The degree of use predictor had 4K lines and four ways

(16K entries), which is four times the size of the predictor used in the original paper [35]. The register cache had 64 entries, organized in a 4-way set-associative table, while the backing file contained 1024 physical registers. It has been assumed that a read operation from the backing file requires 3 clock cycles.

4.4.1 Evaluating the Degree of Use Predictor

The degree of use predictor was first evaluated. Figures 4.4, 4.5, and 4.6 show the hit rate and the accuracy of the predictor for one, two, and four threads, respectively. The hit rate gives the percentage of predictor accesses that had a tag and signature match. This reveals the code coverage of the predictor. The accuracy gives the percentage of predictions that were later verified as correct.

For a single thread, the predictor has an average hit rate of 99% and an average accuracy of 94%. When two threads are running simultaneously, the average hit rate is 98% while the average accuracy of the predictor remains at 94%. When four threads are executed, the average hit rate is 96% and the average accuracy is slightly lower at 92%.

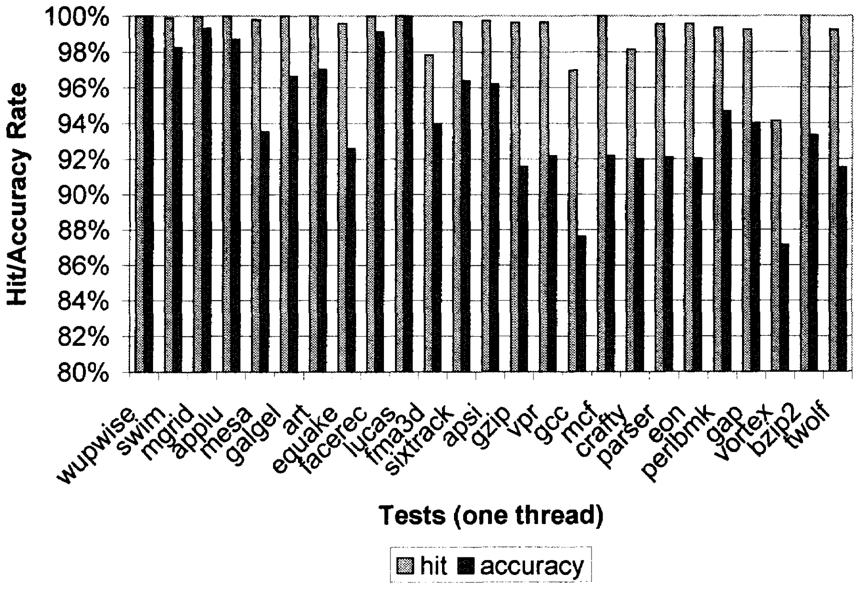


Figure 4.4 Hit rate and accuracy of the degree of use predictor for one thread.

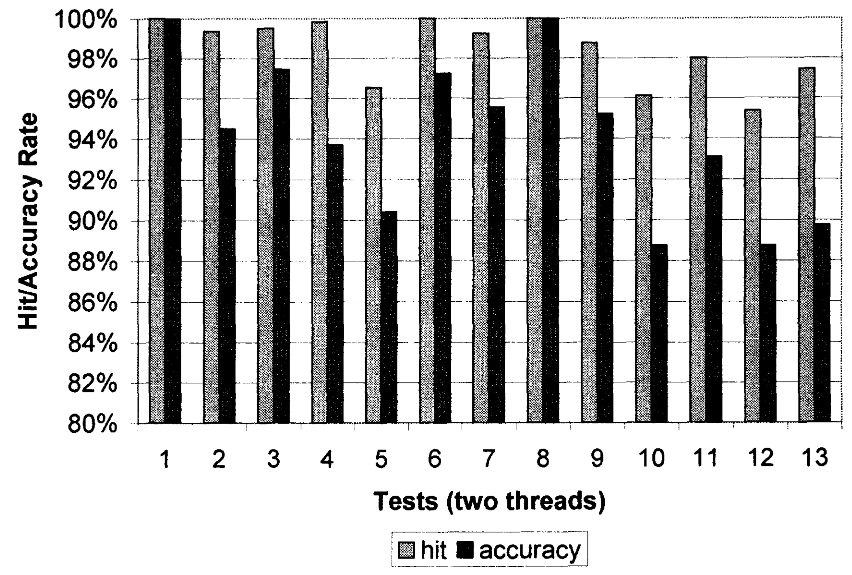


Figure 4.5 Hit rate and accuracy of the degree of use predictor for two threads.

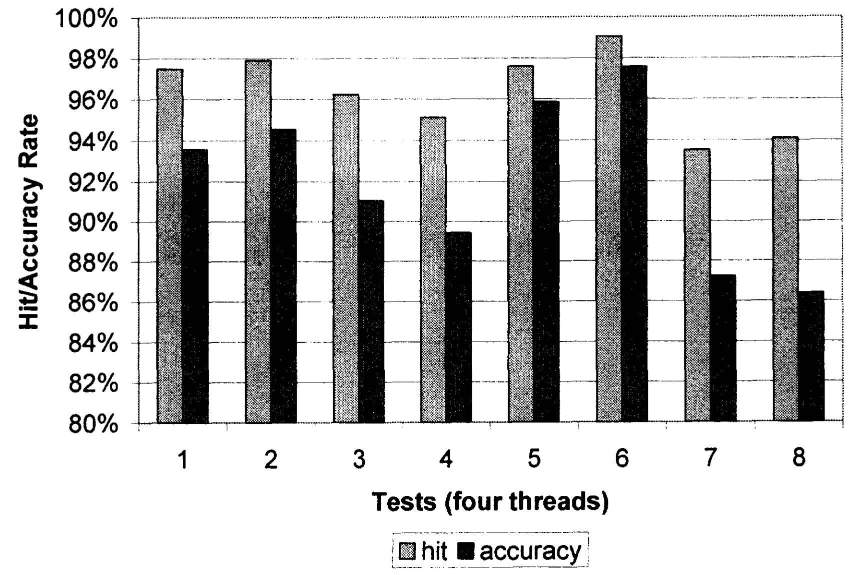


Figure 4.6 Hit rate and accuracy of the degree of use predictor for four threads.

The results indicate that the degree of use prediction table size chosen for the simulations (16K entries) is adequate for simultaneous multithreading of a number of threads.

4.4.2 Evaluation the Use-Based Register Cache

The performance of the use-based register cache was first compared to the performance of a monolithic register file with an access latency of three cycles. Three register cache sizes were simulated: 32, 64, and 128 registers. Figures 4.7, 4.8, and 4.9 show the simulation results for one, two, and four threads, respectively.

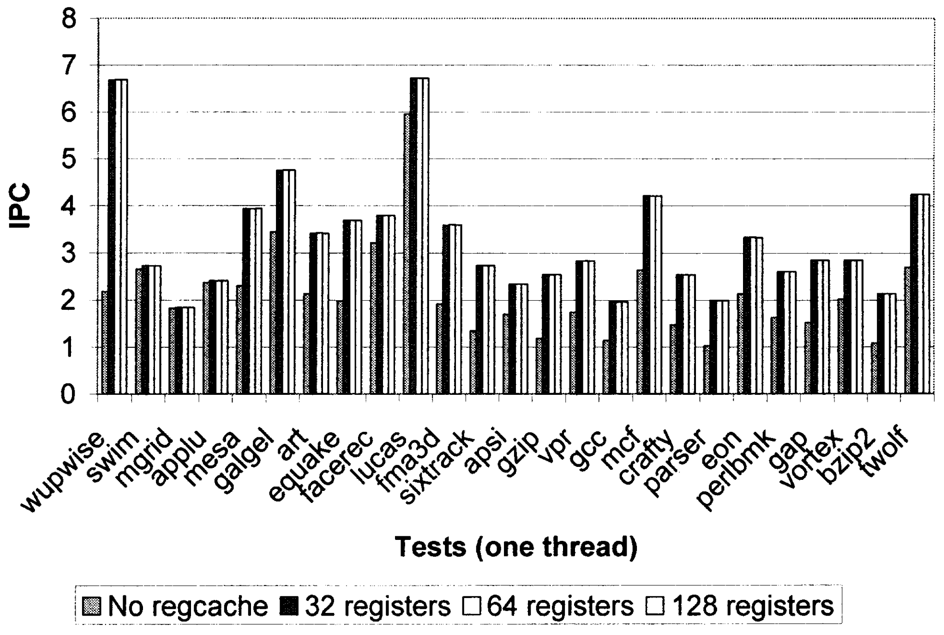


Figure 4.7 IPC with and without register caching for one thread.

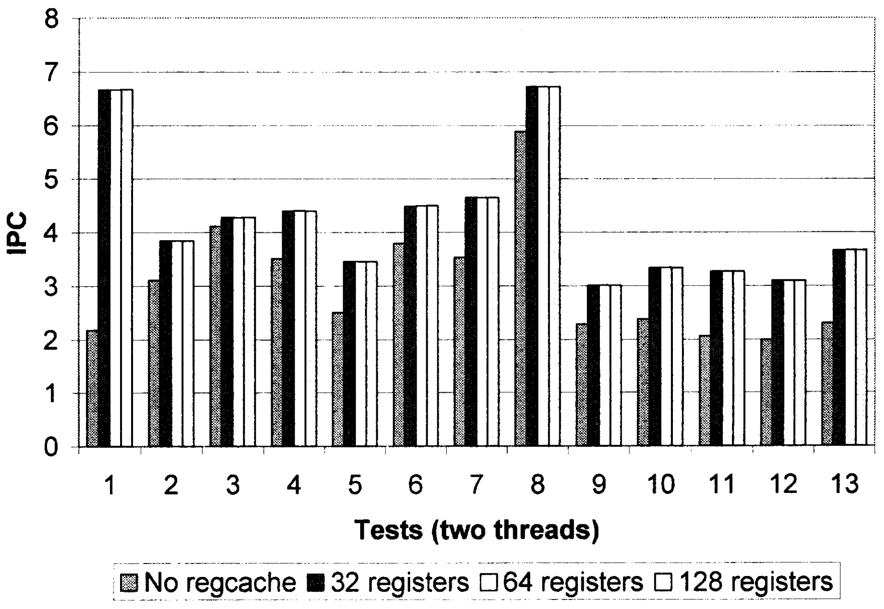


Figure 4.8 IPC with and without register caching for two threads.

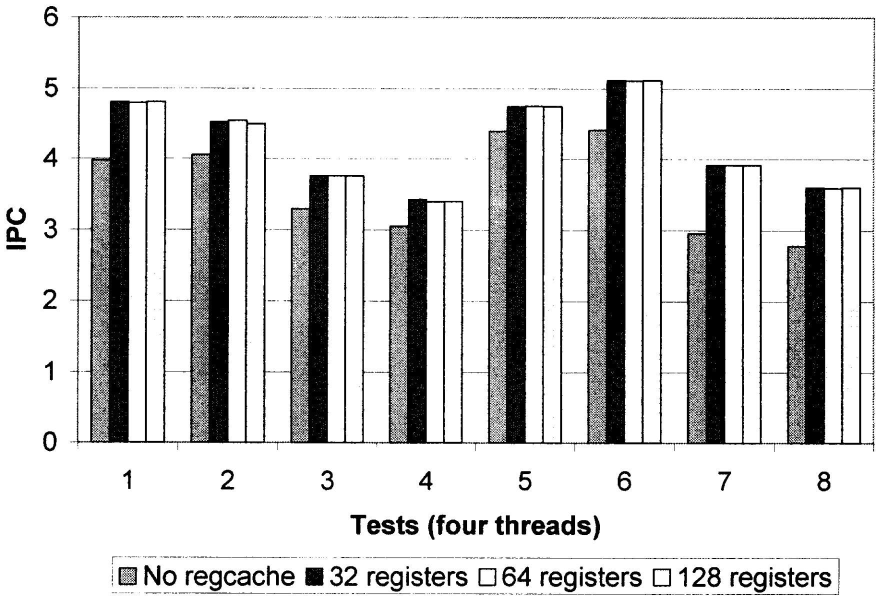


Figure 4.9 IPC with and without register caching for four threads.

The effectiveness of the use-based register cache is obvious from the results above. Simulations showed that the SMT processor using register caching has a speedup over the processor with a monolithic register file averaging 55% for a single thread, 38% for two threads, and 17% for four threads. The SMT processor's ability to dynamically select from independent threads allows it to perform well despite the long access latency of the register file. This is why the effectiveness of register caching with multiple threads is not quite as large as its effectiveness with a single thread.

Even with only 32 registers, the results in Figures 4.7 to 4.9 show that the SMT simulator is able to achieve very high performance. The IPC values for register caching are close to the ideal case where the access latency of the register file is just one cycle.

The next set of simulation results, shown in Figures 4.10 to 4.12, give the register cache hit rates when the register cache is composed of either 32 or 64 registers. When a single thread is running, the average register cache hit rate is 88% for 32 registers and

90% for 64 registers. When two threads are executed simultaneously, the hit rates are 87% and 91% for 32 and 64 registers, respectively. When four threads are running, the hit rate are 81% for 32 registers and 87% for 64 registers.

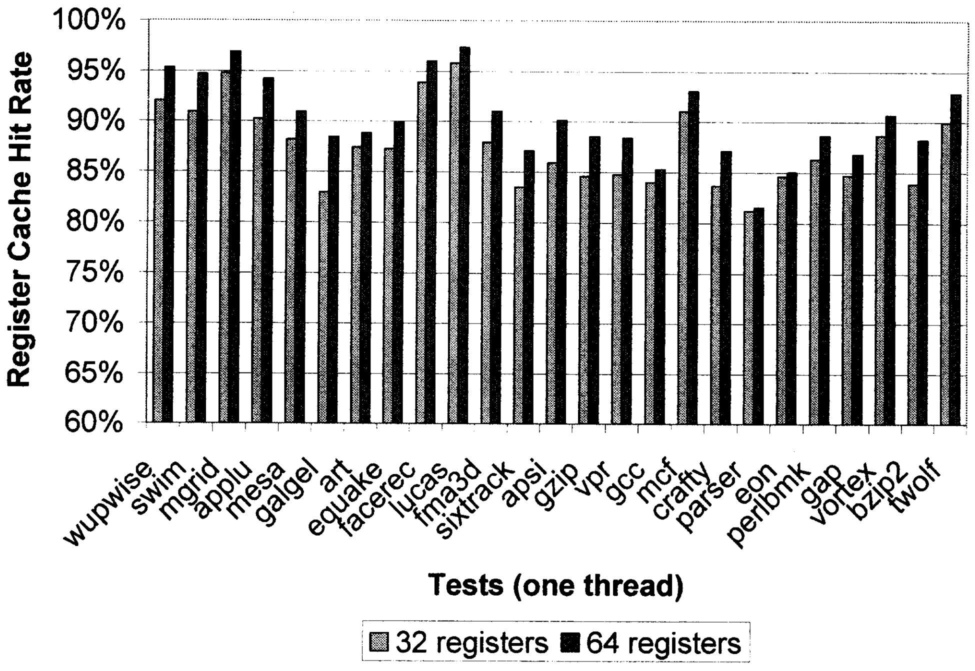


Figure 4.10 Register cache hit rates for one thread.

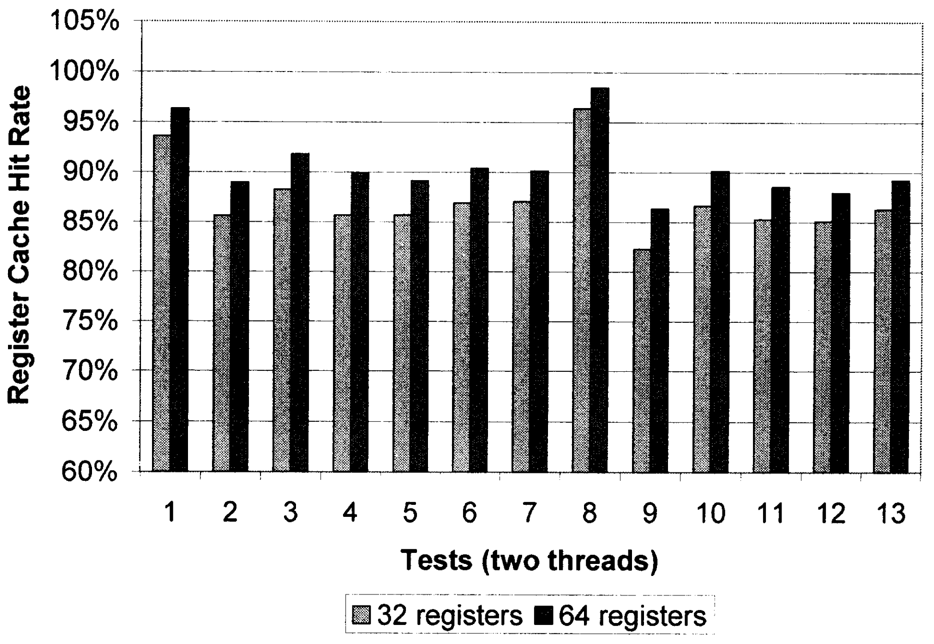


Figure 4.11 Register cache hit rates for two threads.

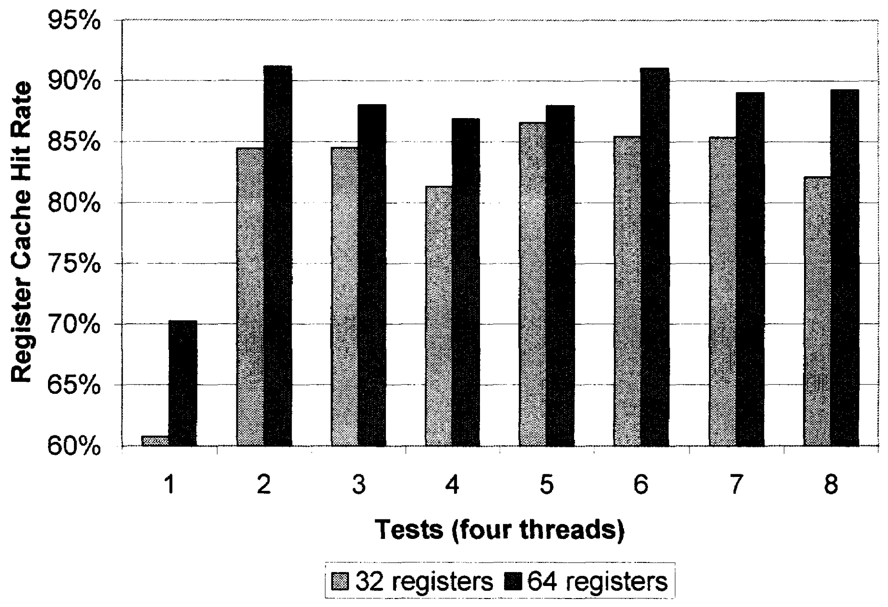


Figure 4.12 Register cache hit rates for four threads.

The results for both IPC (Figures 4.7 to 4.9) and register cache hit rates (Figures 4.10 to 4.12) indicate that the performance of the SMT processor does not depend on the register cache hit rate. This work has assumed an ideal case, where the issue slot of the instruction experiencing a register cache miss can be filled in the same clock cycle with another ready instruction. This is one reason why the performance of the SMT processor remains high despite a decrease in the register cache hit rate.

Smaller cache sizes were not evaluated. The choice of cache size will depend on the acceptable performance for running single threads on the SMT processor, since it is with single threads that the register caching scheme has the greatest impact.

CHAPTER 5

CONCLUSION

A simultaneous multithreading processor simulator was developed from the superscalar pipeline of the SimpleScalar simulation tool, employing the basic changes that were proposed by the original proponents of simultaneous multithreading. The benchmarks executed on the SMT processor simulator verified the functionality of the developed tool by producing correct outputs and achieving performance levels similar to those produced by the original authors.

The SMT simulator was also used to assess the use-based register cache that was designed to improve the effective register file access time in superscalar processors. The register cache provided an improvement in performance for different number of threads but the gain diminishes as the number of threads is increased. When more threads are available, the SMT processor is able to better utilize the resources and the negative impact of the longer register file access latency is reduced. Results showed that the SMT processor using a register cache with 32 registers achieves a performance similar to a register file with an access latency of one cycle.

It is recommended that the simulator be further developed to model various configurations of the SMT architecture (e.g., independent queues for threads, separate integer and floating point queues). It is also recommended that more evaluations of the register cache be made (e.g., varying register cache associativity and size, varying degree of use predictor associativity and size, varying issue width and RUU size, updating the degree of use training table at the dispatch stage).

This thesis has provided a simulator that can be used to evaluate previously proposed components designed for superscalar processors on a simultaneous multithreading environment. It will also serve as a tool for testing future designs of SMT processor elements.

REFERENCES

1. J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, Vol. 83, pp. 1609-1624, December 1995.

2. D. Wall, "Limits of instruction level parallelism." in Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

3. K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson and K. Chung, "The case for a single-chip multiprocessor," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

4. B.J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *SPIE Real Time Signal Processing IV*, pp. 241-248, 1981.

5. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *International Conference on Supercomputing*, pp. 1-6, June 1990.

6. R.H. Halstead and T. Fujita, "MASA: A multithreaded processor architecture for parallel symbolic computing," in *15th Annual International Symposium on Computer Architecture*, pp. 443-451, May 1988.

7. J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A multithreading technique targeting multiprocessors and work-stations," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 308-318, October 1994.

8. A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 104-114, May 1990.

9. R.H. Saavedra-Barrera, D.E. Culler, and T. von Eicken, "Analysis of multithreaded architectures for parallel computing," in *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 169-178, July 1990.

10. R. Thekkath and S.J. Eggers, "The effectiveness of multiple hardware contexts," in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 328-337, October 1994.

11. D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

12. D. Tullsen, S. J. Eggers, H. M. Levy, J. L. Lo, and R. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

13. S. Palacharla, N. P. Jouppi and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

14. R.P. Preston et al., "Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading," in *Proceedings of the International Solid State Circuits Conference*, January 2002

15. E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips", in Proceedings of the Eighth International Symposium on High-Performance Computer Architecture, February 2002.

16. J. Cruz, A. Gonzalez, M. Valero, and N. Topham, "Multiple-banked register file architectures," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

17. R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

18. J. A. Butts and G. S. Sohi, "Use-based register caching with decoupled indexing," *ACM SIGARCH Computer Architecture News*, Vol. 32, No. 2, March 2004.

19. J. H. Tseng and K. Asanovic, "A speculative control scheme for an energy-efficient banked register file," *IEEE Transactions on Computers*, Vol. 54, No. 6, June 2005.

20. R. Sangireddy, "Register organization for enhanced on-chip parallelism," in Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.

21. A. Gonzales, J. Gonzales, and M. Valero, "Virtual-physical registers," in Proceedings of the 4th International Symposium on High-Performance Computer Architecture, 1998.

22. O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Early register deallocation mechanisms using checkpointed register files," *IEEE Transactions on Computers*, Vol. 55, No. 9, pp. 1153-1166, September 2006.

23. S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings of the 31st International Symposium on Microarchitecture*, 1998.

24. S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

25. M. Lipasti, B. Mestan, and E. Gunadi, "Physical register inlining," in Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

26. O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure" in *Proceedings of the 37th International Symposium on Microarchitecture*, December 2004.

27. M. Kondo and H. Nakamura, "A small, fast and low-power register file by bit-partitioning", in *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, February 2005.

28. N. Kato, M. Yamato, O. Tujimoto, M. Sato, K. Sasada, K. Uchikura, M. Namiki, and H. Nakajo, "Impact of dynamic allocation of physical register banks for an SMT processor," in *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2004.

29. J.H. Tseng, K. Asanovic, "Banked register file for SMT processors," presented at the *Boston Area Architecture Workshop*, Boston, MA, January 2004.

30. H. Yang, G. Cui, and X. Yang, "Eliminating inter-thread interference in register file for SMT processors," in *Proceedings of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2005.

31. D. Burger and T. Austin, The Simplescalar Toolset, Version 2.0. Technical report, University of Wisconsin-Madison, June 1997.

32. Digital Equipment Corporation, Digital Unix Assembly Language Programmer's Guide, March 1996.

33. J. Zalamea, J. Llosa, E. Ayguade, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, December 2000.

34. R. Russell, "The Cray-1 computer system," in *Readings in Computer Architecture*, Morgan Kaufmann, pp. 40-49, 2000.

35. J. A. Butts and G. S. Sohi, "Characterizing and predicting value degree of use," in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.