AN ABSTRACT OF THE THESIS OF

Ryan L. Carlson for the degree of Masters of Science in Electrical and Computer Engineering presented on June 4, 1998. Title: A Study of Hardware/Software Multithreading.

Abstract approved: Redacted for Privacy

Ben Lee

As the design of computers advances, two important trends have surfaced: The exploitation of parallelism and the design against memory latency. Into these two new trends has come the Multithreaded Virtual Processor (MVP). Based on a standard superscalar core, the MVP is able to exploit both Instruction Level Parallelism (ILP) and, utilizing the concepts of multithreading, is able to further exploit Thread Level Parallelism (TLP) in program code. By combining both hardware and software multithreading techniques into a new hybrid model, the MVP is able to use fast hardware context switching techniques along with both hardware and software scheduling. The new hybrid creates a processor capable of exploiting long memory latency operations to increase parallelism, while introducing both minimal software overhead and hardware design changes.

This thesis will explore the MVP model and simulator and provide results that illustrate MVP's effectiveness and demonstrate its recommendation to be included in future processor designs. Additionally, the thesis will show that MVP's effectiveness is governed by four main considerations: (1) The data set size relative to the cache size, (2)

the number of hardware contexts/threads supported, (3) the amount of locality within the data sets, and (4) the amount of exploitable parallelism within the algorithms.

A Study of Hardware/Software Multithreading.

by

Ryan L. Carlson

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Masters of Science

Presented June 4, 1998
Commencement June 1999

Masters of Science thesis of Ryan L. Carlson presented on June 4, 1998

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head or Chair of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Ryan L. Carlson, Author

ACKNOWLEDGMENT

I would like to thank Prof. Ben Lee for starting me on the path of a Masters degree for helping develop the MVP model.

Also, I would like to thank my friend and fellow graduate student Hantak Kwak for his effort in collecting data and porting Pthreads to the SimpleScalar tool set. Thanks Hantak, this thesis would have taken much longer without your help.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table | Page

A STUDY OF HARDWARE/SOFTWARE MULTITHREADING

1. INTRODUCTION

In the quest for faster processing of large complex computer algorithms, many schemes have been developed over the past decade. In recent years, two important trends have been observed in computer design and manufacturing. The first trend is for computer designers to focus their attention on building processors that can exploit parallelism available both inside software programs and between totally different programs. By exploiting this parallelism, processors can now execute several instructions at once and designers have turned to two main areas of concentration: Single processors and Multiprocessors.

In the area of single processors, the concept of a superscalar processor has dominated the market. The superscalar design relies on a large content addressable memories (usually known as the issue window) to allow multiple instructions to be issued to the functional units at the same time. In this manner, as long as the issue window is large enough that parallel instructions can be found, multiple instructions can be executed at the same time and performance is greatly enhanced. The second area of focus for designers is the area of multiprocessors. Multiprocessors begin where the single processors left off, in that they get around the issue window limitation by utilizing a whole set of processors to execute instructions in parallel. The current dominating force in the multiprocessor area is the concept of shared memory multiprocessing (SMP). The SMP computers consist of multiple processors that all access the same globally shared memory. The ease of programming in the shared memory environment is what helped the SMP achieve its high market dominance. Another factor in the SMP's rise to dominance has been its ease of construction. SMPs can be made out of groups generic superscalar processors that are either clustered in the same machine using the same memory bus(Clustered SMPs) or

groups of single processor computers linked by a standardized network (such as Networks of Workstations (NOWs)).

However, though both single and multiprocessor systems can execute several instructions in parallel, their performance has been limited by long memory latencies: the second trend of computer designers. Memory latencies result when an either instruction or data is requested from the memory hierarchy. Since, memory is slow in comparison to processors speeds, the processor must wait until the requested information is retrieved. Typically, the processor can execute few or no instructions at this point and performance suffers. To make matters worse, studies have shown that memory latencies are a problem that is getting worse not better. In fact, that while the speed of commercial microprocessors has increased by a factor of twelve over the past ten years, the speeds of memory have only doubled [1]. Unless a radical breakthrough in memory technology is made, the future indicates that the ratio of memory speed to processor speed will only increase even further. Multiprocessors will continue to bear the brunt of this problem as not only do they have the same latencies as the commercial processors that they utilize, but additional memory latency is gained from the interconnections of the processors. Single computer SMPs force the multiple processors to share a single memory bus that leads from the lowest cache to shared memory. Thus, a processor will have to wait until another processor finishes accessing the main memory. For SMPs constructed with computer networks, such as NOWs, the effects are potentially even worse. If a value does not exist within the computers local memory, the computer must send a request for the data across a comparatively slow network What all these memory latencies mean for processors is that, instead of doing useful computations, the processor must wait for the information it requires. Indeed, memory latencies are the current, most limiting factor on processor performance.

Another problem faced by both single and some multiprocessors is the amount of parallelism within a single program. This instruction level parallelism (ILP), is finite and

without creating multiple "loci" of control, or threads, there is only so much performance improvement to be gained from a single program. For example, a typical superscalar processor using an issue window of 16 instructions will receive a 100% performance improvement by increasing the window to 32, 33% improvement with a window of 64, 7% with 128, and <1% with 256.

One of the most promising techniques for tolerating the memory latencies and moving around the problem of finite ILP, is the idea of multithreading. The idea of multithreading is based around the fact that algorithms maintain various portions that are either completely data independent or mostly data independent and can therefore be executed in parallel. These parallel portions of program code can therefore be treated like they are different programs, and become the systems threads. There are many ways that a processor can switch between threads, but there exist two methods that remain the most common. First, when the current thread encounters a long latency operation, the processor can switch control to a new thread. Thus, processor execution can continue while waiting for the long latency operation. The long latency operation can be anything from a long memory latency to a long I/O latency. The second method is to constantly interleave instructions from the threads (known as Simultaneous Multithreading or SMT). Thus, when the instructions from one thread are stalled, instructions from others can continue to move through the processor.

The Multithreaded Virtual Processor (MVP) was developed to test a new strategy for multithreading. The goal of the MVP is to introduce a processor capable of multithreading, but does not require extensive, and thus costly and time consuming, modifications to a current superscalar processor design. To keep this goal, the MVP exploits a hybrid model that combines both software and hardware techniques. The hybrid is, essentially, that software libraries were used to create and manage threads. However, since software by itself cannot detect long memory latencies, the software is supported by minimal hardware modifications.

This thesis shows that by supporting increasing more powerful software tools with minimal hardware modifications, full advantage of multithreading can be realized. Additionally, as more die area becomes available to future processors, this thesis also shows that space should be devoted to supporting multithreading.

In this thesis the MVP model will be explained as well as the tools and modifications thereof that were used to create it. Also, the results of various simulations on benchmark programs will be displayed and explained in detail. The thesis will show that the performance of the MVP is influenced by four main considerations: (1) the data set size relative to the cache size, (2) the number of hardware contexts/threads supported, (3) the amount of locality within the data sets, and (4) the amount of exploitable parallelism within the algorithms. Further, a detailed look of the new bottlenecks imposed by multithreading and potential modifications to modern processors to relieve them is also presented. Finally, the study will show that overall, multithreading is worth the investment of providing additional hardware support.

2. PREVIOUS RESEARCH

The consistently increasing gap between processor speeds and memory speeds has prompted much research into this field for the last ten years. Most research can be classified into two main categories: research for architecture designs to reduce memory latency and research for architecture designs that can tolerate memory latency.

In order to reduce memory latency, several fetch unit and cache modifications have been proposed. Recently, the concept of prefetching has headed the research interest. The new prefetching techniques include simple next-line prefetching to the newer and more complex wrong-path prefetching [2] methods. To take advantage on the fact that most algorithms perform many loop iterations, Trace Caches [3] were created to store an instruction trace of the loop and prevent the re-fetch of the instructions from main memory. To help with the prefetching of data values, Stream Buffers [4] were developed. The stream buffer could prefetch not only next-line data values, but could be expanded to prefetch constant stride data accesses. To help increase the associativity of low level caches, Victim and Second-chance caches were explored. By providing additional storage in the L1 cache, these caches caused less requests to main memory to be made and latency was reduced. Unfortunately, though latency reduction helps, it does not eliminate the memory latency problem. Clearly, latency reduction must be combined with some form of tolerating the remaining latency.

Latency tolerance has led to the development of most of the newest processor designs currently used by industry. Indeed, modern superscalar processors with their multiple out-of-order issue strategies, were designed specifically to tolerate the growing memory latencies. However, with size limitations on the number of instructions issued, superscalar by itself can not cope with the latency, nor can it get by the finite instruction level parallelism (ILP) problem. In order to take more advantage of the long memory latencies and parallelism within threads, research has turned to multithreading.

The concept of multithreading for hiding long latency operations is not a new one. The original dataflow models of computation already implied the general ideas of fine-grain multithreading [5]. Multithreading began strictly as a programming paradigm dedicated to increasing throughput on SMPs [6]. Powerful software packages, some backed even by Operating System (OS) kernel support, have been created to help programmers take advantage of algorithm parallelism. However, these exclusively software packages, such as Pthreads [7], Solaris Threads [8], and Linux Threads, are unable to take advantage of actual hardware conditions. Thus, in no way can software controlled multithreading detect and act upon a cache miss and thus, effect memory latency in this manner. Instead, the software packages are forced to rely on time quantums to switch between threads or to interleave thread execution. The problem with time quantums is that they are not precise enough. For example, a processor could interrupt a non-stalled thread or let a stalled thread remain in the hardware for a time. Interleaving threads causes its own problem of jamming a processor when one thread stalls and the look-ahead of the processor's issue unit is not enough to move other thread instructions past the blockage. This problem is especially true if more than one thread becomes stalled at the same time. Additionally, when software packages switch between threads, the operation creates high software overhead. Thus, purely software controlled multithreading is not a good solution.

In an attempt to solve the limitations imposed by the software packages, multithreading researches began studying hardware modifications. To improve the time required to switch between threads, the concept of multiple hardware contexts was employed. These hardware contexts were implemented as both logical and physical register files and Program Status Windows (PSWs) and can be found on such systems as HEP [8] and TERA [6]. However, these systems require extensive modifications to the processor architecture and enforce the use of very specialized processors. Thus, both architectures move against the trend to use generic components and are neither cost effective nor feasible.

There are also two attempts to add hardware support to existing processor architectures. The first, is the MIT Alewife processor that utilizes a SPARC based architecture called Sparcle [10]. However, Sparcle is based on an old superpipeline design and does not indicate how multithreading would perform on a modern superscalar design. The second attempt was made by Nader Bagherzadeh [13]. Bagherzadeh's processor is based on a modern superscalar design, however, it enforces some limitations. The first one is that the threads are limited by size and by number. The fine grain limitation means that potentially less benefit can be gained, since the processor can run out of instructions to execute and really long latencies can not be fully exploited. The number limitation dictates that only a certain number of threads can exist at the same time within a processor. Thus, even if more parallelism exists to be exploited, the programmer must follow strict guidelines. The second limitation is that thread execution is interleaved and runs into the same problem as the software packages that utilize this SMT strategy, the processor must have a very high look-ahead.

The MVP surpasses the previous attempts in a number of ways. Mainly, multithreading is added to a modern architecture with both software and hardware support. The hardware support allows the use of multiple hardware contexts and to detect cache misses to hide memory latency. The software support allows an almost limitless number of threads to be maintained and drastically limits the amount of hardware modifications. Additionally, the MVP does not utilize interleaved thread execution, but instead switches between threads. This addition enables the MVP to be constructed without the necessity of giant reorder buffers or reservation stations. However, it is important to note that though some software overhead for context switching is introduced, it is less than that of a purely software controlled multithreading package. In essence, the MVP is a new blending of both hardware and software multithreading.

3. THE MULTITHREADED VIRTUAL PROCESSOR MODEL

As stated previously, the purpose of the MVP is to enable a programmer to use a standard software controlled multithreading package. At the same time, the package is transparently extended to take advantage of hardware support for tolerating long memory latencies. This chapter will describe the MVP and the thread execution model that MVP uses to govern the execution of running threads. Also, the chapter will describe the simple prefetching schemes implemented in the MVP to show what effect prefetching can have on the performance bottlenecks of the processor.

3.1 MVP Organization

The organization of the MVP is shown in Figure 1. The processor consists of a standard superscalar processor core based on Sohi's register update unit (RUU) [12]. The RUU acts as a combination reservation station (RS)/re-order buffer (ROB). In addition to the standard superscalar design, multiple register files/contexts and a hardware scheduler have been added. The hardware scheduler has the responsibility to manage all threads that have been scheduled into the hardware for execution by the software thread package.

Each register file, including some additional control bits, represent a thread context. An example of a thread context is shown in Figure 2. Each context consists of a full register file, a valid bit, and a ready bit. The valid bit is used to indicate whether a context contains an executable thread (i.e. a valid PC and register data). If the valid bit is not set the hardware scheduler will not switch execution to that context. The ready bit is used to dictate whether a context has had its memory latency resolved yet. As the hardware switches to a new context in response to a cache miss generated by the current context, the current context's ready bit is cleared. The ready bit remains cleared until the cache miss is

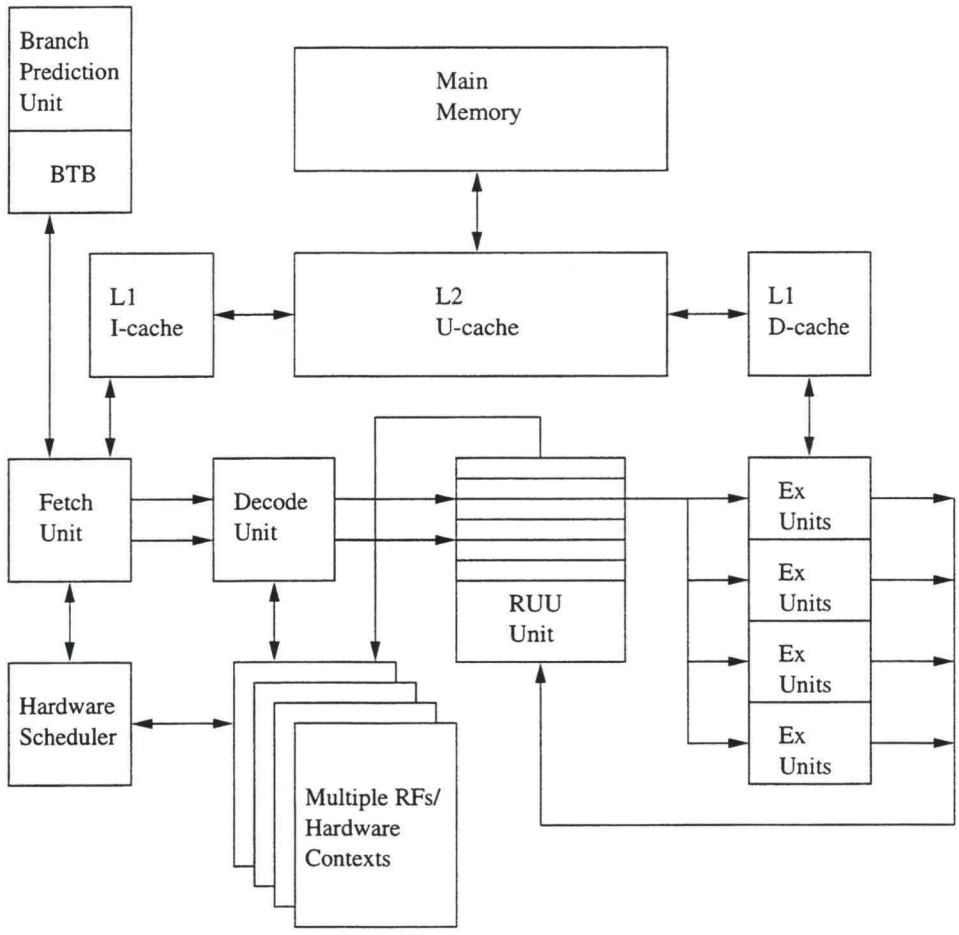

Figure 1: Hardware Organization of the MVP.

resolved and is set when the data becomes available to the thread. The hardware scheduler will not return execution to the non-ready context until the ready bit is set.

In addition to the thread contexts, two global bits are also added to the hardware: Thread_Mode and More_Threads. Thread_Mode is used to designate when the MVP is to be allowed to start executing across multiple hardware contexts. The bit, controlled primarily by the software package, will prevent the hardware scheduler from utilizing more than one context until it is set. This bit is used to prevent a context switch from occurring before any threads have been created by the software. The bit additionally provides a simplistic locking mechanism for preventing the hardware scheduler from generating a

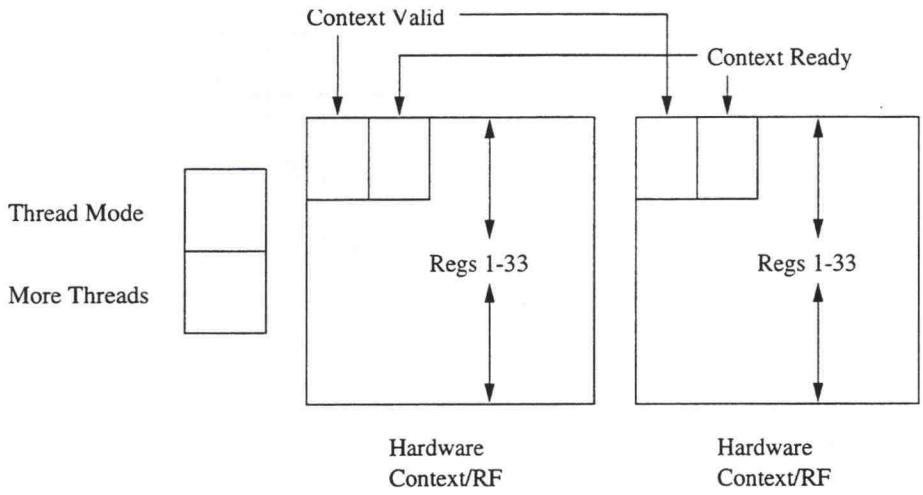

Figure 2: A Hardware Thread Context.

context switch during potentially hazardous times. An example would be when the software scheduling algorithm is being executed in one of the hardware contexts. If a context switch occurs during execution of the software scheduler, a potential race condition could develop when another context also executes the software scheduler algorithm.

The other global bit, More_Threads, is also controlled by the software thread library. This bit indicates whether the software scheduler has more threads to execute on the hardware. If the bit is set, the hardware scheduler is informed that more threads are available for execution and knows it has to call the software scheduler to execute more threads.

3.2 Multithreading Model

The MVP consists of two main components for dealing with multithreading: software and hardware. The software is responsible for creating, managing, and scheduling threads on the hardware. In this case, the POSIX compliant Pthreads package was used. After threads have been scheduled onto the MVP contexts, the hardware

scheduler takes over. The hardware scheduler then switches execution to a different context whenever the current context encounters a long memory latency. In the case of this study, the context is switched upon a detection of a second level cache miss by the memory-management unit (MMU). It is important to note that, though in MVP the second level caches are unified (shown by Figure 1), a context switch is generated only by a second level data cache miss.

The hardware scheduler maintains control over the threads until a thread finishes execution and returns. As a thread returns, control is relinquished to the Pthreads scheduler. The MVP execution model that governs the interactions between the user program, Pthreads library calls, Pthreads scheduler, and the hardware scheduler is shown in Figure 3. The functionality of each of the states is described below:

- main() - While in main(), the MVP is executing the main user program in non-thread mode. When a Pthreads routine is invoked, the model transitions to the Pthreads Library Calls state. For all programs this is the first and last state encountered by the MVP as execution begins in this state and returns once all threads have been executed.
- Pthreads Library Calls - In this state, the MVP is executing a Pthreads routine. When the routine returns, execution will either return to main() or attempt to schedule a thread for execution (Schedule New Thread state).
- Schedule New Thread - This state's action depends upon which transition and/or which Pthreads routine called the Pthreads Scheduler. Regardless, the state will perform either one of the following actions: It either
    * Selects the next thread from the list of threads ready to be scheduled (known as the priority queue), and compares the thread priority with the priority of the currently executing thread. If the currently executing thread's priority is higher, the scheduler returns, if not, the current thread is removed from the

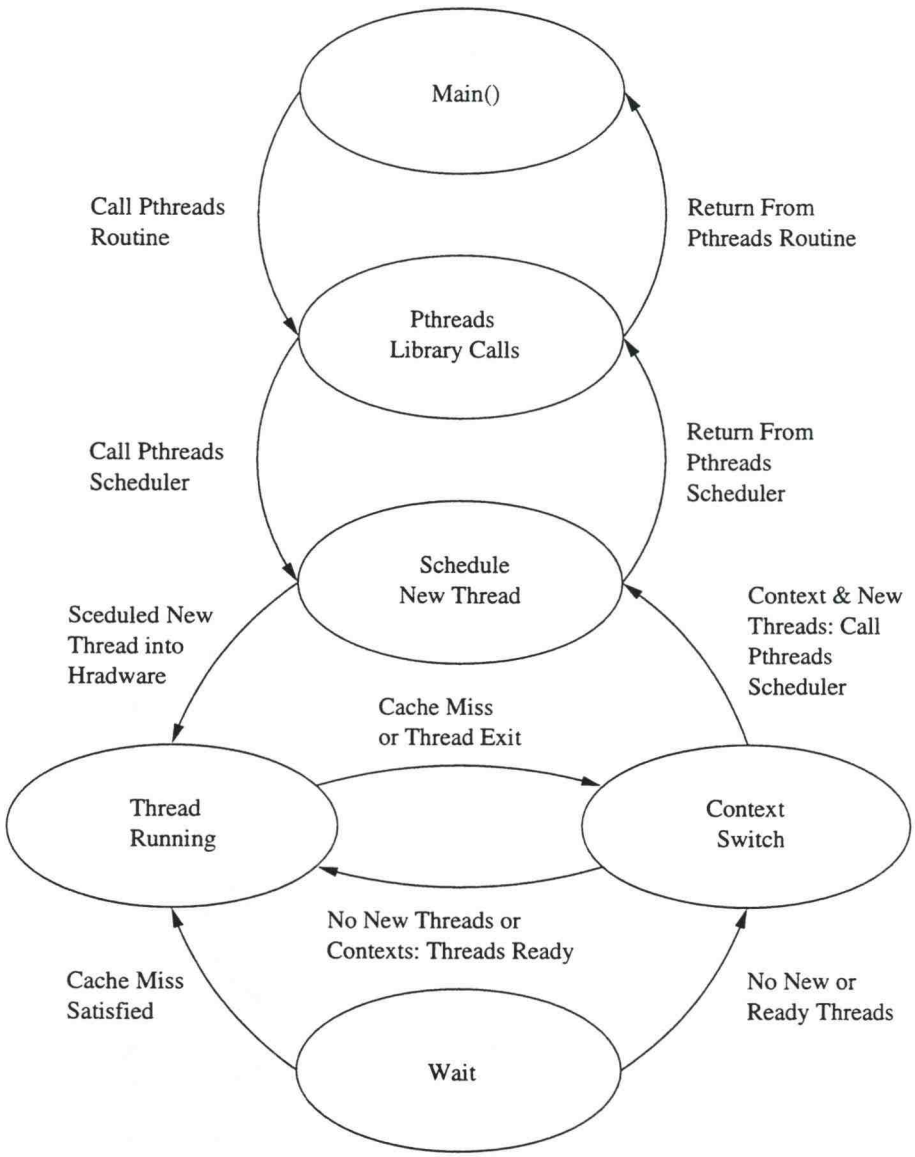

Figure 3: The MVP Execution Model.

hardware and the new thread scheduled. The priority queue (PQ) is maintained by the Pthreads scheduler exclusively.

* Chooses the next thread from the PQ and schedules the thread into the current hardware context. Since all threads are by default equal in priority, this is the most commonly observed action of the state.

- Thread Running - This state signifies that a thread is executing in the current hardware context. Execution leaves this state and proceeds to the Context Switch state when a cache miss or context switch instruction is encountered.
- Context Switch - The Context Switch state performs one of the following actions depending on the current machine state: It either
  * Changes the current hardware context to another valid and ready context (Thread Running transition).
  * Switches control to the Pthreads scheduler if a non-valid hardware context exists and more threads are available for execution. (Schedule New Thread transition).
  * Halts execution and waits if no more threads are available and no contexts are ready (Wait transition).
- Wait - Wait ceases all execution of the MVP and waits for a context to become ready (i.e., a thread to receive its data from a long memory latency operation). When a context is again ready, Wait makes the transition back to Thread Running.

In order to implement the MVP execution model, two main design choices were encountered. First, instead of gang scheduling threads into all of the hardware contexts at once, the decision was made to schedule threads one at a time. The reason for this decision was that scheduling one thread at a time was closest to the current Pthreads operation and so required less modification to the Pthreads scheduling code. Also, gang scheduling the threads would not necessary result in improved performance at the cost of increased hardware complexity. Basically, the choice was a tradeoff between one long scheduling operation and multiple shorter scheduling operations.

Second, it was decided that when a thread finishes its execution it would be better to schedule immediately a new thread to the context (if available), rather than switching execution to a thread already located in another hardware context. This decision was based

on results obtained from preliminary studies [13] that indicated it is always better to keep as many hardware contexts occupied as possible.

3.3 Prefetching Model

It was decided to add prefetching to the MVP primarily for two reasons. First, since most modern microprocessors are now implementing some form of instruction cache prefetching, it is important to study the possible benefits/hindrances that these algorithms pose to the MVP. Second, preliminary research suggested that the MVP suffered from an increased bottleneck in the fetch unit and prefetching was to be explored as a possible solution. In this study, two very simple algorithms were implemented on the MVP. The first was a simple next-line prefetching scheme. In this scheme, when a line is fetched from the I-cache, the next line in the I-cache is also fetched. The next-line prefetching always occurs regardless of the instructions located in the previous line. For instance, even if the previous line contains a predicted taken branch instruction the next I-cache line would still have been prefetched. As this prefetching scheme can be implemented with standard microprocessors, it was used as a base line to compare with MVP running a similar scheme. However, in MVP, the prefetching occurs for only the currently executing hardware context. In addition to the simple next-line prefetching, another prefetching scheme was created exclusively for MVP. The second scheme expanded the next-line algorithm to include prefetches for the next-thread as well. Now, instead of prefetching I-cache lines for the current context, once a cache miss is detected, the next line for the next current context is prefetched. Thus, when a context switch occurs the new context's instructions are either available or already in the process of being fetched into the I-cache and latency is reduced.

An important note about prefetching is that while it may be a benefit to a single processor, multiprocessor systems may be disadvantaged by prefetching. The reason is

that prefetching is a memory latency/bandwidth tradeoff. While single processors may have memory bandwidth to spare, the same is not true of multiprocessor systems. Most notably, clustered SMP machines. In these machines several processors share the same memory bus and usually all memory bandwidth is used with little or none left over for use by prefetching schemes.

4. TOOLS

This chapter will focus on the two main tools used in the creation of the MVP. In order to create and manage threads the Pthreads package was used. On the hardware side, Todd Austin's SimpleScalar simulation tool set [14] was adapted to simulate the MVP and all associated memory hierarchy and hardware scheduler. First, the chapter will focus on the software side governed by Pthreads.

4.1 MIT Pthreads

In order to create and manage threads for the MVP, the Pthreads package was selected. Though several different implementations of Pthreads exist, Chris Provenzano's Pthreads was chosen. The advantage of Provenzano's Pthreads was primarily that it was freely available and that it was implemented as a collection of libraries.

The Pthreads scheduler maintains three distinct algorithms: first-in-first-out (FIFO), round-robin, and user-controlled priority. The main difference between FIFO and round-robin is that round-robin scheduling associates a time quantum for each thread. On expiration of the quantum, the scheduler will schedule a new thread into the hardware. The round-robin algorithm was chosen as it exhibited the closest behavior to the MVP and required little modifications to fit the MVP's needs.

To use Pthreads, a programmer creates and manages the threads by making a series of function calls. All threads to be executed must be explicitly defined and coded by the user. To begin, the user makes a call to create the threads to be executed. Then, to run the threads, the user makes another function call from the main program. Each thread is executed on the hardware for one time quantum. Once the quantum expires, an interrupt is generated calling the Pthreads scheduler and another thread is moved to the hardware. The time quantum is set small enough to help performance by avoiding long I/O operations.

When the last thread finishes its execution, the Pthreads library returns control back to the user program.

Though Pthreads library maintains many functions for thread management, for simplicity, only four functions of the library were used for the simulations:

- pthread_create - This functions creates a new thread. If the newly created thread has a higher priority than the user program (default = lower priority), then pthread_create can call the Pthreads scheduler and begin execution on the new thread.
- pthread_exit - When called, pthread_exit will clean-up the completed thread and call the Pthreads scheduler to either begin execution on a new thread or return execution back to the main user program if no additional threads remain to be executed.
- pthread_join - This complicated function acts as a barrier synchronization between threads and the main user program. It is typically called from the main user program and given a thread as an argument. The function will check to see if the given thread has finished execution, if not, the function will call the scheduler to execute the thread. The main user program will not pass this function call until the requested thread has finished execution. Thus, if pthread_join is used in a loop and given all created threads as arguments, execution will not proceed past the function call until all threads have finished execution.
- barrier_wait - The barrier_wait function enforces a simple barrier synchronization between threads. Just like a standard barrier synchronization, thread execution is paused at this point until a certain number of threads call the barrier_wait function.

An important notice is how Pthreads manages and stores its threads and their data. Pthreads keeps track of the threads by assigning each thread a unique thread ID. The list of thread IDs is kept in the PQ. When a thread is executed on the hardware, it is removed

from the PQ and placed in a variable called pthread_run. Thus, Pthreads can keep track of all available threads and which thread is currently executing. To store the data associated with each thread, a collection of various data registers and pointers, Pthreads uses the OS stack. When threads are created, space is allocated on the stack for each thread and used to store all data associated with the thread in the advent of a interrupt or a swap of a currently running thread for another thread in the PQ. This fact is important to realize as it necessitates the existence of an OS stack in order for the proper function of Pthreads.

4.2 SimpleScalar

The SimpleScalar tool set was created by Todd Austin. The SimpleScalar architecture is derived from a combination of the MIPS-IV ISA and DLX. Essentially, the DLX adds additional addressing modes and a square-root instruction to the ISA. Also, the ISA was expanded to a 64-bit instruction encoding to provide users the option to expand the ISA to suit their own needs. The actual tool set contains various simulators ranging from a simple functional simulator to an advanced out-of-order microarchitectural simulator.

The simulator base used to simulate the MVP was sim-outorder, which simulates as n-way issue, five stage superscalar processor based upon Sohi's RUU shown in Figure 4. In order to fully understand the limitations of sim-outorder it is important to first understand how sim-outorder is both built and operates.

Sim-outorder is based upon a very simple functional simulator developed first, known as sim-safe. The functional simulator simply executes instructions in order with no instruction passing allowed. No architecture is simulated either, causing an instruction to be fetched from memory (no caches), decoded, and then executed before another instruction is fetched. The decode/execute step actually occurs at the same time, so as an instruction is decoded, results are calculated, registers and memory updated, and branches

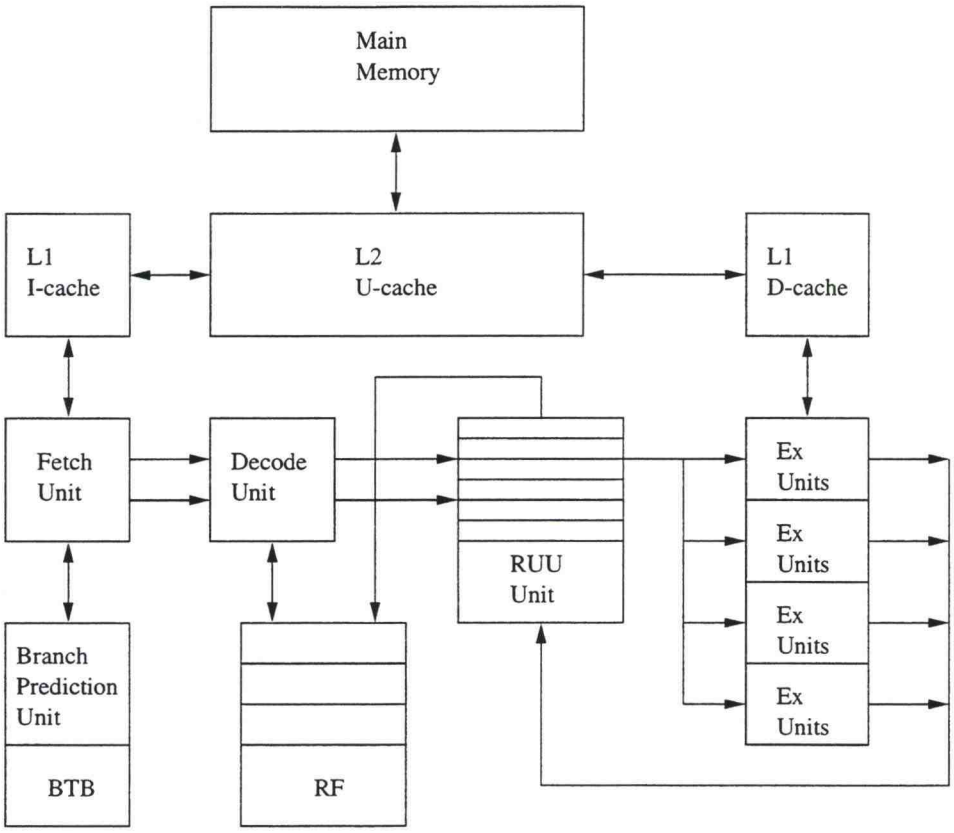

Figure 4: The SimpleScalar Hardware Organization.

calculated and taken. When sim-outorder was built on this core, the core was never modified. Thus, all results and memory modifications still take place in the decode step.

The problem is that with sim-outorder, once an instruction is 'decoded', a copy is sent into a microarchitectural simulator. The microarchitectural part of sim-outorder then moves the already executed instruction through the RUU based pipeline. In essence, the instruction is issued, dispatched, executed, written back, and retired. Also, sim-outorder adds multi-level caches to help compute accurate timing. However, since the instruction has already been executed by the functional simulator, no data is moved/calculated by the microarchitectural simulator, only timing information is collected. Another way to view the interaction between the functional and microarchitectural simulators in sim-outorder would

be to think of the functional simulator as an on-the-fly trace generator that passes each instruction to a trace driven simulator.

To maintain an accurate simulation of wrong path execution (i.e., branch miss-speculation), sim-outorder introduces the concept of spec_mode. When a conditional branch instruction is decoded, sim-outorder compares the actual branch (computed by the functional simulator) to the predicted branch (computed by the microarchitectural simulator). If a match is not made, then the branch is miss-predicted and sim-outorder enters speculation mode (spec_mode). In order that the register file and main memory are not polluted by the functional simulator with 'speculative' instructions, spec_mode forces the functional simulator to access a different 'speculative' memory and register file structures. It is important to note that the speculative memory and register file data structure values are not in any way related to the correct memory and data structure values. This difference results in a big limitation of sim-outorder in that, if execution remains in spec_mode long enough, the bogus data values will result in a fault and halt execution of sim-outorder.

Another limitation of sim-outorder's construction is with precise interrupts. Since the functional simulator executes each instruction the instant it is decoded, the machine state is at the point of decoding an instruction and not at the point of committing an instruction as in all actual superscalar processors. Thus, any interrupt that is generated in the microarchitectural simulator can not result in a precise interrupt.

As a direct consequence of a lack of precise interrupts, sim-outorder enforces another limitation in the form of no OS support. In order to model syscalls used by most benchmark programs, sim-outorder uses the scheme shown in Figure 5. With this model, when a syscall is decoded from the simulated program, sim-outorder makes the same syscall to the native OS on the machine executing the simulator. Though this model works with most syscalls, it will not work for signals and timers. In example, consider that a user program wants to set a timer for one second. A problem occurs when simulating this code,

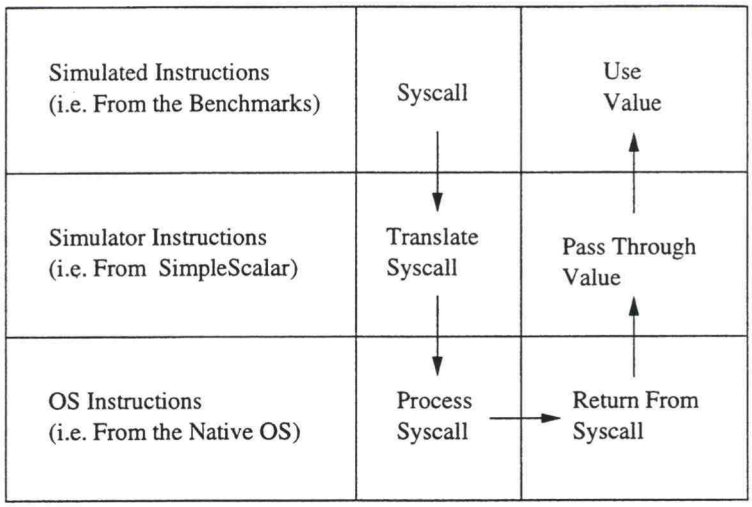

Figure 5: Sim-outorder Syscall Handling.

as sim-outorder sends the timer value outside the simulator to the native OS. The native OS will wait one second and then interrupt sim-outorder which in turn sends the equivalent interrupt to the simulated user program. However, though the native OS waited one second, the simulated time will actually be much less. This problem occurs since now the processor is executing instructions from both the benchmark and the simulator instead of instructions solely from the benchmark program. So instead of the user program waiting the one second it wanted, it could have only waited a simulated time of only .1 seconds.

The five stage pipeline used by sim-outorder consists of the following stages: fetch, decode, issue, writeback, and commit. These stages are implemented by six functions called by the main driver function of sim-outorder: sim_main(). In order to easily move instructions through the pipeline without complex software, the stages/functions are called in reverse order once per clock cycle. The stage functions include:

- ruu_fetch - The fetch stage is responsible for fetching instruction out of main memory and placing them it the fetch queue. The fetch continues until either the fetch queue is full, a branch instruction is reached, or an I-cache miss occurs.

This function also updates whatever I-caches and branch prediction tables exist. In example, the MVP uses a branch target buffer (BTB) along with two branch prediction bits.

- ruu_dispatch - The ruu_dispatch stage removes instructions from the fetch queue, decodes them, and places the new instructions into the RUU. This stage is where sim-outorder makes the translation from the functional to the microarchitectural simulator. As long as instructions exist in the fetch queue, the decode bandwidth is not exceeded, and space is available in the RUU, the actual sequence of events is:
  * Instructions are removed from the fetch queue and decoded.
  * The instructions are executed and their results are stored in the register file and/or main memory.
  * If the instruction is a miss-predicted branch, spec_mode is enabled.
  * The instruction is returned and inserted into the RUU. It is at this point that the instruction enters the microarchitectural simulator.
  * If more instructions exist in the fetch queue, the decode bandwidth has not been exceeded, and space is still available in the RUU, decode another instruction.

* As can be deduced from the sequence of events the functional simulator is strictly an in-order simulator.

- ruu_issue - At this stage ready instructions are issued from the RUU into the functional units. All ready instructions can be issued out-of-order as long as the issue bandwidth is not exceeded and functional units remain open. Loads are also executed in this stage and cache addresses and dirty/valid bits are updated. Note, sim-outorder assumes a simple MMU that only allows loads before any non-resolved store to be executed.

- ruu_writeback - This stage is where interactions are 'completed' in so far as the microarchitectural simulator. In this stage, instructions that are finished with their execution units, have their results written back to the RUU to update the data dependencies. Also, miss-predicted branches are executed and, if necessary, execution is reset to before spec_mode was initialized.
- ruu_release_fu - This small function does not correspond directly to a pipeline stage, but does release the functional units once instructions have finished executing.
- ruu_commit - The commit function removes completed instructions from the RUU and executes any store instructions located at the head of the RUU. Note, instructions have already modified the register and memory in the decode stage, so no values are actually stored in the register files or main memory. A store instruction will, however, result in changes to the cache hierarchy.

5. BUILDING THE MULTITHREADED VIRTUAL PROCESSOR SIMULATOR

In order to study the MVP a simulator was developed (MVPsim) that successfully integrated the SimpleScalar tool set with the Pthreads thread package. This chapter focuses on the changes to both Pthreads and SimpleScalar that were necessary in order for the integration into MVPsim to take place.

5.1 Modifications to Pthreads

As designed, the modifications to the Pthreads library were not lengthy to implement. The modifications were made in three main areas of the Pthreads library: (1) thread scheduling, (2) thread synchronization, and (3) atomic locking.

First, Pthreads was not originally designed to manage multiple threads running at the same time. Previously, Pthreads would keep track of which thread is executing by placing a pointer to the specific thread into the pthread_run variable and removing the pointer from the PQ. When a context switch occurs during use of multiple hardware contexts, Pthreads has no way of knowing which thread is in the current context. Therefore, Pthreads was changed such that whenever the hardware scheduler calls the Pthreads scheduler, the thread pointer is copied from the hardware context into the pthread_run variable. Using this strategy the Pthreads scheduler always has a pointer to the currently executing thread. Additionally, the scheduler had to be modified to support the new More_Threads bit included in the MVP. As explained in section 3.2, the More_Threads bit must reflect whether or not there remain threads in the PQ. The Pthreads scheduler was modified to set the More_Threads bit to zero whenever the status of the PQ changes from filled to empty. Vice versa, the Pthreads scheduler also sets the More_Threads bit to one when the PQ's status changes from empty to filled.

The second and main modification came from Pthreads thread synchronization requirements. Originally, when a thread encounters a barrier synchronization primitive, because that thread can not execute until the barrier is satisfied, Pthreads removed that thread from the hardware and placed it into a wait queue. Next, Pthreads removes another thread out of the PQ and places the new thread into the hardware. Once the wait queue contains all the necessary threads to satisfy the barrier, Pthreads moves all the threads back to the PQ and execution of the threads continues. In the MVP model, a different scheme must be used. When one thread reaches a barrier synchronization point, the other threads in the hardware context will normally also have that same synchronization point. Thus, instead of scheduling a new thread, the MVP needs to switch contexts. In order to easily facilitate the change in Pthreads, the following novel scheme was developed.

A specific group of threads, known as context-switch threads, were designed to first cause a context switch and next, to yield execution to any other threads in the PQ. In other words, a barrier synchronization now proceeds through the following steps:

- A thread reaches a barrier synchronization point and calls the appropriate Pthreads routine. Pthreads removes the thread from the hardware context and places it in the wait queue. Next, Pthreads places a context-switch thread into current hardware context and executes it.
- The context-switch thread initiates a context switch so that the other threads in the hardware contexts have a chance to reach the barrier synchronization point.
- When execution again returns to the context-switch thread, it will immediately yield execution to any thread in the PQ. This yield is performed to either satisfy the barrier synchronization or to continue execution if the barrier has already been satisfied.

Unfortunately, though this scheme resulted in the smallest and easiest change to Pthreads, it did cause the greatest Pthreads overhead during the simulation of the benchmarks (from context switching the extra tthreads).

The third modification to Pthreads was to provide support for an atomic locking mechanism. Pthreads was never meant to run on a multiprocessor system let alone the MVP. The result is that, were MVP to context switch out of a context currently executing Pthreads library instructions, a possible race condition could develop if the next context begins executing the same Pthreads code. The flimsy locking mechanism provided in Pthreads was not even atomic and could not support use by the MVP. To fix the problem of atomic support, the Thread_Mode bit (explained in section 3.1) was added to the hardware and support was added to the Pthreads library. In short, whenever the MVP can switch hardware contexts, the Thread_Mode bit is set to one. Pthreads was modified to set the Thread_Mode bit to zero when interrupting (calling the Pthreads scheduler) and upon a thread reaching a barrier synchronization. In response, Pthreads was modified to set the thread-mode bit to one when: starting to execute threads, returning from an interrupt (beginning execution on a newly scheduled thread), and returning from a call to the barrier synchronization primitives.

As explained, the modifications to Pthreads, while not simple, were kept to a minimum as much as possible. This philosophy was maintained in order to provide an easy transition to a new hardware with little modification to existing software. The philosophy also had the result of requiring the greatest modification to the SimpleScalar simulation tool set.

5.2 Modifications to SimpleScalar

In order to support both the MVP and Pthreads, the SimpleScalar tool set required extensive modifications. First, in order to support Pthreads, SimpleScalar has to provide an OS stack and precise interrupts to save thread information whenever a thread is not in the hardware. To provide the MVP functionality, SimpleScalar had to be further modified to support multiple hardware contexts along with added registers, instructions, and to

generate interrupts upon observing a cache miss. The final modification to SimpleScalar was to provide both next-line and next-thread prefetching techniques to SimpleScalar's fetch unit.

To add an OS stack into SimpleScalar, a simple approach was used to decrease the time and extent of modification required. Since Pthreads requires only the presence of an OS stack for processing interrupts, the OS stack was implemented as an interrupt handling routine. First, a function (interrupt_fetch) was created to handle all interrupts that were needed to support. For use with the MVP, only interrupts generated by the cache miss were implemented while the serial versions used a timer (SIGVTALARM). Then, whenever the appropriate interrupt was generated, SimpleScalar would call the interrupt_fetch routine. The routine (found in signal.c), serves to map the requested interrupt in SimpleScalar to a selected spot in simulated memory. Essentially, memory mapping the interrupt to the predefined interrupt handling routine. The interrupt handling routine (sigisr.S) is included by Pthreads and serves, when accessed, to create the OS stack. The OS stack is implemented by storing all important registers of the current MVP context to memory and increments a pointer to the next free location. If a thread needs to be removed from the OS stack, the pointer is decremented and the information loaded back into the current hardware context. In order to keep from modifying Pthreads, the interrupt that is used for the MVP simulator was SIGVTALARM. Since Pthreads already used this interrupt for its own original time quantum interrupt, no modification had to be made to Pthreads' own interrupt handling routines.

To provide functionality for handling precise interrupts, an extension of SimpleScalar's speculative execution mode was made. As explained in section 4.2, when SimpleScalar enters spec mode, all accesses to main memory or the register file become routed to false locations. To use this system for precise interrupts, an extra field was added to SimpleScalar instructions to alert the simulator that this was the instruction that generated an interrupt. In the case of the MVP, it would alert the simulator that this instruction

caused a cache miss (a faulting load instruction). To get a precise state in the SimpleScalar simulator between the functional and microarchitectural components, the instant an instruction interrupts, the simulator enters a new memory spec mode. The new memory spec mode is the same as spec mode, but it will not allow the processor to leave spec mode until the faulting instruction has been dealt with. Thus, the functional simulator can no longer modify the register file or main memory. Once the instruction reaches the head of the RUU, the two parts of the simulator are equal in state, the simulator can leave spec mode, and execution restarted to the interrupt handling routines. All modifications to the main functions of SimpleScalar are listed below:

- ruu_commit - Changed to detect a faulting load instruction and, if so, to leave spec mode and call the interrupt handler. This function is the only place in the simulator when the simulator can leave memory spec mode.
- ruu_writeback - This function received minor changes to prevent a faulting load from being mistaken for a miss-predicted branch instruction. Thus, preventing statistics from becoming polluted. Also, disallow a miss-predicted branch from leaving spec mode when a faulting load has occurred (i.e. the simulator can't leave memory spec mode until ruu_commit).
- ruu_issue - Since normally SimpleScalar assigns the whole time a faulting load is stalled (until the load has its data), this function had to be modified. To reflect how the MVP will work, only the time its takes to discover that the L2 cache will miss needs to be assigned to the faulting load.
- ruu_decode - This function had to be changed to, if cache miss interrupts are enabled, check all loads (using the cache_probe function) to see if the loads will miss in the L2 cache. If yes, then the function forces the simulator into memory spec mode.

To help create the support for the MVP, the SimpleScalar ISA also had to be modified. As explained in section 5.1, the context-switch thread requires a context switch

instruction. Also, another register had to be added (reg 33) to take care of the More_Threads and Thread_Mode bits. Additionally, another bit was required to be set in response to a request for a context switch. To this extent, the SimpleScalar compiler was modified to allow the use of register 33. Also, the decode file (ss.def) had to be modified to detect the new instruction. For the sake of simplicity, the new instruction was entered in asm code to eliminate the need for further modifying the compiler.

In order to provide multiple hardware contexts, a simple solution of an array was created. Instead of SimpleScalar interfacing with a single register file (regs_R), SimpleScalar now utilizes an array of register files, each one representing a MVP hardware context (HWCT). The index into the array determines which context is the currently executing context. To provide a method to switch between the contexts and to interface Pthreads to SimpleScalar, a new function was added to SimpleScalar called hw_cswitch. The function's primary duty is to perform as the hardware scheduler. When called, hw_cswitch returns the PC for the location that the simulated processor is to proceed next. The function is called by SimpleScalar in two different locations. First, the function is called by ruu_decode when the decode unit detects that a context switch instruction has been processed. The next time hw_cswitch must be called is by the commit stage (ruu_commit). The commit stage must call hw_cswitch when an instruction that caused a cache miss is ready to retire, again at that point the state can be saved and the interrupt processed. Each time hw_cswitch is called, the current PC is sent as an argument and the next PC is returned. The functionality that hw_cswitch supports is in full accordance with how the hardware scheduler is defined to operate by the MVP model described in Chapter 3.

1. The hw_cswitch function checks to see if the context switch was requested by instruction or an actual cache miss and thread mode is on. If it was from an instruction, then the bit is reset (MVP_CSW_REQ) so as not to cause another

context switch, and the context switch proceeds. If thread mode is not enabled, then the current PC is returned.

2. Since a context switch is going to occur, hw_cswitch copies the current context into the appropriate place in the context array. Unfortunately, the copying method was chosen as SimpleScalar had too many direct references to the register file (regs_R), especially in syscall.c, to allow a direct replacement with the array (HWCT[]). The drawback to this approach is that, while easier to implement, it slowed down the simulation speed of SimpleScalar considerably.
3. The hw_cswitch function checks to see if there exist more threads in Pthreads (MVP_M_THREADS bit). If yes, then the next context is selected. If the new context is valid, the context's register file is copied into the active register file and the new PC is returned. If the new context is not valid, in accordance with the model, the PC of the interrupt handler is returned so that a new thread can be scheduled by Pthreads. If there are no more threads to run, hw_cswitch searches for the next executable context. When found, the new context is copied and the new PC is returned. However, if the context found is the one that caused the context switch in the first place, then thread mode is disabled (TD_MODE) and execution returns to the context. Thread mode is disabled at this point since there are no more threads to run in either Pthreads or the other hardware contexts.

Once the new PC is returned from hw_cswitch, the MVP simulator can easily resume execution on the new PC. Either running a new or same context, or executing Pthreads library instructions for scheduling a new thread.

Though the changes to SimpleScalar were the most difficult and lengthy part of the creation of the MVP simulator, once completed simulations were ready to be performed on the new architecture.

6. SIMULATION RESULTS

In order to study the performance and effects seen by multithreading and prefetching on the MVP, several areas of processor performance were studied: These areas include speedup, cache effects, pipeline bottlenecks (in the form of IPC), and prefetching (in the form of speedup). However, this chapter first focuses on the chosen simulated processor and on the simulated benchmarks.

6.1 Simulated Processor

The simulated processor for the MVP is a combination of SimpleScalar defaults and realistic cache effects. The regular superscalar aspect of the MVP is assumed to have the following characteristics:

- Functional units and their corresponding latencies are displayed by Table 1.
- The cache and main memory organizations and latencies are displayed by Table 2. The caching hierarchy is assumed to be only two levels (L1 and L2) and utilizes a blocking scheme. It is important to note that the MVP can operate with more cache levels, but two were chosen in order keep simulation time low. Also, though the main memory latency is rather conservative in its outlook compared to

Table 1: Simulated Functional Units for MVP.

FUNCTIONAL UNIT	LATENCY	PIPELINED
Integer ALU/Branch	1	Yes
Integer Multiply	3	Yes
Integer Division	12	No
Load/Store	2	Yes
FP Addition	2	Yes
FP Multiplication	4	Yes
FP Division	12	No

Table 2: Cache and Main memory Latencies.

	L1 I-CACHE	L1 D-CACHE	L2 CACHE
Size	16K Bytes	16K Bytes	256/512 K Bytes
Associatively	Direct-mapped	4-way Set	4-way Set
Line Size	32 Bytes	32 Bytes	32/64 Bytes
Latency (hit)	1 CPU Cycle	1 CPU Cycle	6 CPU Cycles
Latency (miss)	6 CPU Cycles	6 CPU Cycles	100 CPU Cycles

modern technology (i.e. UltraSparc IIi maintains a 72 cycle main memory latency [15]), it is expected that memory latency will grow in the future. Moreover, multiprocessor systems will add even more latency derived from their shared-bus protocols.

- The fetch, decode, and issue width of the pipeline is 4 (which gives the MVP the ideal Instructions executed Per clock Cycle (IPC) of 4.0). The number of RUU entries, and therefore both the number of ROB and reservation station entries, used by the MVP is 16.
- Branch prediction in the MVP is assumed to use a 2K-entry BTB with 2-bit branch prediction bits.

The new hardware added to the processor to support fast multithreading (multiple register files and hardware scheduler) is assumed to have the following characteristics:

- Assuming that the process for context switching is supported entirely by dedicated hardware, the process is very similar to recovering from a miss-predicted branch and requires a penalty of 3 cycles. The process for switching execution from one hardware context to another involves:
  * Powering down one register file and powering up the register file corresponding to the new thread.
  * Flushing the ROB of all instructions issued after the faulting load instruction.

    * Setting the fetch unit to begin fetching instructions from the new thread's PC.

- Context switching to a new thread in the MVP is initiated when an L2 cache miss is detected. Since the L1 caches have such a minimal latency of only 6 CPU cycles, it was not worth context switching to a new thread. However, the MVP is not constricted to using only the L2 cache, but can utilize any miss in the memory hierarchy that results in a sufficient delay of CPU cycles.
- The number of threads created for each simulation trial was kept equal to the number of hardware contexts. It was empirically derived that varying the number of threads in regards to a constant number of hardware contexts had only a minimal increase on the MVP's performance. This minimal increase was the result of a small incremental increase seen in performance as the number of threads was increased. However, this increase in performance was offset by an incremental increase in Pthreads software overhead. The best performance was obtained by keeping the software overhead as minimal as possible by only creating enough threads for the hardware contexts.

6.2 Benchmarks

In order to both validate and obtain processor performance information for the MVP, a set of five benchmark programs were selected. Both Matrix Multiplication (MMT) and Gaussian Elimination (GE) were hand-coded. The three other benchmarks, Radix Sort (RS), MP3D, and Fast-Fourier Transform (FFT) were selected from the SPLASH-2 benchmark suite [16]. Developed to benchmark performance of shared-memory multicomputers, the SPLASH-2 suite uses ANL macros to create and manage the programs threads. To port the SPLASH-2 benchmark suite to the MVP simulator, the ANL macros were directly replaced with their Pthreads equivalent statements. It is important to note that

no optimization was attempted in the port to help prevent the data obtained from representing only the best-case scenarios. To get an understanding of what each benchmark strives to test, a brief description of each follows:

- FFT implements a complex 1-D version of the sqrt(n) six step FFT algorithm. The algorithm has been pre-optimized for minimizing inter-thread communication. The data set consists of n complex data points and another n complex data points called the roots of unity. Therefore, each thread is responsible for transposing a contiguous submatrix sqrt(n)/p * sqrt(n)/p with every other thread as well as transposing a single submatrix by itself. The data set between threads is very localized and even though optimized against it, still supports much data sharing between the threads.
- GE partisans an n-by-n matrix into threads by using the row-rise, block-cyclic method. Next, one thread calculates its pivot and performs the division step followed by all other threads performing their elimination steps. The threads created by GE tend to exhibit very separate and distinct data sets from each other with only minimal data sharing caused by the shared pivot values. GE is very similar to LU decomposition in SPLASH-2, except for the fact that GE generates only the upper triangular matrix.
- MMT is the simplest benchmark used on the MVP simulator. MMT parses the matrix into blocks and assigns those blocks directly into threads. The data set for the threads is very disjoint, but the row by column effect does produce considerable data overlap between the threads. Due to MMT's simplicity, there is no intercommunication or synchronization between threads.
- MP3D is a simple simulator for rarefied gas flow over an object in a wind tunnel. To prevent a long initial file read operation, the geometry of the object is created as a data structure in main memory at start-up. The algorithm assigns given particles into threads and the threads spend most of their execution time in a loop

moving their particles through a single time step. Each thread continuously detects every possible collision of its particles with any other particle within a pre-defined cell space. Whenever a collision is detected, knowledge of that collision must by forwarded to all participating particles. In essence, the MP3D algorithm contains a data set that is very localized and must share much of that data set among the other threads.

- RS performs the classic linear sort algorithm in parallel. The algorithm passes over the assigned key values to each thread and, based on those key values, each thread generates its own local histogram. Next, all the threads combine the local histograms into a large globally shared histogram. Finally, each thread iterates over its assigned array and by utilizing the global histogram, permutes its keys into a new sorted array. The RS algorithm results in almost two distinct data sets. First, whenever the global histogram is accessed, the data set is very localized and shared. However, when not accessing the global histogram, the data set becomes very disjoint between threads. The RS algorithm also has most complex synchronization of the MVP's benchmark suite.

6.3 Results

Four sets of simulation runs were performed on each benchmark for comparison purposes. The first set, serving as the control, was obtained by running a sequential version of the benchmarks on SimpleScalar. These sequential results, known as the Serial versions, served to illustrate the base performance that MVP should beat in order to show improvement. The other three simulation runs obtained data for MVP with 2, 4, and 8 hardware contexts in operation. The results were obtained by simulating approximately 160 million instructions (MP3D) to 1.1 billion instructions (GE and RS) and were grouped into four categories: speedup, cache effects, bottlenecks, and prefetching.

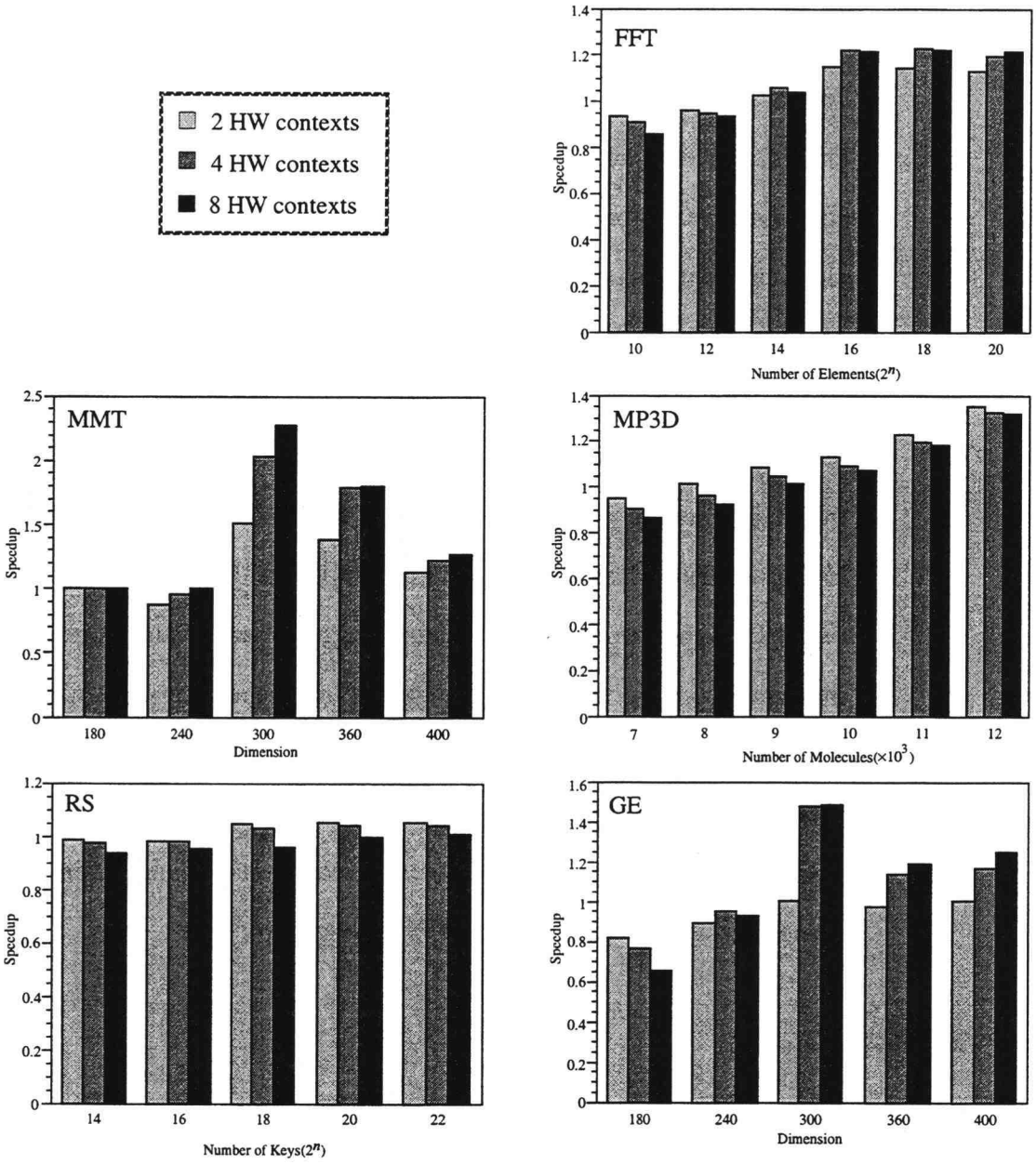

Figure 6: Speedup Results for Various Benchmarks.

Figure 6 displays the relative performance of the MVP for the five benchmark programs. The results have been normalized relative to the performance of the serial versions. The data shows that though Pthreads does introduce some overhead, as the data set size grows, the MVP overcomes the overhead and performs better than the serial

versions. A good example of this effect is delineated by MP3D. FFT also shows a good improvement in performance as the data set size increases and the algorithm begins to take advantage the latency tolerance of the MVP. Another effect displayed by Figure 6 is that the use of more hardware contexts does not necessarily result in improved performance, as seen by both RS and MP3D. This result is obtained from the a combination of the benchmarks' high synchronization requirements, relatively small parallel portions, and increased overhead from more hardware contexts. Although the performance degrades as more hardware contexts are added, the performance margin between the cases narrows as the problem size increases. In essence, the cases with 4 and 8 hardware contexts will overcome the corresponding overhead increase and eventually outperform the 2 hardware context case. Other effects seen by Figure 6, include improved, but varied, performance seen by both GE and MMT. The results garnered from GE for the 300 case appears to be based on a radically lower L2 miss rate (compared to the serial version) caused by the addition of threads. The threads' data sets seem to have resulted in a very good mapping of the 300 by 300 matrix into the L2 cache. MMT performance seems to also be seriously effected by the L2 cache. With MMT, multithreading caused a much lower L2 miss rate (again compared to the serial versions), while the 240 case resulted in a much higher L2 miss rate. Clearly, though multithreading results in improved performance for large data sets, the L2 cache effects have a high impact on how great that performance improvement will be.

To gain a closer understanding of the effect that the caches have on multithreading, two sets of graphs were obtained. The first set monitored the L2 miss rate to determine the effects that are seen by the L2 cache to main memory bus. The second set of graphs displays the number of accesses received by the L2 cache. This data set is further broken down to show which accesses were issued from the L1 I-cache or the L1 D-cache. In essence, by using the second set of graphs, the performance of the L1 caches for the benchmarks can also be observed.

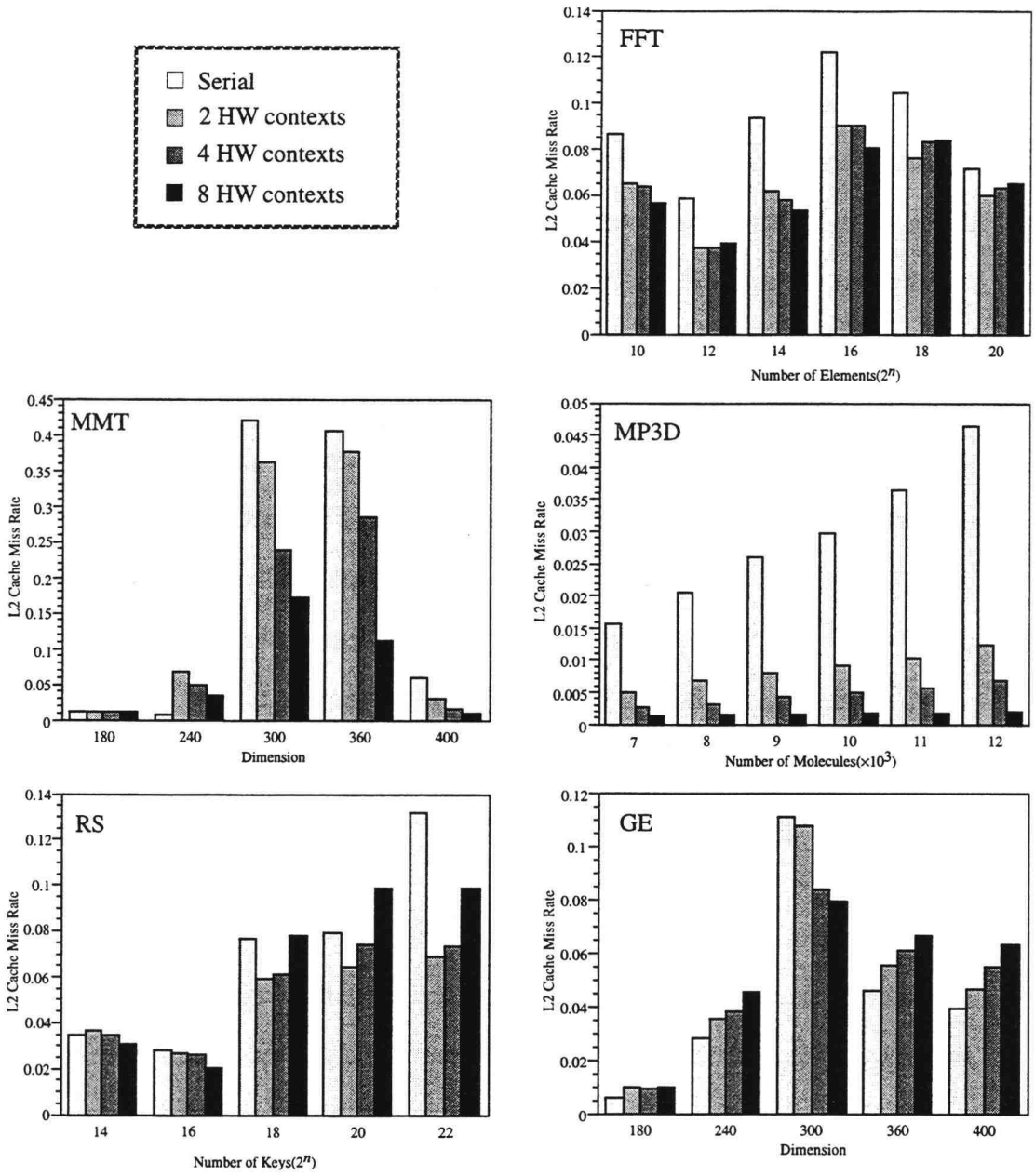

Figure 7: L2 Cache Miss Rates for Various Benchmarks.

The data graphs of the first set are shown by Figure 7, and give some rather interesting results. It is apparent that, for the majority of the benchmarks, the L2 miss rates for the MVP are lower than the serial versions. In fact, the only benchmark not displaying

this result is GE. The lower miss rates observed are the result of the fact that the data sets used by the benchmarks have very high localities. This data locality creates a unique situation where one thread can help out another in the caches. In essence, one thread can cause a cache miss that will bring in data that another thread will need to access later on. To illustrate this point, consider the RS benchmark algorithm. When one of RS's threads accesses the global histogram for a data value, other data values that the thread will not need but other threads might, will also be brought into the cache. Thus, another thread could potentially find the global histogram data that it needs already residing in the cache, and will not generate a miss. Sequential programs cannot perform this sequence and result in worse performance by generating a cache miss, using a small portion of the data, and then replacing the line before the rest of the data can be used. Though MMT does not utilize a very localized data set, the nature of its algorithm results in a very similar effect. This result is obtained by having MMT multiply a row of one matrix to a column of another. By distributing the rows of both matrices in a row-wise block manner among the threads, when one thread cache misses on a column value, the thread will inadvertently load other column values to be used by other threads. These extra data values are obtained by the fact that a cache line is part of a row of a matrix. When a thread creates a cache miss while attempting to find a column value, other values belonging to the same row are also loaded into the cache. Therefore, when another thread goes looking for its column values, the values could already be in the cache. Here again, the serial programs tend to replace the data before it can be reused, and thus generate more cache misses.

In other words, there are two effects that the MVP exploits to lower the L2 miss rates: Data sharing and data locality. Data sharing, occurs when one thread brings in a data value that it and another thread will use. Data locality occurs when one thread brings in data values that other threads will use that happen to exist on the same cache line that has the data value that the first thread will use. In both cases lower miss rates than the serial

versions result as MVP threads help prefetch data that can be used multiple times before being replaced.

Another effect seen by the L2 cache is an increase in the L2 miss rate as shown by RS. This increase is caused by the sorting portion of the algorithm. When the threads sort their individual keys of the array, the data set becomes very disjoint and results in an increased L2 miss rate from conflict misses. As seen by the graph, the sorting portion begins to have a more profound effect on performance as the number of hardware contexts increases. Similarly, GE also uses very distinct and low locality data sets among its threads. Thus, a thread, upon generating a cache miss, simply brings in more rows belonging only to the same thread. The net result is that now threads are competing for space in the L2 and the higher number of conflict misses results in a higher overall L2 miss rate than the serial versions.

There also remains two unique L2 effects displayed a both GE (at 300) and MMT (at 240). At these data set sizes, the graph illustrates a reversal of the general trend between miss rates seen by all the rest of the data set sizes. It is hypothesized that these results are obtained from the simple fact that some data set sizes tend to map themselves into the L2 cache much better or much worse than others.

The last interesting effect on the L2 cache shown by Figure 7 is displayed by RS and FFT. Both RS and FFT exhibit a minimum cache miss rate for cases 16 and 12, respectively. At these data points an interesting transition occurs as the caches are nearly full (so compulsory misses are offset and conflict/capacity misses are at a minimum). Increasing the number of L2 accesses (i.e. increasing the data set size) causes an increase in both conflict and capacity misses, while decreasing L2 accesses (i.e. decreasing the data set size) emphasizes the effects of the compulsory misses.

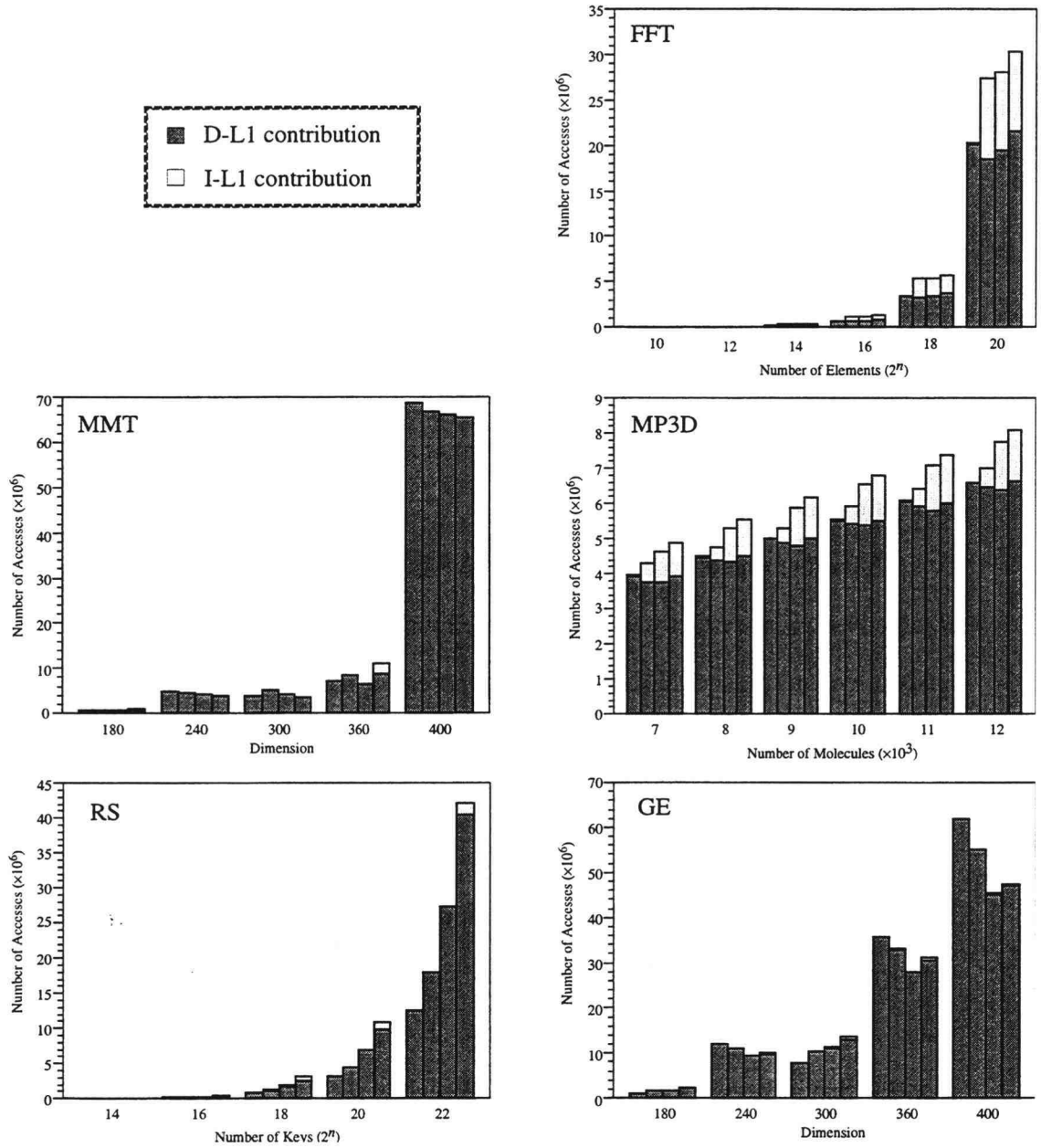

Figure 8: L1 Cache Miss Accesses for Various Benchmarks. Four Graphs for Each Point Represent, From Left to Right, Serial, 2 HW Contexts, 4 HW Contexts, and 8 HW Contexts.

Figure 8 displays the effect on the L1 caches. For FFT, MP3D, and RS, the results suggest that while multithreading works well in the L2 cache, the exploitation of locality is hindered in the L1 D-cache. The cause is the L1 D-cache's smaller line size, no longer can

threads as effectively help one another as in the L2 cache. This evidence is especially strong for the RS algorithm, which can take advantage of almost all data from the global histogram that is loaded into caches. It also appears that the data separation experienced by GE and MMT helps to result in an improved L1 D-cache performance. With these two benchmarks the non-local data serves to map the threads better into the L1 D-caches than the serial versions and results in less L2 cache accesses. However, it is important to note that once again matrix size plays a very important role as both GE (at 300) and MMT (at 360) show completely opposite results compared to the rest of the data gathered for those two benchmarks.

Another aspect illustrated by Figure 8 is the L1 I-cache effects. The results obtained are almost exactly what is expected when using multithreading. As the number of hardware contexts, and therefore threads, increases, the threads begin to fight over space in the L1 I-cache and conflict misses increase.

Figure 9 delineates the average IPC. These results were collected in order to explore any new bottlenecks that multithreading might cause to a superscalar pipeline. The IPCs were obtained by dividing the total number of instructions executed by the total number of cycles it took to complete each benchmark. Assuming a fetch, decode, and issue width of 4, leads to an ideal IPC of 4. To find bottlenecks, the portions of IPC lost (i.e. IPC that lowered the ideal IPC from the actual IPC) were obtained from each of the Fetch stage, Dispatch stage, and Issue stage. Next the graphs were broken down to show what percentage of the lost IPC was incurred from a bottleneck at each stage. The pipeline stage bottlenecks that are graphed include: IPC lost due to exceeding fetch bandwidth (Fetch_lost), RUU full (RS/ROB_lost), busy execution units (FU_lost), and issue bandwidth limitations (Issue_lost). Results from bottlenecks at the Decode and Commit stages were also obtained, but were dropped from the graph when it became apparent that the bandwidths were never exceeded in any of the simulations executed.

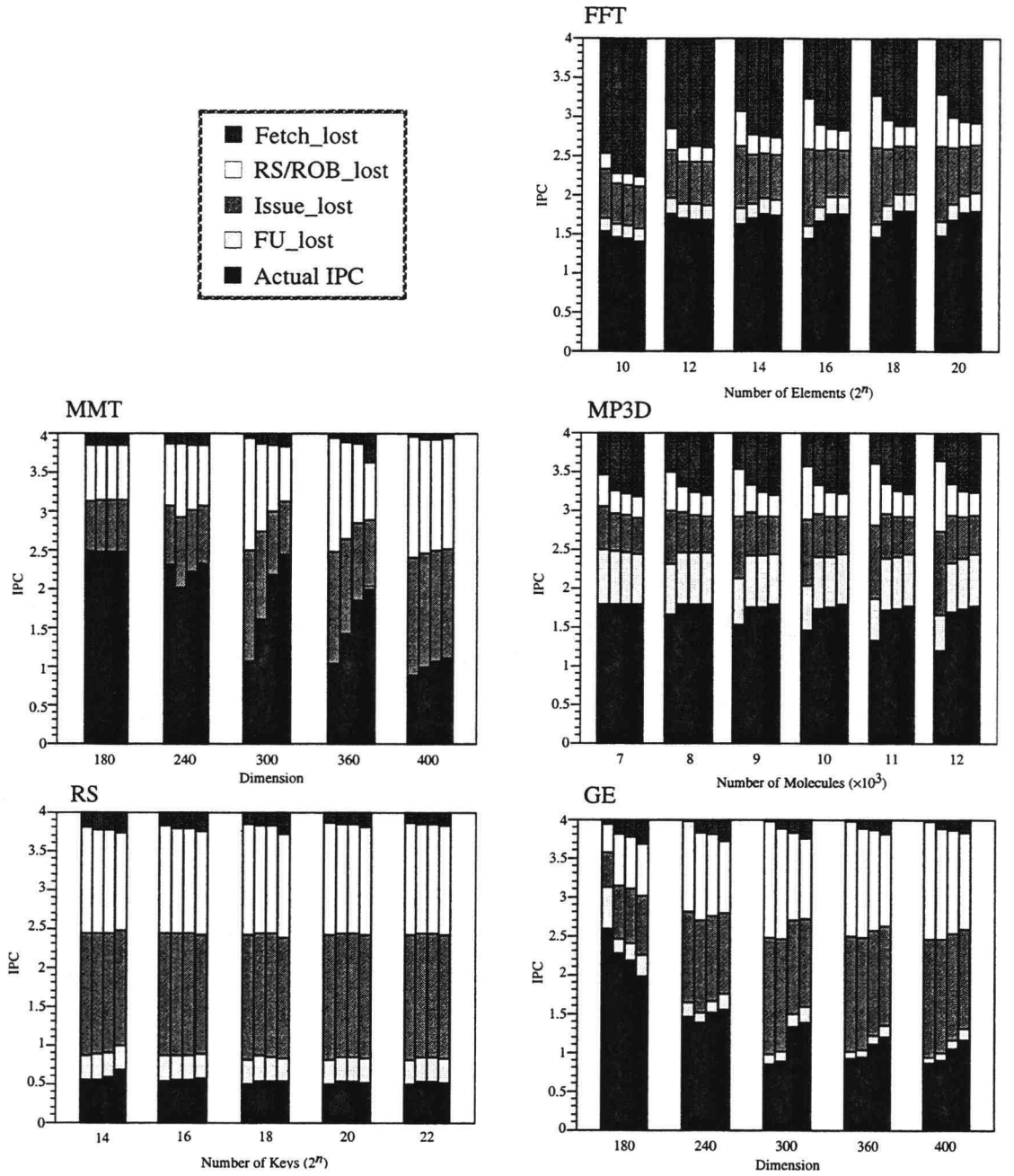

Figure 9: IPC for Various Benchmarks. Four Graphs for Each Point Represent, From Left to Right, Serial, 2 HW Contexts, 4 HW Contexts, and 8 HW Contexts.

From the graphs for IPC, it is clear that multithreading creates additional stress on the fetch bandwidth. This additional stress is due to the fact that program locality is

reduced by context switching between threads. By switching contexts the PC lowers the average number of instructions before a branch instruction is reached. Therefore additional stress is placed on the fetch unit to fetch across more branches resulting in increasing levels of speculation. In short, fetch bandwidths have to be improved [17]. Another effect that can be observed is a decrease in IPC lost at the issue stage. By switching threads on a cache miss, long latency data dependencies are avoided and consequently, more instructions can be issued.

Also exhibited by the graphs is almost no change in IPC by RS and only small amounts of difference noticeable by FFT and MP3D. The reason is the level of synchronization and therefore parallelism available in the algorithms. RS has much synchronization, while FFT and MP3D have some and MMT has none at all. Though GE (with large amount of synchronization) exhibits large changes similar to MMT, the result is caused by the fact that GE has an extremely large parallel portion in comparison to its small sequential portions.

The final effect seen in Figure 9 is that for some of the smaller data set sizes the serial version has a higher IPC than the MVP versions. The reason is that Pthreads code has a relatively lower IPC then the threads. Mainly Pthreads cannot context switch so all memory latencies must be waited for and as a result IPC suffers. As the data set size increases, IPC begins to be increase as the added parallelism offsets the sequential Pthreads instructions.

The final study of the MVP examines how two simple instruction prefetching techniques can help mitigate the fetch bottleneck. Unfortunately, the prefetching strategies did not have a high impact on improving the performance. However, an interesting result developed when it was found that while the simple next-line strategy resulted in a performance improvement, next-thread prefetching resulted in a performance increase of only .1%. Also, whenever next-line prefetching did not result in a performance

improvement, if observed, next-thread prefetching was directly responsible for all performance degradation.

Next-line prefetching resulted in an only modest performance improvement of 3% at the most. Since, in essence, prefetching is designed to help remove stalls due to cache misses, the relatively small proportion of L1 I-cache misses for most of the benchmarks dictates that next-line prefetching will result in this modest performance improvement. However, it is also clear that because of multithreading's higher L1 I-cache miss rate shown by Figure 8, next-line prefetching provides more help than for serial versions. In fact, due to the incredibly low miss rate of the serial versions L1 I-cache, no serial version had a measurable improvement from next-line prefetching. This observation can be seen best by FFT and MP3D. In FFT and MP3D, serial performance was degraded by .33% and 0% ave., while the MVP performance was improved by .70% and 1.2% ave. respectively. Both FFT and MP3D have the highest L1 I-cache miss rates and therefore, exhibit the best performance improvement for the multithreading. Thus, though the performance improvement is modest, it can be shown that multithreading has the best chance of exploiting the benefits of next-line prefetching.

A disappointment was the performance of next-thread prefetching. Due to the proportionally small L1 I-cache misses that occurred from context switching, next-thread prefetching results in only very small improvements in performance. Additionally, next-thread prefetching managed to degrade performance of the benchmarks by generating conflict misses and replacing Pthreads instructions. Typically, this performance degradation was small or modest in the case of MMT (8 hardware contexts 1% ave.) and GE (4 hardware contexts 1.8%), except for two cases of FFT and MP3D. In these cases, the combination of the number of threads, data set size, and pre-fetching served to generate an unusually high number of conflict misses in the L1 I-cache (11% and 13% respectively).

Again, it must be remembered that simple next-line prefetching resulted in almost double the memory bandwidth of a single processor. With a gain of only less than 1% on

average, prefetching is seen as not worth the design time for any type of SMP machine but may be worth further investigation of advanced methods for single processor use.

7. SUMMARIES AND CONCLUSIONS

In summary, this thesis explored the simulation study of the MVP. The results showed that the MVP derives its increase in performance from exploiting both parallelism and data locality. Also, it was observed that the cache sizes relative to the data set sizes created profound shifts the performance of the MVP. Though it was discovered that simple next-line prefetching helped to relieve slightly the fetch-unit bottleneck created by multithreading, it was shown that MVP has the potential to benefit more from prefetching's existence. It is the conclusion of this thesis that, for future microprocessors, multithreading provides good performance for additional die space..

However, there is still much more research to be conducted to help the MVP achieve the greatest possible benefit of multithreading. Future research areas include more study into prefetching. Unfortunately limited by the use of SimpleScalar as a simulator base, more modern approaches to prefetching beyond next-line could not be studied. In particular, the MVP has the potential to erase the fetch bottleneck by combining branch prediction with instruction alignment to increase the number of fetched instructions. Also, the use of wrong path prefetching could stand lower the number of I-cache misses even further. Again, however, bandwidth limitations must be considered when dealing with prefetching techniques.

Another performance effect that was unable to be tested was the use of the MVP in a multiprocessing environment. With the addition of multiple shared memory MVPs, the existence of a cache coherence protocol is mandatory. Cache coherence has the potential to increase both the cache miss rate, by invalidating cache lines, and the memory latency, by placing more contention on the shared memory bus. It is easy to surmise that both of these effects would have an impact on the performance of the MVP.

Finally, the MVP has the potential to be modified to utilize Simultaneous Multithreading (SMT). As described previously, SMT is the process of interleaving thread

instructions so that multiple threads are executed at the same time. Again, though this approach necessitates a large ROB and reservation stations, the large grain of the threads used by the MVP works to a greater advantage. The large grain threads are better able to exploit both ILP and TLP (thread level parallelism) [11,18]. Thus, with more instructions available to the execution units, processor utilization will increase. Combing a SMT MVP (SMVP)with a multiple RUU system will help decrease the dependency on large scale ROBs. Another benefit of SMT is the ability to execute non-related threads; that is, a single SMVP could execute threads from different programs and act very much like a small-scale, shared-memory multiprocessor.

Indeed, though prefetching needs more research to offer any substantial rewards, it is clear that the MVP offers a clear benefit for future processors and provides various stepping stones to more advanced designs of SMT processors. Perhaps as people begin turning more and more towards the concept of multithreading, the future will dictate that MVPs will be a dominant force in computing.

BIBLIOGRAPHY

1. Culler, D. And Singh, J. P., Parallel Computer Architecture: A Hardware/Software Approach, Draft, Morgan Kaufmann, 1997.

2. Pierce, J., Mudge, T., "Wrong-Path Instruction Prefetching." *Proceedings of the International Symposium on Microarchitecture*, pages 165-175, December 1996.

3. Rotenberg, E., Bennett, S., Smith, J. E., "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching." *Proceedings of the International Symposium on Microarchitecture*, pages 24-34, December 1996.

4. Farkas, K. I., Jouppi, N. P., Chow, P., "How Useful are non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? (DEC WRL Technical Report version)." *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 78-89, January 1995.

5. Lee, B. And Hurson, A. R., "Dataflow Architectures and Multithreading." *IEEE Computer*, Vol. 27, No. 8, 1994, pp. 27-39.

6. Catanzaro, B., Multiprocessor System Architectures, Prentice Hall, 1994.

7. Butenhof, D. R., Programming with Posix Threads, Addison Wesley, 1997.

8. Smith, B., "The Architecture of HEP." *in Parallel MIMD Computation: HEP Supercomputer and Applications*, edited by J.S. Kowalik, MIT Press 1985.

9. Alverson, R. *et al.*, "The Tera Computer System." *Proc. Int'l. Conference on Supercomputing*, June 1990.

10. Agarwal, A. *et al.*, "Sparcle: An Evolutionary Design for Large-Scale Multiprocessors." *Proc. Of Workshop on Scaleable Shared Memory Multiprocessor*, Kluwer Academic Publishers, 1991.

11. Loikkanen, M. And Bagherzadeh, N., "A Fine-Grain Multithreading Superscalar Architecture." *Parallel Architectures and Compilation Techniques '96*, October 1996.

12. Sohi, G. S., and Vajapeyam, S., "Instruction Issue Logic for High-Performance Interruptible Pipelined Processors." *Proceedings of the 14th Annual Symposium on Computer Architecture*, June 1987, pp. 27-34.

13. Oeriz, D., Lee, B., Yoon, S. H., and Lim, K. W., "A Preliminary Study of Architectural Support for Multithreading." *30th Hawaii International Conference in System Science*, Software Track, January 7-10, 1997.

14. Burger, D. C., Austin, T. M., and Bennett, T., "Evaluating Future Microprocessors-The SimpleScalar Tool Set." *UW Computer Sciences Technical Report #1308*, July, 1996.

15. Normoyle, K., "UltraSPARC IIi™ - A Highly Integrated 300 MHz 64-bit SPARC V9 CPU." *HOT Chips IX*, August 1997.

16. Woo, S. C. *et al.*, "The SPLASH-2 programs: Characterization and Methodological Consideration." *Proc. 22nd Annual Symposium on Computer Architecture*, June 1995, pp. 24-36.

17. Conte, T. *et al.*, "Optimization of Instruction Fetch Mechanisms for High Issues Rates." *Proc. Of the 22nd International Symposium on Computer Architecture*, June 1995.

18. Lo, J. L. *et al.*, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading." *ACM Transactions on Computer Systems*, August 1997, pp. 322-354.