AN ABSTRACT OF THE THESIS OF

Daniel Ortiz-Arroyo for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on December 12, 2002. Title: The Dynamic Simultaneous Multithreaded Processor.

Abstract approved: Redacted for Privacy

Il-Beom Lee

This dissertation investigates diverse techniques to support multithreading in modern high performance processors. The mechanisms studied expand the architecture of a high performance superscalar processor to control efficiently the interaction between software-controlled and hardware-controlled multithreading. Additionally, dynamic speculative mechanisms are proposed to exploit thread-level-parallelism (*TLP*) and instruction-level-parallelism (*ILP*) on a *Simultaneous Multithreading (SMT)* architecture.

First, the *hybrid multithreaded execution* model is discussed. This model combines software-controlled multithreading with hardware support for efficient context switching and thread scheduling. A thread scheduling technique called *set scheduling* is introduced and its impact on the overall performance is described. An analytical model of the hybrid multithreaded execution is developed and validated by simulation. Through stochastic simulation, we find that the application of the hybrid multithreaded execution model results in higher processor utilization than traditional software-controlled multithreading.

Next, in the main part of this dissertation, a new architecture is proposed: the *Dynamic Simultaneous Multithreading* (DSMT) processor. In this architecture, multiple threads are identified and created speculatively at runtime without compiler help. Subsequently, a SMT processor core executes those threads. The performance of a DSMT processor was evaluated with a new execution-driven simulator developed specifically for the purpose. Our experimental results based on simulation show that DSMT architecture has very good potential to improve SMT processor's performance when there is only a single task available for execution.

©Copyright by Daniel Ortiz-Arroyo

December 12, 2002

All Rights Reserved

The Dynamic Simultaneous Multithreaded Processor

by

Daniel Ortiz-Arroyo

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 12, 2002
Commencement June 2003

Doctor of Philosophy thesis of Daniel Ortiz-Arroyo presented on December 12, 2002.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of the Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of the 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

Daniel Ortiz-Arroyo, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF ACRONYMS

ANL	Argonne National Laboratory
API	Application Programming Interface
BTB	Branch Target Buffer
CC-UMA	Cache Coherent UMA
CM	Clustered Multithreading
CMP	Chip Multiprocessor
DMT	Dynamic Multithreading
DSMT	Dynamic Simultaneous Multithreading
EMSIM	Extendible Microarchitecture Simulator
FU	Functional Unit
GUI	Graphical User Interface
ILP	Instruction Level Parallelism
IPC	Instructions per Clock cycle
IQ	Instruction Queue
ISA	Instruction Set Architecture
LDBTB	Loop Detection Branch Target Buffer
MOB	Memory Order Buffer
LSST	Loop Stride Speculation Table
MMU	Memory Management Unit
MPI	Message Passing Interface
MVP	Multithreaded Virtual Processor
NUMA	Non Uniform Memory Access
OO	Object Oriented
OS	Operating System
PISA	Portable Instruction Set Architecture
ROB	Reorder Buffer
RAW	Read After Write
RS	Reservation Stations
SMT	Simultaneous Multithreading
UMA	Uniform Memory Access
TLP	Thread Level Parallelism
TCIU	Thread Control and Initiation Unit
UML	Uniform Modeling Language
VLIW	Very Long Instruction Word
WAR	Write After Read
WAW	Write After Write
XML	Extensible Markup Language

DEDICATION

To my wife: Sandra. She taught me the true nature of love, guiding me with her intense inner light during the time when I was lost. Her unconditional love, support, patience and understanding kept me always focused on reaching my goals.

To all my kids Sandra, Mariana and Daniel for all the love and joy they give me everyday; for their understanding on all those days when they needed me, and daddy was absent.

To my parents: Daniel and Blanca, for their endless love and support. I own them everything I am; they taught me what is truly essential in a human being.

To my brother, and sisters for their love and encouragement.

To my old friends: Ernesto, Victor, Roberto, and Sergio for their friendship during good and bad times. To my new friends: Francisco, Pablo, Mario, and Trevor for sharing with my family and me uncountable special occasions.

To all my mentors, but specially my elementary school teachers; they seeded in me the interest in this never-ending journey.

To all developers of public domain software that made, in large part, this research feasible.

Finally, to all those who still believe that a better world is possible and work everyday to make it happen someday.

THE DYNAMIC SIMULTANEOUS MULTITHREADED PROCESSOR

1 INTRODUCTION

Modern, high performance *superscalar* processors are designed to exploit dynamically the *instruction level parallelism* (ILP) available in compiler-generated code. In contrast, *Very Long Instruction Word (VLIW)* processors exploit the ILP extracted statically by a specialized compiler [23,31,58].

In superscalar processors, instructions are fetched *in order* but may be executed *out of order* sequentially or in parallel. However, out of order execution of instructions must preserve the semantic contents of a program to produce correct results. Accordingly, internal mechanisms in the architecture force instructions to commit their values in order to the register file, ensuring at the same time, precise interrupt handling [57].

Superscalar processors are *single-threaded* i.e., they can fetch and execute instructions from a single program or *control flow*. In these processors, instructions fetched from a single thread are decoded and their register operands renamed. The renaming mechanisms eliminate false dependencies among instructions (WAR and WAW). The residual *true dependencies* (RAW) represent the *data-flow* limit in ILP.

After the renaming step, independent instructions are issued for execution and will eventually commit their results to the register file. In contrast, dependent instructions are placed in the *instruction window*. These instructions will remain in the instruction window until their dependencies are resolved [23,58]. When this occurs, those instructions are issued for execution and later on, in the final stage of processing, their results are committed to the register file.

Branches disrupt the sequential fetching of instructions from memory. The result of a conditional branch instruction is not known until late in the pipeline, until the execution stage. For this reason, the fetching mechanism employs branch prediction methods [59] to avoid unnecessary delays when a branch is found in the instruction stream. Using branch prediction, the most probable branch direction and target address are predicted based on the past history of a branch. Predicting the direction of a branch allows the processor to continue fetching with no interruption when the prediction is correct. However, when the prediction turns out to be wrong, all instructions fetched after a branch must be flushed from the pipeline; then, fetching resumes execution in the correct direction. Instructions located at both the taken and not-taken path following a branch are called *control-flow dependent*. Branch misprediction is detrimental to a processor's performance because clock cycles are consumed processing useless instructions.

Unfortunately, control-flow and data-flow dependencies occur very often in programs. Hence, they reduce the amount of ILP that superscalar processors are capable of exploiting. This limitation in ILP is critical and generates a significant under-utilization of processor resources that degrades performance. Fortunately, research on data prediction and control-flow speculation [22, 36 and 40] has found that the performance limitation on ILP due to data and control-flow dependencies is not absolute.

To overcome superscalar processor limitations in exploiting ILP, researchers have proposed exploiting *Thread Level Parallelism (TLP)* [37]. In TLP a sequential stream of instructions or *thread* is the basic unit of parallel execution. Maximum performance is obtained when several threads are executed concurrently either in a single multithreaded processor or in multiple single-threaded processors.

In multithreaded processors, TLP is used to tolerate long latencies or to compensate for the lack of ILP in a thread. In the first case, multiple thread execution is overlapped; the processor executes useful instructions from another thread while the current thread latency is being fulfilled. In the second case, TLP increases instruction window size, which compensates for the lack of per thread ILP; therefore, both ILP and

TLP parallelism can be exploited on architectures with multiple thread processing units capable of issuing simultaneously instructions from different threads.

To exploit TLP, threads have to be extracted within a program. One way to extract multiple threads is through static techniques. Static techniques are used in binary annotators [33,61] or in parallelizing compilers to identify threads [78]. Alternatively, in other approaches to exploit TLP, an operating system task or program are the threads of execution [73]. However, static methods have their own particular problems. Parallelizing compilers become useless if the original source code is unavailable; binary annotators tend to exploit fine-grain parallelism (e.g., basic blocks and inner loops) but require in some cases modifications to the *instruction set architecture* (ISA).

Alternatively to static methods, in dynamic methods, threads are extracted using run-time thread detection mechanisms. Without compiler or user help, dynamic methods of thread generation generally employ speculative techniques. Threads in this case, are created mainly from loop boundaries, subroutines or from exception handling routines [3, 70 and 82].

Dynamic and static methods to generate threads have their own merits. However, no single method is sufficient to exploit TLP. Next generation processors will have to harness all forms of parallelism to continue increasing performance.

In addition to generating threads, successful exploitation of dynamic TLP requires support in the processor for controlling the execution of multiple threads. Also, special mechanisms are required to resolve inter-thread, register and memory dependencies, when threads are not completely independent. Lastly, a synchronization mechanism is required to coordinate the execution of multiple threads.

The degree of support in hardware for multithreading can vary greatly. For example, it can be as simple as register windows for supporting multiple hardware contexts [1] or as complex as in the TERA multiprocessor where each processor supports up to 128 processor states [4]. In order to support multiple thread execution in hardware, the state of a thread is stored in its *context*. A thread context contains the register file and the program counter associated with a thread.

Hardware-based models of multithreading have been roughly categorized as: *fine-grain, coarse-grain* and *simultaneous multithreading*. The first two classifications, fine and coarse grain multithreading, are based on the number of instructions executed by a processor before switching contexts. The third category, *Simultaneous Multithreading* (SMT), is not based on context switching frequency. Instead, multiple threads are executed simultaneously, thereby keeping all contexts active without switching between them.

In fine grain multithreading, processors switch contexts every clock cycle and consequently resources are shared at a fine granularity. However, performance degrades when there is only a single thread in the system. Additionally, even when multiple threads are in execution, only a single thread can issue instructions in a single cycle. An example of a fine grain multithreading system is the TERA multiprocessor [4].

In contrast, coarse grained multithreaded processors switch contexts when a thread of execution gets blocked. A thread is blocked when long latency operations, such as cache misses or thread synchronization, occur in the instruction stream. This type of multithreading achieves high efficiency by overlapping the execution of a new thread when the current thread of execution is blocked. Given that in this model of multithreading only one thread is active at a time, instruction issuing is again limited to a single thread. Hence, when there is not enough ILP to exploit in a thread, processor resources are underutilized. Examples of this type of architecture are APRIL [1] and MAJC [65, 67]. This model of multithreading is sometimes called *vertical multithreading*.

Vertical multithreaded systems are especially suited for shared memory multiprocessing systems. In these systems, data may be obtained from either local memory as in the case of Uniform Memory Access (UMA) machines, or from remote memory, as in the case of Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines [28]. A multithreaded processor in these machines hides memory latency by overlapping the execution of a new thread with access to memory.

The third form of multithreading, *Simultaneous multithreading* (SMT) has received considerable attention in recent years [73]. SMT is a technique that converts

TLP into ILP. Under this vision, SMT accommodates variations in parallelism. Namely, when a program has a single thread, i.e., lacks TLP, the ILP extracted from a thread uses all resources available. When more threads exist, TLP can compensate for the lack of per-thread ILP. SMT, in contrast with coarse grain and fine grain multithreading, enables multiple threads to issue multiple instructions every clock cycle. Therefore, when a thread lacks ILP, another thread with enough ILP can compensate for this deficiency increasing processor utilization through efficient resource sharing. For this reason, SMT surpasses superscalar processor performance at the cost of extra complexity required to fetch from multiple threads, control thread execution, and issue from multiple threads.

To achieve maximum performance, SMT workloads are either multiple independent programs or a single program that has been parallelized. A thread in the former case is an entire independent program. In the latter case, a thread is a parallel fraction of the whole program to which all dependencies have been removed by the compiler.

The concept of multithreading is important in hardware for improving processor performance but is also important in the software realm. Software support for multithreading in uniprocessor machines enables applications to respond and perform better by interleaving the execution of several tasks. For instance, special threads are used in applications to manage lengthy I/O or communication requests without blocking the processor.

Software support for multithreading is generally provided at two levels (herein referred to as *software-controlled multithreading*): *user and kernel-level* threads.

At user-level, multithreading is generally made available to programmers through language constructs or collections of library functions. These constructions and libraries supply the methods required to create, execute, synchronize and schedule threads without OS intervention. The Java language, for instance, includes specific methods to create, execute and synchronize multiple threads in applets or applications [6].

Alternatively, support for user-level multithreading has been incorporated into software libraries. An example of this approach is *Pthreads*, a user-level API employed to create multithreaded applications. This API is also known as POSIX threads [15]. Pthreads supports a wide variety of programming constructs for multithreading such as: thread creation and synchronization, mutual exclusion, conditional variables, etc. The Pthreads library for the C language is available on numerous platforms.

The main advantage of user-level multithreading is efficient context switching as threads share the same addressing space. However, the disadvantage of this approach is that the OS is not fully conscious of all user-thread activity. Therefore, the OS will not be able to detect when a thread is jammed and hence can not kill the thread or signal the deadlocked condition to the user-level thread manager. Moreover, the OS could inadvertently degrade application response time by scheduling for execution a task that contains an idle thread.

Operating systems like *Solaris*, *Linux* and *Windows* provide support for *kernel-level* multithreading in which the kernel manages all thread activities. Hence, the OS has more control over the threads at the expense of higher context switching time. This is because each thread has its own addressing space and saving and restoring thread state is expensive.

To obtain the advantages of both approaches, kernel and user-level threads can be combined in a hybrid software multithreaded system. Using this strategy good performance and control over the threads is achieved at the same time. In this multiple-level model of multithreading, several user-level threads are multiplexed to one or more kernel-threads [5].

Software-controlled multithreading at the user or kernel level improves overall application response time on single-threaded processors. However, it is when threads are executed in parallel in either a single multithreaded processor or on multiple single threaded processors that the advantage of multithreading is most noteworthy. Lastly, software-controlled multithreading has become a very important program structuring mechanism for software design. The reason is that many applications are suitable for design as a collection of small tasks or threads. In this *divide and conquer* approach to software design, each task provides one or more services. In comparison, more complex designs use a single large monolithic task to provide all services. For this reason, and for its improvement in performance, numerous client, server, and client-server applications are being designed with multiple threads.

On the hardware facet of multithreading, numerous techniques have been proposed to incorporate support for multithreading in modern processors. Moreover, the recently proposed SMT architecture has been incorporated in new commercial processor designs [62]. The emergence of these multithreaded processors in microprocessor industry indicates that many of current generation processors will adapt or continue adapting multithreading as a performance improving methodology.

This dissertation studies diverse techniques to support multithreading in hardware. In the first part, a simplified model is used to describe how the different layers of support for multithreading impact processor utilization.

Afterward, in the main contribution of this research, special mechanisms are proposed to generate speculatively threads from a single program. Subsequently those threads are executed on a SMT processor core augmented with checking mechanisms that evaluate the outcome of speculation, squashing those threads whose speculation turned to be wrong. Unlike other similar architectures, the *Dynamic Simultaneous Multithreading* (DSMT) processor proposed in this dissertation uses simple mechanisms to keep track of inter-thread dependencies in registers and memory. The following section describes these topics in more detail.

1.1 Scope of the Thesis

New commercial processors like MAJC [65, 67] with multithreading-multiprocessing capabilities; the PowerPC [10] with support for coarse-grain multithreading, and the new Xeon processor from Intel [29] which is based on SMT technology indicate that multithreading is gaining wide acceptance in microprocessor industry.

In these multithreaded systems, the scheduling mechanisms involved at software and hardware levels play an important role in creating high performance systems. The *hybrid multithreaded* model discussed in chapter two provides a high-level, general representation of this type of system. Using a simplified model, the impact of different thread scheduling techniques on processor's performance is evaluated using stochastic simulation.

The hybrid multithreaded model studies the problem of distributing efficiently software threads into hardware contexts which are capable of detecting and tolerating long latency operations without sacrificing performance. For this reason, it surpasses the performance of traditional single-threaded architectures. However, this type of latency-tolerant multithreaded architecture is still limited in performance by the ILP contained within a single thread. Furthermore, other multithreaded architectures such as SMT also suffer from the same limitation in ILP when there is a single task in execution. To avoid this deficiency, a novel architecture, the *Dynamic Simultaneous Multithreaded (DSMT)* processor is proposed in this dissertation. This architecture employs diverse speculative methods to extract dynamically ILP and TLP from sequential programs. Threads are generated from loop boundaries and later executed on a SMT processor core.

SMT is an architectural approach to processor design, which was originally proposed to overcome the limitations of single-thread issue architectures. This technology is very cost effective because it improves processor utilization without adding significant complexity. SMT technology is an evolutionary approach to current processor designs since it is based on augmenting a superscalar core with support for the simultaneous execution of multiple independent threads. The main motivation of SMT is to share processor resources among threads, improving overall processor utilization. Moreover, since modern OS workloads are comprised of multiple independent tasks, these workloads are efficiently executed on SMT. For all these advantages, recent processor designs [19, 29] have adopted the SMT model of multithreading. However, one drawback of SMT architectures is that they do nothing to improve performance when there is only a single program or task available for execution.

SMT and previous research on thread level speculation [37, 41, 44, 64] provide the framework for the DSMT architecture proposed in this dissertation. DSMT augments a SMT processor core with dynamic speculative generation and execution of threads. Therefore, whereas a SMT processor is underutilized when the software system is unable of supplying enough independent tasks, a DSMT processor will still be able of extracting and executing multiple threads dynamically from a single program. Hence, DSMT architectures can overcome one of the main limitations of SMT.

DSMT employs various forms of speculation to extract dynamically TLP and ILP from sequential programs. Unlike other similar architectures, DSMT provides good performance using simple mechanisms to synchronize threads and keep track of inter-thread dependencies, both in registers and memory. The mechanisms proposed for DSMT employ the information obtained during the sequential execution of code segments as a *hint* to speculate the future behavior of multiple threads. Moreover, DSMT utilizes a novel greedy approach that chooses those sections of code that are most likely to provide highest performance based on its past dynamic behavior.

To study dynamic multithreaded architectures such as DSMT, accurate simulation tools are required. These simulation tools must be capable of reproducing the variety of dynamic events that cause performance degradation in speculative architectures such as: intra-thread and inter-thread mispredictions, as well as synchronization, squashing and invalidation of threads. These events occur very often during the execution of multiple speculative threads, and therefore, including their detrimental effect on the overall performance is essential. However, there are only a few simulators publicly available with the capabilities to emulate the dynamic behavior of high performance processors, and such simulators in general are difficult to modify. Unfortunately, developing detailed simulators requires dedicating substantial time during the design, testing and validation phases. However, in spite of all these facts, simulators are invaluable tools to study and evaluate new architectures.

For all the reasons previously mentioned, a new execution-driven procedural simulator called *DSMTSim* was designed and implemented specifically for this research. This simulator is capable of reproducing, in detail, mispredictions occurring during the dynamic speculative execution of threads. For instance, DSMTSim executes misspeculated intra-thread control paths due to branch mispredictions, recovering the state of the processor when a branch is mispredicted. Also, it creates and passes register values *on-the fly* from producer instructions to the consumers at intra and inter-thread levels, squashing those threads that might have used a value too early. Moreover, mispredicted threads are created, synchronized and flushed by DSMTSim at run-time.

DSMTSim is also capable of simulating either a single-threaded wide-issue superscalar processor or the multithreaded version of the DSMT architecture. The fast mode included in DSMTSim allows skipping non-representative initialization sections in programs generated by a compiler derived from gcc.

During the development process of the simulation tools, the *EMSim* simulator was also created. EMSim is an object-oriented simulator for superscalar architectures, which can be extended to simulate more complex processors. Unlike other similar simulators, EMSim provides a modern, modular software architecture that is reusable and easy to modify. Different aspects related to the development of the simulation platform used in this research and the techniques used to evaluate DSMT performance are described in detail in this thesis.

1.2 Organization

This dissertation is organized as follows. Chapter 2 describes the hybrid execution model of multithreading and shows how processor utilization is affected by different thread scheduling policies. Chapter 3 discusses the concept behind the DSMT architecture describing its micro-architecture in detail. Also, in this chapter, previous related research work on dynamic multithreaded architectures is examined. Chapter 4 describes in detail the simulation environment designed to study the DSMT architecture and discusses also some simple techniques that may be used to perform benchmark simulation efficiently. Chapter 5 discusses the results obtained during simulation on the performance of DSMT. Chapter 6 offers a conclusion.

2 THE HYBRID MULTITHREADED EXECUTION MODEL

The hybrid multithreaded model is a software-controlled multithreaded system extended with hardware support to efficiently perform context switching and thread scheduling. The model assumes that threads are generated by the software system, and the processor contains support for multiple contexts.

The central idea of the hybrid model is to depict a system where all the existing features in a software-controlled multithreaded system are used, and at the same time, some of the responsibilities of thread management are handled by the hardware.

In our model, the software thread scheduler selects a *set* of threads for execution onto a multithreaded processor. *Set scheduling* acts as an interface between hardware and software and provides a transparent view to the programmer. The main advantage of the hybrid model is that expensive software context switching and thread scheduling costs occur only when threads are initially scheduled onto the processor. Any subsequent context switching or thread scheduling is implemented in hardware. Over time, this leads to considerable reduction in the overhead cost thereby resulting in high processor utilization.

To illustrate the advantage of having hardware support for multithreading, consider the software-controlled multithreaded execution model illustrated in Figure 2-1. This model assumes a single threaded processor with N software threads ready for execution. Each thread issues a remote reference at an interval of R cycles, i.e., its run-length, and becomes blocked for L cycles waiting for the response to return before resuming execution. L depends on the memory access time and the delay through the interconnection network to and from memory. This model assumes that the processor is able to indicate to the software system, perhaps through an interrupt, that a cache miss has occurred.

Also, in software-controlled multithreading, a context-switch occurs at a cost of C cycles between run-lengths. The cost of thread scheduling is included in the context switch overhead. The processor utilization, U_{SC}, based on this execution model is given by equation (1):

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

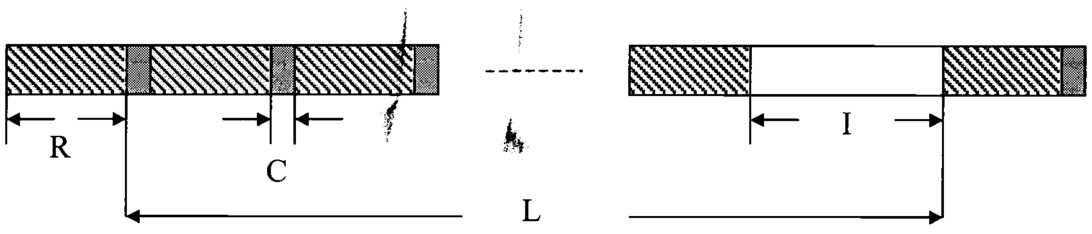


Figure 2-1. Execution model of software-controlled multithreading

If the number of contexts supported is not sufficient, the processor will not be able to completely hide the memory latency L, causing the processor to idle for I cycles (as in the case of Figure 2.1). On the other hand, if there are a sufficient number of contexts, the processor utilization U_{SC} depends only on R and C. The important parameter in this last case is the ratio C/R, which points out that, unless context switching is extremely cheap, the remote reference rate must be kept low.

The model described by equation (1) for software-multithreading was originally proposed by Saavedra *et al.* [54] for a multithreaded processor. In Saavedra's model the multithreaded processor switches context during every cache miss. In the software-controlled model the software is responsible of performing context switching.

As Equation (1) shows, processor utilization is directly related to context switching and thread scheduling costs. In software-controlled multithreading, each thread is associated with a context that contains a thread ID, a set of registers including a PC, a thread priority and a pointer to the stack. Whenever a context switch occurs, a new thread has to be selected (i.e., scheduled) from a pool of ready threads, all the registers associated with the current thread must be flushed onto the stack and registers are loaded with the top frame of the new thread. The *Thread Management System* performs this operation in software automatically, which is expensive. To reduce this cost, the hybrid multithreaded model provides part of these features in hardware to make multithreading as efficient as possible, and providing at the same time, a transparent view to the programmer. The advantage of software-controlled multithreading is that no changes are required in hardware, therefore any processor can benefit from this type of support for multithreading. Figure 2-2 shows the model assumed by the software-controlled model.

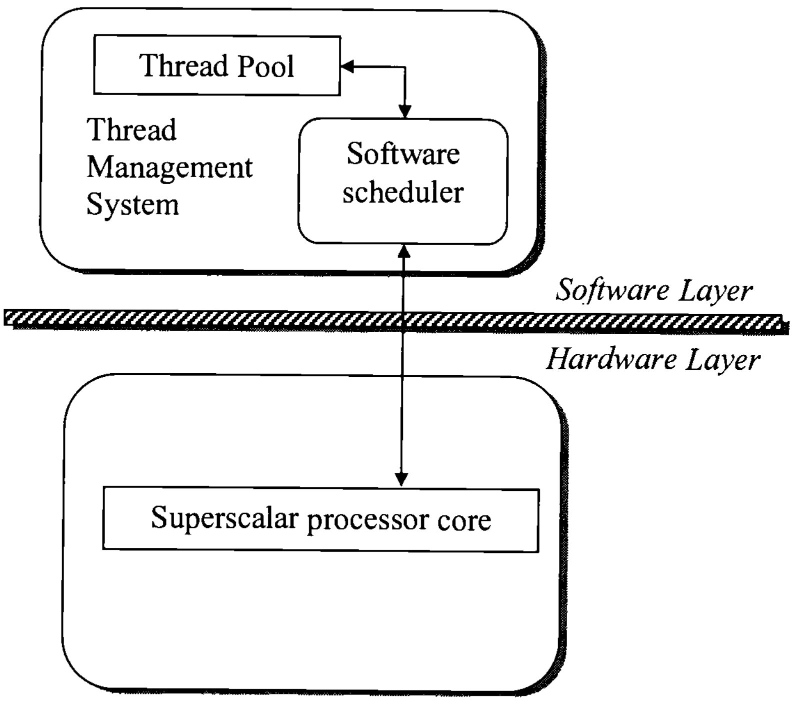


Figure 2-2. Software-controlled multithreading model

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

Figure 2-3 shows the hardware and software schedulers that coordinate thread selection and execution in the hybrid multithreaded model.

The software side of our model consists of an existing *Thread Management System* that contains a *Software Scheduler* that manages *Thread Pool* execution. In most systems, the Thread Pool is implemented as a multi-level priority queue. In these systems, a thread has a priority assigned by either the Thread Management System or the user. The hybrid model assumes that a priority has been given to each thread.

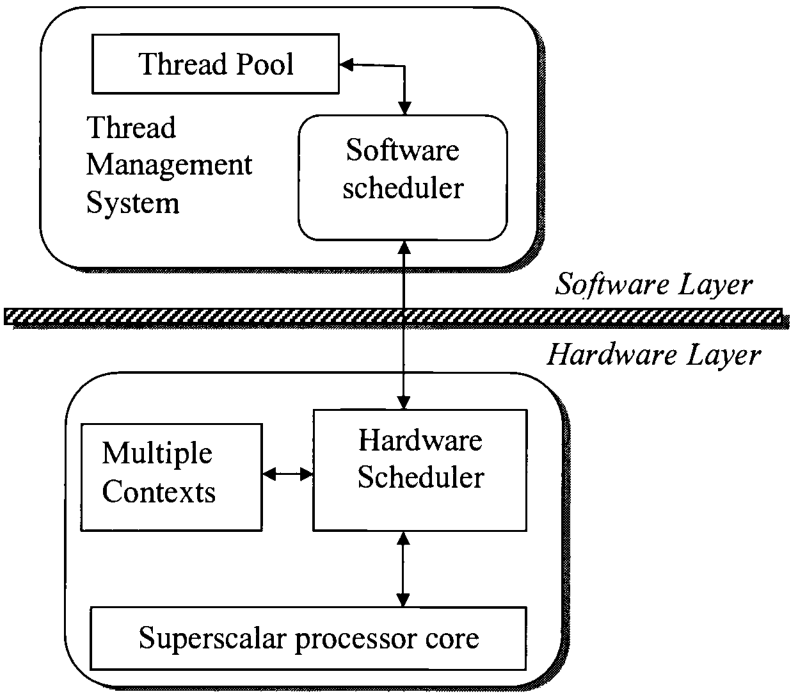


Figure 2-3. Hybrid multithreading model

The responsibility of the Software Scheduler is to select a set of threads from the Thread Pool and schedule them onto the Hardware Scheduler of the processor that supports multiple contexts in hardware. The Software Scheduler, having the objective of maximizing processor utilization, groups threads into *sets*. There are a number of possible policies that can be used to schedule a group of threads onto the Hardware Scheduler. One simple approach is to schedule the next set of threads only after the previously selected threads have completed their execution. This approach is the most appropriate if thread run-lengths are about the same. However, if the thread run-lengths vary other possible scheduling policies are available. We explore these possibilities in Section 2.3.

Hardware support for our model consists of a conventional superscalar processor core augmented with a *Hardware Scheduler* and multiple contexts. Once a thread has been scheduled onto the processor, it can be in one of the following three states: *running*, *ready*, or *sleeping*. Thread state transitions are shown in Fig 2-4.

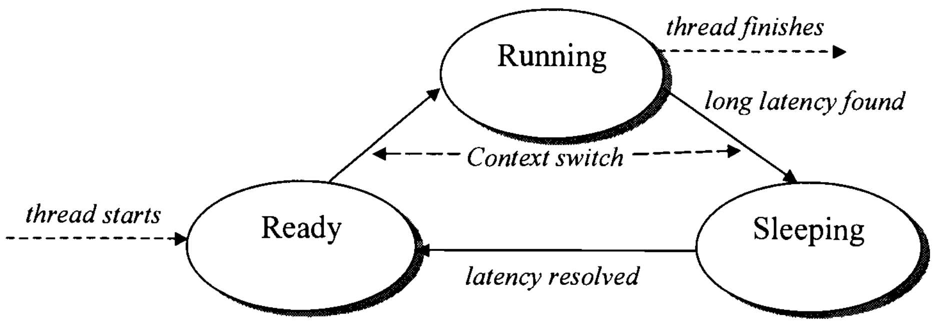


Figure 2-4. Thread state transitions during thread's life

The responsibility of the Hardware Scheduler is to maintain the control of thread states that have been scheduled onto the processor by the Software Scheduler. The *Ready-thread Queue* (*RQ*) and the *Sleeping-thread Queue* (*SQ*), store the threads that are ready for execution or waiting for execution respectively.

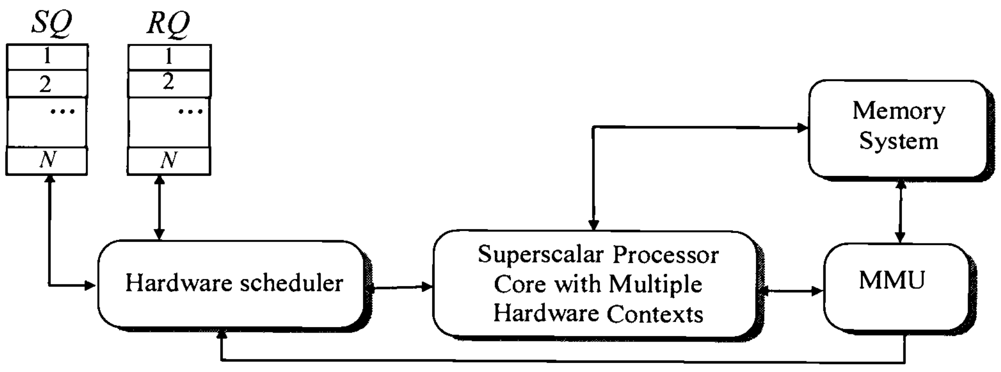


Figure 2-5. Support in hardware for multithreading

A long latency operation detected by the memory management unit (MMU) causes a thread to *context-switch*. The Hardware Scheduler accomplishes this task by placing the blocked thread in *SQ* and immediately scheduling a new thread from *RQ* for execution. In addition to the hardware support shown in Figure 2-5 to keep control of threads during execution, a processor also needs multiple hardware contexts where each thread keeps its own state information. Support for hardware contexts can be implemented in a number of ways. One possible method is to provide separate, fixed-size contexts using a hardware managed register file (in the form of either register windows or duplicated register sets). However, this fixed and inflexible partitioning of the register file results in a waste of scarce high-speed registers. Since the number of registers required by thread contexts varies, a more flexible approach, called Register Relocation, was proposed in [75]. This method relies on the compiler or run-time system to manage the allocation and use of contexts. Instruction operands specify context-relative register numbers, which are numbered consecutively starting with register 0. These context-relative register numbers are dynamically combined (using an OR operation) with a special register relocation mask to form absolute register numbers that are used during instruction execution. Any of these implementations could be used, since the model does not take the implementation of hardware contexts into account.

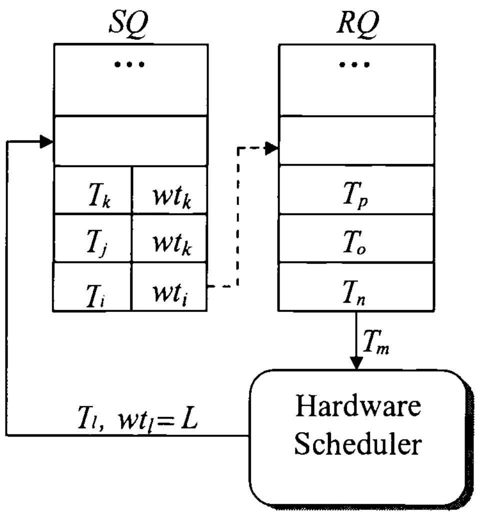


Figure 2-6. Support in hardware for multithreading

In order to manage multiple contexts, a tag *T* containing a thread ID, a PC and a pointer to the thread stack represent each context inside the processor. When the

Hardware Scheduler schedules threads for execution, thread tags are downloaded onto the *RQ*. A thread is scheduled for execution when its tag is de-queued from *RQ*, the stack register is updated and the first instruction pointed to by PC is fetched. When a thread is blocked, its tag is placed in the *SQ* and a context-switch occurs to the next thread in *RQ*. Later, when the blocked thread changes its state to ready, it is en-queued back onto *RQ*. When all the threads from *RQ* (i.e., within a processor) have completed their execution, the Software Scheduler schedules a new set of threads.

In order to keep track of the transition between sleeping and ready threads, each thread, *T*, in *SQ* is associated with a timer, *wt* as is shown in Figure 2.6. When a context switch occurs during the execution of thread T_l, it is sent to *SQ* with *wt* set to *L* and a new thread, T_m, is selected for execution from *RQ*. T_l will remain in *SQ* for *L* cycles waiting for the memory to respond to its request. Eventually, when *L* cycles have elapsed, the Hardware Scheduler will place *Tl* into *RQ* and its state will be changed to ready. To use a timer for controlling thread scheduling must be assumed that L is known. This assumption is valid in UMA single-threaded processor systems with multiple-levels of cache memory, where the latency to access one of these levels is known, but not otherwise.

If *R* and *L* are constant, *SQ* will behave as a FIFO queue and thus each thread will be retired from *SQ* in order. However, these assumptions are not realistic for multiprocessor system because, for example, in UMA machines bus contention will cause *L* to vary. Moreover, in CC-NUMA machines, network contention and the routing algorithm will affect *L*. Variation in memory latency can be handled by mapping cache line tags to *wt*. The Hardware Scheduler then simply identifies threads whose request has been served, and en-queues them onto *RQ*. The hybrid model assumes this method is used when the latency changes stochastically.

2.1 An Analytical Model for Hybrid Multithreaded Execution

An analytical model for the hybrid multithreaded system allows us to study the effect that set scheduling operations have on processor utilization. Figure 2-7 shows the multithreaded execution model through a series of set scheduling operations. During each operation, the Software Scheduler of the Thread Management System schedules N threads onto the RQ at a cost of S cycles, i.e., $S=NC$. Between set scheduling operations, there are a total of G hardware context switches, each with a cost of c cycles, among the N contexts scheduled onto the processor.

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

$$U_H = \frac{NR}{R + L + \frac{NC}{G}} \qquad (2)$$

where G represents the total number of context switches for all the threads. Therefore G/N represents the average number of context switches in a thread. In the second case, the number of contexts is sufficient to hide the latency, thus performance loss comes from the context switching overhead and the set scheduling cost (as in the case of Figure 2-6), i.e.,

$$U_H = \frac{R}{R + c + \frac{NC}{G}} \qquad (3)$$

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

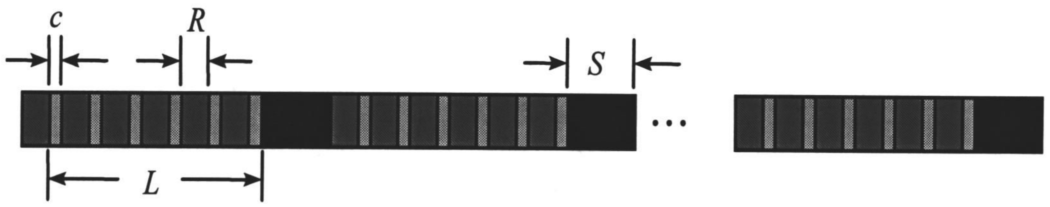


Figure 2-7. Hybrid multithreaded execution model.

The hybrid multithreading model expands Saavedra's model [35] for a multithreaded processor by considering the overhead produced by thread scheduling in software. Also, although our model is still very general, it provides some details on how to support multiple thread scheduling.

In [54], Saavedra finds an algebraic equation that describes how processor utilization changes with context number when the thread run length changes stochastically. However, in that model memory latency changes too, but deterministically. The process to obtain a processor utilization equation consists of building the Petri net that represents the multithreaded execution. Then, the Markov chain is obtained by computing the reachability set of the Petri net. Finally, looking for common patterns in the system reduces the equations found using this procedure. The advantage of this method is that the solution is represented as a single equation. However, the disadvantage of this approach is the complex analysis. Moreover, when the analysis must include one or more extra stochastic variables, a new calculation needs to be performed.

To avoid these hurdles, stochastic simulation was used to study the hybrid model. Using this approach, we were capable of replicating Saavedra's [54] equation graph, which was obtained analytically. Then, the simulator was used to compare our simple analytical model with results obtained by simulation and additionally, to experiment with different scheduling techniques. The experimental process is described in following section.

2.2 Simulation Results

To evaluate the performance of the hybrid multithreaded system described in the previous section, a stochastic simulation study was conducted. The structure of the hybrid model was emulated with a special purpose simulator written in C++. Figure 2-8 shows a simplified UML diagram of the main classes in the simulator.

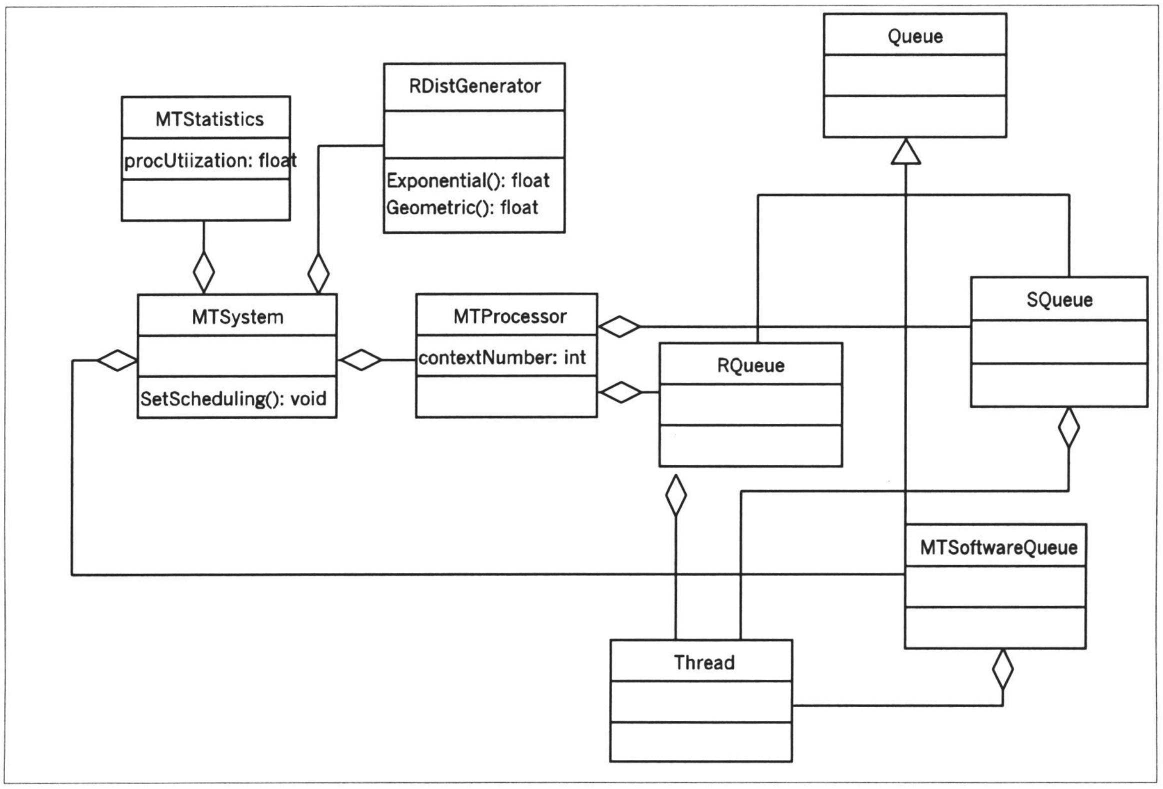


Figure 2-8. UML class diagram of the simulator for the hybrid model.

Simulator class structure is based on containment and inheritance replicating the model structure shown in Fig 2-2. The Thread class contains random number generators that are able to produce different stochastic distributions. These stochastic distributions were used to model different thread run-lengths and memory latency delays. The simulator is also able of generating an output file in Matlab format. Using Matlab and the data file created by the simulator we obtained the graphs described subsequently.

Figure 2-9 compares the theoretical results and the results obtained from our simulation for the hybrid multithreaded model on processor utilization when *R* and *L* are constant average values and *C* is changed. Plots were obtained by running 1,000 threads with *R*=100 cycles, *c*=2 cycles, and *L*=500 cycles.

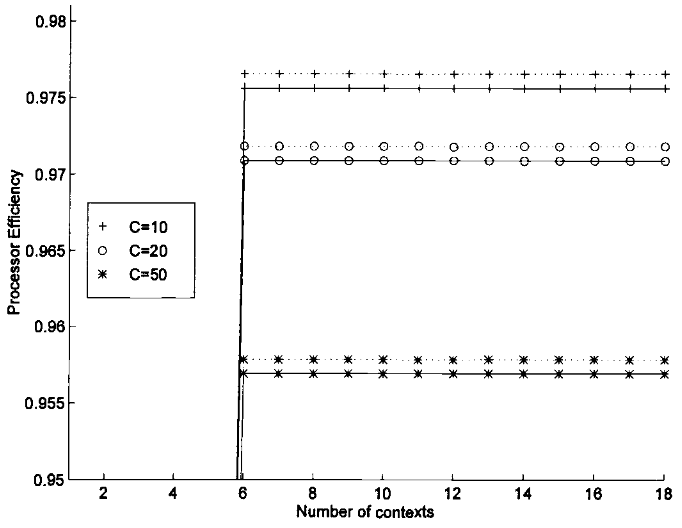


Figure 2-9. Comparison between theoretical (solid lines) and simulation (dotted lines) results.

The graph shows that the simulations results were comparable to the theoretical results from equations (2) and (3). More importantly, as *C* increases from 10 tq 50, processor utilization only decreases by approximately 2%. The primary reason for this is that the set-scheduling cost is incurred only once and all subsequent context switches are done in hardware. Therefore, the hybrid method is more immune to variations in *C*.

To obtain a more realistic evaluation of our hybrid model, stochastic distributions were considered for *R* and *L*. In this case, variations on run-lengths model cache misses that provoke context switch. Variations on L model the behavior of multiprocessing systems where cache access is not deterministic.

Figures 2-10 and 2-11 show the effects for both the hybrid and software-controlled models of stochastic distributions for R and L. *R* was modeled with a geometric distribution, and *L* with a negative exponential distribution. Our simulation results were obtained by running 1000 threads with an overall execution time of approximately 500,000 cycles.

Figure 2-10 compares the performance when *R* has a mean value of 100 cycles, *L* has a mean value of 500 cycles, and *c* = 2 cycles. Results show that not only does the hybrid model outperform its software-controlled counterpart, but also, because it is more immune to variations in *C*, the performance (i.e., processor efficiency) gap widens as *C* increases. Our findings also indicate that the performance of the software-controlled execution model is strongly affected by the granularity of threads. This result is shown in Figure 3.2b where *R* has a mean value of 20 cycles, *L* has a mean value of 100 cycles, and *c* = 1 cycle. When *C*/*R* is large, the performance of the software-controlled model is affected severely by the software scheduling and context switching costs.

Another interesting observation is that when thread run-lengths vary, the utilization goes down (see Figures 2-9, 2-10 and 2-11). This is because when total thread run-lengths are the same, all threads complete their execution at about the same time. Therefore, scheduling the next set of threads only after the previous set of threads has completed execution will cause minimal idling. However, when thread run-lengths vary, some threads will complete first reducing the number of threads from which to context switch.

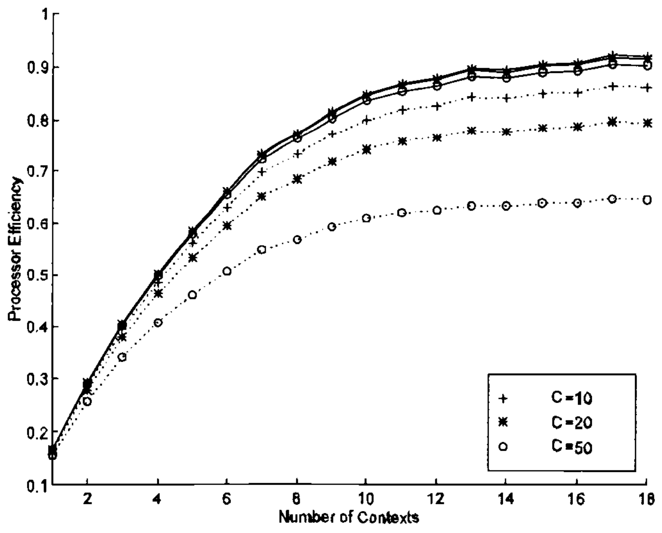


Figure 2-10. Comparison between hybrid (solid lines) and software-controlled (dotted lines) execution models: R = 100, L = 500, and c=2.

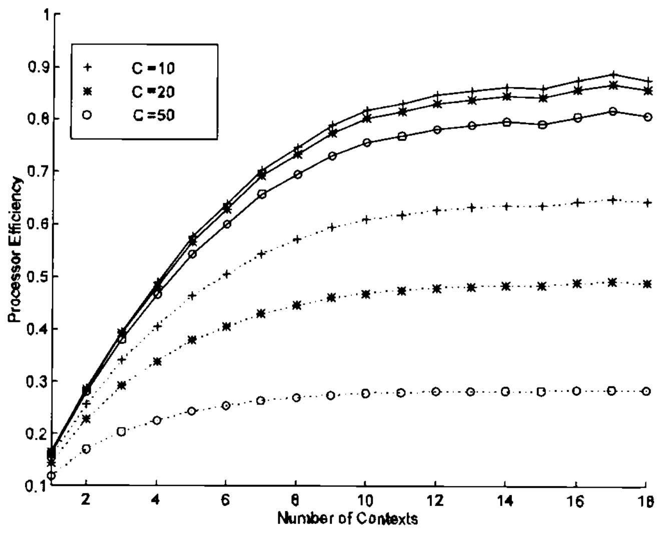


Figure 2-11. Comparison between hybrid (solid lines) and software-controlled (dotted lines) execution models: R = 20, L = 100, and c=1.

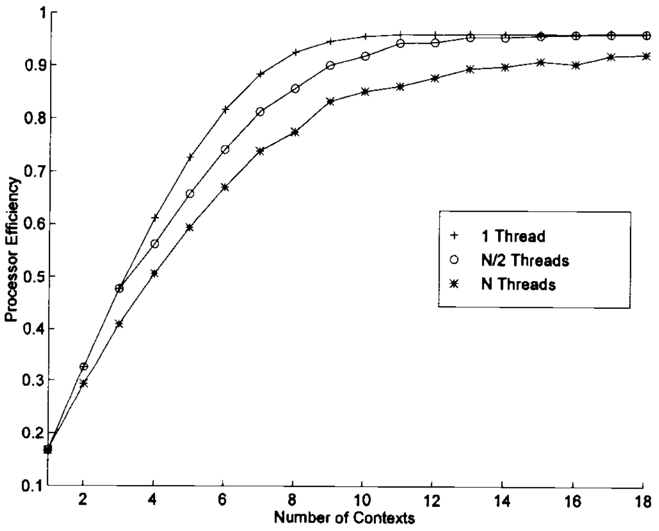


Figure 2-12. Effects of scheduling policies when C=10, R=20, L=100 and c=1.

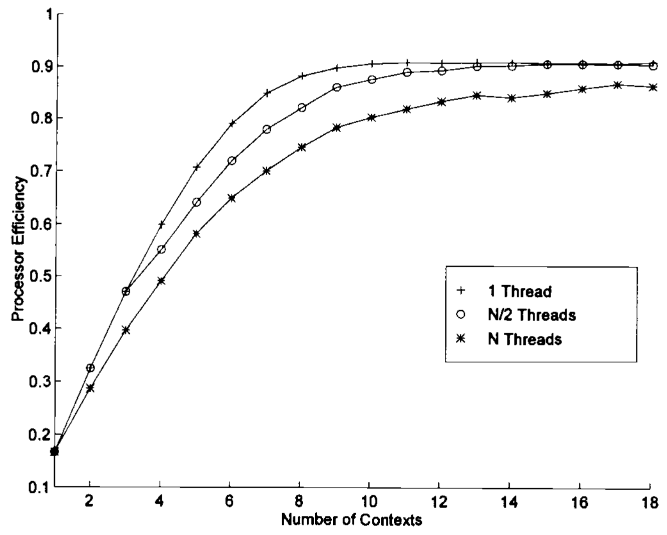


Figure 2-13. Effects of scheduling policies when C=20, R=20, L=100, and c=1.

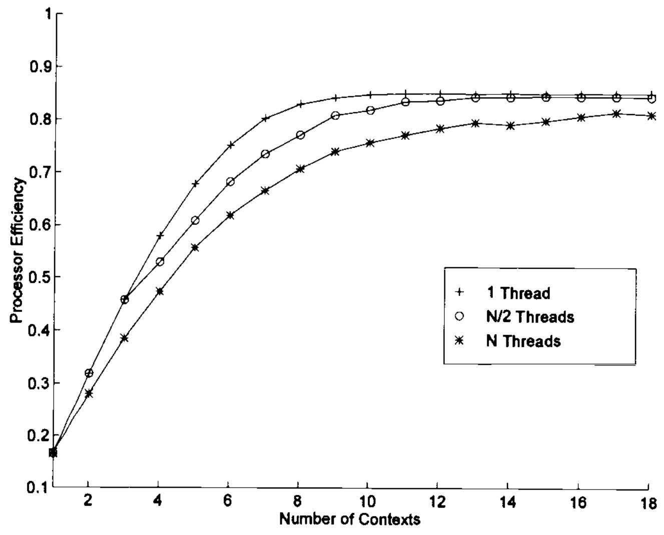


Figure 2-14. Effects of scheduling policies when C=50, R=20, L=100, and c=1.

To determine how to overcome this deficiency, different scheduling policies were tested. Those policies were:

(1) A new thread is scheduled immediately after a thread completes its execution

(2) Schedule *N*/2 new threads when *N*/2 threads complete their execution.

Results of these two scheduling policies were then compared against scheduling *N* new threads when *N* threads finish their execution.

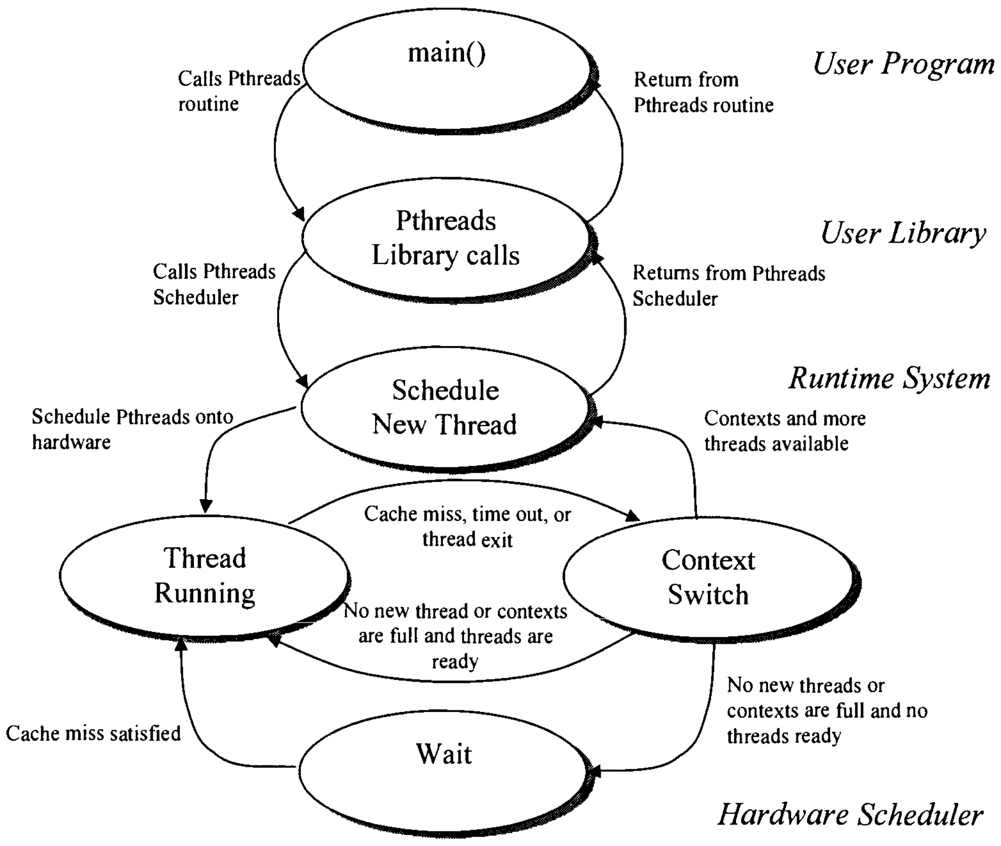

main()
Calls Pthreads routine
Return from Pthreads routine
User Program
Pthreads Library calls
User Library
Calls Pthreads Scheduler
Returns from Pthreads Scheduler
Schedule New Thread
Runtime System
Schedule Pthreads onto hardware
Contexts and more threads available
Cache miss, time out, or thread exit
Thread Running
Context Switch
No new thread or contexts are full and threads are ready
No new threads or contexts are full and no threads ready
Cache miss satisfied
Wait
Hardware Scheduler

Figure 2-15. Thread states from the user-level to the hardware level.

Processor utilization results are shown in Figures 2-12, 2-13, and 2-14 when these different scheduling policies are used for various values of *C*. In these graphs, *R* was modeled by a geometric distribution with a mean value of 20 cycles, *L* by a negative exponential distribution with mean value of 100 cycles, and c=1 cycle. These results show that, for all three values of *C* it is always better to schedule a new thread immediately after a thread completes its execution. Thus, scheduling one at a time will minimize idling due to lack of threads from which to context switch. Using the same general idea of the hybrid model, the simulator for the *Multithreaded Virtual Processor* (MVP) model described in [34, 35] studied how user-level multithreading affects L1 instruction cache miss rate due to interference between threads. Also research on MVP concluded that context switching to hide cache latency and data sharing between threads improves performance. The hybrid model uses the high-level simplified scheme depicted in Figure 2-2 to show how software and hardware schedulers interact. In contrast, MVP provides a very detailed description of how these two schedulers interact. This scheme is depicted in Fig 2-15. Figure 2-15 shows that the runtime system, which is functioning as a small kernel, schedules a new thread onto a hardware context when the processor indicates that a context is available. This policy was used in MVP after we learned from hybrid model simulations that scheduling threads in this way produces the highest processor utilization. To evaluate the performance of the MVP model, benchmark programs were required. Research on SMT or single thread architectures generally uses the SPEC95 benchmarks or the new SPEC2000 benchmarks to evaluate processor performance. In the parallel processing arena, SPLASH-2 benchmarks are used to test parallel architectures [80]. Unfortunately, there are no equivalent benchmarks to perform evaluation of multithreaded architectures. For this reason a group of multithreaded benchmark programs was written based on SPLASH-2 benchmarks using Pthreads. As reported in [34], these benchmarks were used to study MVP performance. To simplify the task of writing the multithreaded benchmarks, the SPLASH-2 suite of programs originally written for shared-memory machines was transformed to use Pthreads calls instead of the original ANL macros [38]. To perform this task, some primitives not available in Pthreads, like barrier synchronization, were added to emulate their counterpart in ANL macros. The multithreaded benchmark programs written were: FFT, MP3D and Radix Sort but all other SPLASH-2 benchmarks can be converted to their multithreaded version by running a macroprocessor with the appropriate macro conversion file. This file contains the translation of ANL macros to Pthreads calls. The conversion process is depicted in Fig 2-16. Additionally to the benchmarks mentioned above, new versions of Matrix Multiplication, Gaussian elimination and Sieve algorithms were manually written to be multithreaded and included in the benchmarks. No optimizations were attempted when converting the serial handwritten programs or the parallel SPLASH-2 benchmarks. In spite of their advantages to tolerate long latencies, the vertical multithreading model used by MVP and the hybrid model described in this chapter have limited capabilities to improve performance. The main reason for this is that only a single thread is issuing instructions at a time. Therefore, processor resources are underutilized when there is not enough ILP to exploit in a particular thread.

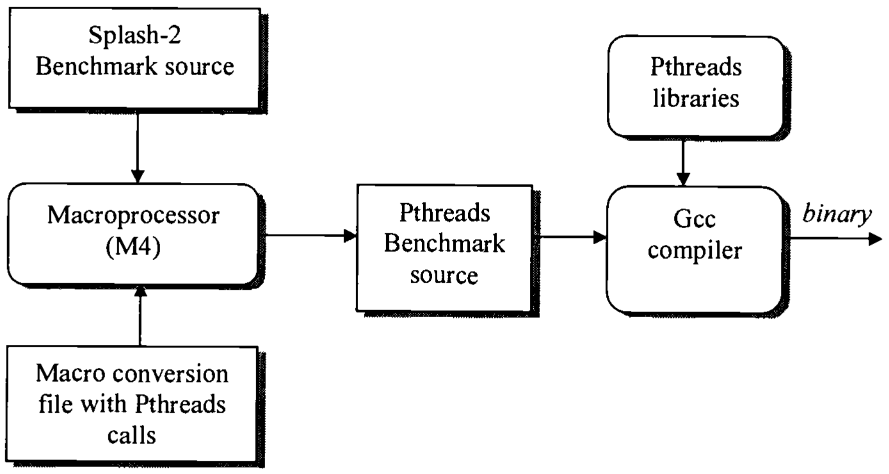

Splash-2 Benchmark source
Pthreads libraries
Macroprocessor (M4)
Pthreads Benchmark source
Gcc compiler
binary
Macro conversion file with Pthreads calls

Figure 2-16. Conversion of benchmarks to multithreading version.

Dynamic speculation and simultaneous multithreading are architectural approaches to exploit effectively TLP and compensate for the lack of ILP in a single thread. The DSMT processor described in the next chapter uses these techniques to overcome the limitations of single-thread single-issuing architectures.

3 THE DSMT PROCESSOR

3.1 Motivation

Improving performance has been the main goal of most processor designs and is also the goal of the DSMT processor. However, it is interesting to note that recent research [11] has recognized that performance should not be the only goal of modern computer system design. Other features in a computing system such as software and hardware failure tolerance and recovery, which were partially addressed in the past, are becoming increasingly important, especially for general-purpose computer systems.

Processor performance is measured by the total execution time consumed executing a program. Total execution time is calculated by the following equation.

$$T_e = T_c \times N \times \frac{1}{IPC} \tag{4}$$

Where T_e is total execution time, T_c clock cycle time, N is the total number of instructions in a program and IPC the number of *instruction executed per clock cycle.*

Equation (4) indicates that by reducing T_c (using a faster clock) or N (compressing several instructions into one) the total execution time of a program will be reduced. Therefore, is not surprising that most recent improvements in current superscalar processors have been obtained applying the most recent advances in VLSI technology aimed at reducing T_c. In these processors, a very short clock cycle allows instructions to move quickly through the pipeline. However, with faster frequencies the problems of clock generation and distribution in synchronous designs are exacerbated. Also, with faster clocks the amount of processing that a pipeline stage is able to perform in a single cycle decreases. Therefore, to compensate for this effect, deeper pipelines are required in fast processors. However, in deep pipelines mispredictions become more expensive since when they occur pipeline stages must be completely flushed and then, refilled with new instructions. This flush/refill process consumes additional clock cycles as the length of the pipeline increases. Finally, it is known that future improvements obtained using the fast clock approach will reach a limit in the long-term due mainly to physical and technological constraints [30].

The other factor in Equation (4), the IPC, represents the average amount of ILP exploited by a processor during every clock cycle. Increasing the value of this ratio is the aim of mechanisms proposed to exploit ILP and TLP. IPC depends on a number of factors including the inherent parallelism in a program, the size of the instruction window, the issue width and other characteristics of the scheme used to extract parallelism. Higher values in IPC can be obtained by using more complex issue logic. However, increasingly complex issue logic can limit the clock speed of the microarchitecture [49,58].

Dynamic speculative techniques of TLP have been proposed to improve processor performance [62]. In dynamic multithreading, threads are generated speculatively at run time. These speculative threads are constantly monitored to detect when a speculation has mispredicted a value. When this happens, the offending thread is squashed and execution resumes at the point of misprediction.

A number of methods for dynamic thread generation have been proposed in literature [3, 70, 82]. Most of the proposed methods are based on detecting one or more of the following conditions:

- Loop boundaries (*loop-iteration*)
- Procedure calls
- Exception handling routines
- Continuation code after a loop body (*loop-continuation*)
- Continuation code after a procedure (*procedure-continuation*)

Loops and procedures are programming structures used very frequently in programs. For this reason, these structures are good candidates to create threads. Numerical applications based mainly on matrix calculations contain many loops with several nesting levels. Moreover, structured programming and modern OO languages that provide encapsulation mechanisms have encouraged the use of procedures or methods through the use of *get* and *set* methods.

On the other hand, previous research [32] reported that half of total execution time of some benchmarks is consumed inside loops. Later, [70] found that from the total number of instructions executed in SPEC95-FP benchmarks, 64% on average correspond to instructions from loops. However, this same percentage for SPEC95-Int benchmarks is 30% on average. Hence, optimizing loop execution is fundamental for improving total execution time in many programs, but especially in applications that perform numeric calculations. This fact has been recognized by the plethora of research done into parallelizing compilers [9, 21 and 78].

From all proposed methods of extracting threads, loops are very challenging programming structures. This is because, very frequently, they contain dependencies that serialize loop execution. These dependencies are called *loop carried dependencies*, since variable values are communicated between iterations. Loops that contain loop carried dependencies are called *do-across* and those who do not are called *do-all* in the literature. Exploitation of do-all loops produces better performance because threads are independent and can execute in parallel. On the contrary, do-across loops cause threads to get blocked when a value produced by a previous iteration is needed by the current iteration, serializing the execution of a loop. Additionally, for do-all and do-across loops, the dynamic behavior produced by internal conditional branches may complicate executing efficiently its iterations in parallel. Moreover, in nested loops an optimal thread selection mechanism must decide which nesting level is likely to provide better performance. Inner loops provide parallelism at a finer grain compared with outer loops. All these factors make exploitation of loops for TLP very challenging.

Certainly, thread generation based on loop iterations do not preclude that one or more of the other methods be also used in conjunction. However, using several methods to generate threads requires more complex mechanisms to generate and control multiple thread execution.

On the other hand, in spite of the multiple advantages that the SMT architecture offers for processor design, one of its drawbacks is that this architecture does nothing to improve performance when there is only a single program or task available for execution. To alleviate this hurdle, we propose the DSMT architecture.

The goal of DSMT is obtaining high IPC through the dynamic exploitation of TLP and ILP on a SMT processor. DSMT enhances a SMT processor core with dynamic speculative generation of threads. Threads are generated from a single program without compiler help. Therefore, DSMT overcomes SMT limitations in performance when the software system is unable of supplying more tasks.

To achieve its goal, DSMT multithreaded model uses speculation at several levels. DSMT, like many superscalar processors, uses intra-thread speculation to predict branches, perform out of order loading of values from memory and to predict the return address of subroutines. Speculation is also used to generate threads, predict the number of iterations in a loop and to predict register and memory dependencies.

Unlike other similar architectures, DSMT uses simple mechanisms to synchronize threads, and keep track of inter-thread dependencies both in registers and memory. The mechanisms proposed in DSMT employ the information obtained during single thread execution as a *hint* to speculate the future behavior of multiple threads. Moreover, DSMT utilizes a greedy approach that chooses those sections of code that are more likely to provide highest performance based on its past dynamic behavior.

The following section describes previous research in academia and industry on some multithreaded architectures closely related to DSMT.

3.2 Previous Related Work

The proposed architecture was drawn from a plethora of related works on exploiting both ILP and TLP from a single program. These proposals share many similarities on how thread level speculation is supported and basically differ on how much of this support is provided in hardware versus software.

One of the first proposals to overcome limitations in ILP was the *Multiscalar* processor [61]. The Multiscalar architecture increases the instruction window size of a processor using multiple tasks extracted from a program. In this architecture, a special compiler or binary annotator divides a program into tasks that are executed on processing units. Then, a special sequencer mechanism determines task order execution and assigns the tasks to processing units. Processing units are connected using a unidirectional ring topology that forwards information from one unit to the next.

A couple of other related proposals are the Trace Processor [53] and a technique called Threaded MultiPath Execution (TME) [76]. TME is based on a technique called Disjoint Eager Execution, which allows a processor to speculatively execute down multiple branch paths in a program. TME as well as DSMT are variations of SMT. TME executes speculatively both possible paths of conditional branches that have been difficult to predict, when there are fewer threads than hardware contexts. Once the correct result of a branch is known the incorrect path is flushed from the pipeline. TME shows a performance improvement of 14%-23% on average for programs with high misprediction rates.

The Trace Processor improves the instruction fetch bandwidth and thus increases the size of the Instruction Window by capturing dynamic instruction sequences of fixed length (i.e., a trace line). This concept has been also used to execute multiple trace lines simultaneously through a modified superscalar processor [74].

In the following sections a brief description of other architectures related to DSMT is provided.

3.2.1 SMT

SMT architecture allows multiple independent threads to issue multiple instructions simultaneously to the execution units without any context switching delay. This is the main characteristic that differentiates SMT from other types of multithreaded architectures.

In SMT, multiple threads compete for and share available processor resources every clock cycle. TLP and ILP are used interchangeably in SMT, accommodating variations in TLP and ILP. Greater throughput and effective use of functional units is achieved by exploiting whichever type of parallelism exists. Figure 3-1 depicts the architecture of a SMT processor.

In [73], it was reported that SMT compares advantageously with wide-issue superscalar processors and chip multiprocessor systems (CMP). In the first case, SMT provides better resource utilization than a wide-issue processor because TLP is used to compensate for the lack of ILP in a single thread. In the second case, SMT performs better than CMP because it is able to select dynamically the appropriate resources required by the execution of multiple threads. In contrast, in a CMP system, resources are statically allocated once threads are scheduled for execution; therefore, lack of ILP in a thread causes resource under-utilization in a processor.

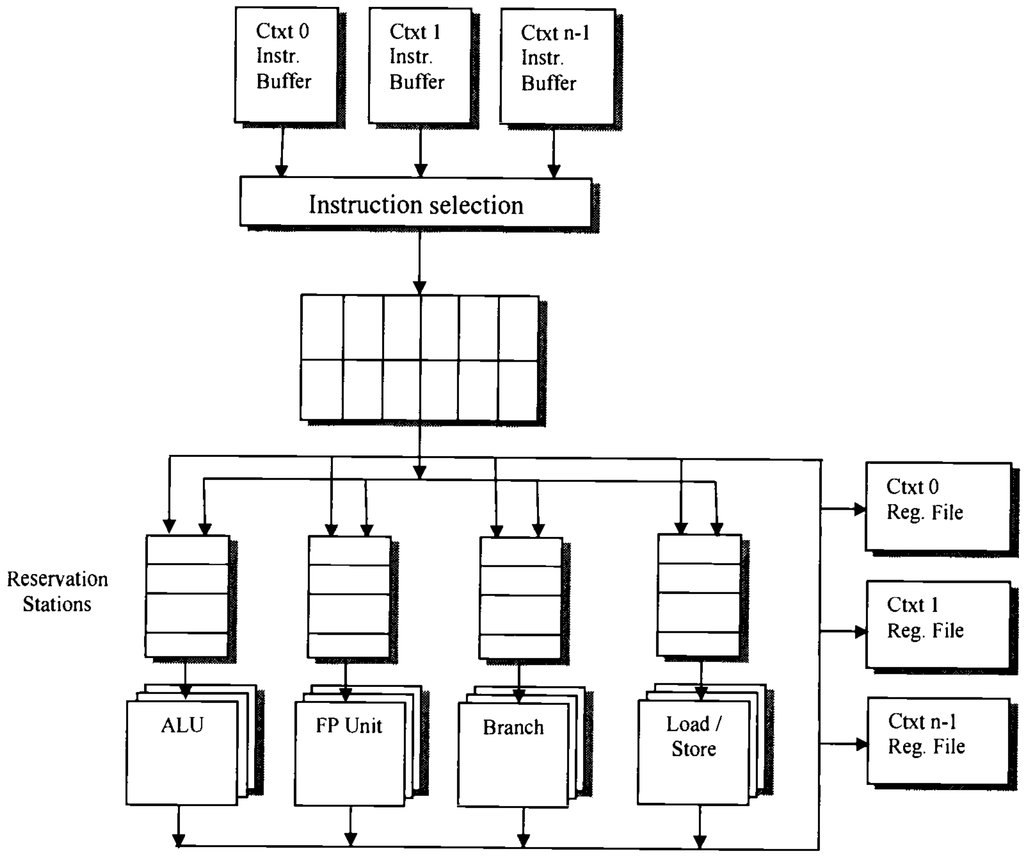


Figure 3-1. SMT architecture

To support SMT the hardware context of a superscalar processor is replicated and also per-thread mechanisms for pipeline flushing, instruction retirement, trapping, precise interrupts and subroutine return stack are added. Additionally per-thread address identifiers in BTB and TLB are needed. Lastly, the fetching mechanism is redesigned to allow fetching from multiple threads. Adding support for SMT on a superscalar processor has a small impact on processor performance. In [73], it was found that single thread performance in a SMT processor is around 1.5% worse than that of a single-thread wide-issue superscalar processor. The reason for this is because two extra stages are added to the SMT pipeline to compensate for a larger register file that requires longer access time.

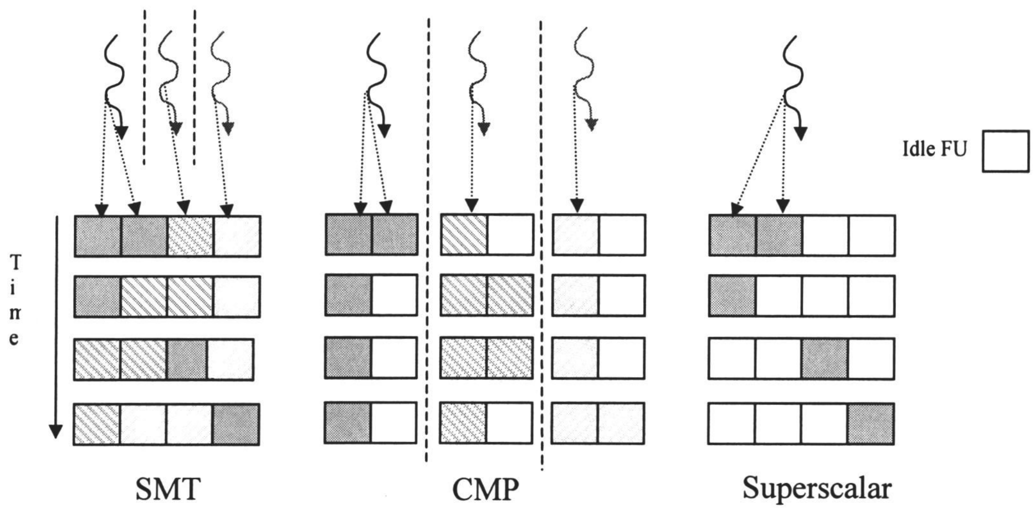


Figure 3-2. Functional unit utilization in several technologies.

Figure 3-2 shows how functional unit utilization improves on SMT with respect to several other technologies.

As Figure 3-2 indicates, processor utilization in SMT is higher that in CMP; more functional units per clock cycle are busy executing instructions. This is because resources are statically partitioned in CMP and therefore, threads scheduled for execution in a processor will be limited to the ILP contained in those threads. In contrast, in SMT, resources are dynamically assigned to each thread and therefore, if a thread lacks ILP, the ILP contained in other threads can compensate for this effect.

Functional unit utilization is also higher in SMT than in the coarse or fine grain multithreading models. The cause is that in both cases, coarse and fine grain multithreading, only one thread can issue instructions and therefore utilization is limited to the ILP contained in a single thread.

Figure 3-3 shows an example comparing functional unit utilization in fine and coarse grain multithreading. In this example, when the number of threads is the same, the efficiency of fine and coarse grain multithreading is comparable. However, in general, support for fine grain multithreading is less complex but its performance is significantly lower than coarse grain multithreading during single thread execution.

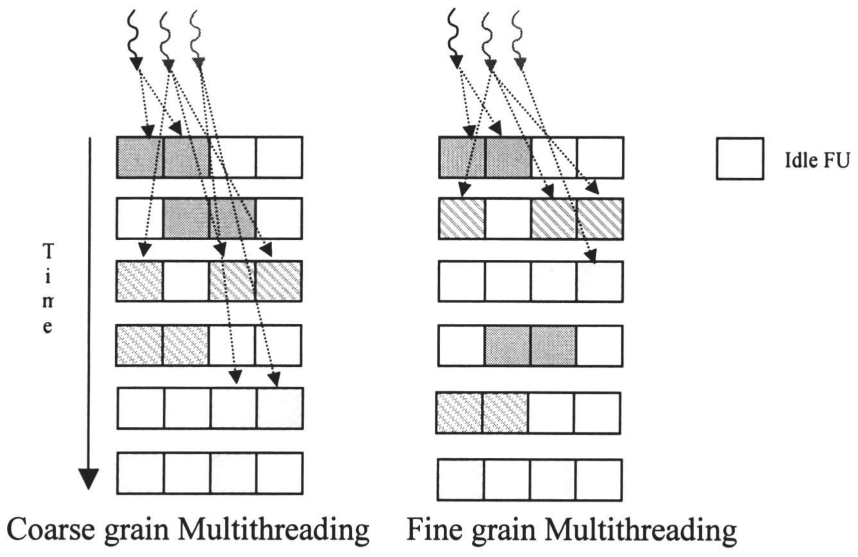


Figure 3-3. Resource utilization in fine and coarse grain multithreading.

Efficient utilization of processor resources is one of the main advantages of SMT over other multithreading architectures. However, SMT has also some potential problems. When there is only a single thread of execution or the number of threads is less than the number of contexts, the resources required to control multiple-thread execution are wasted.

In SMT, resource contention caused by sharing many of the hardware structures among all independent threads, can lead to competition of those resources, producing potentially more cache misses and branch mispredictions. Also, sharing the cache hierarchy among multiple programs limits the efficiency of these resources in comparison to running a single program.

Additionally, [25] found that the design of the memory hierarchy is an important factor for SMT performance. Specifically, it was found that caches must be associative, and L1 cache size should be large enough so that contention in L2 does not degrade performance when a small L1 cache is used. Also, [24] found that for four or more contexts the gain obtained using an out-of-order engine is not cost effective in SMT.

From a software perspective, SMT looks like a multiprocessor system. This means that OS and user programs can schedule processes or threads to logical processors as they would do on conventional physical processors. Therefore, OS changes to schedule programs into the multiple processor contexts are minor, since most modern OSs already support multiprocessor systems [51]. From a microarchitecture perspective, instructions from these logical processors will persist and execute simultaneously on shared execution resources.

DSMT, like SMT, activates simultaneously multiple contexts to exploit TLP. However, unlike SMT, DSMT identifies and executes threads dynamically from a single program.

3.2.2 DMT

Dynamic multithreading (DMT) generates threads dynamically at run time and is capable of executing in parallel a loop, a procedure and the code after the procedure [3]. DMT is designed around a SMT processor core.

This architecture employs trace buffers to keep the state of the speculative threads. In case of an incorrect speculation, a recovery sequence is initiated, re-dispatching misspeculated instructions for execution. Special *trace buffers* located out of the main pipeline, enable DMT to keep larger instruction windows compared to a superscalar processor.

In DMT the hardware breaks up a program into loops and procedure threads that execute simultaneously on a superscalar processor. Data speculation on the inputs to a thread is used to allow speculative threads start its execution immediately.

Control logic keeps a list of the order and the start PC of each thread. A thread is forced to stop fetching instructions when it reaches the start of the next thread in the order list. However, if for some reason a thread never reaches the start of the next thread in list order, is considered mispredicted and consequently squashed.

Threads communicate through registers and memory. Communication between threads is one way only and dictated by their order. A new thread uses as input the register contents of its parent thread. Loads are issued to memory speculatively assuming that there are no dependencies with stores from previous threads. Data misspeculation is common, since threads do not wait for their inputs to be ready. DMT uses selective recovery on mispredicted instructions as soon as the correct input is available. Trace buffers outside the main pipeline are used to hold all speculative instructions and their results. During recovery, instructions are fetched from the trace buffers and re-dispatched into the execution pipeline.

3.2.3 Superthreaded Architecture

This architecture integrates compilation and hardware support to exploit TLP and ILP in programs [68, 69]. Thread level control speculation is done during compilation and data-dependency checking is performed at run-time.

The compiler partitions the control flow graph of a program into threads. The granularity of the threads is typically one or several iterations of a loop. A thread forks successor threads on other processing elements until all thread processing units are busy. The execution of a thread is partitioned into several stages, namely, continuation, target-store-address-generation (TSAG), computation and write-back. When a thread finishes performing the tasks of one stage, data is passed to the next thread, which in turn does the same operation.

In the continuation stage, recurrence variables such as loop indices are computed. These values are forwarded to the next thread before its thread activation. A fork instruction at the end of the continuations stage causes the next thread to begin.

In the TSAG stage, the addresses of the target stores are calculated. These addresses are forwarded to all concurrent threads. In the computation stage, the main computations of a thread are performed. If the address of a load matches some of the entries in the store buffer, data is either, read from the entry if is available or the thread will wait until the data arrives from an earlier concurrent thread. If the value of a target store is computed during this stage, the current thread will forward the data and address to all successor concurrent threads.

To maintain correct memory state, concurrent threads must perform their write-back stages in their original order. Threads wait for a flag from the predecessor thread before performing its writeback stage. This flag indicates that a thread can perform its writeback stage once the previous thread has done it, preserving the in-order state of memory.

3.2.4 Speculative Multithreaded Architecture

Speculative multithreaded architecture (SMA) [39, 41] consists of several thread units (TU) that concurrently execute different threads of a sequential program. These threads are dynamically obtained by a control speculation mechanism based on identifying loops and executing speculatively different iterations of a loop. Thread units are interconnected through a ring topology, and iterations are allocated to thread units following their execution order. Each thread unit has its own physical register file, register map table, instruction queue, functional units, local memory and reorder buffer.

Inter-thread data dependencies through registers and memory are predicted with the help of a history table called a *loop iteration* table.

When a speculative thread is created, its logical register file and its register map table are copied from its predecessor. A live and predictable register will be initialized with the instruction `add Ri,Ri,stride` which is not part of the static code. Another table is used to store the number of remaining writes to a register. That table is initialized with the number of writes performed by the last iteration of the loop. If an instruction having register `Ri` as its destination is retired, the corresponding entry in the register write table is decreased. When this entry becomes zero, the result of the instruction is written in the *i*-th entry of the live-in register file of the succeeding thread unit. When this event occurs, the instruction from the next thread (that was stalled waiting for the value) can continue execution.

Inter-thread memory dependence speculation is performed by means of a *multi-value* cache. This cache memory stores, for each address, as many different data words as the number of thread units. For each store whose base register is predictable a special instruction (not part of the static code) is inserted in the instruction queue. That instruction adds the register with the corresponding offset (taken from the loop iteration table). Figure 3-4 shows the microarchitecture of a SMA processor.

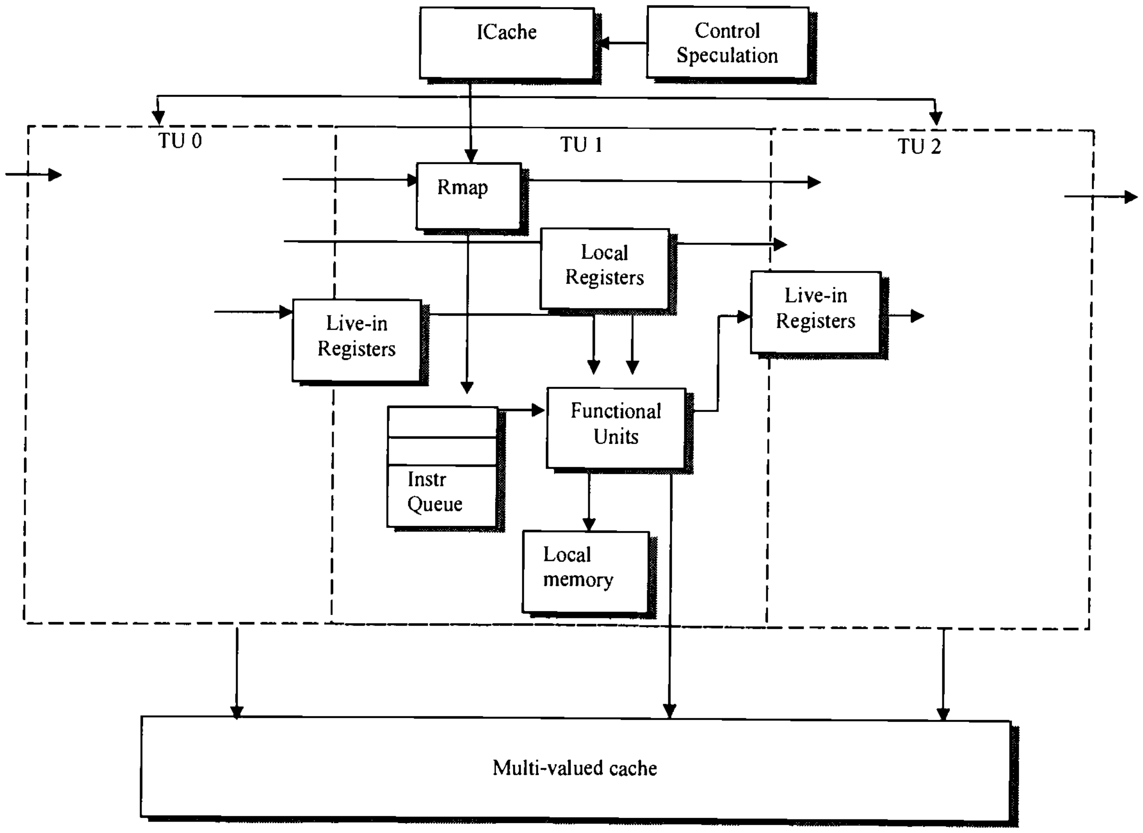


Figure 3-4. Speculative Multithreaded Architecture

3.2.5 Multithreaded Commercial Processors

3.2.5.1 MAJC

The MAJC [65, 67] processor is based on a variable length VLIW architecture. The MAJC processor unit consists of four functional units, each of which with its own set of registers. In the MAJC architecture, unlike a traditional architecture, any functional unit can operate on any type of data. Also any MAJC register can hold any type of data. These features allow efficient use of processor resources.

Multiple processor units are organized on the same chip in a *processor cluster.* Using this architecture, a multithreaded program would distribute its threads among the various processor units in a cluster.

Speculative execution of threads is supported in MAJC. If a thread is executing and it hits long execution time operations such as loops, under certain conditions, MAJC can spawn off an entirely new speculative thread. This thread will execute the instructions following the end of the loop boundary. The thread can be allocated to another processor entirely, so that the non-speculative thread is not slowed down. The new spawned thread uses its own set of registers, which eliminates inter-thread, WAW and WAR hazards. During execution, both threads communicate in real-time, so that the speculative thread knows the results of the non-speculative thread execution. In case RAW hazards occur, the speculative thread is rolled-back and its execution is restarted again. This technique is called *Space-Time Computing* (STC).

In addition to STC, MAJC uses vertical multithreading; when a thread is executing and a cache miss is detected, a context switch occurs. In this case, another thread is executed while the blocked thread is waiting for the data to load from memory. MAJC was designed only as a coprocessor.

3.2.5.2 Intel Hyper-threaded Technology

Intel [29] has announced a new generation of processors based on hyper-thread technology, which is very similar to SMT. These new processors are based on the IA-32 architecture, which has been extended to support SMT. One hyper-threaded processor has two *logical* processors in the same chip, each keeping its architectural state comprised of data, control, segment, debug registers and its own programmable interrupt controller.

Each logical processor can be individually halted, interrupted, or directed to execute a specific thread, independently from the other logical processor. Both logical processors share the execution resources of the processor core, which include functional units, the caches, the systems bus interface, and the firmware. Instructions from both threads are simultaneously dispatched for execution by the processor core. Later, both threads are executed concurrently using out-of-order dynamic scheduling to efficiently utilize functional units each clock cycle.

Intel hyper-threaded processors will appear to the software as two independent processors. Therefore, OS will run unmodified if it supports multiprocessing, as is the case of most contemporary OSs. A special instruction called *CPUID* is used to detect the presence of hyper-threaded processors. According to Intel a hyper-threaded processor can provide a performance gain of up to 30% compared to a non hyper-threaded processor.

3.2.5.3 Alpha EV-8

EV-8 is an 8-way issue SMT processor [19]. Four threads are supported by the architecture in Thread Processing Units, sharing the internal resources of the processor. On every cycle the fetch unit fetches eight instructions form each of two threads that are not currently processing an instruction cache-miss. Instructions are selected for dispatch form the first thread until either a branch or an end-of-cache line is encountered, at which time instructions from the second thread are selected. The two threads used for fetching are selected according to ICount policy [72] that gives priority to the fastest-moving threads. ICount also prevents thread starvation, since threads with fewest instructions in the pipeline are the first to get new fetch cycles. From a programming point of view the EV-8 processor is seen as a virtual CMP with four processors that share L1 data without the overhead of cache-coherency mechanisms. Unfortunately, the EV-8 processor project was canceled.

3.3 DSMT Processor

The DSMT processor contains a superscalar processor pipeline as one of its main components. Additionally, DSMT contains support for the execution of multiple threads, which are active simultaneously hence multiple independent programs can also be executed in DSMT as in SMT architectures. To enable the SMT mode of operation in DSMT, the executive software must disable inter-thread register and memory dependency handling. However, since the main scope of this thesis is to describe the DSMT model, SMT mode of operation will not be described any further.

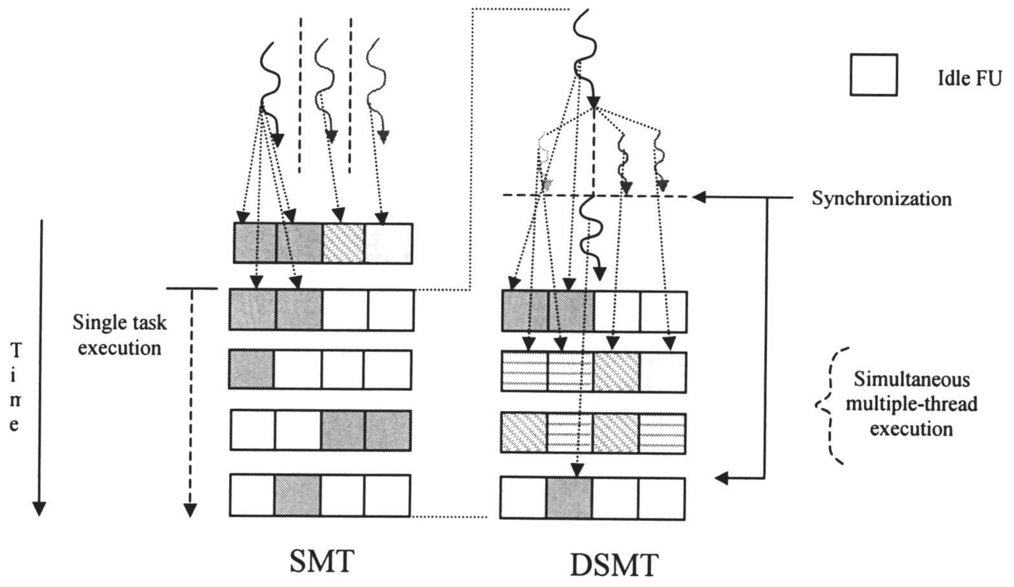


Figure 3-5. Functional unit utilization in SMT and DSMT during single task execution

Figure 3-5 compares functional unit utilization in SMT and DSMT technologies. When there are several independent threads, SMT utilizes efficiently all resources available in the processor. However when there is a single task, SMT does nothing to improve performance; its performance is limited to the ILP contained in a single thread. However in this case, DSMT unlike SMT is capable of generating dynamically multiple threads, which will use more efficiently processor resources. Moreover, since DSMT is based on SMT, it has all the advantages of SMT.

3.3.1 DSMT Operating Modes

The DSMT processor operates in either *DSMT* or *non-DSMT mode.* In the non-DSMT mode, there is only a single thread in execution and thus the DSMT processor behaves as a superscalar processor. In the DSMT mode, multiple threads are executed speculatively, and there is always a single *non-speculative* thread. DSMT enters this mode of execution when a loop is detected in the instruction stream fetched from memory.

In DSMT, speculative threads can be cloned only while the processor is in the non-DSMT mode by the non-speculative thread. Thus, speculative threads are not allowed to clone other speculative threads. This feature reduces the complexity that would be required to control multiple speculative stages.

To identify loops, each time a backward-taken branch instruction is detected, its branch and target addresses are recorded in the LDBTB during fetch stage. Later, during write-back stage, if another taken branch instruction with the same branch target address is found, the processor enters pre-DSMT mode, which is still part of the non-DSMT mode. Notice that this will occur during the second iteration of a loop if the branch information is already in the LDBTB or during the third iteration if there was a LDBTB miss.

During pre-DSMT mode, the decode/dispatch stage marks those registers that will be *live* (i.e. those that will be used as destinations) in the local flags associated to each register. At same time, the loop stride speculation table is initialized in the form described in Section 3.8.

During full-DSMT mode, the overlapped execution of loop iterations occurs, while the processor speculates all subsequent branch instructions (those with the same target address as the one stored in the TCIU) as loop iterations. Using this policy, loops with two or fewer iterations are discarded, eliminating in this way false backward branches that do not belong to a loop and simplifying dynamic loop detection.

In addition, among all branch instructions that could potentially branch backwards, those corresponding to system calls or jumps to subroutines are discarded. The reasons for this are twofold. First, in DSMT simulation environment system calls are handled by the host OS. However, even if this were not the case although some parallelism could be exploited from system calls, they occur relatively infrequent in programs. Second, jumps to subroutines in the body of a loop generate mispredictions in the simple value prediction mechanisms used by DSMT since registers are reused inside the subroutine. Figure 3-6 shows the operating modes of DSMT.

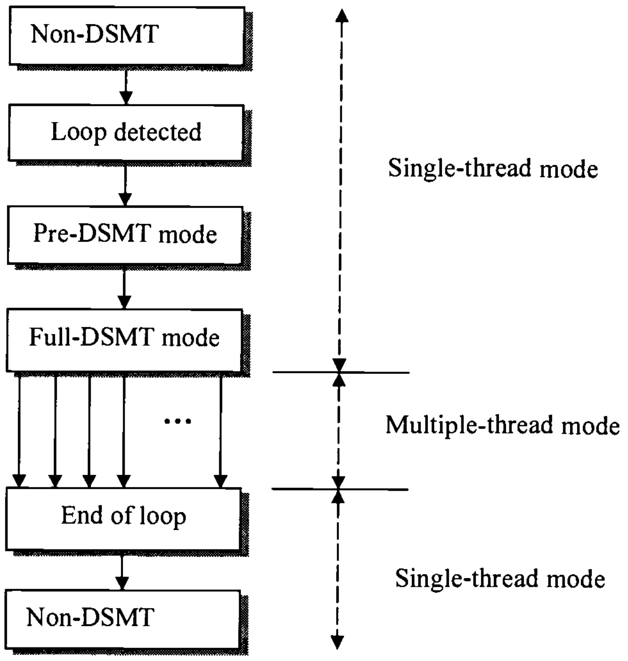


Figure 3-6. DSMT Operating modes.

3.3.2 Context States

As Figure 3-6 shows, DSMT architecture changes its operating mode from non-DSMT mode to DSMT-mode and vice versa. However, once the processor enters DSMT mode a context may go through a series of changes in state. These transitions are caused by the dynamic conditions that occur during dynamic multithreading execution. A context, which is spawned by the non-speculative context is labeled as speculative and placed in its valid *running* state. Additionally, when a speculative context finishes executing one loop iteration, it is placed in a *synchronizing state* waiting for the non-speculative thread to synchronize. Also, a speculative context may be placed in the *hold* state when an inter-thread dependency that could not be solved is found. Moreover, misspeculation causes a speculative context to be invalidated. On the other hand, a non-speculative context may be blocked in the *hold* state during a cache miss. Figure 3-7 shows the possible state changes that a context may go through, during DSMT mode execution.

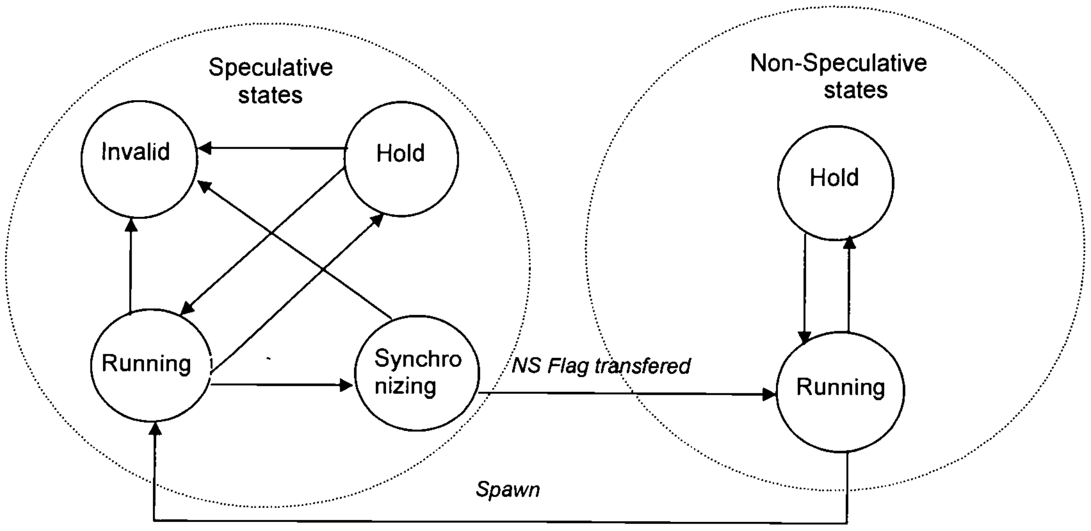


Figure 3-7. Context state transitions

In the following section we describe in detail the microarchitecture of the DSMT processor and the tasks performed at each pipeline stage.

3.3.3 DSMT Microarchitecture

Figure 3-7 shows the microarchitecture of the DSMT processor. During the Fetch stage, a block of instructions is fetched in the usual manner a superscalar processor performs this task. However, DSMT can also fetch instructions from different threads based on various scheduling policies offered by the *Schedule*r.

To support simultaneous execution of multiple threads, each thread in DSMT has its own set of *Instruction Queue* (IQ), *Reorder Buffer* (ROB), *Context*s, and *Memory Order Buffer* (MOB), additionally to multiple *Return Stacks*. The multiple IQs provide a more flexible means of dispatching instructions to the *Reservation Stations* (RS) compared to a single large IQ. This fact reduces stalls caused by instructions that cannot be dispatched due to inter-thread register dependencies. The multiple Return Stacks enable each processor's context to predict a different return address from subroutines.

In DSMT as is in SMT architecture, each Context represents the state of a thread. Multiple contexts are also interfaced to the *Thread Creation and Initiation Unit* (TCIU), which controls how threads are cloned and executed.

DSMT also contains the *Loop Detection Branch Target Buffer* (LDBTB) in the fetch unit, which is responsible for detecting loops, and supplying target addresses so that multiple threads can be cloned. This special BTB is shared by all contexts.

Threads in DSMT are generated speculatively at loop boundaries and executed on a simultaneous multithreaded architecture. Each context handles the execution of a single iteration in a loop.

DSMT architecture controls intra-thread dependencies using the same ROB tagging method that superscalar processors employ. Internal mechanisms in the architecture keep track of inter-thread dependences, thread synchronization, and data memory dependencies. These mechanisms are explained in Section 3.4.

In the following sections, we describe in more detail the tasks performed by each of DSMT pipeline stages.

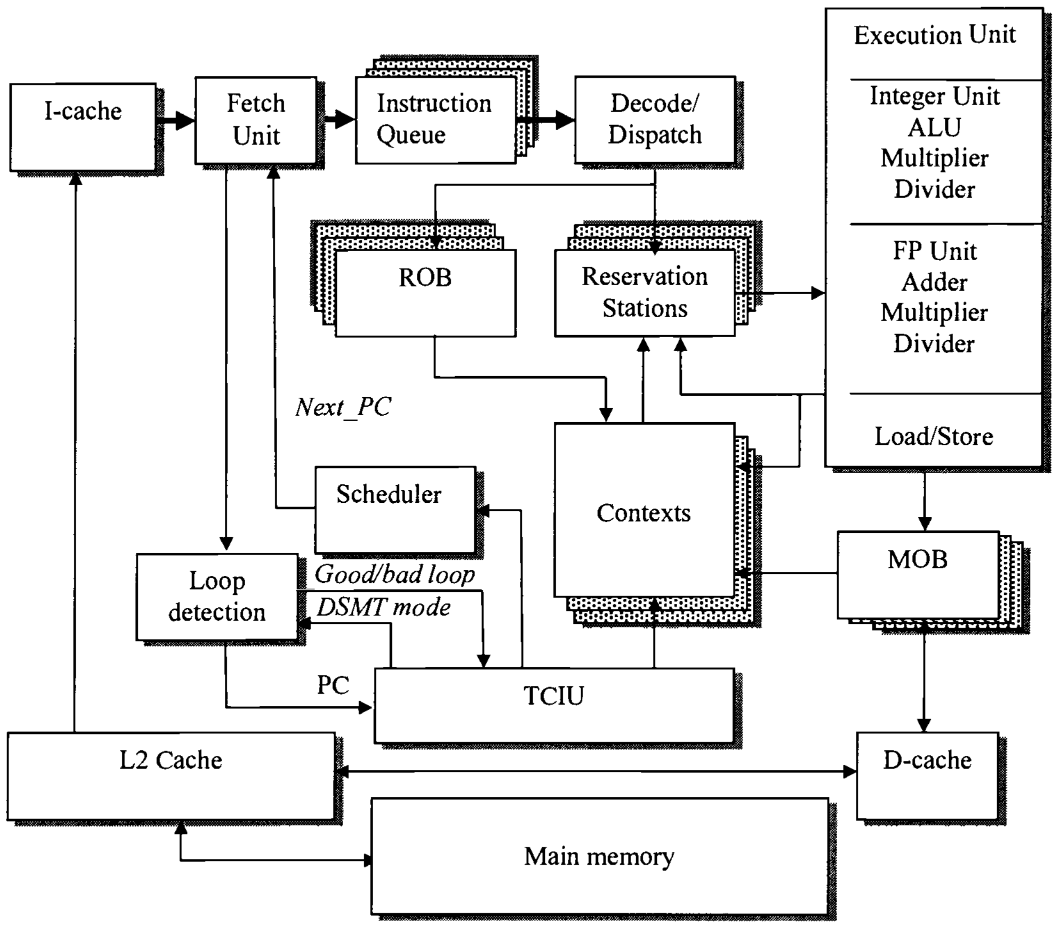


Figure 3-8. DSMT microarchitecture

3.3.4 DSMT Pipeline

The DSMT processor is organized into seven pipelined stages: Fetch, Decode/Dispatch, Issue, Execute, Memory, Writeback and Commit.

DSMT pipeline is depicted in Figure 3-9. Pipeline stages communicate instructions from one stage to the other through internal queues and buffers. Instructions processed in one stage are sent to the following stage for processing. However, the memory stage is used only by load and store instructions being bypassed by all other instructions. The following sections describe in more detail the function and the tasks performed by each pipeline stage during DSM and non DSMT modes.

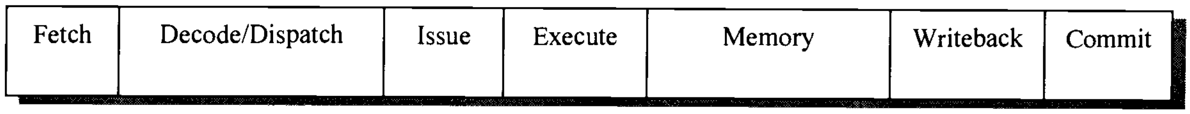


Figure 3-9. DSMT processor pipeline

3.3.4.1 Fetch Stage

The Fetch Unit fetches one cache line from each cache read port. This unit has multiple PCs and an interface to the Thread Control Unit that decides which PC(s) to fetch from. Then, it sends some subset of the cache line off to the instruction queue. A cache line may not be completely full of instructions if a branch instruction was found in the cache line. Also, the starting PC may be offset from the beginning of the cache line due to the occurrence of a previous branch, while a predicted branch prior to the end of a cache line will cause the remainder of the cache line to be discarded.

The Fetch Unit stalls a PC on an immediate jump instruction until the address of the jump is known. This occurs in the decode stage where the address of an immediate jump is computed. However, for conditional branches the fetch unit must predict the result of a branch to avoid stalling the pipeline until the branch address is known.

For taken branch instructions, the branch address is passed to the Branch Target Buffer (BTB) to access its prediction information. The BTB predicts if the branch will be taken or not. If the Branch is predicted as taken, the remainder of the cache line is discarded, and the PC updated to reflect the new fetching address. This predicted address is stored in the BTB with the instruction. Then at writeback stage, the result of a prediction is checked; if the prediction was wrong then the context that fetched the mispredicted instruction is flushed from the pipeline.

Since DSMT is based on a SMT processor core, it implements the ICount2.8 fetching policy. Using this policy, two fetch ports are employed to fetch up to eight instructions per thread. However, the original ICount2.8 policy described in [72] was slightly modified for DSMT due to its particular characteristics. The fetching policy used is called ICount2.8-modified. The modifications made to the original fetching policy are the following:

1) One fetching port is always allocated to the non-speculative thread unless that context is being blocked due to a cache miss. If the non-speculative thread is blocked, then the next speculative context with lesser number of instructions in its IQ and ROB is enabled to use the fetch port reserved to the non-speculative context.
2) Among all speculative threads which are not: blocked due to cache misses, invalid or that are placed in the hold or synchronization states, the thread with the smallest number of instructions in its IQ and ROB is given the highest priority.

As occurs in SMT architectures, this policy favors both, the non-speculative thread and the speculative threads, which are moving faster in the DSMT processor pipeline.

3.3.4.2 Dispatch Stage

During this stage, instructions from a context are taken out from the corresponding instruction queue; at same time, a ROB entry is allocated for the instruction, and the appropriate reservation station is also allocated by filling out all fields in the Reservation Station data store. Additionally, for load and store instructions, slots are allocated in the corresponding MOB in order.

The large number of locations an operand value may come from complicates considerably the dispatching process.

An operand may come from the following sources:

- An immediate value from the instruction itself.
- A register value from the Register File
- A *renamed* value from the ROB.

- The writeback bus at a later time. In this case the dispatched value needs the ROB tag to watch for on the writeback bus.
- Another predecessor context, when inter-thread dependencies are detected.

A tag obtained from the tag generator is used as an identifier to the destination register that an instruction will produce. Since the value of the destination register will be known until the execution stage the register tag of the destination register is marked as *waiting (or busy).* Using the ROB tagging mechanism, WAW and WAR hazards are eliminated by obtaining new tags for the destination registers involved in such false dependencies. However, when there is a RAW hazard and the producer instruction does not have the corresponding value ready, the dependent instruction is given the tag associated with the register that will produce such value. During write-back stage those instructions, which are waiting for register values, are updated. Additionally, once an instruction has been executed, the ROB tagging mechanism facilitates dependent instructions being enabled to read the required register value directly from the ROB.

To manage inter-thread dependencies the dispatch stage performs also the following tasks. The *Ready* bit and the *anchor* bit (discussed in section 3.4) associated with a source register are read; if the anchor bit of a register is set (meaning that another context will produce the value required) and the current context *Ready* bit of a register is not set then the predecessor context register file is accessed to obtain the value. However, since a loop may exhibit dynamic behavior, if the *Ready* bit of the previous context is not set, then all other previous speculative context's register files are scanned looking for the context that has its local *Ready* bit set. The search is stopped when the non-speculative context is reached.

The dispatch task is further complicated by the possibility that a register file has its accesses blocked. This means that the dispatch unit may stop the dispatching of instructions from a particular context when a context is blocked. Blocking may occur in DSMT mode when a context is invalidated because speculation went wrong or when a context has reached the backward branch address that caused entering DSMT mode in first place. In this last case, the context must synchronize changing context state to *hold* and wait until the non-speculative context is being able to handle it the non-speculative flag. Also instruction dispatching in a context may stop when all the following conditions are met for a source register in an instruction:

- No instruction which has been already located in the local ROB will update the register
- The register value will not be produced locally (i.e., its *Ready* bit is not set).
- The register was written in the *Anchor* bits (i.e., other context will produce this register value)
- The register value is not available in the predecessor context.

3.3.4.3 Issue Stage

The task of the Issue unit is to wait for entries in the reservation stations to be marked ready for issue. This implies that they will have all of their necessary operand values resolved, and therefore can be sent for execution. Instructions from different contexts, speculative or non-speculative, have entries in the reservation stations. Once one of these instructions is ready for issue it is sent to the execution unit. In case several instructions are ready for issue, the oldest ones up to the issue bandwidth are issued for execution to the functional unit associated with a reservation station. During this process, priority is given to instructions belonging to the non-speculative context.

3.3.4.4 Execution Stage

The execution unit is composed of functional units. These units can be categorized by their latency and pipeline nature. Thus a functional unit may be single cycle latency, no pipelining (i.e. it accepts new instructions every cycle) or multi-cycle deeply pipelined depending on the type of the functional unit.

Memory access instructions form a special category of instructions to be handled by the execution unit. For these memory instructions, specialized functional units calculate the value of the target address.

3.3.4.5 Memory Stage

In this stage, the data cache memory is accessed by the load/store operations. Speculative threads are allowed to read from memory, or from the current context's MOB, bypassing memory but only the non-speculative thread can store values in memory. However, speculative threads can store values in their MOB speculatively without blocking the thread's ROB. Non-speculative store instructions are retired in order keeping consistent the state of memory. Also, in this stage the mechanism (described in Section 3.7) that keeps track of dependencies in memory is updated.

Instructions that do not access memory are sent directly from the execution stage to the writeback stage bypassing the memory stage.

3.3.4.6 Writeback Stage

The write back unit takes the results coming out of the execute unit and places them on the write back bus. This mechanism allows instructions waiting for the result in the reservation stations and the ROB to simultaneously latch the value. During write back stage results of register jumps (*jr $r*) and conditional branches are obtained. In the case of register jumps, the writeback unit updates the fetch unit with the corrected PC. In the case of branches, it determines the validity of the predicted target address. Then, it forwards the results of the prediction to the BTB to help in predicting future branches in the same branch address. Finally, when a context has mispredicted a branch it flushes the speculative instruction stream after the branch.

3.3.4.7 Commit Stage

In the commit stage, instructions are retired from the ROB *in order* and the physical register file of the context to which the committed instruction belongs is updated. Also in this stage, the TCIU is updated. The TCIU decides, based on the calculated IPC in non-DSMT mode and in pre-DSMT mode, if it is worth it or not to exploit or continuing exploiting a loop. If the decision is to enter full-DSMT mode in the next cycle, the TCIU unit sets the DSMT mode bit and spawns the new threads by activating the contexts and copying the register file from the non-speculative context to the register file of the new context spawned.

Additionally during this stage, the non-speculative context transfers the flag, which indicates which context is working as non-speculative to the next successor context. At same time the TCIU unit is updated with the context number of the new non-speculative thread. This occurs when a loop continuation instruction is committed. However, since it may occur that speculative contexts could finish a single loop iteration before the non-speculative thread does, these contexts are put in a *hold* state which disables them from fetching/dispatching new instructions until they are enabled again by obtaining the non-speculative flag.

When a loop termination instruction commits (i.e., when a non-taken branch instruction is found) at this stage, the TCIU flushes the remaining speculative contexts, changes the state of the processor to non-DSMT mode and updates the LDBTB with the final IPC obtained during DSMT mode.

3.4 DSMT Support for multiple contexts

In addition to multiple PCs that access the instruction cache to fetch instructions for the multiple contexts, DSMT uses the following structures to support multiple contexts.

3.4.1 Instruction Queue and MOB

The compacted results of instruction fetches after branches are fed to an instruction queue (IQ). There is one IQ per context and each IQ structure is implemented as a circular FIFO. During dispatching, instructions are taken from the head of the queue up to the dispatch bandwidth associated to each thread.

One *Memory Order Buffer* (MOB) is also assigned to each context. During dispatching, slots in the MOB are allocated to each load or store operation. Stores are committed in order, but speculative and non-speculative loads are allowed to bypass stores.

3.4.2 Reservation Stations

The reservation stations (RS) are used to hold instructions that may or may not be ready for issue. Each entry in the RS contains immediate values or register values that were read at dispatch. These operands are marked as valid. When all such operands are marked valid the instruction is ready for issue.

For operands not yet valid, (i.e. those who have a dependency in which another instruction will provide the required operand) each entry contains the tag name that is used to latch the correct value from the writeback bus. Additionally, the reservation station contains the ROB tag of the location to which the result should be written. Reservation stations are shared by multiple contexts.

3.4.3 Reorder Buffer and Register File

The Reorder buffer is used to keep the correct order of instructions during retiring or committing. To perform this task a circular FIFO queue is used. An entry in the reorder buffer has an implicit tag in the form of an index. Additionally, the reorder buffer serves as a register renaming mechanism in that it contains register values produced during the execution phase. These values are then available to dispatched instructions before they haven been committed to the register file. In this way artificial WAW and WAR hazards are eliminated. The Reorder buffer, like the instruction queue, is more efficiently implemented as multiple reorder buffers (one per context).

Additionally, DSMT contains one register file per context. The register file from a context is able to send and receive register values from other context's register files.

3.5 Loop detection mechanism

The main module of the loop detection mechanism in DSMT is the LDBTB. The structure of the LDBTB consists of a specially modified Branch Target Buffer (BTB) augmented with additional fields to facilitate loop identification. In addition to the typical fields found in most modern superscalar processors BTB (e.g., branch address, target address, op-code, and branch prediction information), it contains the following information: a flag indicating that the target address of this branch is the starting address of a loop; the number of iterations that this loop has executed (i.e., the number of consecutive taken branches); and a type information indicating whether this is a "good" or "bad" loop for speculative execution based on its previous behavior.

LDBTB contains a field that provides feedback on how loops behaved in their previous executions. Three criteria are used to determine whether a loop is good or bad:

1. Number of iterations a loop executes,
2. Number of contexts currently available in DSMT,
3. How much overlapped execution of cloned threads exists and
4. Thread run-length.

The first two criteria determine the potential TLP in the cloned threads. However, even in loops with a large number of iterations, it is possible that they may exhibit very low ILP during execution. This could be caused by a large number of inter-thread dependencies in the loop or by frequent miss-speculation of loop iterations. In this case, the third criterion is used to determine the effectiveness of the DSMT execution. This criterion associates an Instructions-per-Clock (IPC) measure during the execution of a loop in DSMT mode.

On the other hand, the fourth criterion together with the first two indicates how sustainable the overlapped execution can be.

The four criteria are combined to form the *sustained IPC* (SIPC) measure for a loop. A loop is labeled as *good* or *bad* based on a "break even" policy, where the observed IPC during DSMT execution is compared against observed IPC for a non-DSMT execution of the same loop. If the IPC measure breaks even then the loop is labeled as a good loop for speculative execution. In this way, we can guarantee that DSMT mode execution will result in as good or better performance than in non-DSMT mode execution when speculation is correct.

Nested loops present opportunities to select the thread granularity that DSMT exploits. In these loops, the SIPC measure is also used to select a particular loop in the nested loop structure that will provide the best performance. Control of nested loops is handled by a special stack structure associated with LDBTB. Inner loops are stored at the bottom of the stack and outer loops at the top of stack. This simple mechanism is capable of detecting nested loops by storing the branch address and the target address of a loop in the stack of loops. Then, when a new loop is detected, its branch and target address are compared with the addresses stored at the top of the stack. If the new loop's branch and target addresses include the range of addresses stored at the top of the stack, the loop is pushed onto the stack. The stack is used to select for execution the loop in the nest that will provide the best performance.

3.6 TCIU and multiple contexts

The structure of the TCIU and multiple contexts are shown in Figure 3-9. When LDBTB detects a loop using the policies described earlier, it latches the target address of the thread to be cloned to the *Continuation* register and sets the *M-bit* indicating the processor is in full-DSMT mode. In this mode, the thread cloning process starts by copying the target address in the Continuation register to the *PC*. Then it copies the values in the physical register file to the *Logical Registers* and sets the *Valid* (V) and *Speculative* (S) mode bits of each context. The *V-bit* indicates that the context is valid and, therefore, the Fetch stage can fetch instructions from its PC and update its local registers.

In full-DSMT mode, only one context executes in non-speculative mode and all other contexts are executed speculatively. In this last case, the *S-bi*t, when set, indicates a context is running speculatively.

After the speculative threads have been cloned, the *Loop-Stride Speculation Table* (LSST) is used to *"start"* the cloned threads. LSST keeps a list of register increments (i.e., induction variables) for eventual speculation. LSST contains also confidence bits associated with the prediction associated to each register.

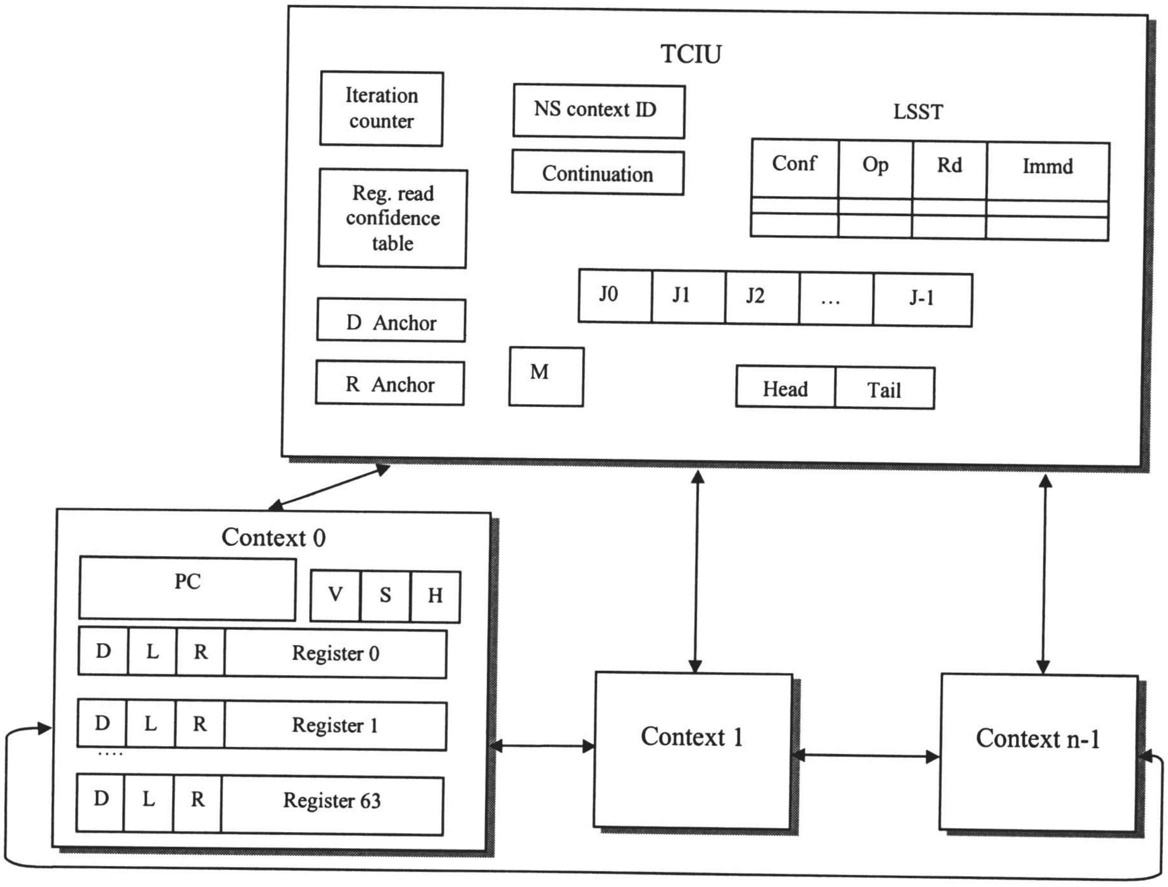


Figure 3-10. Thread control and multiple contexts.

In full-DSMT mode, each context has an associated *Join* (J) bit, which is set to indicate to the TCIU that thread execution has completed. This bit is used to synchronize thread execution.

The TCIU also has a set of Anchor bits, *D_Anchor* and *R_Anchor* bits, which are the status of D and R bits respectively (explained below), of the thread just before threads were cloned. These anchor bits provide a means of:

(1) Speculating whether cloned threads should read the registers of their own context or from the registers of other contexts and

(2) Generating a new set of D bits for future speculation.

The logical register file is the local group of registers in a context. The logical register files of different contexts are mapped to a single physical register file where the actual state of the processor is stored. The logical register files allow register values to be shared across different contexts and, additionally provide each context with a distinct logical view of its state. They also provide fast register access in a context.

To facilitate forwarding between logical register files, in addition to the usual ROB entry tags and busy bits found in a register file of modern superscalar processors, each logical register is associated with a set of utility bits that keep track of inter-thread register dependencies. They are:

1. *Ready* (R) bit – When set it indicates some instruction(s) logically preceding this one in the thread's program order has committed a value to the register; otherwise, no value has been committed to the register and there are no instruction(s) in the local ROB that will commit to this register (which is detected by checking the register's Busy bit). R bits in conjunction with busy bits reflect whether registers should be speculatively read from its own context or another context. In addition, R bits of context *i* are interfaced to the context *i-1*. Whether these contexts are physically connected or not, depend on the *Head* and *Tail* registers of TCIU. This flag also indicates that successor contexts need look no further than this context to get the value.
2. *Dependency* (D) bits – Keeps track of registers that have inter-thread dependencies. When a register is read, if its R-bit is zero and there are no other

instructions in the ROB that will commit to the register, a check is then made to see if its R_Anchor-bit is set. If both of these conditions are true (register not written in the current context and the other context has written that register previously), it means inter-thread dependence exists for the register. These bits will serve as D_Anchor bits to facilitate speculation on how registers are accessed.

3. *Load* (L) bit – When an instruction attempts to read a register in its own context, its L-bit is set if its R-bit is zero (meaning the current thread has not written to this register) and there is no instruction(s) in the ROB that will commit to this register. L-bits indicate that registers have been speculatively read from their own context.

At this point, there are two important implementation details to take into account. First, before threads can be cloned, a couple of loop iterations must be executed in non-DSMT mode to establish the contents of the Continuation register, LSST entries, and D_ and R_Anchor bits. Second, all cloned threads execute speculatively. The only non-speculative thread is the parent thread, which means the parent thread's context has the precise state of the processor. Thus, when the non-speculative thread completes, its successor thread then becomes the non-speculative thread, and the Anchor bits located in the TCIU unit are updated with the R and D bits of the just completed non-speculative thread.

Lastly, is important to mention that during the DSMT mode of execution the register file of the non-speculative thread contains the precise state of the processor and therefore to guarantee a precise state interrupts must be disabled when the non-speculative flag is transferred from one context to the next in the sequence.

3.7 Resolution of Inter-thread register and Memory Dependences in Speculative Execution

Register dependencies between iterations are resolved by speculatively accessing registers based on D_Anchor bits. If R-bit is set for a register, the register value can be read directly from its own context. Otherwise, first-level speculation, called *register dependence speculatio*n, is performed based on its D_Anchor bit to determine which thread the value should be read from. For example, when an instruction in a thread tries to read its own register with its R-bit equal to zero, the D_Anchor bit for the register is checked. If the D_Anchor bit is set, it indicates that previous executions of the iterations had an inter-thread dependency on the register. Therefore, the speculation assumes that this inter-thread dependency will likely exist in the current execution of speculative threads, and the register, when ready, is read from its immediate predecessor thread. If the register dependence speculation turns out to be wrong due to dynamic behavior of loop iterations and the immediate predecessor thread does not generate the register value, the second-level speculation is used. This involves searching back for the last thread that generated a value for the register.

On the other hand, if the D_Anchor bit of a register is zero, it indicates that previous executions of the loop iterations did not have inter-thread dependence on the register. However, since dynamic behavior of loop iterations may have changed a register due to an inter-thread dependent register, the speculation used is to assume that predecessors may have modified the register and read the register from the last thread that wrote to this register. If no predecessor threads have modified the register, it is read from its own context.

Since the DSMT processor relies very heavily on speculation, cloning and speculative execution of threads, require a method to detect and squash threads when misspeculation occurs. The detection of misspeculation is done when registers are written. Whenever a thread writes to a register, its L bits of the successor threads are checked to see if any thread has read the register. If so, that thread and all of its successor threads are squashed and reinitiated. However, when this condition occurs a check is made before the context is squashed to see if the register value has changed. If the value has not changed then the context is not flushed since this means that the register value used is still correct, otherwise the context is flushed. Additionally, every time a context is flushed due to an early reads in a context the confidence bits associated to the register are updated as described later in this Section.

In order to ensure the proper ordering and yet maximize the overlapped execution of the cloned threads, a new iteration is initiated in the context whenever any completes. Thus, when a thread completes its iteration, the context signals to the TCIU by setting the appropriate J-bit. Since the just completed iteration has properly updated its R and D bits, these bits become the new set of Anchor bits. In addition, the just completed thread's successor now becomes the non-speculative thread. The TCIU can therefore reinitiate the next iteration by appropriately changing the Head and Tail registers and cloning the new thread.

In DSMT, loads from different threads can be executed speculatively. However, only the non-speculative threads must perform stores. To ensure that sequential semantics are not violated, the *Memory Dataflow Resolution Table* (MDRT) is used. The organization of the MDRT is shown in Figure 3-11. Load/Store operations are sorted in one of the *Memory Order Buffer* (MOBs) according to its tag. In addition to allowing loads to bypass stores, the MOB also acts a buffer so that stores can speculatively commit. This prevents uncommitted stores from speculative threads blocking the ROB. The Priority Logic selects Load/Store operations based on a priority and forwards it to the memory subsystem. The MDRT checks these operations to ensure memory locations are not corrupted.

MDRT is a fully associative buffer with each entry containing a valid (V) bit, a word address *(addr*) and a value. In addition, each associated with each thread has an L-bit and an S-bit indicating whether the memory word has been loaded or stored, respectively by a particular thread.

The data cache used by DSMT has four ports. One of them is assigned to the non-speculative context. The rest are assigned to the speculative contexts using a round-robin policy. However when there are not enough valid speculative contexts, the non-speculative context may use the available data cache ports. Loads can proceed normally for non-speculative threads. However, speculative threads performing a load check first to see if an entry exists based on *addr*. If none exists, an entry is allocated for the *addr*. If an entry is found, its L-bit for the thread is set and the load is allowed to proceed with its operation. A store instruction is not allowed to update the memory unless it is from a non-speculative thread and it is at the head of its ROB. This guarantees that the precise state of the processor can be maintained. When a non-speculative thread performs a store, it sets the S-bit for the thread. In addition, it checks to see if other threads have their L bits set. If any thread has read previously the memory value that the non-speculative thread has committed, such thread and all of its successors are squashed.

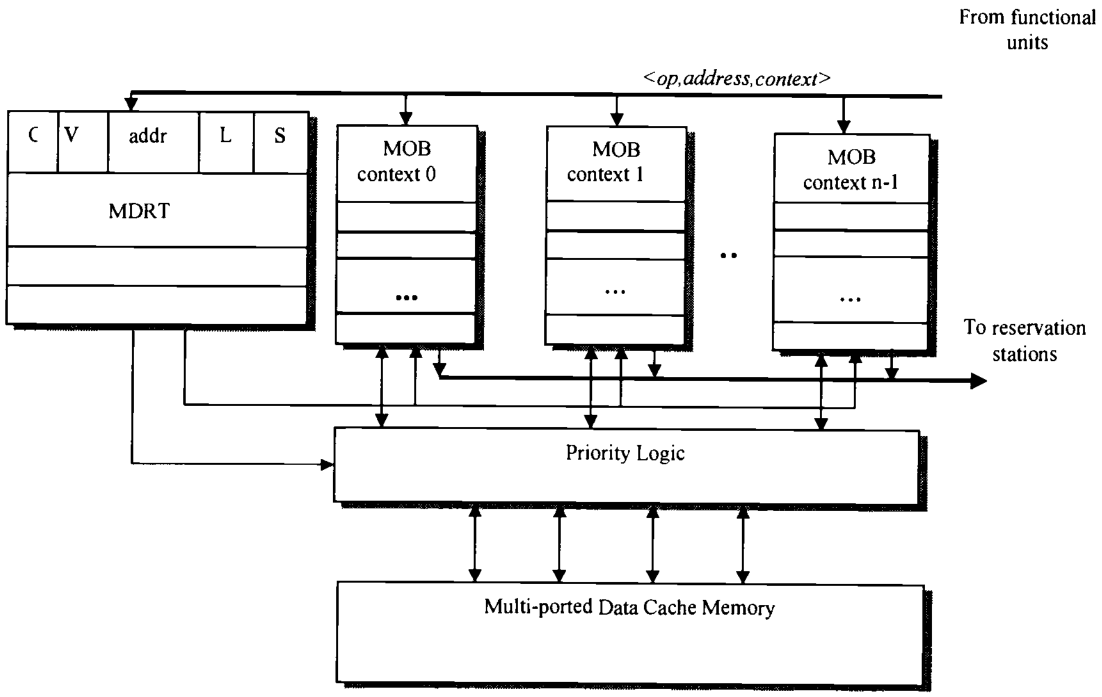


Figure 3-11. Memory dependency resolution mechanism.

Blind speculation may cause continuous squashing of threads who keep reading the same register value too early all time. To avoid this detrimental effect on performance, the TCIU contains a table of confidence bits (C) associated with each register. Two bits are used to assign a confidence value to the speculation performed when a register value is read from another context. The 2-bits counters are incremented every time a thread is squashed due to wrong speculation and decremented when the speculation turns out to be correct i.e., when a register value is read early but it is found that the value did not change. A similar mechanism is used in the MDRT to keep track of early loads from a thread. The confidence bits in TCIU and MDRT are reset to zero every time the processor exits DSMT mode.

The confidence bits associated with the TCIU and MDRT are checked during the dispatch stage, delaying if necessary, instruction dispatching when it has been determined that the instruction has an inter-thread dependency but is very probable that it will be squashed since the confidence is low.

3.8 Value Prediction

DSMT value prediction mechanisms are used to predict induction variables using the *Loop Stride Speculation Table* (LSST). Induction variables are used very often in compiler generated code to keep track of the number of loop iterations, or to accessing memory in regular patterns. Induction variables have normally the prototype `addiu $r,$r,#imm` or its signed version `addi $r,$r,#imm` in the PISA instruction set. These types of instructions may serialize loop iteration execution since the register values that are used depend on the loop iteration number. For instance, in the code section shown in Figure 3-12, which corresponds to the inner loop of an optimized version of the matrix multiplication algorithm, all instructions that employ immediate addressing mode are marked as *candidates* for loop stride speculation during the decoding/dispatching stage.

Then, during pre-DSMT mode, the `#imm` values associated with each source register in the candidate instruction are stored in the LSST. Later, at context spawning (in full-DSMT mode), the register values are speculatively calculated as: *$r+iteration_number*imm* by specialized hardware units. Notice that the current iteration number executed by the non-speculative thread is contained in the TCIU. This number is updated every time that the non-speculative thread commits its results. When the non-speculative thread spawns new threads, it assigns to each one an iteration number that is used later to calculate the stride.

```
<main+90> addu $v0[2],$t1[9],$v1[3]
<main+98> l.s $f2,0($a0[4])
<main+a0> l.s $f0,0($v0[2])
<main+a8> mul.s $f2,$f2,$f0
<main+b0> l.s $f0,0($a2[6])
<main+b8> addiu $v1[3],$v1[3],80
<main+c0> add.s $f2,$f2,$f0
<main+c8> addiu $a1[5],$a1[5],1
<main+d0> addiu $a0[4],$a0[4],4
<main+d8> slti $v0[2],$a1[5],100
<main+e0> s.s $f2,0($a2[6])
<main+e8> bne $v0[2],$zero[0],00400280 <main+90>
```

Figure 3-12 Inner loop in the matrix multiplication algorithm

At the end of an iteration, to check the result of LSST speculation, the final register values obtained during the execution of the non-speculative thread are checked against the speculated values used by the successor speculative context. If those values are different, the context associated with the mispredicted register value and all its successors are flushed from the pipeline.

As Figure 3-12 indicates, is possible to dispatch instruction `addu $v0[2],$t1[9],$v1[3]` in parallel from all contexts since register `$t1[9]` never changes (*is dead*) and the value of register `$v1[3]` is speculatively generated by the LSST. The same argument is valid for instruction `l.s $f2,0($a0[4])` and `l.s $f0,0($v0[2])` because registers `$a0[4]`and `$v0[2]` are assigned a value speculatively. On the contrary, if those register values were not speculatively generated, then the first three instructions in the loop will not be enabled for dispatching, causing serialization of that portion of code. This fact was verified experimentally executing the matrix multiplication algorithm code shown in Figure 3-12 with the LSST turned off. When this experiment was performed very little improvement in performance was obtained even for such very parallel loop.

3.9 IPC Measurement Mechanism

DSMT sustained IPC policy to decide whether the parallelism available in a loop is worth exploiting is based on measuring the IPC obtained during DSMT mode as described in section 3.5. Since IPC is a ratio, its calculation, a fixed point division, may take several clock cycles. However, the decision to enter full DSMT mode should be done in a single cycle. Moreover, additionally to calculate IPC, the values obtained during sequential execution in pre-DSMT mode must be compared with those obtained during parallel execution in full-DSMT mode. The comparison is performed to decide which mode of operation will give better performance. However, when a precise measurement of IPC is expensive to carry out, an approximate method may be used. Using an approximate method, a comparison of the numerator of the IPC ratio (number of instructions committed) in DSMT mode is performed while keeping fixed the denominator of the ratio (number of clock cycles). One possible simple implementation of this mechanism using counters and comparators is depicted in Figure 3-11.

In this implementation, the TCI unit specifies each clock cycle, which is the current operating mode of the processor that is either, non-DSMT pre or full-DSMT mode. Once the DSMT architecture is operating in pre or full-DSMT mode, the cycle counters keep track of the number of cycles that the processor has spent on an operating mode. At the same time, instructions counters keep track of the number of instructions committed.

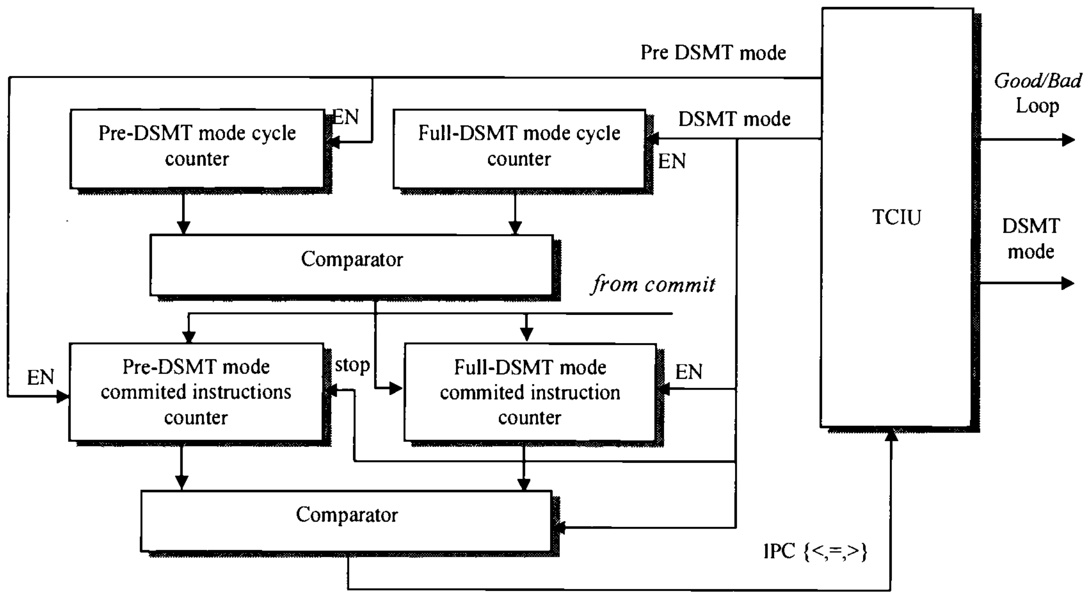

Pre DSMT mode
DSMT mode
Good/Bad Loop
DSMT mode
TCIU
Pre-DSMT mode cycle counter
EN
Full-DSMT mode cycle counter
EN
Comparator
from commit
EN
Pre-DSMT mode commited instructions counter
stop
Full-DSMT mode commited instruction counter
EN
Comparator
IPC {<,=,>}

Figure 3-13. Mechanism employed to calculate relative IPC.

The cycle and the instruction counters corresponding to pre-DSMT mode are halted when the processor decides to enter full-DSMT mode. Those values are used later as the reference that indicates in which clock cycle the TCIU will decide if the IPC obtained in full-DSMT mode is better or worse that the performance obtained during pre-DSMT mode.

4 DSMT SIMULATION ENVIRONMENT

A simulation platform centered on a new simulator, *DSMTSim*, was created to study the performance of DSMT architecture. DSMTSim is an execution-driven simulator that is able to operate in different execution modes. These modes are:

- Fast mode simulation.
- Wide issue, single context out-of-order simulator.
- SMT multiple contexts.
- DSMT simulator.

DSMTSim is an execution-driven simulator that accepts the binary files generated by Simplescalar's gcc compiler (SS-gcc) [14]. SS-gcc generates instructions for PISA, which is a modified version of the MIPS instruction set.

DSMTSim loads a binary program into internal memory and then simulates cycle by cycle, all operations performed by the internal pipeline during single and multiple-thread execution. Values are passed from producer to consumer instructions or memory *on fly*. Hence, detailed simulation of the out of order engine is completely simulated. DSMTSim is able of simulating single and multiple context execution with a command line option.

In DSMT instructions in a thread commit their results out of order. However, threads may also commit their results out of order with respect to other threads. This dynamic behavior causes microarchitectures such as DSMT being difficult to simulate. For this reason, several simulators of superscalar architectures were evaluated before deciding to develop DSMTSim. The following section describes some of those simulators in detail.

4.1 Simulation Tools in Computer Architecture

There exist a number of simulation tools that contain detailed models of today's high performance microprocessors. *SimpleScalar* (SS) tool suite 14 is a popular simulation platform that provides several classes of simulators of varying accuracy/speed. Among these, *sim-outorder* simulates a superscalar microarchitecture, and is the most complex simulator of the tool suite; this simulator is a hybrid of a functional and trace simulator. Traces are generated on-the-fly by the front-end simulation engine, executing those instructions in-order, modifying the values of registers and memory. Later, in the back-end, those traces are used to emulate an out-of-order processor, this time without modifying registers or memory.

Sim-outorder handles system calls by passing them to the host operating system. The host OS executes the system calls and passes the results back to sim-outorder. SS tool suite is being widely used in computer architecture research. SS is written in C, executes only user-level application programs, and has been ported to many different platforms. A large percentage of the research published in major conferences and journals is done using SS. However, sim-outorder is not easy to modify due to its structure, and additionally the trace front-end that it contains makes difficult simulating dynamic multithreading architectures since the simulation must reconstruct the effects of the dynamic events from the trace.

In contrast to the SS approach, SimOS simulates all the hardware in a computer system, including I/O devices such as hard disks and network interfaces 52. SimOS simulates the hardware components in sufficient detail to boot and execute a complete OS. Using this simulator, it is possible to study the effects of more realistic workloads on the performance of a complete computer system. SimOS is written in C, models the MIPS R4000, R10000 and Digital Alpha processor families and executes IRIX and Digital Unix OS. SimOS comes with an in-order processor simulator but an out-of-order version (MXS) 20 is also available for some platforms. However, one drawback of SimOS is that runs only the binaries of Irix OS.

PSim 16 is a simulator for the PowerPC architecture. PSim implements the three levels of the PowerPC *instruction set architecture* (ISA): User, virtual, and operating environments. In the user mode, PSim can run static programs compiled for any of the following operating systems: NetBSD, Solaris or Linux. This simulator comes integrated with the gdb debugger.

Other superscalar processor simulators were designed as teaching tools. Examples of this type of simulator are: SuperDLX [43], and SATSim [79]. There are also simulators that are variations of SS, such as SIMCA [27], which has multithreading capabilities. This special purpose simulator requires support from the compiler to generate threads. In addition, some simulators run only on specific platforms or require special compilers such as MIPS 17 or SMTSim 71. All these simulators are execution-driven.

In contrast, there are simulators that are both, event-driven and execution-driven, e.g., RSIM 48. RSIM simulates an out-of-order processor similar to MIPS R10000 and is partially written in C and C++. RSIM is also capable of simulating a multiprocessor system using event-driven simulation.

Another hybrid simulator is fMW 8, which is a descendent of the trace simulator VMW 18. This simulator contains a trace engine called MW that directs the order of instruction execution of PSim. PSim calculates results and sends the data back to MW, which calculates IPC and processor utilization.

Unfortunately, in spite of their simplicity, simulators based on traces are unlikely to capture all the processor's behavior [98]. For instance, speculative loads may not show up in a memory reference trace. Moreover, the dynamic speculative behavior of threads is very difficult to recreate from a trace. For these reasons, and due to the lack of detail required to simulate dynamic multithreading in existing trace and non-trace simulators, a new simulation environment was designed for DSMT architecture. Next section, describes the simulation environment created for this purpose.

4.2 Simulation Environment

During the simulation tool development process for DSMT architecture, two new simulators were designed and implemented. The first simulator called EMSim [46] is an object-oriented simulator that emulates the architecture of a generic superscalar processor. Therefore, this simulator supports only single context simulation. The second simulator called DSMTSim is a procedural simulator, which simulates DSMT in detail with multiple contexts.

Both simulators in their current version are compatible with the compiler, linker, assembler, and libraries of the SS tool suite. As a result, they share with SS the way in which data, stack, and code areas are mapped into memory. However, parts of the macros that define the implementation of the instruction set and the system calls of SS were modified to make them compatible with DSMTSim and EMSim. In both simulators, the simulation parameters can be configured from the command line or from a text file in both simulators. The configuration parameters include: size and associativity of cache memories and BTB; size of the instruction queue, ROB and reservation stations; and number and type of functional units.

DSMTSim and EMSim are execution-driven simulators. These simulators are capable of operating in different execution modes, which are: (a) fast, in-order simulation and (b) detailed wide-issue, out-of-order simulation in case of EMSIM and multiple context simulation in case of DSMTSim. The fast simulation mode employed by both simulators, allows the user to quickly place a simulator in a particular section of the benchmark code, skipping non-representative parts like initialization. During fast mode simulation, instructions are read directly from memory and executed in sequence. Conversely, in the detailed simulation mode, all the memory hierarchy and the pipeline stages of the simulator are exercised. Using the detailed mode, the simulators load a binary program into their internal memory and then simulate in detail, cycle-by-cycle, all the processing performed by the pipeline. During instruction processing, register values are calculated and passed from producer instructions to consumer instructions or memory *on-the-fly*. On-the-fly value passing closely emulates the action of real superscalar processors, and is also used as a means for checking the correct operation of the tagging and out-of-order execution mechanisms included in each simulator. In this way, correct manipulation of instruction values is ensured during speculative execution.

Functional validation of EMSim and DSMTSim was performed in two ways. First, the contents of memory and the internal registers on each simulator were compared on a cycle-by-cycle basis with the corresponding values obtained by simplescalar's sim-outorder executing the same benchmark. Furthermore, special benchmarks, which perform intensive mathematical calculations, were executed on a real machine. The output result sent to the console obtained by these benchmarks was compared with the results obtained from EMSim and DSMTSim. In both cases, the simulators obtained the same results as sim-outorder and the real machine. This test confirmed that all the out of order mechanisms and the control speculation were correctly implemented.

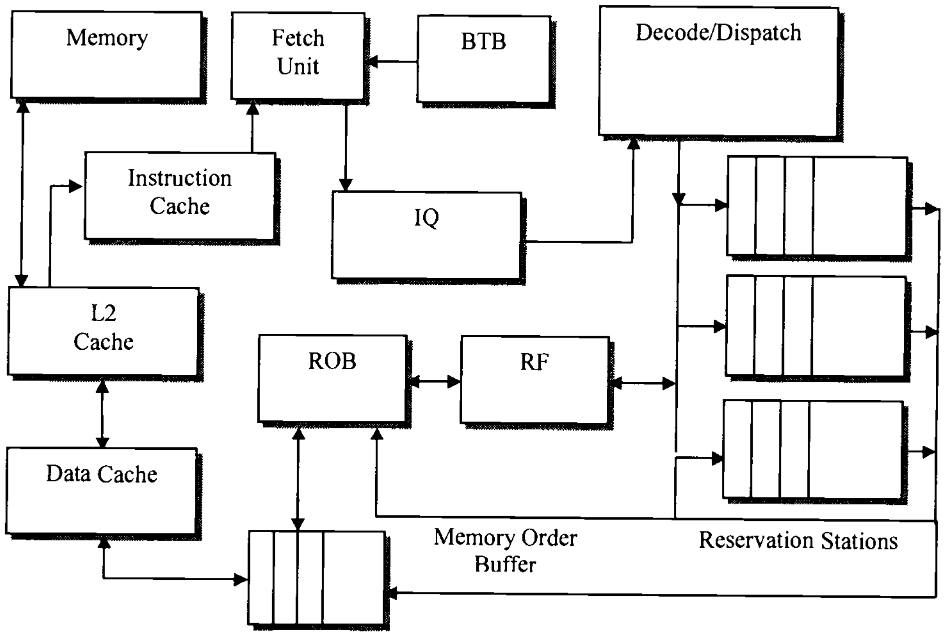


Figure 4-1. EMSim's superscalar model.

Next sections describe in more detail the two simulators developed for this research: EMSim and DSMTSim.

4.3 EMSim

EMSim was designed to develop eventually into the simulation platform for DSMT architecture. The specific superscalar architecture that EMSim simulates is shown in Figure 4-1 EMSim consists of seven main stages: Fetch, Decode/ Dispatch, Issue, Execute, Write-Back, Memory and Commit.

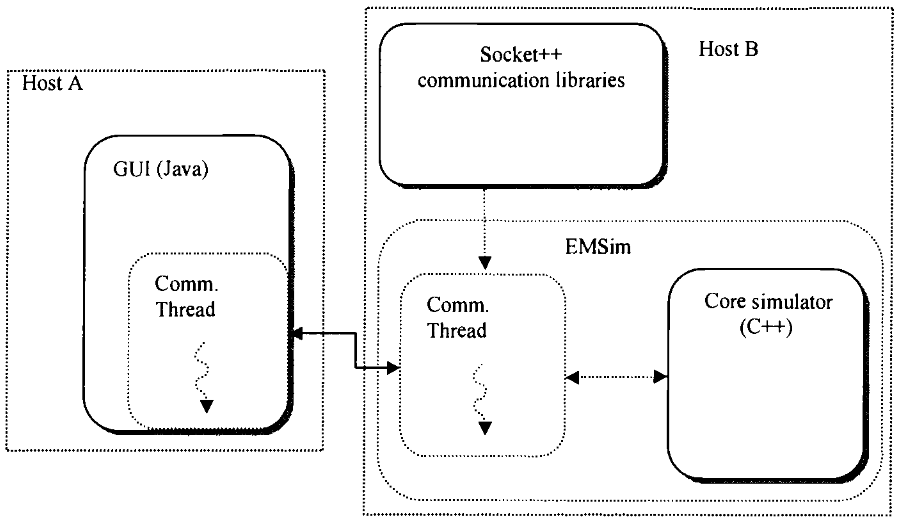


Figure 4-2. EMSim's distributed architecture.

EMSim was designed using *object-oriented* (OO) techniques. The advantages of an OO approach to software design in general are well documented [13]. They include many well-accepted design goals of quality program development, such as modularity, modifiability, and maintainability [13]. Moreover, designs centered on objects are especially suited to use in simulation.

Simulation speed is obviously an important factor in a simulator. However, the features that the OO approach provide to software design in general, are equally or perhaps more important in a simulator. Languages such as C++ offer useful OO mechanisms, such as *inheritance*, *polymorphism*, and *templates*. Templates support the design of software using *generic programming* techniques. In generic programming, software components are created so that they can be easily reused in a wide variety of situations. The data structures and algorithms in the Standard Template Library (STL) [7] are examples of the application of generic programming. In this library, software components such as queues, sets, lists, etc., are able to handle different types of objects employing different algorithms.

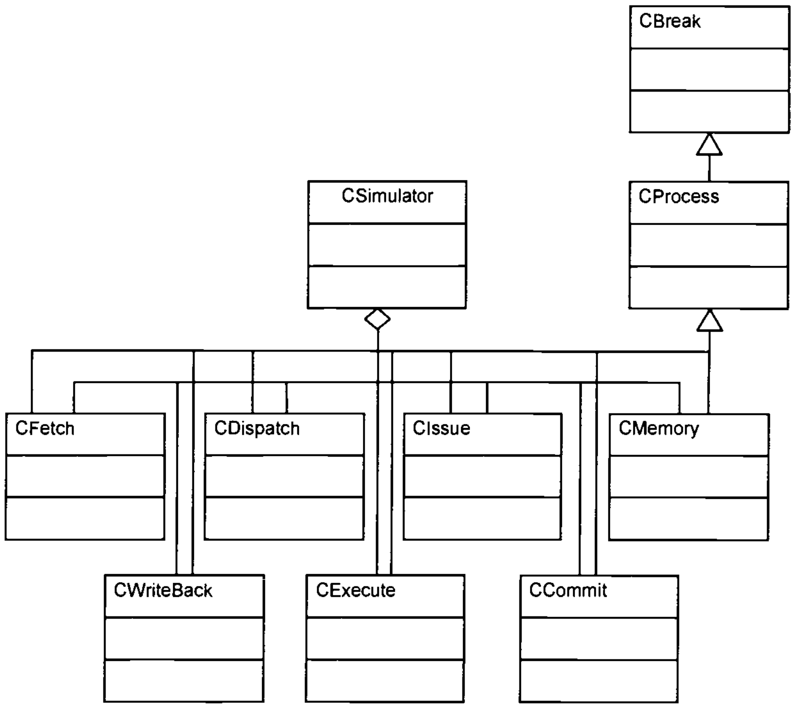


Figure 4-3. UML diagram of EMSim's main classes

EMSim provides modularity, code reutilization, and extendibility through the use of classes, inheritance and generic programming. Modifications to EMSim are easily integrated since these features are available to a developer. To obtain fast execution speed, EMSim was developed in C++. In addition, the implementation of EMSim was carried out employing STL's generic containers and iterators. Moreover, the *interfaces* defined using virtual functions allow subclasses to specialize methods with their particular implementation.

EMSim has a distributed architecture that consists of a graphical user interface (GUI), a core simulation engine, and communication facilities. This is in contrast with existing simulators that consist of only a single executable with a simple command line user interface where all simulation parameters are specified.

The Java language provides, through the *Swing* library, abundant graphical elements to build complex user interfaces that are portable across different platforms [6]. To make use of these capabilities and without sacrificing simulation speed, EMSim was designed using Java's Swing libraries for the user interface and C++ for the core simulator. The simulation environment designed with this architecture allows decoupling the GUI from the simulator, a useful feature during testing or when benchmarking is performed.

EMSim's user interface handles user events and creates a special thread to communicate with the main simulator through TCP/IP sockets. On the other hand, the core simulator creates a Posix thread (*pthreads)* to execute the communication routines.

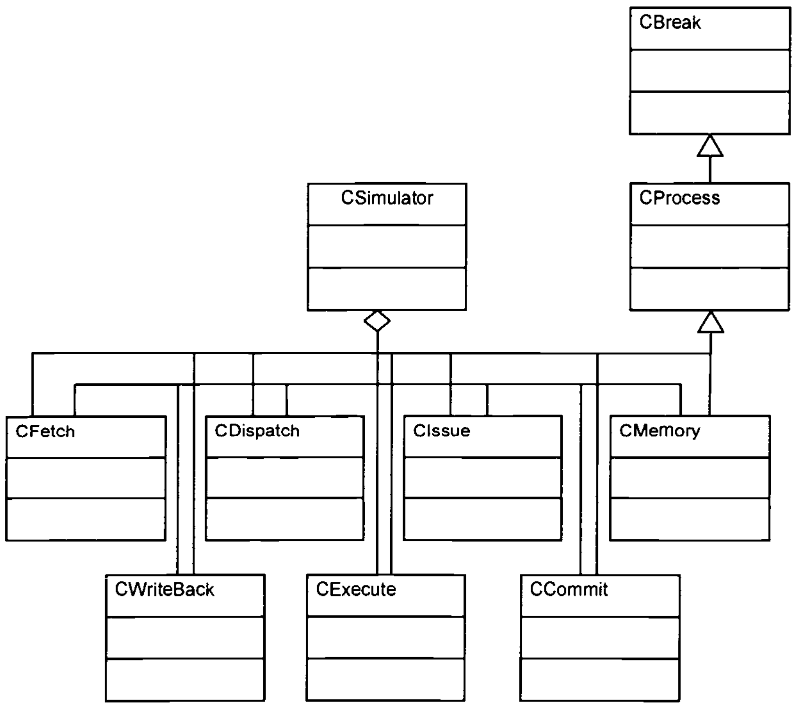


Figure 4-4. UML diagram of EMSim's pipeline

The communication thread in the GUI receives commands from the user to control simulation execution. A simple protocol was designed to transfer data between the simulator and the GUI. The communication library socket++ [60] is employed to receive/send the C++ input/output streams from/to the simulator. EMSim's distributed architecture makes feasible to use the simulator with the GUI on a single computer or remotely over the network. EMSim is also able to run on a single computer without the GUI in text mode. In the latter case, the output streams generated by the simulator are re-directed to the standard output instead of through the TCP/IP sockets.

EMSim's design is organized in a hierarchy of classes. A partial UML diagram of the main classes in EMSim is shown in Figure 4-3. The base clase CObject provides common methods and variables to all classes derived from it. The Instruction-cache (CICache), Data-cache (CDCache), and BTB (CBTB) classes are specializations of the base classs CCache used to store instructions, data, and branch prediction information respectively. Other classes are also defined to keep processor state and provide the actions performed by the register file (CRegFile), main memory (CMMem), reservation stations (CResStat), instruction pool (CinsPool), and writeback queue (CWBQueue).

EMSim pipeline was modeled using the UML class hierarchy diagram shown in Figure 4-4. The class CSimulator contains the classes used to model the pipeline stages and the main simulation loop. As is illustrated in Figure 4-4, classes were defined to emulate a processor pipeline consisting of fetch (CFetch), decode/dispatch (CDecode), issue (CIssue), execute (CExecute), write-back (CWriteBack), and commit (CCommit) stages. In addition, memory instructions are processed during the memory stage (represented by CMemory). The base class CProcess provides the features that are common to all derived classes representing the pipeline stages (e.g., bandwidth). Utility classes (not shown) were also designed to handle statistics, exceptions, memory data, clock, timers, etc.

The design of EMSim also provides support for debugging. The class CBreak in Figure 4-4 allows breakpoint conditions to be declared. This class also contains methods to dump the state of a pipeline stage when a breakpoint condition occurs during simulation. The information dumped includes the state and contents of the queues handled by the stage. A new class derived from CBreak may override the dump method to print any other information required by the user. In its design, EMSim employs different STL generic containers such as sets, lists and queues. Although the design of STL was optimized in some parts for fast execution, this library adds an overhead to the total simulation time. Hence, to minimize this overhead, STL's generic containers were employed only in those parts of the simulator that do not negatively impact EMSim's performance. Therefore, containers of objects that are used very frequently were implemented using templates and arrays instead of STL generic containers. EMSim's main loop processes all the pipeline stages at each simulation cycle as the following code shows:

```
obj_list_iterator p;
for(;;;){
     for(p=s_RunList.begin();
     p!= s_RunList.end();
     p++)
       (*p)->Run();
     clock.Tick();
}
```

Figure 4-5. Main loop of EMSim

As this code segment illustrates, there are no specific references to any particular pipeline stage in the main loop. Pipeline objects are stored in the STL generic container `s_RunList` during initialization, and then, accessed through the object *iterator* `p`. This design reduces the amount of changes required and simplifies modifying EMSim. For instance, to add a new pipeline stage into EMSim, a new class derived from CProcess is created. This class will include the implementation of the virtual method `Run` shown in the segment of code above. Virtual methods are the interface that is implemented in different ways at each pipeline stage. Later, an instance of the class would be stored in `s_RunList`. Pipeline objects are stored using the `push_back()` method of the container `s_RunList`. The iterator in the main loop of EMSim will automatically process the new pipeline stage by calling its `Run` method. Using this generic programming approach, the main loop of the simulator remains unchanged regardless of how many pipeline stages are added (changes occur mainly in the new class code).

In EMSim, pipeline stages communicate through instruction queues. Hence, after being processed, instructions must be placed in the appropriate output queue so that the next pipeline stage could access those instructions. Thus, no global structures are accessed during this process. In contrast, adding a new pipeline stage in SS's sim-outorder requires changing the main simulation routine. In addition, the new procedure must update, in an appropriate way, the global structures and variables (e.g., queues and flags) that keep track of the processor's state in the RUU unit [99]. However, since the RUU unit is a centralized structure, this process must be performed very carefully to avoid causing unintended effects in other parts of the simulator.

The OO structure of EMSim facilitates extending its capabilities to simulate other types of architectures and allows also a quick visualization of the changes that will be required to perform those extensions. For instance, extending EMSim to simulate a SMT processor entails creating additional object instances for the register file (or processor's *context*), instruction queue, ROB, and Memory Order Buffer (MOB). In addition, using the generic programming approach, such instances would be stored in generic containers that the appropriate pipeline stage would access in order to process instructions corresponding to different contexts. Other changes to EMSim required to implement a SMT processor would involve extending the data cache memory to support multiple ports, overriding the fetch method that access the instruction cache memory to support a new fetching policy, overriding the method that is used to dispatch instructions, etc.

4.4 DSMTSim

DSMTSim is the procedural simulator for the DSMT architecture. This simulator has all the elements and mechanisms required to simulate a SMT architecture, such as: multiple contexts, multiple instruction and data-cache ports, multiple ROB's, MOB's, and multiple return stacks. Additionally, DSMTSim includes the data structures required to support the DSMT model described in detail in Section 3.3. For instance, DSMTSim implements the code to simulate elements such as: TCIU, MDRT, LDBTB, anchor bits etc., as described in Section 3.3.

Figure 4-6 shows the programming environment used by DSMTSim and EMSim. Both simulators were written for the Linux platform and the SS compiler (gcc 2.72 ported to generate PISA code) was used to generate all of the benchmarks used in simulation.

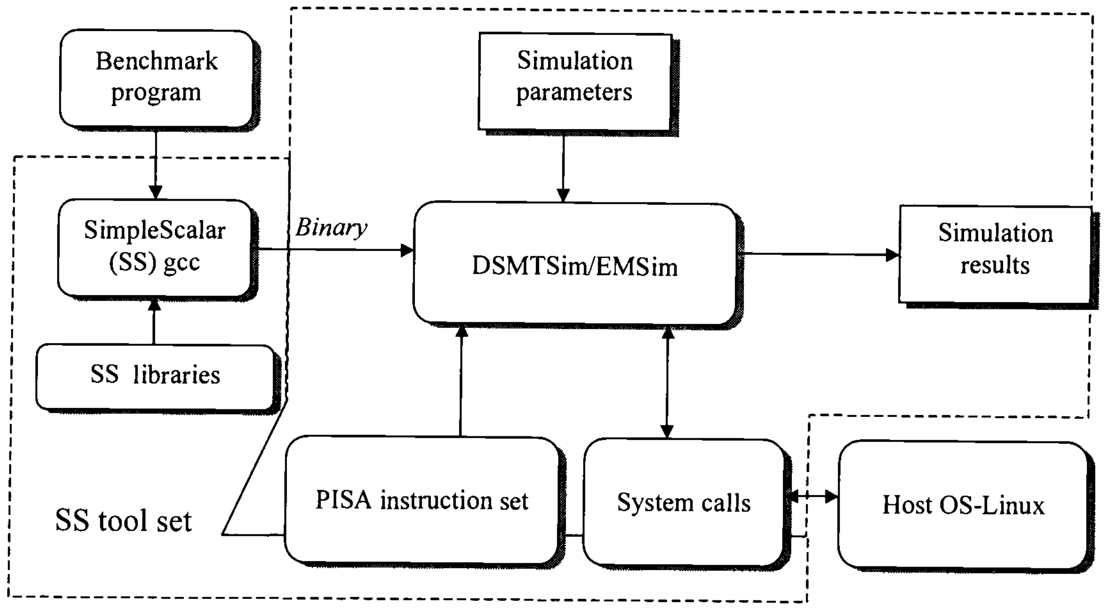


Figure 4-6. DSMTSim and EMSim's programming environment.

Feature	**SimpleScalar sim-outorder 3.0b**	**DSMTSim 1.0b**	**EMSim 1.0a**
Simulator type	Hybrid (trace front-end)	Execution Driven	Execution Driven
Multiple contexts	No	Yes	No
Object Oriented	No	No	Yes
Fast execution mode	No (only in sim-fast)	Yes	Yes
Distributed architecture	No	No	Yes
Configurability	Very	Some	Minimum
On-the-fly value passing	No	Yes	Yes
Multiplatform	Yes	No	No

Table 4-1Comparison of features

Table 4-1 compares some of the capabilities provided by Simplescalar's sim-outorder, DSMTSim and EMSim in their current versions.

In spite of the fact that the main goal of DSMTSim and EMSim was not to achieve fast simulation speed, two code optimizations were included in the design of both simulators. First, the maximum number of instructions that a simulator is able of processing in its internal data structures is allocated only once during initialization, and placed in an instruction pool. When a context fetches an instruction from memory, such instruction is marked as *busy* in the instruction pool. Later, when that instruction commits its results it is marked as *free*. This recycling policy enables instructions to be reused, avoiding the cost of allocating new blocks of memory every time an instruction is processed. Second, instructions processed in one pipeline stage are passed from one stage to the next by changing only the instruction pointers associated with them.

4.5 DSMTSim Incremental Development and Functional Validation

DSMTSim was developed incrementally to ease debugging and testing. During development, several versions of the simulator were created, namely: single context *DSMTSim-fast*, single context superscalar *DSMTSim-SS* and multiple contexts DSMTSim. Simpler versions of the simulator were tested and used as the basis for the more complex simulators. This process facilitated debugging and the smooth, incremental development of DSMTSim.

DSMTSim-fast is a fast and very simple version of a single context processor. This version is roughly equivalent to SimFast from the SimpleScalar toolset. DSMTSim-fast does not have cache memory or pipeline stages. Instructions are taken from memory, and decoded and executed in sequence. The goal of this simulator was twofold; first to test the modules that are shared by all simulators, such as program loader, decoder, register file and the instruction set; and second to provide the fast simulation mode that DSMTSim uses to execute benchmark initialization code. Initialization code is generally discarded when obtaining performance measurements.

Functional validation for DSMTSim was performed at several levels. First, since DSMTSim is actually passing values between instruction registers, the output generated by the simulator was compared with the output of the same program using a real processor, i.e. the output sent to the console for both the simulator and the real processor was compared. Second, to debug and validate DSMTSim-fast, a special test program was written. The testing program is able of comparing the state of the internal memory and the register values with the contents of memory and registers in SimpleScalar's sim-fast simulator, when both simulators are executing the same benchmark program. However, Simplescalar's sim-outorder simulator was modified to generate the output in the same format as DSMTSim does.

In order to perform validation using sim-outorder simulation, a test program called DSMTTest was written. This test program spawns two Unix processes corresponding to both simulators: DSMTSim and SimpleScalar's sim-outorder. Then, each simulator, working in parallel, executes a single instruction and sends all its registers values and its memory contents to the parent process (DSMTTest) through OS pipes. Later, the parent process compares the received values and generates a report file in case an error in the register file or in memory is found. Otherwise, simulation continues normally.

Figure 4-7 shows a block diagram of the method used to validate DSMTSim using DSMTTest and sim-outorder.

On the other hand, the full-fledge superscalar simulator EMSim was debugged/validated using an instance of EMSim in fast mode. The fact that EMSim is written in C++ made testing phase easier, since instances of EMSim fast mode and EMSim regular superscalar objects are created easily. To allow comparing EMSim fast mode with EMSim superscalar output, the commit stage of EMSim superscalar was changed to stop committing instructions after the retirement of a single instruction. Figure 4-8 shows how C++ objects facilitated the testing procedure.

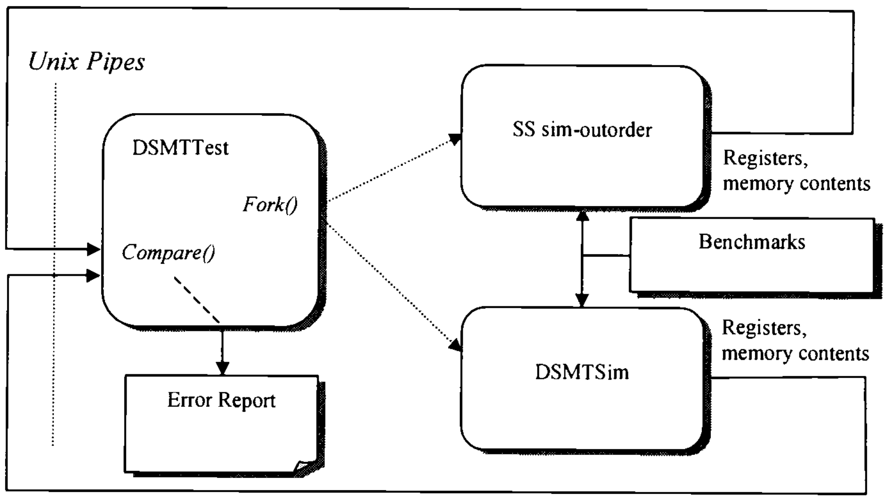


Figure 4-7. Functional validation of DSMTSim.

```
EMSim-Fast emsimFast = new EMSim-Fast(); // create simulators
EMSim emsimSS=new EMSim();
emsimFast.execute(1);//executes and commits a single instruction
emsimSS.execute(1);//all pipeline used, commits one instruction
for (int i=0;i<EMSim::register_num;i++) {//compares register values
      if (emsimFast.getRegister(i) != emsimSS.getRegister(i)){
            generateReport();  // if error generates report
            break;
      }
 }
 // do the same with memory
```

Figure 4-8. Code used to compare states of EMSim and EMSim-fast .

4.6 Software Tool for Parallel Benchmarking

In computer architecture research and in many other engineering areas, performance evaluation requires very time-consuming simulation runs. These simulations can take a long time to produce performance results. During simulation, a benchmark program is usually executed using a specific set of simulation parameters. Later, a new simulation is started, but this time with some of the input parameters changed. These steps are repeated for different benchmarks using different simulation parameters. Finally, when all simulations have finished, simulation results are collected and sorted for final analysis. This entire process makes performance evaluation a long and error-prone process. However, total simulation time may be improved if a group of networked computers is employed. In this case, each computer performs different simulations, reducing considerably total benchmark simulation time because all computers are working in parallel. However, simulation data results still have to be collected and sorted manually. Furthermore, in this environment, there is no way to guarantee good load balancing in the network and therefore, different simulations could finish at unpredictably times.

A parallel computing cluster, like the SWARM Beowulf cluster [45], can ameliorate all of these hurdles. SWARM, along with software specifically written to work in this parallel environment, substantially reduces the time required to perform detailed architectural simulation studies. At the same time, load balancing is obtained through specialized software that is in charge of analyzing network load, assigning tasks to nodes that have a lesser load. A software tool was created to work on the SWARM for improving benchmark simulation time and data collection.

Simulation nodes in a parallel environment like SWARM need to communicate in order to send simulation results. Message Passing Interface (MPI) [47] is a collection of library functions that provide communication primitives using message passing. DSMTSim was modified with MPI calls to send and receive simulation data.

Figure 4-10 shows the parallel simulation environment created for parallel benchmarking of DSMTSim. This environment is composed mainly of:

- A GUI application written in Java
- A colection of Perl and shell scripts
- DSMTSim modified with MPI calls.

Additionally, the following tools, already available on SWARM, were used as part of this simulation environment.

- The MPI library
- The LSF job queue management system

The GUI allows the user to configure the simulation parameters. Some of the information provided by the user includes: a variable to measure (IPC, cache miss rate etc), range of simulation (context number range), benchmark program to execute and format of output results. Once the user enters all this information, a text file is created with the script data needed to generate a desired simulation cycle.

A Perl script reads this simulation information and executes the load balancing software program in SWARM given the required number of simulation nodes. Once the simulation processes are sent to the queue of processes, simulation nodes are initialized by the MPI system.

During initialization, each node calculates its identification number. A particular simulation node to get simulation parameters such as the context number uses this number. A node in SWARM will execute the simulation corresponding to a single context execution, another to two contexts and so on.

Figure 4-9 shows partially the code used in the root node to spawn simulation nodes and to send/receive data from slave nodes.

During parallel benchmarking execution, the root node coordinates simulation execution but each node reads in parallel the information needed to perform a particular simulation run. When a node finishes one simulation cycle, it synchronizes with all other nodes in the MPI communication world and sends its results back to the root node. At this time the node simulation is free to start a new simulation cycle.

```
MPI_Init(&argv,&argc);  //initializes the MPI system
MPI_Comm-rank(MPI_COMM_WORLD,&id_num);//get id number for the node
MPI_Comm_size(MPI_COMM_WORLD,&num_proc);//get number of process
...
MPI_Barrier(MPI_COMM_WORLD);
getSimulationParameters(id_num);//obtain simulation parameters
do {
      PerformSimulation(id_num); // call DSMTSim to execute benchmark
      MPI_Barrier(MPI_COMM_WORLD); // synchronize nodes
      for (int i=0; i< num_data; i++)
      {
            getSimulationValue(i,&value);
            If (!id_num) //root process

            MPI_Recv(&value,1,MPI_FLOAT,I,tag,MPI_COMM_WORLD,&status)
            ;
            else // node process
                MPI_Send(&value,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
      }

} while(!simulations_done);
MPI_Finalize();
```

Figure 4-9. MPI code to send and receive simulation data in parallel bechmarking.

When all nodes terminate the total benchmark simulation cycle, they send their final simulation results to the root node. With this data, the root node creates a single file with the complete simulation results. Also at this point, the LSF system used by SWARM sends an email to the user indicating that a simulation cycle has ended.

At this point the user can execute the Java GUI to read the simulation results file and optionally to generate a graph of those results.

Figure 4-10 shows the components of the DSMT simulation software tool set on SWARM, and Figure 4-11 depicts the GUI used to configure simulation parameters for DSMTSim during parallel benchmarking.

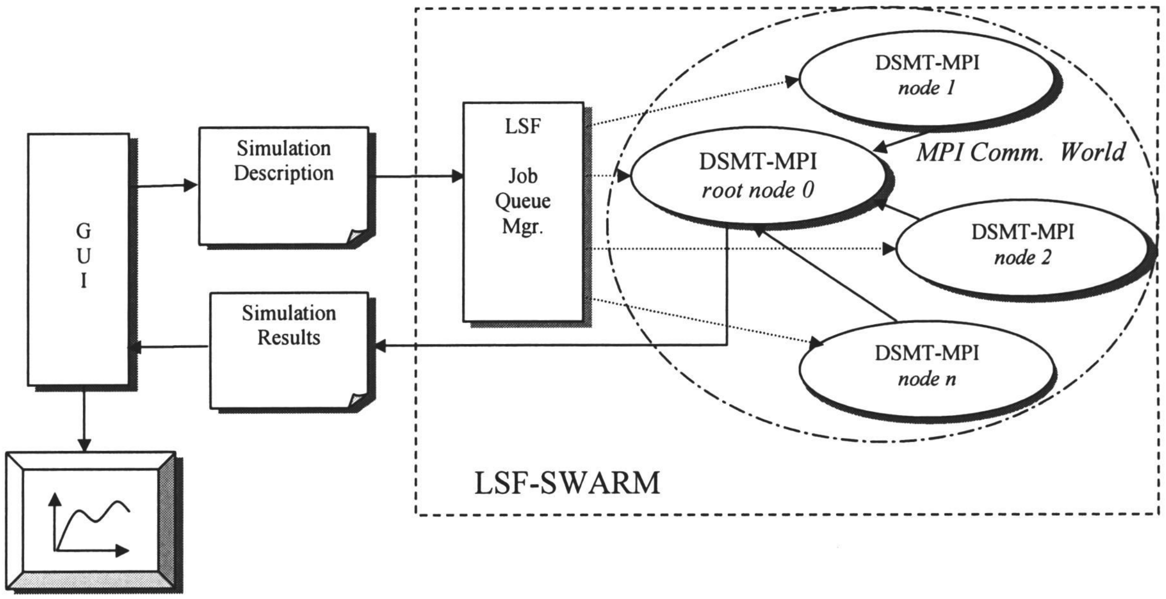
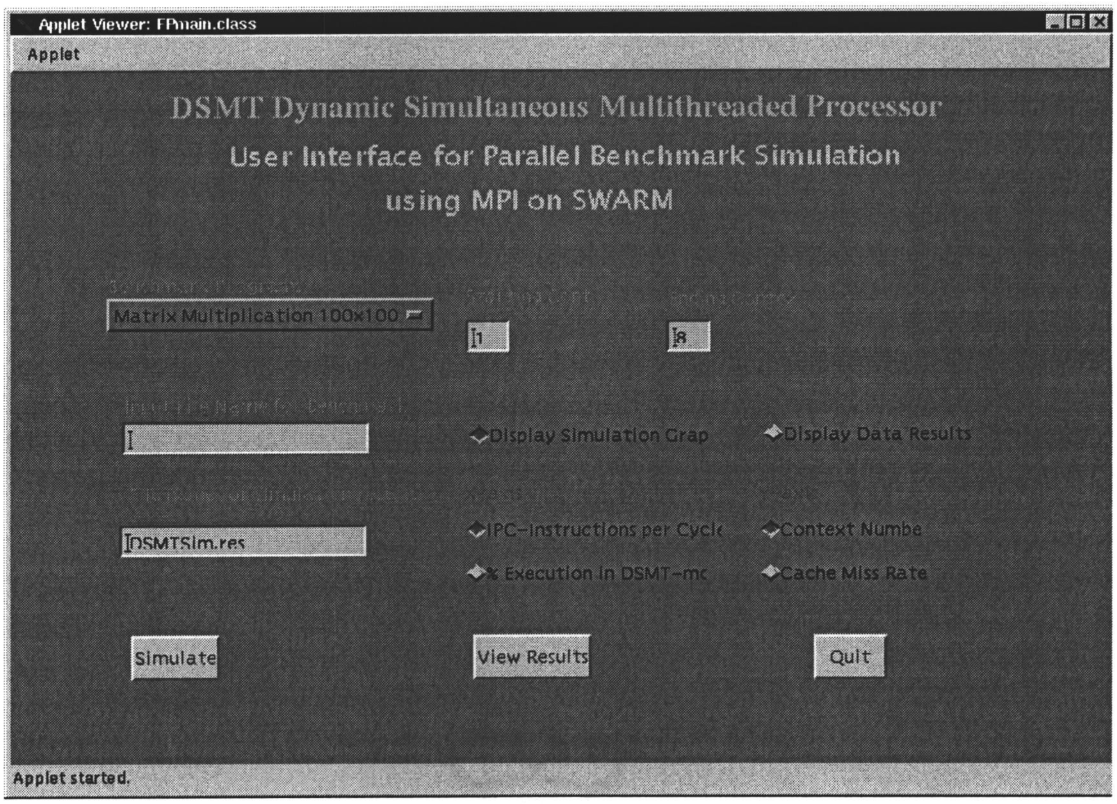


Figure 4-10. Parallel benchmarking on SWARM


Applet Viewer: FPmain.class

Applet

DSMT Dynamic Simultaneous Multithreaded Processor

User Interface for Parallel Benchmark Simulation

using MPI on SWARM

Matrix Multiplication 100x100

1

8

Display Simulation Grap

Display Data Results

DSMTSim.res

IPC-instructions per Cycle

Context Numbe

% Execution in DSMT-mc

Cache Miss Rate

Simulate

View Results

Quit

Applet started.

Figure 4-11. GUI for DSMTSim parallel benchmarking.

5 DSMT SIMULATION PERFORMANCE RESULTS

This section discusses the simulation results obtained during DSMT performance evaluation. First, the well-known Livermore loops are used to evaluate the effect of different fetching policies on DSMT. The Livermore loops where chosen for their features, generality, and variety. Livermore loops consist of 24 core calculations used in familiar numerical algorithms such as: matrix multiplication, Cholesky's conjugate gradient, Monte Carlo's search etc. All the kernels were compiled without modifications to the original C source code, using gcc with the O3 optimization flag turned on.

FU Type	Int ALU	Int Mul	Int Div	FP Add	FP Mul	FP Div	L/S
Number of FU	8	2	1	2	2	1	2
Reservation Stations	8	2	2	4	4	2	4

Table 5-1 DSMT functional unit configuration

DSMTSim's functional unit configuration used during simulation is based on Table 5-1. Table 5-2 shows the simulation parameters used in the experiments.

Instr. Queue size/context	L/S Queue size/context	ROB size/context	Inst/Data/L2.Cache	Shared BTB size
64	64	32 (2 bits)	128/128/256 KB, 2-way	2 K, 2-way

Table 5-2 DSMT configuration.

To assess the effect of the number of fetching ports and the issue width on DSMT's performance, an ideal configuration was used in the first in the experiments, i.e. the fetch unit was enabled to use as many ports as contexts are available in the processor. Also, each context was allowed to issue up to four instructions per clock cycle.

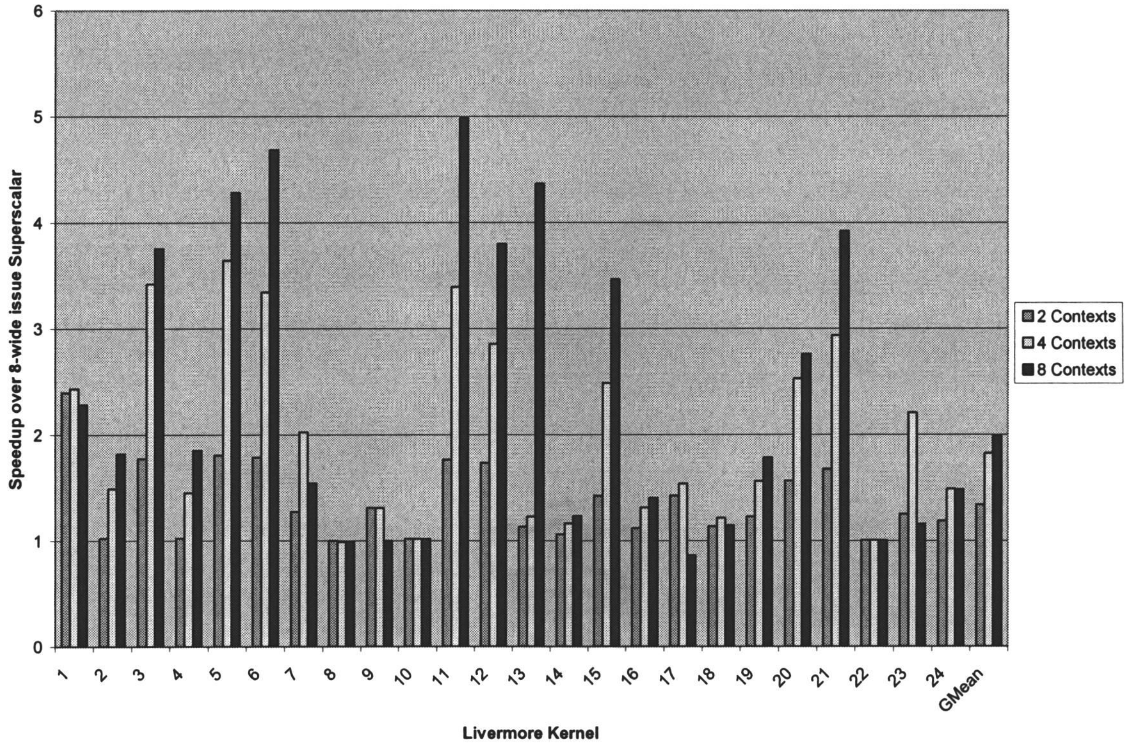


Figure 5-1. DSMT performance with variable number of fetching ports

Figure 5-1 shows the speedup obtained by the DSMT processor with 2, 4, and 8 contexts when compared to a single context 8-wide issue superscalar processor executing the same benchmarks. In the graph shown, GMean represents the geometric mean of the speedup obtained by all twenty four kernels.

As Figure 5-1 shows, the maximum speedup obtained by DSMT was 100% on average with 8 contexts, followed by 4 contexts with a speedup of 84% and 2 contexts with speedup of 34% on average. Performance results on Figure 5.1 show that for a few loops the speedup was negligible. However, statistics gathered by the simulator showed that in Kernel-8 (integration), for instance, DSMT's LSST mechanism has a high misprediction rate. This occurs because the pattern to access memory used in that loop is more complex, and therefore the simple value prediction method used by LSTT is unable to predict correctly the value of the induction variables used in that loop.

On the contrary, Kernel-10 (difference predictor) low performance is due to a combination of two factors: LSST's low prediction rate and high thread synchronization rate. As was explained in Section 3.7, a speculative thread gets blocked when due to dynamic conditions it is able to finish execution before the non-speculative context does. Since the non-speculative context is the only one capable of enabling other speculative contexts to become non-speculative, the gap produced when the non-speculative thread is delayed causes degradation in performance.

On the other hand, the other factor that is causing degradation in performance during the execution of kernel 22 (plankian distribution) is a high branch misprediction rate.

Figure 5-1 also indicates that DSMT does not always obtain the best performance from the maximum number of threads. The reason for this is twofold. First, since each context executes nearly the same code (when there are no conditional branches inside), the requirements of each thread in terms of processor resources are very similar during each iteration. Therefore, increased competition exists for the shared resources. This issue is especially critical in loops with either a very large number of numerical calculations, a large number of memory accesses or both, such as kernels: 8 (integration), and 10 (difference predictor).

Lastly, dynamic behavior inside loops is the other main cause for the poor performance exhibited by some loops on an eight context DSMT processor. Loops with dynamic behavior tend to generate higher inter-thread mispredictions. Mispredictions occur mainly when register values are read too early by the speculative threads as is in the case of Kernel-17 (conditional computation) which contains five *goto* statements in the loop body. In this loop, statistics gathered by the simulator indicate that mispredicted threads that are squashed when the end of loop was found cause the decrease in performance for eight contexts. An analysis of the code indicates that some internal backward jumps are confused by the loop detection mechanism as loops. However, since these backward branches are dependent of internal variables in the loop sometimes a backward branch is taken and some other times is not. Therefore, using the maximum number of contexts increases the probability that many more of threads will be squashed when the conditional branch is not taken.

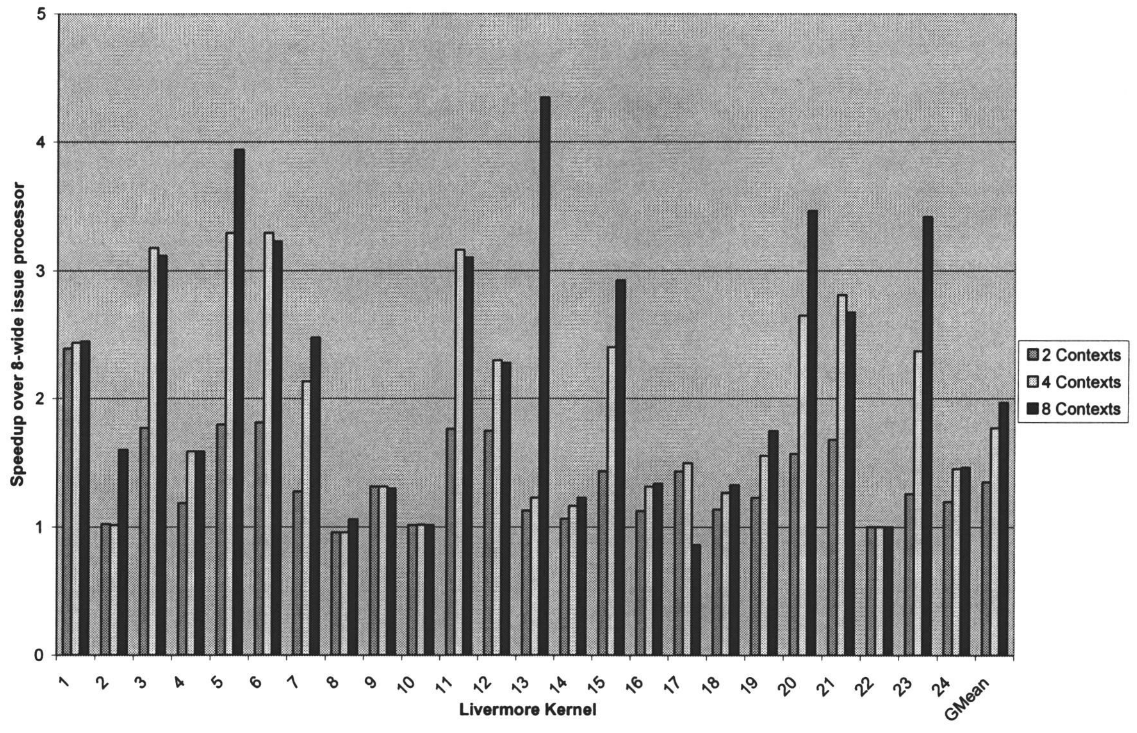


Figure 5-2. DSMT performance when ICount2.8-modified policy is used

Figure 5-2 shows DSMT performance when a more realistic configuration for the number of fetching ports and issue bandwidth is used for DSMT. In this simulation, the I-cache is two-ported [77] and the fetching policy used is similar to ICount2.8 [72]. However, as is described in Section 3.3.4.1, SMT's original ICount2.8 policy was slightly modified (called ICount2.8-modified).

Results in Figure 5-2 show that the difference in performance between the two configurations used (ideal and ICount2.8-modified) is 0%, 5%, and 3% on the average for 2, 4 and 8 contexts respectively. This indicates that the performance obtained using two fetching ports with ICount2.8-modified policy is very close to the one obtained by the ideal configuration. The reason for the small difference in performance is twofold. First, the policy used in DSMT to maintain the precise state of the processor forces the speculative threads that have reached the join state to wait for the non-speculative context before committing. Therefore, the non-speculative thread should be kept running as fast as possible so that speculative threads may be taken out of that waiting state as quickly as possible. This policy is enforced in ICount2.8-modified, which gives priority to the non-speculative thread. Second, due to the limited precision of the prediction mechanisms used (i.e. LSST and value prediction for registers and memory), an increased fetch bandwidth augments the probability that misspeculated instructions may enter the pipeline. Therefore, the competition in resources among non-speculative and speculative (but correctly predicted) instructions on one hand, and misspeculated instructions on the other, produces diminishing returns in performance.

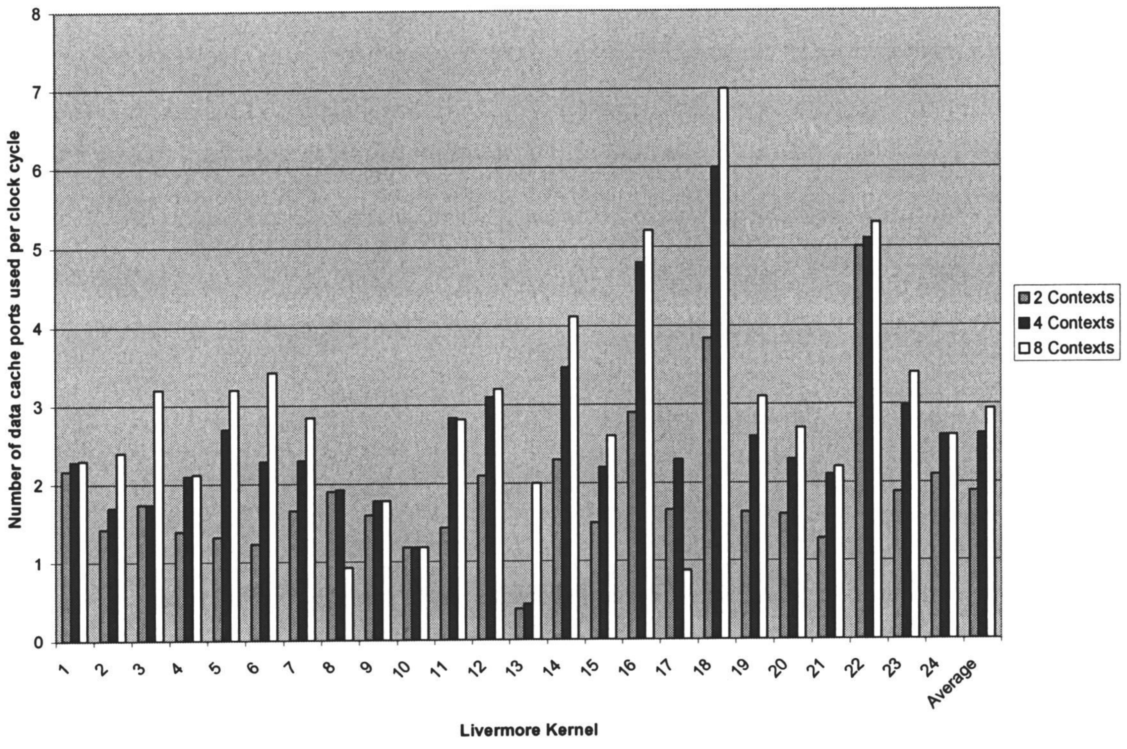


Figure 5-3. Data cache accesses per clock cycle

Figure 5-3 shows the average number of data cache ports accessed per clock cycle when the Livermore loops are executed on DSMT. As this figure indicates for most of the loops, with more contexts in the processor more pressure is exercised on the data cache ports every cycle. However, Kernel-8 and Kernel-17 reveal a different pattern in accessing data cache: with 8 contexts less data cache ports are used. DSMT statistics shows that in these loops LSST misspredictions occur very often (especially in Kernel-17) due to dynamic behavior in the loop and therefore, with more contexts, more threads are squashed reducing the pressure on the data cache. However, in these loops performance also degrades with more threads.

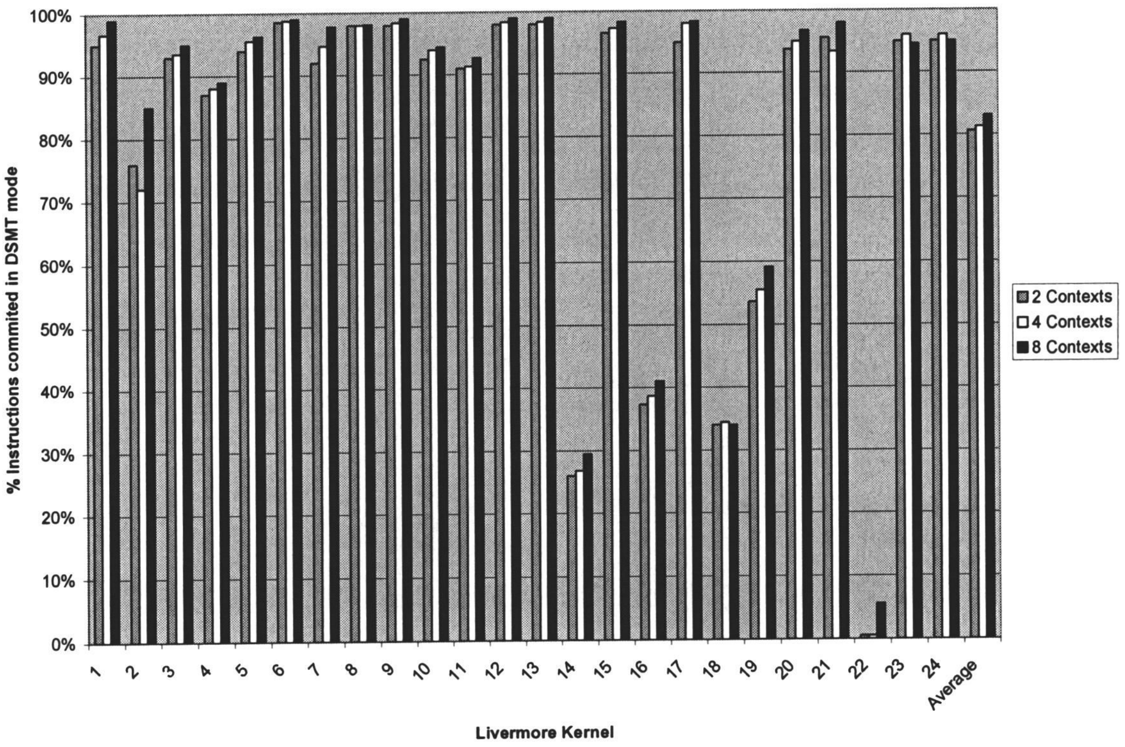


Figure 5-4. Instruction committed in DSMT mode for Livermore loops

Figure 5-3 suggests that the number of data cache ports required by DSMT is around 3. However, to take into account the more demanding requirements in terms of memory of more complex programs DSMT uses four fetching ports. It is interesting to note that reducing the fetch bandwidth of DSMT (using ICount2.8-modified), reduces also the pressure on the data cache ports. Therefore there is a tradeoff between increasing the fetch bandwidth to improve performance, and also limiting the fetch bandwidth if misspeculations occur very often.

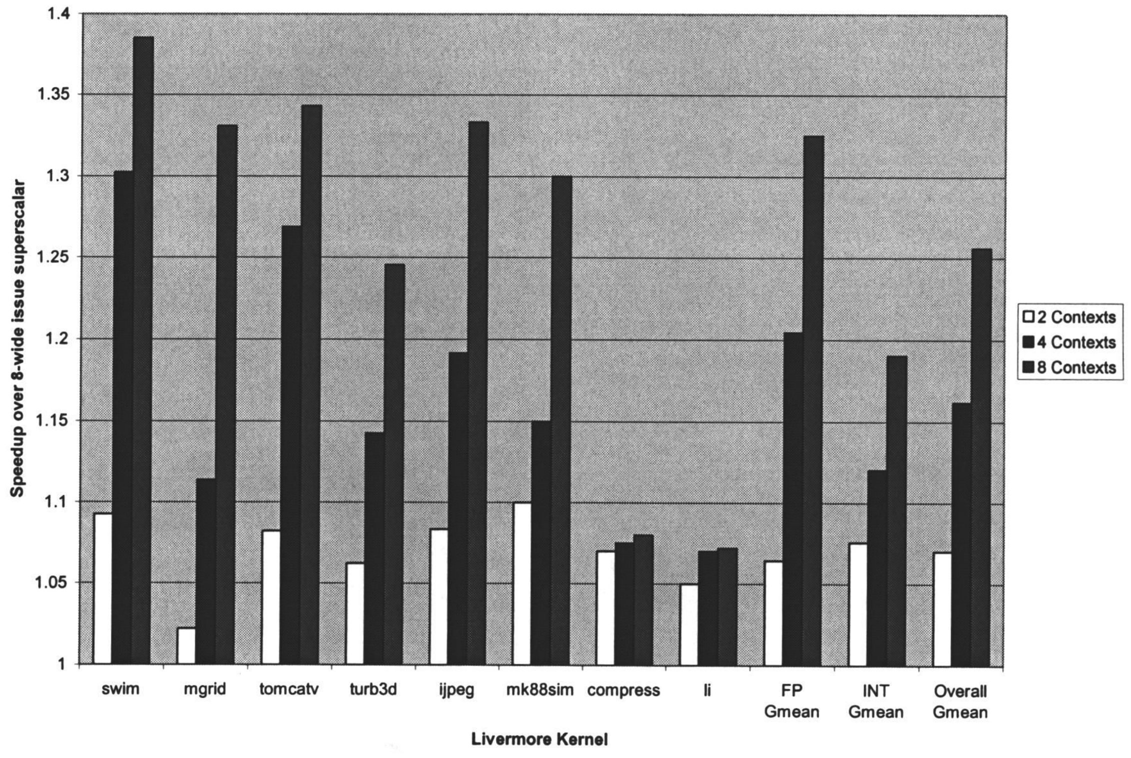


Figure 5-5. DSMT Performance using SPEC95 benchmarks

Figure 5-4 shows the percentage of instructions committed in DSMT mode for each of the Livermore loops. On the average nearly 80% of the instructions were committed in DSMT mode, which indicates that DSMT loop detection mechanism is very effective in detecting and exploiting these loops. However, notice that the number of committed instructions includes also instructions that could have been committed and later squashed due to misspeculation.

As Figure 5-4 indicates, DSMT is able to find very good amounts of TLP in kernels 8 and 10. However, as Figure 5-2 illustrates the speedup obtained in those loops is negligible due to misspeculation. On contrary, DSMT is unable to find enough TLP in kernel 22 due to a very high branch misprediction rate. DSMT statistics also show that Loops 14, 16 and 18 which have several internal loops and *if-then-else* statements in the loop body produce a relatively high number of branch mispredictions.

The Livermore loops were used to explore part of the design space of DSMT and to characterize its dynamic behavior. However, the real advantage of DSMT is when more complex applications are executed. Figure 5-5 illustrates DSMT performance results during the execution of several SPEC95 benchmarks.

All SPEC95 benchmarks were executed on DSMTSim using the configuration shown in tables 5.1 and 5.2 with an ICount2.8-modified policy. In the simulations 500 million of instructions were executed. The first 200 hundred million instructions corresponding to code initialization were skipped using the fast simulation mode provided by DSMTSim. These code sections are non-representative of the general behavior of a program because generally they contain many system calls used to read initialization or input files. The reference inputs of the SPEC95 benchmarks were used during all SPEC95 simulations.

Figure 5-5 shows the performance results obtained by DSMT for both the SPEC95 floating point (SPEC95-FP) and the integer benchmarks (SPEC95-Int). Results show that the average speedup obtained by DSMT in all benchmarks is 26% on average for 8 contexts, 16.5% for 4 contexts and approximately 7% for a two-context DSMT processor. Speedups obtained were relative to a single-threaded 8-wide issue superscalar processor with the same internal configuration as the DSMT processor.

As other previous studies have found [22, 39, 40, 70], in the DSMT architecture SPEC95-FP benchmarks provide the better speedup (32.5% on average with 8 contexts). The reason is that essentially they contain many more loops than SPEC95-Int. SPEC95-Int obtained an average speedup of 19% average speedup with 8 contexts. As these results show, numerical applications will benefit more from the DSMT model than non-scientific applications.

6 CONCLUSIONS

This dissertation presented efficient mechanisms in hardware to support multithreading on modern high performance processors. First, a simple model of a system with support in software and hardware for multithreading was introduced. The model describes the interaction between software and hardware thread schedulers, and how different scheduling policies affect processor utilization.

Then the *Dynamic Simultaneous Multithreading* (DSMT) processor was introduced and its simulation environment was described in detail. The simulation environment consists mainly of two simulators, designed and implemented especially for this research. During the development process of these simulators, it was acknowledged that object oriented technologies are specially suited for computer architecture simulation. However, a few simulators have been developed using this methodology. As occurs in other areas of software development, these methodologies ease the design, debug, testing and validation of a simulator. It was also envisioned that the features of an OO simulator jointly with the expressiveness of an XML [81] schema used to describe processor microarchitectures could create powerful simulation platforms for the next generation of simulators in computer architecture. For instance, XML could be used to describe microarchitectures, in such a way that simulators capable of processing XML could be able to read such descriptions and simulate the required architecture.

The DSMT architecture was described in detail and its performance results were presented and evaluated. DSMT employs aggressive forms of speculation to extract TLP and ILP from sequential programs dynamically. Unlike other similar architectures, DSMT uses simple mechanisms to synchronize threads and keep track of inter-thread dependencies, both in registers and memory. The novel mechanisms proposed in DSMT employ the information obtained during the sequential execution of code segments as a hint to speculate the subsequent behavior of multiple threads. Moreover, DSMT utilizes a novel greedy approach which chooses those sections of code that are more likely to provide the highest performance based on its past dynamic behavior.

The simulation results of DSMT were obtained using a very accurate simulator, which is capable of executing mispredicted paths of execution, jointly with run-time generation, control and synchronization of multiple threads. In contrast, other similar architectures that showed the potential of exploiting loops for improving performance used trace simulation [41, 39].

DSMT simulation results show that speculative dynamic multithreading based on extracting threads from loops only has very good potential to improve SMT's performance when only a single task is available for execution. DSMT obtained nearly 100% speedup executing the Livermore loops and 26% of improvement on average when the SPEC95 benchmarks were executed. However, the improvement in performance obtained by the DSMT model, which is based on exploiting only loops, is limited, especially for non-numerical applications. The reasons for this are threefold. First, the relative lack of loops, jointly with the limited parallelism available in the loops found in some applications creates a strong limitation for a model based on exploiting only loops such as the one used by DSMT. Second, Amdahl's law imposes an absolute limit on the performance that a processor may achieve within an application that consists of sequential and parallel sections of code [44]. Finally, as the simulations results have shown, the dynamic behavior of speculative multithreading causes frequent mispredictions in some loops that additionally to a higher resource competition among speculative and non-speculative threads may produce a detrimental effect on processor's performance.

Simulation results also showed that a tradeoff exists between increasing the amount of TLP that DSMT is capable of exploiting (increasing for instance the accuracy of the loop detection mechanism, the number of contexts, fetch bandwidth etc.) and reducing at same time the frequency of misspeculations.

There are a number of ways the DMST architecture can be improved. First, as [44] found, the combined exploitation of procedures and loops will provide higher improvements in performance. However, an analysis on the tradeoff between the complexities required by this *superspeculative* architecture and its expected overall performance needs to be evaluated. Second, as simulation results showed, more sophisticated value prediction mechanisms will lead to improving noticeably DSMT's performance at the cost of additional complexity.

An important bottleneck of DSMT observed during simulations, was the memory dataflow mechanism, which consists of MDRT, context's MOB and the data cache ports used. Long running threads that access a large number of memory locations may cause MDRT to quickly fill-up when the number of data cache ports is insufficient resulting in the entire pipeline to backup. This was the reason why the dynamic loop detection mechanism did not choose the outer-loop in some of the benchmarks, such as Kernel-21 (matrix multiplication), to clone threads. But, even when the middle-loop is chosen, its speculative threads generate a large number of loads and stores, especially stores that cannot commit. This causes the MDRT to backup and the result of this bottleneck percolated all the way back up to the IQs.

Also, the current limitation with the dynamic thread generation method employed by DSMT architecture is that the thread granularity is based on the chosen loop level. Although this approach of defining threads is consistent with a number of proposals in the literature [3, 22], a method that "throttles" the thread execution is needed to avoid filling up the MDRT too quickly. For example, in a doubly nested loop, the thread detection mechanism will clone each iteration of the outer-loop if it has been determined that cloning the inner loop resulted in a poor performance. However, since having a large number of iterations in the inner-loop may cause a bottleneck, each iteration of the outer-loop could be subdivided into smaller threads and cloned onto the multiple contexts. This approach combined with the already existing feedback mechanism to identify "*good*" threads will provide more flexibility in choosing an appropriate granule of threads.

Simulation results also show that choosing the best number of contexts for execution may be critical for some applications. However, selecting at run-time an optimal number of threads to exploit is difficult. However, one possible solution is to carry this information from the compiler to the architecture using a technique similar to the one described in [50].

Finally improving the branch prediction mechanism employed by DSMT, which is currently based on a simple 2-bit saturating counter, will increase also DSMT's performance on some benchmarks.

In summary, considering the very good improvement in performance obtained by DSMT, this architecture turns out to be more attractive when the low complexity of the mechanisms used to keep track of inter-thread dependencies in register and memory is considered jointly with all the advantages offered by its SMT core.

BIBLIOGRAPHY

1. Agarwal A., Lim B. H., Kranz D. and Kubiatowicz J., "APRIL: A Processor Architecture for Multiprocessing." *In Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pp. 104-114, June 1990.

2. Agarwal A., "Performance tradeoffs in Multithreaded Processors." *IEEE Transactions on Parallel and Distributed Systems*, 1991.

3. Akkary H. and Driscoll M., "A Dynamic Multithreading Processor." *31st Int'l Symp. on Microarch.*, Dec 1998.

4. Alverson R. *et a*l., "The Tera Computer System." *International Conference on Supercomputin*g, 1990, pp. 1-6.

5. Anderson T.E., Bershad, B.N., Lazowska, and Levy, H.M. 1992. "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism." *ACM Trans. on Comp. Syst. 10*, 4 (Feb), 53–70.

6. Arnold Ken, Gosling James, The Java Programming Language, *Addison-Wesley*, Reading Massachusetts, May 1996.

7. Austern M., Generic Programming and the STL, *Addison-Wesley*, 1998. ISBN 0-201-30956-4.

8. Bechanan C., *et al,* "An Integrated Functional Performance Simulator." *IEEE Computer*, May-June 1999.

9. Bomans L., Hempel R. and Roose D., "The Argonne/GMD Macros in FORTRAN for Portable Parallel Programming and their Implementation on the Intel iPSC/2." *Parallel Computing.* 15, 119 (1990).

10. Borkenhagen J. M., *et al.*, "A Multithreaded PowerPC Processor for Commercial Servers." *IBM Journal of Research and Development.* Vol. 44 No 6 Nov. 2000.

11. Brown A., and Patterson D.A., "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)." *High Performanace Processing Symposium*, Asilor, Ca, October 2001.

12. Bryan B., A, et al. "Can Trace-Driven Simulators Accurately Predict Superscalar Performance?." *In Proc. of International Conference on Computer Design*, pp. 478--485, Oct. 1996.

13. Budd T., An Introduction to Object Oriented Programming (2nd Edition), *Addison Wesley*, Reading, Massachusetts, 1997. 452pp. ISBN 0-201-82419-1.

14. Burger D., Austin T. M., The "SimpleScalar Tool Set, Version 2.0." *Computer Architecture News,* Vol. 25, No. 3, pp. 13-25, June 1997. http://www.simplescalar.org

15. Butenhof D. R., Programming with POSIX Threads, *Addison Wesley*, 1997.

16. Cagney A., PSim PowerPC simulator. http://sources.redhat.com/psim

17. Clemson University. MIPS Superscalar Simulator (for Sparc-Solaris big-endiand platform). http://www.eng.clemson.edu/~ilp/sim.html.

18. Diep T. A., and Shen J. P., "VMW: A Visualization Based Microarchitecture Workbench." *IEEE Computer*, 28(12):57--64, December 1995.

19. Diefendorff K., "Compaq Chooses SMT for Alpha." *Microprocessor Report*, Vol. 13, No. 16, December 6, 1999, p. 1.

20. Flynn M. J., Hung P., Peymandoust A, "Using Simple Tools to Evaluate Complex Architectural Trade-offs." *IEEE Micro magazine*, July-August 2000.

21. Hall M., W. *et al.*, Maximizing Multiprocessor Performance with SUIF Compiler. *IEEE Computer*, Dec. 1996.

22. Hammond M., Wiley M., and Olukotun K., "Data Speculation Support for a Chip Multiprocessor." *Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

23. Hennessy J., and Patterson, D., Computer Architecture-A Quantitative Approach, 2nd Edition, *Morgan Kaufmann*.

24. Hily S., and Seznec A., "Out-of-order Execution may not be Cost Effective on Processors Featuring Simultaneous Multithreading." *International Symposium on High-Performance Computer Architecture*, January 1999.

25. Hily S., et al., "Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading." *INRIA internal report* 3115, 1997.

26. Hirata H., et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads." In the *19th Annual International Symposium on Computer Architecture* (May). 136–145.

27. Huan J., The Simulator for Multithreaded Computer Architecture (SIMCA). http://www.mount.cs.umn.edu/Research/Agassiz/simca.html

28. Hwang K., Advanced Computer Architectures, Parallelism, Scalability, Programmability. *McGraw-Hill*, Inc. 1993.

29. Intel. Introduction to Hyper-Threading Technology. Document number 250008-002. 2001.

30. Issac R. D., "The future of CMOS technology." *IBM Journal of Research and Development.* Vol. 44 Num 3, 2000.

31. Johnson M. Superscalar Microprocessor Design. *Prentice Hill*, Englewood Cliffs, NJ, 1990.

32. Kobayashi M., "Dynamic Characteristics of Loops." in *IEEE Transactions on Computers*, C-33(2), pp. 125-132, 1984.

33. Krishnan V., Torrelas J., "Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support." *International Conference on Supercomputers*, 1998.

34. Kwak, H., "A Performance Study of Multithreading." Ph. D Thesis OSU, 1998.

35. Lee B, Kwak H., T., Carlson R., Yoon S. H., Han W. J., "Simulation Study of Multithreaded Virtual Processor." *IASTED International Conference on Parallel and Distributed Systems (EuroPDS98)*, Vienna, July 1-3, 1998.

36. Lipasti M. H., Shen J. P., "Exceeding the Dataflow Limit via Value Prediction." *The 29th Annual International Symposium on Microarchitecture,* pp. 226-237, December 1996.

37. Lo J., *et a*l., "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading." *ACM Transactions on Computer System*s, Aug. 1997, pp. 322-354.

38. Lusk E. L., Overbeek R. A., "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors." *Technical Report No. ANL-84-51*, Rev. 1, Argonne National Laboratory, June 1987.

39. Marcuello P., Gonzales A., Tubella J., "Speculative Multithreaded Processors." *International Conference on Supercomputing*, pp 77-84, 1998.

40. Marcuello P., and Gonzalez, A., "Control and Data Dependence Speculation in Multithreaded Processors." *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (MTEAC'98), January 1998.

41. Marcuello P., and Gonzales, A., "Clustered Speculative Multithreaded Processors." *Proceedings of ICS'99.*

42. Moshovos B., Vijaykumar T. N., and Sohi G. S., "Dynamic Speculation and Synchronization of Data Dependences." in *Proc. of Int. Symp. on Computer Architectur*e, pp. 181-193, 1997.

43. Moura C., SuperDLX a Generic Superscalar Simulator. ACAPS technical memo. McGill University, School of Computer Science. April 13,1993.

44. Oplinger J., Heine D., Lam M. S., "In Search of Speculative Thread Level Parallelism." *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, October 1999.

45. Oregon State University, Computer Science Department. SWARM cluster. http://www.cs.orst.edu/swarm.

46. Ortiz-Arroyo D., Lee B., and Yu C., "EMSim: An Extendible Simulation Environment for Studying High-Performance Microarchitectures." *The 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2002)*, July 14-18, 2002, Orlando, Florida.

47. Pacheco P., "Parallel Programming with MPI." *Morgan-Kauffman Publishers Inc.* 1996.

48. Pai V. S., Parthasarathy R., Adve S. V., "RSIM: An Execution-driven Simulator for ILP-based Shared-memory Multiprocessors and uniprocessors." *IEEE TCCA Newsletter.* October 1997. http://www-ece.rice.edu/~rsim

49. Palacharla S., Jouppi N. P. and Smith J. E., "Complexity-Effective Superscalar Processors." in *Proc. of Int. Symp. on Computer Architecture*, pp.206-218, 1997.

50. Puppin D., Tullsen D., "Maximizing TLP with Loop-parallelization on SMT." *Proceedings of the 5th Workshop on Multithreaded Execution, Architecture, and Compilation*, Austin, Texas, December, 2001.

51. Redstone J. A., Eggers S. J. and Levy H. M., "An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture." *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

52. Rosemblum M. *et al.* "Using the SimOS Machine Simulator to Study Complex Computer Systems." *In ACM TOMACS Special Issue on Computer Simulation*,1997.

53. Rotenberg E., Bennet S., Smith J., "Trace Processors: Exploiting Hierarchy and Speculation." IEEE Transactions on Computers.

54. Saavedra-Barrera, R., Culler, D.E., and von Eicken, T. "Analysis of Multithreaded Architectures for Parallel Computing." *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architecture*, July 1990, pp. 169-178.

55. Sazeides Y., Smith J. E. "The Predictability of Data Values." *The 30th Annual International Symposium on Microarchitecture*, pp. 248-258, December 1997.

56. Sirer E. G., "Measuring Limits of Fine Grain Parallelism." Undergraduate thesis, *PrincetonUniversity*,1993.http://www.cs.washington.edu/homes/egs/mipsi/mipsi.html

57. Smith J. E., Pleszkun A. R. "Implementing Precise Interrupts in Pipelined Processors." *IEEE Transactions on Computers,* Vol. 37, pp. 562-573, May 1988.

58. Smith J. E., and Sohi, G. S., "The Microarchitecture of Superscalar Processors." *Proceedings of the IEEE,* December 1995.

59. Smith J. E., "A Study of Branch Prediction Strategies." *The 8 th Annual International Symposium on Computer Architecture,* pp. 135-148, May 1981.

60. Socket++ source code and user manual. http://www.cs.utexas.edu/users/lavender/courses/socket++/

61. Sohi G. S., Breach S. E., Vijaykumar T. N., "Multiscalar Processors." *The 22nd Annual International Symposium on Computer Architecture,* pp. 414-425, June 1995.

62. Sohi G. S., Roth A., "Speculative Multithreaded Processors." Internal Report. University of Wisconsin", 2001.

63. Srivastava A., and Eustace A., "ATOM: A System for Building Customized Program Analysis Tools." in *Proc. of the 1994 Conf. on Programming Languages Design and Implementation*, 1994.

64. Steffan J. G., and Mowry T. C., "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization." *Proceedings of the Fourth Int'l Conf. on High-Performance Computer Architecture (HPCA-4),* Feb. 1998.

65. Sun Microsystems. MAJC Architecture Tutorial. 2000.

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

67. Tremblay M., "MAJC: An Architecture for the New Millennium." *Proc. Hot Chips 11*, pages 275-288, Aug. 1999.

68. Tsai, J., *et a*l.,"The Superthreaded Architecture: Thread Pipelining with Run-TimeData Dependence Checking and Control Speculation." *Proceedings of International Conference on Parallel Architectures and Compilation Technique*s, Oct. 1996.

69. Tsai J., Jiang Z., Ness E., Yew P. C. "Performance Study of a Concurrent Multithreaded Processor." *The 4th International Symposium on High-Performance Computer Architecture,* pp. 24-35, February 1998.

70. Tubella J., and González A., "Control Speculation in Multithreaded Processors through Dynamic Loop Detection." in *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture (HPCA-4).*

71. Tullsen D. M., "Simulation and Modeling of a Simultaneous Multithreading Processor". In *Computer Measurement Group Conference*, December 1996.

72. Tullsen D. M, Eggers S. J., Emer J., Levy H. M, Lo J. L., and Stamm R. L. "Exploiting choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." In *International Symposium on Computer Architecture*, May 1996.

73. Tullsen, D. M, *et al*., "Simultaneous Multithreading: Maximizing On-Chip Parallelism." *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 392-403.

74. VajapeyamS. and Mitra T., "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences." *Proc. of the Int. Symp. on Computer Architecture*, pp. 1-12, 1997.

75. Waldspurger C. A., Weihl W. E., "Register Relocation: Flexible Contexts for Multithreading." *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 1993.

76. Wallace S., Calder B., Tullsen D. M., "Threaded Multiple Path Execution." *The 25th Annual International Symposium on Computer Architecture,* June 1998.

77. Wilson K., M., Olukotun K., and Rosenblum M., "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors." *Proc. of the 23rd International Symposium on Computer Architecture*, June 1996.

78. Wolf M. E., Improving locality and Parallelism in Nested Loops. Ph.D. thesis, Stanford University, Computer Systems Laboratory, August, 1992.

79. Wolf M., Willis L., SATSim a Superscalar Architecture Trace Simulator Using Interactive Animation. *Workshop on Computer Architecture Education*, June 2000.

80. Woo S. C., *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations." In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, 1995.

81. XML Extensible Markup Language. http://www.w3.org/XML.

82. Zilles C. B., Emer J. S., Sohi G. S., "The Use of Multithreading for Exception Handling." *Proceedings of Micro-32*, 1999.