AN ABSTRACT OF THE THESIS OF

Hantak Kwak for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on December 7, 1998. Title: A Performance Study of Multithreading.

Abstract approved: _____

Ben Lee

As the performance gap between processor and memory grows, memory latency will be a major bottleneck in achieving high processor utilization. *Multithreading* has emerged as one of the most promising and exciting techniques used to tolerate memory latency by exploiting thread-level parallelism. The question however remains as to how effective multithreading is on tolerating memory latency. Due to the current availability of powerful microprocessors, high-speed networks and software infrastructure systems, a cost-effective parallel machine is often realized using a network of workstations. Therefore, we examine the possibility and the effectiveness of using multithreading in a networked computing environment. Also, we propose the Multithreaded Virtual Processor model as a means of integrating multithreaded programming paradigm and modern superscalar processor with support for fast context switching and thread scheduling. In order to validate our idea, a simulator was developed using a POSIX compliant Pthreads package and a generic superscalar simulator called SimpleScalar glued together with support for multithreading. The simulator is a powerful workbench that enables us to study how future superscalar design and thread management should be modified to better support multithreading. Our studies with MVP show that, in general, the performance improvement comes not only from tolerating memory latency, but also due to the data sharing among threads.

A Performance Study of Multithreading

by

Hantak Kwak

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed December 7, 1998

Commencement June 1999

Doctor of Philosophy thesis of Hantak Kwak presented on December 7, 1998

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Head of Department of Electrical and Computer Engineering

Dean of Graduate School

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

Redacted for Privacy

Hantak Kwak, Author

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor, Dr. Ben Lee. He has inspired me throughout my academic years, and this thesis work would have not been possible without his encouragement and patience. Also, I wish to thank Dr. James Herzog, Dr. Shih-Lien Lu, Dr. Un-Ku Moon, and Dr. Mina Ossiander for acting as my program committees.

My friends and colleagues at the Department of Electrical and Computer Engineering have always offered an inspiring and friendly atmosphere. In particular, I would like to thank Ryan Carlson, Mark Dailey, Daniel Ortiz, Mike Miller, YongSeok Seo, YunSeok Kim, KyoungSik Choi, HoSeok An, and many others for their help.

My special thank goes to all my families for their understanding and devotion. They made my life joyful and worthwhile. I would like to thank my father for his unconditional support and encouragement, wife for believing in me, and parent-in-laws for their kindness. Finally, this thesis is dedicated to my mother who is fighting her illness.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF TABLES

A PERFORMANCE STUDY OF MULTITHREADING

1. Introduction

In an effort to overcome the limitations of circuit technology, computer architects have resorted to exploiting instruction-level parallelism (ILP) for improving processor utilization. In addition, techniques such as on-chip caches, superpipelining, long instruction words, branch prediction and speculative execution are also employed. The superscalar architecture, which almost all of the current microprocessors are based on, has been very successful in exploiting ILP. However, recent studies have shown that the single-threaded paradigm used by conventional programming languages and run-time systems do not fully utilize the processor's capabilities. This is due to the fact that the advances in VLSI technology have led to faster clocks and processor designs that can issue multiple instructions per cycle; yet the performance of memory system has not increased proportionally. The speed of commercial microprocessors has increased by a factor of twelve over the past ten years while the speed of memories has only doubled [24].

The widening of processor-memory gap reduces the processor utilization since the processor has to spend more cycles for memory accesses. Also, stalls due to cache misses severely degrade the overall performance by disrupting the pipeline. The result is the decrease in the number of potential instructions that can be issued in each cycle for superscalar, thus effectively reducing the ILP. Consequently, no matter what microarchitectural techniques are used, the performance of a processor will soon be limited by the memory characteristics. Therefore, this thesis examines an alternative technique, called multithreading, as a solution to the memory latency and the limited ILP problems.

Multithreading has emerged as one of the most promising and exciting techniques to tolerate the memory latency. While a single threaded model of computation

does not provide mechanisms to tolerate the memory latency, multithreading provides the processor with a pool of threads and context switching occurs between the threads not only to hide memory latency, but also other long latency operations such as I/O and synchronization operations. The processor may also interleave instructions on a cycle-by-cycle basis from multiple threads to minimize pipeline breaks due to dependencies among instructions within a thread. This allows the threads to dynamically share the processor's resources and increases the processor utilization.

To provide better understanding of multithreading, Chapter 2 introduces analytical models of multithreading that characterize the performance of multithreading as well as its limitations. Also, a comprehensive survey of multithreaded systems is provided that includes programming languages, compiling techniques and various architectures along with the latest multithreading techniques.

As an alternative to expensive Massively Parallel Processors (MPP) or Symmetric Multiprocessor (SMP) systems, it is possible to build a low-cost parallel machine by utilizing a network of workstations (NOWs). A software infrastructure system facilitates a construction of a virtual parallel machine using a collection of workstations. A shared-memory abstraction can also be provided on top of physically distributed memory system. However, the performance of a parallel machine based on NOWs may suffer from long and unpredictable memory latency due to the distributed nature of the underlying physical memory system. Therefore, Chapter 3 studies the viability and effectiveness of multithreading in a network-based multiprocessor system. In particular, we examine a matrix multiplication problem on a network of workstations to see how effective multithreading is in a distributed shared-memory (DSM) environment.

Finally, Chapter 4 investigates how future processors can adopt the multithreaded architecture such that the departure from current microarchitecture is minimal, yet provides better performance. We propose the Multithreaded Virtual Processor

(MVP) system that exploits the synergy between the multithreaded programming paradigm and the well-designed contemporary microprocessors.

2. Multithreading

In the last decade, researches on multithreading techniques have led to substantial improvements over traditional single threaded abstractions. This chapter provides a summary of these efforts in designing multithreaded systems that includes new programming languages, new compiling techniques, and architectures. The organization of the chapter is as follows: Section 2.1 presents analytical models of multithreading. Section 2.2 discusses the multithreading in terms of programming models. Section 2.3 provides an overview of various multithreaded architectures along with their key features. Finally, Section 2.4 presents current research efforts in multithreading.

2.1 Performance Models

A multithreaded system contains multiple *loci of control* (or threads) within a single program, and the processor switches among threads to hide long latencies. Therefore, multithreading allows the exploitation of thread-level parallelism and improves the processor utilization. However, there is a limitation to the improvement that can be achieved. The most important limitation is due to the fact that applications running on a multithreaded system may not exhibit sufficiently large degrees of parallelism to permit the identification and scheduling of multiple threads to the processor. Even if sufficient parallelism exists, the cost of multithreading should be traded off against any loss in performance due to active threads sharing the cache and processor cycles wasted during context-switch.

The performance of a multithreaded processor depends on several architectural and program parameters—the memory latency, the remote reference rate, the number of threads, the thread length, the context switching cost, etc. A simple analytical model of

multithreaded processor behavior provides a better understanding of (1) how these parameters are interrelated, (2) how the number of threads affects the processor utilization, (3) how much context switching costs contribute to the overall performance, and (4) how the program should be partitioned into threads.

2.1.1 Basic Model

Saavedra *et al.* proposed a simple multithreaded processor model based on a set of parameters that reflects the architectural and software characteristics [76]. Assume the processor switches between threads only on long latency operations, such as remote memory accesses. Let L denote a fixed latency for such operations. Let R be the average amount of time that each thread executes before encountering a long latency operation. Let C be the fixed overhead in switching between threads. The processor utilization of a single thread model can be described by

$$U_1 = \frac{R}{R+L}. \tag{2.1}$$

Equation 2.1 shows that the utilization is limited by the frequency of long latency operations (i.e., $1/R$), and the average time required to service the long latency operation L. Therefore, the large memory latency has an adverse effect on the utilization in a processor with a single thread.

If L is much larger than the time to context-switch between threads, C, then useful work can be performed during long latency operations. In addition, if the number of threads is sufficiently large, the latency can be completely hidden as shown in Figure 2.1. In such a case, the processor utilization can be described as

$$U_{sat} = \frac{R}{R+C}, \tag{2.2}$$

where the number of threads required to totally mask L is assumed N_{sat}. This saturation number of threads satisfies the following requirement:

$$N_{sat} \geq \frac{R+L}{R+C}.$$

Note that increasing the number of threads beyond N_{sat} will not increase the processor utilization.

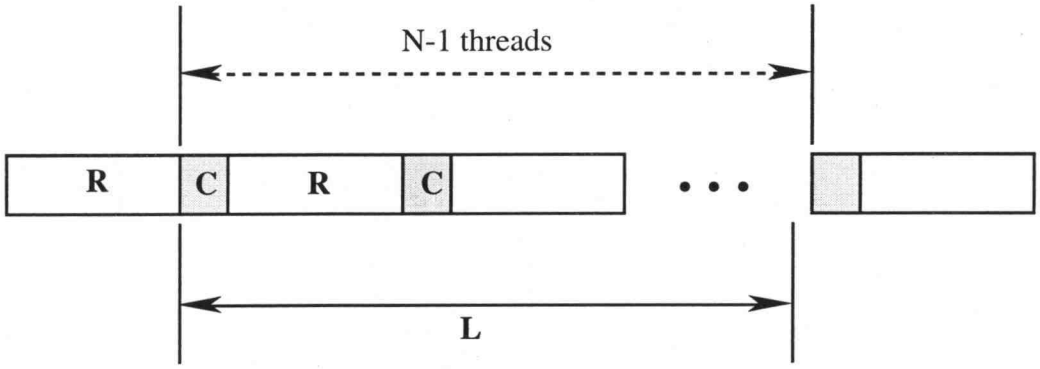


Figure 2.1: Memory latency L is completely hidden in saturation.

If there are insufficient number of threads to totally mask the latency L, the processor utilization can be described by

$$U_N = \frac{NR}{R+L},$$

where N is the number of threads. This equation shows that the utilization increases linearly with the number of available contexts until the number reaches the saturation point N_{sat}. Note that U_{sat} is the upper limit on the utilization achievable through multithreading in this performance model. Using the above equations, the speedup that can be achieved is given by

$$S_N = \frac{U_N}{U_1} = \left\{ \begin{array}{ll} N & \text{if } N < N_{SAT} \\ \dfrac{R+L}{R+C} & \text{otherwise} \end{array} \right\}.$$

As shown above, the minimum number of threads needed to achieve maximum utilization, $N_{sat} \geq (R+L)/(R+C)$, depends on time between context-switches (R), the time to service long latency operation (L), and the context switching overhead (C). For example, assuming a fine grained multithreaded system with R=1 and the context switching overhead is negligible (e.g., using multiple hardware contexts), the optimum utilization requires at least (1+L) threads. When C is not negligible, R should be much larger (i.e., coarser-grain multithreading) to achieve useful performance gains. Also, the improved processor utilization comes at an increased cost of supporting multiple threads, such as additional hardware for multiple contexts and the processor cycles wasted for context switching.

The above model ignored the performance impact due to higher cache miss rates in a multithreaded system and higher demands on the network placed by higher processor utilization. In addition, the above model assumed fixed latencies, and fixed frequency of long latency operations in threads. Assuming a context-switch occurs on every cache miss, we can equate cache miss rate m with the frequency of long latency operations, $p = 1/R$. Then, the speedup of a multithreaded system can be rewritten as

$$S_N = \frac{U_N}{U_1} = \left\{ \begin{array}{ll} N & \text{if } N < N_{sat} \\ \dfrac{1+mL}{1+mC} & \text{otherwise} \end{array} \right\}.$$

The cache miss penalty is the primary contributor to L. Note that we assume constant cache miss rate and miss penalty in the above equation. The effect of context switching on other long latency operations such as synchronization delays can also be added to above equation.

2.1.2 Cache and Network Model

In the previous section, it is assumed that cache miss rate and miss penalties are not affected by multithreading. The analytical performance model, proposed by Agarwal [6], considers multithreading with cache interference, network contention, and context switching overhead. In multithreaded processors with caches, multiple threads are simultaneously active and they interfere with each other, creating a higher cache miss rate and a higher network contention. The cache miss rate is negatively affected by increasing the degree of multithreading. Likewise, the miss penalty increases with the number of threads due to higher network utilization—leading to longer delays in accessing remote memory modules. Therefore, a mathematical model of multithreaded processors must reflect these characteristics. In conducting this analysis, the following assumptions are made:

- All threads resident in a processor have the same cache miss rate and working set size.
- The processes execute useful instructions between cache misses.
- Context-switch occurs only on a cache miss and the processor cycles spent for context switching are considered wasted.

- Network requests occur only on cache misses and cache invalidation.
- Data sharing and compulsory cache misses (i.e., misses that occur when the part of resident working set or new blocks are brought in the cache) are ignored.

The parameters used to develop the model are as follows:

- p - the number of threads resident in a processor.
- C - context switching overhead.

Suppose that a process executes useful instructions for R cycles, and then a cache miss occurs. Assuming the cache miss requires L cycles before the process can resume, Equations 2.1 and 2.2 can be rewritten such that the processor utilization is expressed in terms of cache miss rates $m(p)$. Since cache the miss rate is simply the inverse of R, the utilization is then,

$$U(p) = \begin{cases} \dfrac{p}{1+Lm(p)}, & \text{for } p < \dfrac{1+Lm(p)}{1+Cm(p)} \\ \dfrac{1}{1+Cm(p)}, & \text{for } p \geq \dfrac{1+Lm(p)}{1+Cm(p)}. \end{cases}$$

First, consider the network effect. The cache miss service time, L, depends on the network latency and memory access time. In particular, for a packet-switched k-ary n-cube interconnection network with cut-through routing, the network parameters and assumptions can be defined as follows:

Parameters

- M - memory access time.
- B - message size.
- k - network radix.
- n - network dimension.

- ρ - network channel utilization.
- k_d -average distance a message travels.
- h - network switch delay (i.e., the number of hops a message make).

Assumptions

- Uniform traffic rate for all nodes.
- Uniform distribution of message target nodes.
- Infinite buffering at each intermediate node.

Then, the average L for buffered k-ary n-cube direct network is given in [2] as

$$T(p) = \left[1 + \frac{\rho B \frac{1}{k_d}\left(1 - \frac{1}{k_d}\right)}{(1-\rho)} \right] h + M + B - 1. \tag{2.3}$$

This cache miss service time is calculated as the sum of memory access time, M, the pipeline latency of the message size, B-1, and network switch delay h (i.e., $h/2$ for request and $h/2$ for response). Assuming separate communication channels for both directions, the expected number of hops, k_d, between two randomly chosen node in a 1-dimensional array can be estimated by the ratio of the sum of distances for all source-destination pairs to the total number of such pairs [2].

$$k_d = \frac{k^2 - 1}{3k} \approx \frac{k}{3}. \tag{2.4}$$

The probability of a network request on any cycle is $2p/[R+L]$ (the factor two accounts for both memory requests and responses), and the channel utilization factor ρ is given in [6] as

$$\rho = \frac{2p}{R+L} \cdot \frac{Bnk_d}{2n} = \frac{p}{R+L} Bk_d . \tag{2.5}$$

Substituting Equation 2.4 and 2.5 into 2.4, and setting $R = 1/m(p)$, we arrive at

$$L = \frac{T_o}{2} + \frac{Bpk}{6} - \frac{1}{2m(p)} + \frac{1}{2}\sqrt{\left(T_o - \frac{Bpk}{3} + \frac{1}{m(p)}\right)^2 + 8pB^2 n\frac{k}{3}\left(1 - \frac{3}{k}\right)},$$

where $T_0 = h + M + B - 1$. Note that L increases almost linearly with p, the number of threads.

Now consider the cache effect. The cache model must characterize the increase in the miss rate as the number of threads resident in a processor increases. The assumptions in this model include:

- direct-mapped cache and uniform address mapping
- working-set cache model (i.e., only small portion of program is required in the cache at a time for the execution of a program)

In order to study the impact of multiple threads on cache miss rates, the cache misses are classified into four categories [3]: *nonstationary*, *intrinsic interference*, *multiprogramming*, and *coherence-related* misses. The nonstationary misses, m_{ns}, occur when a miss brings a new block into the cache for the first time. The intrinsic interference, m_{intr}, results when the blocks interfere with each other in the cache during block replacements. The multiprogramming misses, $m(p)$, account for the cases when one process (or

thread) displaces the cache blocks of another thread. Coherency related invalidation, m_{inv}, occurs in multiprocessor systems where the changes made in one processor may require invalidation of other processor cache entries. The cache model is based on the following additional parameters.

- S - cache size.
- u - working set size in blocks.
- τ - period between measurement of working set.
- c - collision rate used for interference misses.
- v - size of the carry-over set.
- m_{fixed} - fixed miss rate assumed in this model, i.e., $m_{fixed} = m_{ns} + m_{inv}$.

The computation of intrinsic interference and multiprogramming miss rate requires the estimation of the carry-over set size. The size of carry-over set, v, is the number of blocks a thread leaves behind in the working set when it switches out. Assuming every block maps into the cache with the same probability $1/S$, the v is then given by

$$v = S\left[1-(1-1/S)^{u}\right], \tag{2.6}$$

where the term $\left[1-(1-1/S)^{u}\right]$ indicates the probability that at least one block maps into a cache set. When $S >> 1$, Equation 2.6 simplifies to

$$v = S(1-e^{-\frac{u}{S}}).$$

Let $v'(p)$ denote the steady state carry-over set size under the condition that p processors share the cache for short duration of context switching. In a multithreaded cache, a thread effectively sees a smaller cache, resulting in an increase in both the intrinsic interference and multiprogramming miss rates [6]. Also, the additional context switching component of miss rate, $m'_{cs}(p)$, is introduced due to restoring of thread's displaced blocks when a new thread is scheduled on a processor. The $m'_{cs}(p)$ can be obtained as a function of $v'(p)$ as

$$m'_{cs}(p) = \frac{v - v'(p)}{v} \cdot \frac{u}{\tau}. \tag{2.7}$$

If we consider the probability that a block of intervening $(p-1)$ threads maps to the top of a block, an alternative expression for $m'_{cs}(p)$ is also obtained:

$$m'_{cs}(p) = \frac{v'(p)}{S}(p-1) \cdot \frac{u}{\tau}. \tag{2.8}$$

By equating (2.7) and (2.8), the $v'(p)$ is expressed as

$$v'(p) = \frac{v}{1 + v\frac{(p-1)}{S}}. \tag{2.9}$$

Equation 2.9 signifies two occasions: (1) when the cache is very large $(S >> v)$, the approximation $v'(p) \approx v \approx u$ holds, indicating the cache can hold the entire working set of all the threads, and (2) when $S = v$, the effective size of cached working set of each thread becomes v/p.

The intrinsic interference miss rate in a direct-mapped cache given in [1] is

$$m_{intr} = \frac{c}{\tau}\left[u - S \cdot bin(u, \frac{1}{S}, d = 1)\right] = \frac{c}{\tau}\left[u - u\left(1 - \frac{1}{S}\right)^{(u-1)}\right]$$

$$\approx \frac{c}{\tau}\left(u - ue^{\frac{u}{S}}\right), \qquad (2.10)$$

where $bin(u, 1/S, d) = \binom{u}{d}\left(\frac{1}{S}\right)^{d}\left(1-\left(\frac{1}{S}\right)\right)^{u-d}$. The binomial distribution above indicates the probability that d blocks from the working set of size u map into one of the S cache sets. In general, the size of the carry-over set of a thread in a multithreaded cache is smaller than the size in a multiprogrammed cache (i.e., $v'(p) \leq v(p)$). The number of colliding blocks in a multithreaded caches also increases. The colliding blocks are the blocks that map into the same cache set. Assuming random placement of blocks in the cache, the estimated number of non-colliding blocks in multithreaded cache is given by

$$u\left(1 - \frac{1}{S}\right)^{u-1} \frac{v'(p)}{v} \approx ue^{-\frac{u}{S}} \frac{v'(p)}{v}. \qquad (2.11)$$

From (2.10) and (2.11), the intrinsic interference miss rate of the multithreaded cache becomes

$$m'_{intr} = \frac{c}{\tau}\left[u - ue^{-\frac{u}{S}} \frac{v'(p)}{v(p)}\right].$$

Thus, the net increase in the miss rate of multithreaded cache, $m'(p)$, is

$$m'(p) = m'_{cs}(p) + m'_{intr}(p) - m_{intr}(p)$$

$$= v'(p)\frac{(p-1)}{S}\cdot\frac{u}{\tau}+\frac{c}{\tau}ue^{-\frac{u}{S}}\left[1-\frac{v'(p)}{v(p)}\right].$$

Finally, the overall cache miss rate is computed by considering three components of fixed miss rate m_{fixed}, the single thread interference miss rate m_{intr}, and the multithreading components m'_{intr}, yielding

$$m(p) = m_{fixed} + m_{intr}(p) + m'(p)$$

$$= m_{fixed} + m_{intr}(p)\left(1+\frac{(p-1)(1+1/c)}{1+(p-1)\frac{u}{S}}\right),$$

where $m_{intr}(p) \approx \frac{c}{\tau}\cdot\frac{u^2}{S}$.

Increasing the degree of multithreading will effect both the intrinsic-interference and the multiprogramming component of the cache misses. When more threads occupy the cache, we can assume that each thread is allocated a smaller working set, and this in turn leads to higher intrinsic conflicts. Likewise, as the number of threads increases, the multiprogramming-related component also increases since there is a higher probability that cache blocks of active threads displace those of inactive threads. It is interesting to note that with sufficiently large cache memories, the multiprogramming related component of the cache miss rate is not affected by the number of threads. This is because the cache memory is large enough to hold the working sets of all resident threads. Set associativity is another issue that significantly affects the performance of cache memories for multithreaded systems. The higher associativity of cache can compensate for the increased intrinsic interference in a multithreaded system.

2.2 Programming Models

A thread can be viewed as a unit of execution that can be active within a process sharing certain resources such as files, address space with other threads in the process space. The notion of threads or lightweight processes permits the programming of applications using *virtual processes* such that a process can continue execution even when one or more of its threads are blocked. This concurrency can be supported in many different ways. For example, thread library, such as C-threads [23] and Pthreads [15], provide the programmers with API (Application Programming Interface) for creating, invoking, and scheduling threads.

The library implementation of threads usually supports the coarse-grain block multithreading. The block multithreading often requires synchronization among threads using semaphores, mutexes, and conditional variables. Also, thread scheduling mechanism is usually implemented using join, suspend, detach, and terminate calls. On the other hand, functional programming languages such as Multilisp [36] and Id [65] have proposed a different approach on multithreading, often supporting fine-grain threads. In such languages, traditionally blocking or synchronous function calls are made non-blocking or asynchronous. For example, when a function is invoked in conventional languages, the control transfers to the called function (blocking the execution of the caller) and the control is returned to the caller upon its completion. However, Multilisp function calls, e.g., *futures,* are non-blocking so that several futures can be invoked without waiting for their completion. In the following subsections, two programming models that support multithreading, namely *Cilk* and *Cid*, are introduced. The description of Pthreads is deferred until Chapter 4.

2.2.1 Cilk

Cilk language is an extension of C which provides an abstraction of threads in explicit continuation passing style [12]. The Cilk runtime supports *work stealing* for scheduling threads and achieves load balancing across a distribute processing environment. A Cilk program consists of a collection of procedures, each in turn consists of threads. These threads of a Cilk program can be viewed as the nodes of a directed acyclic graph as shown in Figure 2.2. Each horizontal edge represents a creation of a successor thread, and a downward edge represents the creation of child threads while data dependencies are represented by the dashed lines.

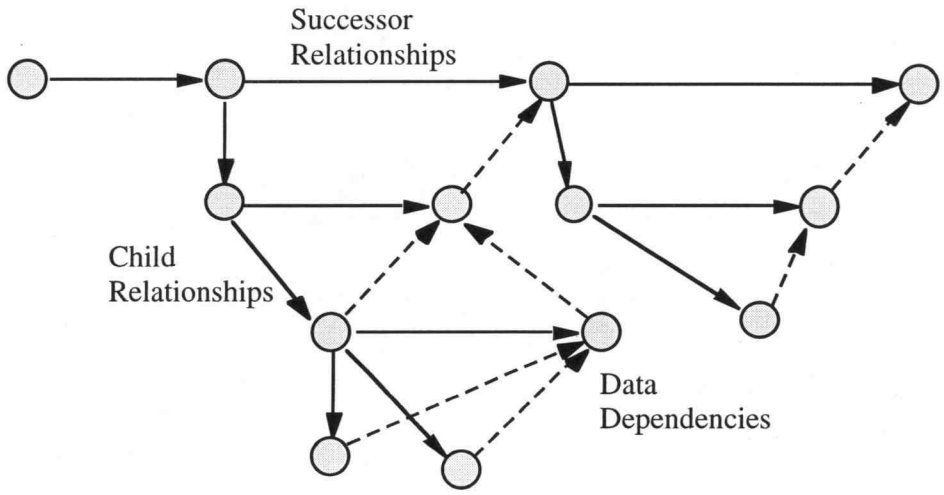


Figure 2.2: An example of Cilk program.

Like TAM threads, Cilk threads are non-blocking. This requires the creation of successor threads that expect the results from the child threads. The successor thread is blocked until the necessary synchronization events (or release conditions) are satisfied.

Cilk threads can spawn child threads to execute a new procedure. The child threads normally return values or synchronize with the successor threads created by their parent thread.

The runtime system keeps track of the active threads and threads awaiting for initiation. The data structure used for thread management is called a *closure*. A closure consists of a pointer to the code of the thread, a slot for each of the input parameters, and a join counter indicating the number of missing values (or synchronization events). The closure (hence the thread) becomes ready to executed when the join counter becomes zero; otherwise the closure is known as waiting. The missing values are provided by other threads using *continuation passing*, which identifies the thread closure and the argument position within the thread.

The following shows a Cilk program segment for computing the n^{th} Fibonacci number.

```
thread fib (cont int k, int n)
{
        if (n<2)
                send_argument (k, n)
        else{
                cont int x, y;
                spawn_next sum (k, ?x, ?y);
                spawn fib (x, n-1);
                spawn fib (y, n-2);
        }
}

thread sum (cont int k, int x, int y)
{
        send_argument (k, x+y);
}
```

The program consists of two threads, fib and its successor sum (which waits for the recursive fib calls to complete and provide the necessary values to sum). The fib threads test the input argument n, and if it is greater than 2, it spawns the successor thread sum

by passing the continuation *k*. Since sum requires two inputs x and y before becoming enabled, it spawns two child threads with *n-1* and *n-2* as their arguments as well as the slots where they should send their results (specified by the cont parameter). The statement send_argument sends the results to the appropriate continuation. The closure structure for the above Fibonacci program is shown in Figure 2.3.

Cilk run-time system uses an innovative approach to load distribution known as *work stealing* [11]. In this scheme, an idle worker (processor) randomly selects a heavily loaded processor and steals a portion of its work, thus achieving load balancing. However, only the ready threads are stolen in order to avoid the complications of relocating the continuation slots of the stolen threads.

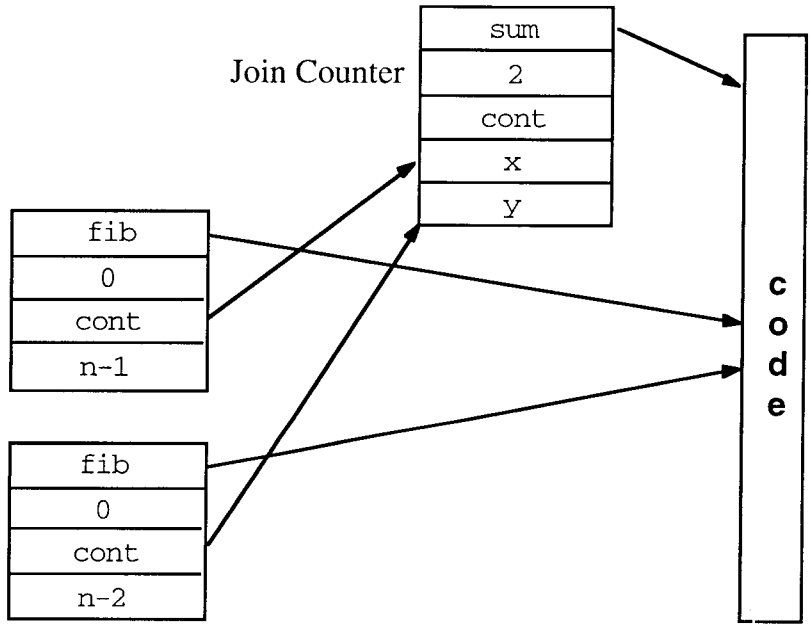


Figure 2.3: An example of Fibonacci in Cilk.

2.2.2 Cid

Cid is a parallel extension of C that provides a MIMD threads plus shared objects programming model even though it runs on distributed memory machines, such as workstation farms (Note that there is no explicit message-passing in Cid) [64]. Cid provides all threads with a shared memory space of objects. A Cid thread is an asynchronous C function call. Any C variable or data structure can be registered as a Cid global object, and each global object is associated with a pointer which is unique to across all processors. Given its global pointer, any thread on any PE can access any object, and the Cid run-time system manages all the necessary coherency of the objects. Also, Cid run-time scheduler can automatically achieve load-balancing based on a work-stealing mechanism. However, a thread does not further migrate between PEs once initiated.

Unlike TAM and Cilk, Cid threads can block, waiting for synchronization. Each Cid thread can be viewed as a C function with appropriate mechanism to specify synchronization. The simplest type synchronization is based on Join and the associated variables. Consider the following Cid implementation of the Fibonacci function.

```
int fib(int n)
{
        int fibN1, fibN2;
        cid_initialized_jvar(joinvariable);
        if (N<2) return n
        else
        {
                cid_fork(joinvariable;)
                fibN1 = fib(n-1); fibN2=fib(n-2);
                cid_jwait(&joinvariable);
                return fibN1+fibN2;
        }
}
```

When the value of N is greater than 2, two new threads are forked using cid_fork to compute fib(n-1) and fib(n-2). The cid_fork also indicates that these computations synchronize using join on the joinvariable specified. The parent thread will wait for the completion of the child threads and then returns the sum of fib(n-1) and fib(n-2) and signals appropriate joinvariable. Note that the Cid system is responsible for initializing the joinvariable as indicated by cid_initialized_jvar.

2.3 Examples of Multithreaded Systems

This section provides an overview of various multithreaded architectures and discusses some of the software and hardware features that represent the past and the current research efforts in the multithreading community. The architectures included in the discussion are TAM, Tera, MIT's Alewife, M-Machine, Electrotechnical Lab's EM-X, DEC/MIT's StarT-Next Generation, Stanford's FLASH, and Simultaneous Multithreading.

2.3.1 Threaded Abstract Machine

David E. Culler and his colleagues at U. C. Berkeley proposed the Threaded Abstract Machine (TAM) as an efficient execution model which maps the dataflow execution model onto a self-scheduled control flow execution model [27]. The novel aspect of TAM is how it allows the compiler to integrate the interactions among the scheduling of parallel threads, the asynchronous message events, and the utilization of the storage hierarchy. TAM exposes the scheduling of threads so that the compiler can optimize the storage resources (e.g., registers, local memory, etc.) by scheduling the related instructions together within a thread or even across threads. This improves the register utiliza-

tion and the cache behavior by enhancing the locality of references, which is not present in the traditional dataflow architecture.

Thread partitioning in TAM involves several steps: Id90 to dataflow graphs, program graphs to TAM threads, and finally TAM to native machine code. The threaded machine language, TL0, was designed to permit programming using the TAM model. TAM recognizes three major storage resources—code-blocks, frames, and structures—and the existence of critical processor resources, such as registers. A program is represented by a collection of re-entrant code-blocks, corresponding roughly to individual functions or loop bodies in the high-level program text. A code-block comprises a collection of threads and inlets. Invoking a code-block involves allocating a frame—much like a conventional call frame—depositing argument values into locations within the frame, and enabling threads within the code-block for execution. The compiler statically determines the frame size for each code-block and is also responsible for correctly using slots and registers under all possible dynamic thread orderings. The compiler also reserves a portion of the frame as a *continuation vector*, used at run-time, to hold pointers to enabled threads. The global scheduling pool is a set of frames that contain enabled threads.

Executing code-block may invoke several code-blocks concurrently because the caller is not suspended as opposed to conventional languages. Therefore, the set of frames in existence at any time forms a tree (the activation tree) rather than a stack, reflecting the dynamic call structure shown in Figure 2.4. An activation is enabled if its frame contains any enabled threads, and multiple subset of enabled activations may be resident in the processor at any time.

Threads can be categorized as synchronizing or non-synchronizing. A synchronizing thread specifies a frame slot containing the entry count for the thread. The compiler is responsible for initialization of the correct entry counts for synchronizing threads. Each *fork* to a synchronizing thread causes the entry count to be decreased by

one, and the thread may execute only when the count reaches zero. Synchronization occurs only at the start of a thread, and the thread executes to completion when the synchronization requirement is met. A thread ends with stop instruction, causing another thread to be scheduled.

Conditional flow of execution is supported by a switch instruction, which forks one of two threads based on a boolean input value. Also, fork operations may occur anywhere within a thread and cause additional threads to be enabled for execution. Long latency operations, such as I-Fetch or Send, implicitly fork a thread that resumes when the request completes. This allows the processor to continue with useful work while the remote access is outstanding.

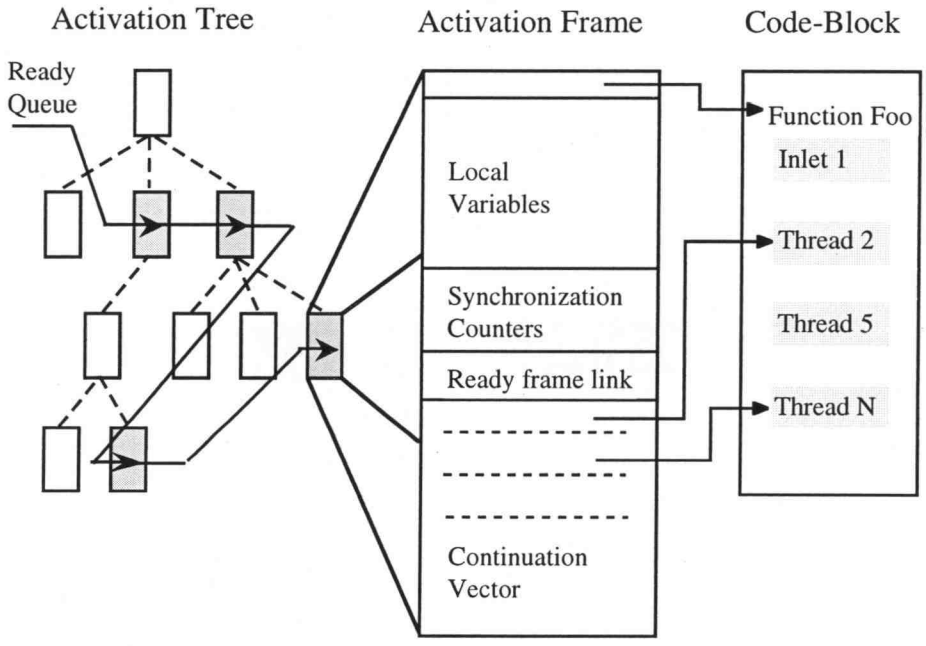


Figure 2.4: TAM activation tree.

2.3.2 Tera MTA

Tera MTA (MultiThreaded Architecture) computer is a multistream MIMD system developed by Tera Computer Company in the late 1980's [8], and it is the only commercially available multithreaded architecture. The designers of the system tried to achieve the following three goals: (1) high-speed, highly-scalable architecture, (2) be applicable to a wide variety of problems, including numeric and non-numeric languages, and (3) ease the implementation of compiler.

The interconnection network of Tera system is composed of pipelined packet switching nodes in a three-dimensional mesh with a wrap-around. On every clock cycle, each link can simultaneously transmit a packet containing source/destination addresses, an operation, and 64-bit data in both directions. For example, a 256 processor system consists of 4096 switching nodes arranged in 16×16×16 toroidal mesh, among which 1280 nodes are attached to 256 processors, 512 data memory units, 256 I/O cache units, and 256 I/O processors as shown in Figure 2.5. In general, the number of network nodes grows as a function of $p^{\frac{3}{2}}$, where p is the number of processors in the system.

In Tera, each processor can simultaneously execute multiple instruction streams from one to as many as 128 active program counters. On every clock cycle, the processor logic selects an instruction stream that is ready to execute, and a new instruction from a different stream may be issued in each cycle without interfering with the previous instruction. Each instruction stream maintains the following three states: one 64-bit Stream Status Word (SSW), 32 64-bit General Registers (R0-R31), and eight 64-bit Target Registers (T0-T7). Each processor also has a large number of registers (128 SSWs, 4096 General Registers, and 1024 Target Registers) in order to support context

switching on every cycle. Program addresses are 32 bits long, and the program counter is located in the lower half of the its SSW. The upper half is used to specify the various modes (e.g., floating-point rounding), trap mask, and four recently generated condition status. Target Registers are used for branch targets, and the computation of a branch address and the prediction of a branch are separated, allowing the prefetching of target instructions. A Tera instruction typically consists of three operations: a memory reference operation, an arithmetic operation, and a control operation. The control operation can also be of another arithmetic operation, i.e., if the third operation specifies a floating-point operation, it will perform two floating-point operations per cycle.

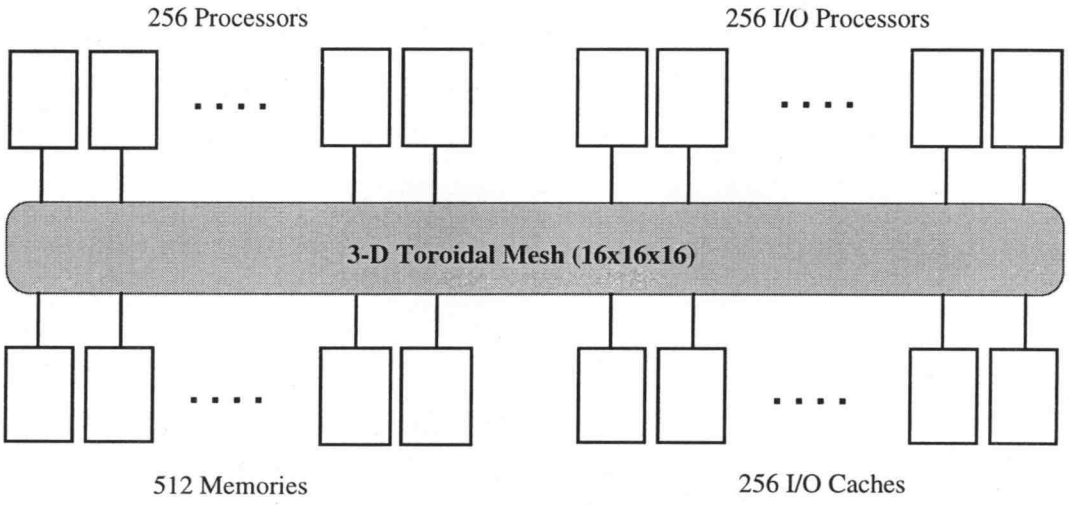


Figure 2.5: The organization of Tera MTA.

Each processor needs to execute on the average about 70 instructions to achieve peak performance by hiding remote latencies (i.e., the average latency for remote access is about 70 cycles). Therefore, if each instruction stream can execute some of its instructions in parallel, less than 70 streams are required to achieve the peak performance. To reduce the required numbers of streams, Tera introduced a new technique called *ex-*

plicit-dependence lookahead to utilize instruction-level parallelism. The idea is that each instruction contains a three-bit lookahead field that explicitly specifies how many instructions from this stream will be issued before encountering an instruction that depends on the current instruction. Since seven is the maximum possible lookahead value with three bits, at most eight instructions can be executed concurrently from each stream. Therefore, only nine streams are needed to hide 72 clock cycles of latency in the best case, compared to 70 different streams required for the worst case.

A full-size Tera system contains 512 128-Mbyte data memory units. Memory is 64-bit wide and byte-addressable. Associated with each word are four additional *access state bits* consisting of two data trap bits, a forward bit, and a full/empty bit. The trap bit allows application-specific use of data breakpoints, demand-driven evaluation, run-time exception handling, implementation of active memory objects, stack limit checking, etc. The forward bit implements invisible indirect addressing, where the value found in the location is to be interpreted as a pointer to the target of the memory reference rather than the target itself. The full/empty bit is used for lightweight synchronization.

Load and store operations use the full/empty bit to define three different synchronization modes along with the access control bits defined in the memory word. The values for access control for each operation is shown in Table 2.1. For example, if the value of the access control field is 2, the store operation waits for the memory location to be written before writing to the location, and sets the full/empty bit to full. When a memory access fails, it is placed in a retry queue and memory unit retries the operation several times before the stream that issued the memory operation results in a trap. Retry requests are interleaved with new memory requests to avoid the saturation of the communication links.

Table 2.1: The access control values.

Value	LOAD	STORE
0	read regardless of the state of the full/empty bit	write regardless and set the full/empty bit to full
1	not used	not used
2	wait for full state and read	wait for full state and write
3	read only when full and set the bit empty	write only when empty and set the bit full

2.3.3 StarT-NG

StarT-NG (Next Generation) is a joint project between MIT and Motorola, which attempts to develop a general-purpose parallel system using commodity components [20], such as PowerPC 620—a 64-bit 4-way superscalar processor with a dedicated 128-bit wide L2 cache interface/128-bit wide L3 path to memory. StarT-NG is a symmetric multiprocessors (SMP) system that examines how the multithreaded codes can run on a stock processor and emphasizes the importance of cache-coherent global shared-memory supported by efficient message-passing. Influenced by the predecessor StarT [66], multithreading in StarT-NG relies heavily on software support. The instruction fork creates a thread by pushing a continuation specified in registers onto a continuation stack. The compiler is required to generate switch (jump) instructions in the instruction stream for thread switching. Also, the compiler needs to generate the necessary save/restore instructions to swap the relevant register values from the continuation stack, resulting in a large context switching cost.

StarT-NG has 4-processor card slots, where one to four slots are filled with Network-Endpoint-Subsystem (NES) cards. Each NES contains a single PowerPC 620

processor with 4 MBytes of L2 cache and a Network Interface Unit (NIU) as depicted in Figure 2.6. Each site has an Address Capture Device (ACD) on the NES board, which manages bus transactions. When an access to global shared-memory is necessary, a processor is used as a Service Processor (SP) for servicing the bus transactions. Otherwise, a processor is used as an Application Processor (AP) for running applications.

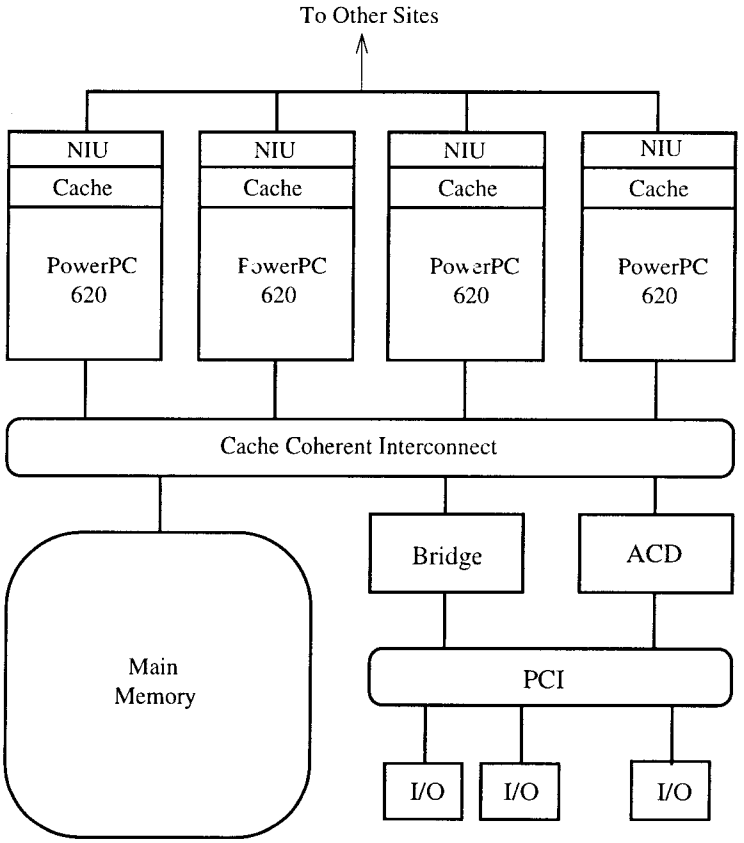


Figure 2.6: A site structure of *T-NG.

StarT-NG is built on a *fat-tree* network using MIT's Arctic routers connected to NIU [13]. The NIU's packet buffers are memory-mapped into an application's address space enabling users to send and receive messages without kernel intervention. The arrival of a message is signaled either by polling or interrupt. Generally, PowerPC 620

will poll the NIU by reading a specified location of the packet buffer with minimum overhead. On the other hand, when the frequency of the message arrival is estimated to be low, an interrupt mechanism can be used either for a kernel message or a user message in order to minimize the overhead of polling.

Cache coherent distributed shared-memory in a prototype StarT-NG is implemented in software to experiment with the directory-based protocol by programming the Shared Memory Unit (SMU) consisting of ACD and SP. When a cache miss occurs, the local SMU determines whether the operation is local or global by examining the appropriate bits of the address—a higher order bit of the physical address distinguishes the global and local address space. If it is local, the directory information is updated and the cache line is read from the local memory. If it is global requiring remote access, cache-coherence action message is sent out to either invalidate remote caches or flush a dirty cache-line.

2.3.4 EM-X

The EM-X parallel computer, which is a successor to EM-4 architecture [78], is being built at Electrotechnical Laboratory in Japan [77]. Based on dataflow model, EM-X integrates the communication pipeline into the execution pipeline by using small and simple packets. Sending and receiving of packets do not interfere with the thread execution. Threads are invoked by the arrival of the packets from the network or by matching two packets. When a thread suspends, a packet on the input queue initiates the next thread. EM-X also supports direct matching for synchronization of threads, and the matching is performed prior to the buffering of the matching packets. Therefore, one clock cycle is needed for pre-matching of two packets, but the overhead is hidden by simultaneously executing other threads.

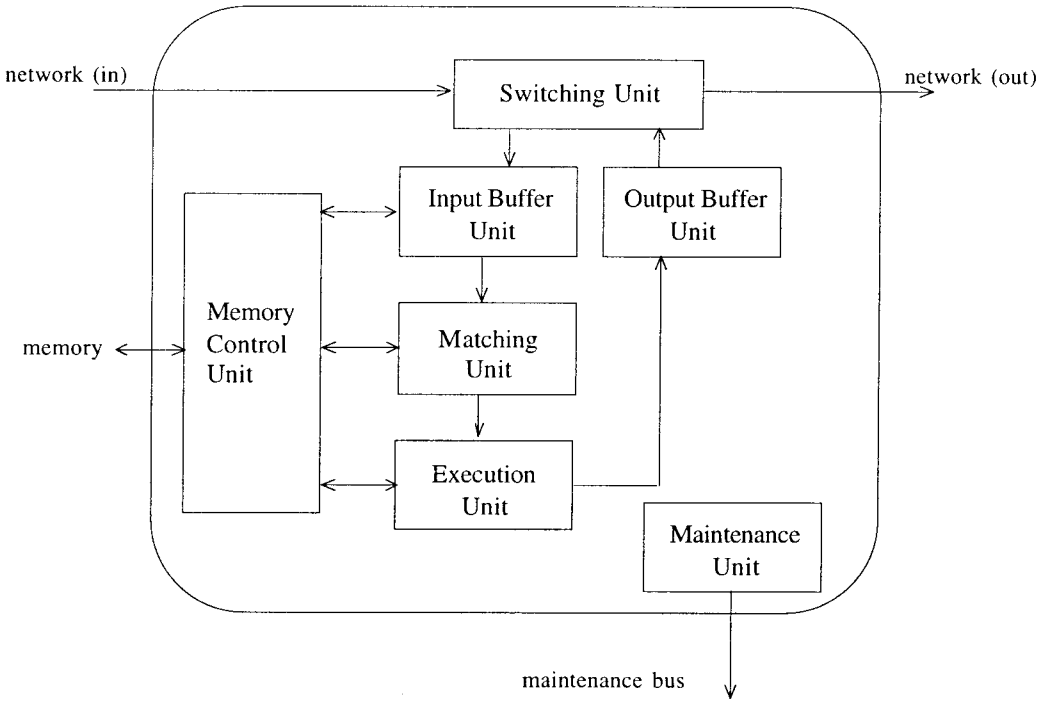


Figure 2.7: The structure of EMC-Y.

The EM-X consists of EMC-Y nodes connected in a circular Omega Network with virtual cut-through routing scheme. The EMC-Y processor is capable of 20 MIPS and 40 MFLOPS (e.g., the instruction fma, FP multiply and add, performs two single-precision floating-operations simultaneously) with clock rate of 20 MHz, and Figure 2.7 shows its internal the structure. The Switching Unit is a 3-by-3 crossbar connecting input and output of network and the processor. Packets arriving at the processor are received in the Input Buffer Unit (IBU). The IBU has an on-chip packet buffer which holds up to 8 packets. When the on-chip buffer overflows, packets are stored in the data memory and automatically restored back when the buffer space becomes available.

EM-X implements a flexible packet scheduling by maintaining two separate priority buffers. The packets in the high priority buffer are first transferred to the Matching Unit (MU), and low priority packets are transferred only when the high priority buffer is empty. The MU prepares the invocation of a thread by using the direct matching scheme

[55]. This is done by first extracting the base address of the operand segment from the incoming packet. The operand segment is an activation frame shared among threads in a function and holds the matching memory and local variables. Next, the partner data is loaded from the matching memory specified in the packet address, and the corresponding presence flag is cleared. Then, a template (i.e., a code frame) is fetched from the top of the operand segment, and the first instruction of the enabled thread resident in the template is executed on the execution unit (EXU).

The EXU, a RISC-based thread execution unit with 32 registers, provides four SEND instructions for invoking a thread, remote memory access, returning the result from the thread execution, and implementation of variable size operand segments or a block access of remote memory [77]. EM-X performs a remote memory access by invoking packet handlers at the destination processor, and the packets are entirely serviced by hardware which does not disrupt the thread execution in the execution pipeline. The round trip distances of the Omega Network in EM-X are 0, 5, 10, and 15 hops for request/reply sequences with the average of 10.13 hops requiring less than 1μsec on a unloaded network. On a loaded network, the latency is 2.5μsec on the average with random communication of 100 Mpackets/sec.

2.3.5 Alewife

MIT Alewife machine improves scalability and programmability of modern parallel systems by providing software-extended coherent cache, global memory space, integrated message-passing, and support for fine-grain computation. Underneath the Alewife's abstraction of globally shared memory, each PE has a physically distributed memory managed by a Communication and Memory Management Unit (CMMU). This CMMU maintains the locality by caching both private and shared data on each node, and a scalable software-extended scheme, called LimitLESS, manages the cache coherence

[5]. The LimitLESS scheme implements a full-map directory protocol that supports up to five read requests per memory line directly in hardware and more by trapping into software.

Each Alewife node, shown in Figure 2.8, consists of a Sparcle processor, 64 Kbytes of direct-mapped cache, 4 Mbytes of data and 2 Mbytes of directory, 2 Mbytes of private unshared memory, a floating-point unit, and mesh routing chip. The nodes communicate via two-dimensional mesh network using wormhole routing technique. Sparcle is a modified SPARC processor that facilitates block multithreading, fine-grain synchronization, and rapid messaging. In Sparcle, the register windows of SPARC are modified to represent four independent contexts: one for trap handlers and other three for user threads. A context-switch is initiated when the CMMU detects a remote memory access and causes a synchronous memory fault to Sparcle. The context switching is implemented by a short trap handler that saves the old program counter and status register and switches to a new thread by restoring a new program counter and status registers. Currently, the context switching takes 14 clock cycles, but it is expected to be reduced to four clock cycles.

Sparcle provides new instructions that manipulate the full/empty bits in memory for data-level synchronization [2]. For example, ldt (read location if full, else trap) and stt (write location if empty, else trap) instructions can be used to synchronize on an element-by-element basis. Fast message handling is also implemented via special instructions and memory-mapped interface to the interconnection network. To send messages, Sparcle first writes a message to the interconnection network queue using stio instruction, and then ipillaunch instruction is used to launch the message into the network. A message usually contains the message opcode, the destination node address, and data values (e.g., content of a register or address/length pair which invokes DMA on blocks from memory). The arrival of a message invokes a trap handler that loads the incoming

message into registers using ldio instruction, or initiates a DMA sequence to store the message into memory.

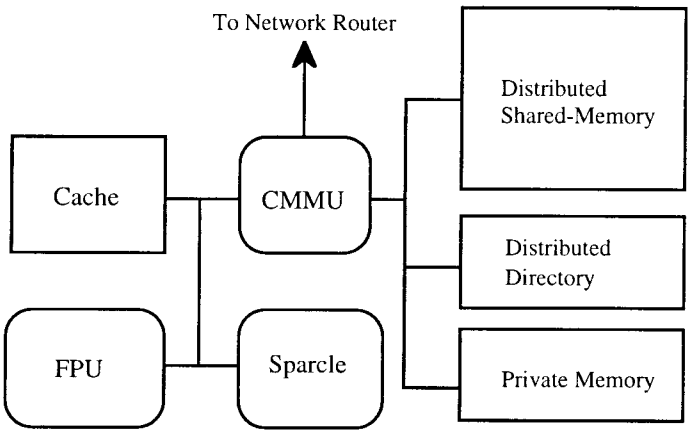


Figure 2.8: The organization of an Alewife node.

2.3.6 The M-Machine

The M-machine is an experimental multicomputer being developed by MIT. The M-Machine achieves better utilization by devoting more chip area to the processor, and it is claimed that a 32-node M-Machine system with 256 MBytes of memory has 128 times the peak performance of uniprocessor with the same memory capacity at 1.5 time the area, yielding 85 times improvement in peak performance/area [30]. The M-Machine consists of a collection of computing nodes interconnected by a bidirectional 3-D mesh network. Each node consists of a multi-ALU (MAP) and 8 MBytes of synchronous DRAM. A MAP contains four execution clusters, four cache banks, a network interface, and a router. Each of the four MAP cluster is a 64-bit, three-way issue, pipelined processor consisting of a memory unit, an integer unit, and a floating-point unit as

shown in Figure 2.9. The memory unit is used for interfacing memory system and cluster switch. The cache is organized as four word-interleaved 32-KByte banks to permit four consecutive accesses. Each word has a synchronization bit which is manipulated by special load and store operations for atomic read-modify-write operations.

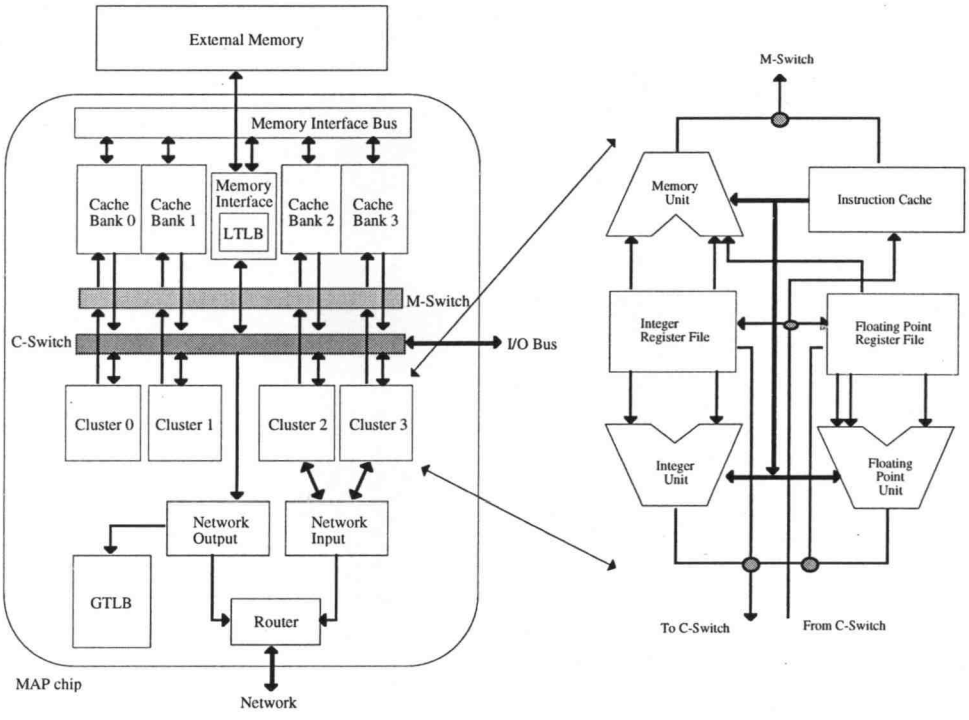


Figure 2.9: The MAP architecture and its four clusters. A cluster consists of 3 execution units, 2 register files, an instruction cache, and interface to the memory and cluster switches.

The M-Machine supports a single global virtual address space through a global translation lookaside buffer (GTLB). GTLB is used to translate the virtual address into physical node identifiers in the message. Also, messages are composed in the general registers of a cluster and launched atomically using a user-level SEND instruction. Arriving messages are queued in a register-mapped FIFO, and a system-level message handler performs the requested operations specified in the message.

Each MAP instruction contains one to three operations and may complete out-of-order. The M-Machine exploits instruction-level parallelism by running up to 12 parallel instruction sequences (called H-Thread) concurrently. Also, the MAP interleaves the 12-wide instruction streams (called V-Thread) from different threads of computation to exploit thread-level parallelism and to mask various latencies that occurs in the pipeline (i.e., during memory accesses and during communication). Six V-Threads are resident in a cluster, and each V-Thread consists of four H-Threads. A V-Thread consists of a sequence of 3-wide instructions containing integer, memory, and floating-point operation. Within an H-Thread, instructions are issued in order, but may complete out of order. Synchronization and communication among H-Threads in the same V-Thread is done using a scoreboard bit associated with each register. However, H-Threads in different V-Threads may only communicate and synchronize through massages and memory. The M-Machine provides a fast user-level message passing substrate through hardware support. Also, register-to-register communication is provided to reduce the memory accesses.

2.3.7 Simultaneous Multithreading

Simultaneous multithreading (SMT) is a technique that allows multiple independent threads from different programs to issue multiple instructions to a superscalar processor's functional units. Therefore, SMT combines the multiple instruction-issue features of modern superscalar processors with the latency-hiding ability of multithreaded architectures, alleviating the problems of long latencies and limited per-thread parallelism. This means that the SMT model can be realized without extensive changes to a conventional superscalar processor architecture.

Figure 2.10 shows the processor organization of an 8-thread simultaneous multithreading machine proposed by Tullsen D. M. *et al.* at University of Washington [57].

The processor execution stage is composed of three Floating-Point Units and six Integer Units. Therefore, the peak instruction bandwidth is nine. However, throughput of the machine is bounded to eight instructions per cycle due to the bandwidth of Fetch and Decode Units. Each Integer and Floating-Point Instruction Queue (IQ) holds 32 entries, and the caches are multi-ported and interleaved. Also, there are 256 physical registers (i.e., assumed 32-register instruction set architecture per each of 8 threads) and 100 additional registers for renaming.

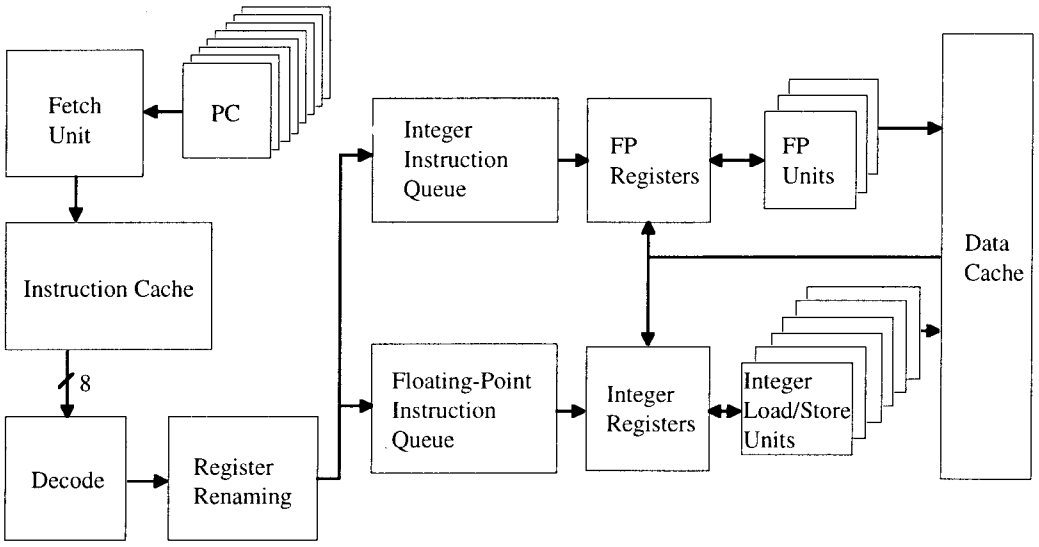


Figure 2.10: An overview of SMT hardware architecture.

When running on a single thread, the throughput of the basic SMT system is 2% less than a superscalar with similar hardware resources due to the need to accommodate longer pipeline for a large register file; on the other hand, its estimated peak throughput is 84% higher than that of a superscalar processor when running on multiple threads. However, it is shown that the actual throughput peaks at IPC of 4 even with eight threads. This saturation is caused by the three factors: (1) small IQ size. (2) limited

fetch throughput (only 4.2 useful instruction fetch per cycle). (3) lack of instruction-level parallelism.

The fetch throughput can be improved by using techniques such as partitioning of fetch unit among threads, selective fetching, or eliminating conditions that block the fetch. It has been shown that the best performance is obtained when the Fetch Unit is partitioned in such a way that eight instructions are fetched from two threads, and the priority is given to the threads with the fewest instructions in the decode stage. Also, fetch misses can be reduced by examining the I-cache tag lookups a cycle early and selecting only threads that do not cause misses. However, this scheme requires extra ports in the I-cache tags and increases misfetch penalties due to one more pipeline stage needed for early lookup. The resulting performance shows that a 2.5 throughput gain over a conventional superscalar architecture when running at 8 threads, yielding an IPC of 5.4. The conclusions drawn from the experiments are as follows:

- Techniques such as dynamic scheduling and speculative execution in a superscalar processor are not sufficient to take full advantage of a wide-issue processor without simultaneous multithreading.
- Instruction scheduling in SMT is no more complex than that of a dynamically scheduled superscalar processor.
- Register file data paths in SMT are no more complex than those in a superscalar, and the performance implication on the register file and its longer pipeline is minimal.
- The required instruction fetch throughput is attainable without increasing the fetch bandwidth by partitioning Fetch Unit and employing selective fetching.

2.4 New Generation of Multithreading

The current multithreaded systems can execute several threads concurrently by providing multiple hardware contexts and hardware scheduler mechanisms to support fast context switching and thread management within the processor. The concurrent execution of multiple threads requires the programs to be expressed as multiple interacting threads. In order to obtain such threads, traditionally two approaches have been taken: (1) The programmers write programs using thread packages, where threads are explicitly expressed in APIs provided by the thread packages. (2) Multiple programs are used, where each program represents a thread. However, some of the programs do not lend themselves to parallel programming, making it difficult to express the program using threads.

On the other hand, new emerging multithreaded systems use various software and hardware speculation techniques to obtain multiple threads from a sequential program, eliminating the need to program using threads. These next generation multithreaded systems use compiler techniques and hardware mechanisms to identify threads in a speculative fashion and provide the support for resolving inter-thread register dependencies and memory disambiguation. This section introduces such multithreaded architectures, namely *Multiscalar*, *I-ACOMA* and *DeSM*.

2.4.1 Multiscalar

Traditional processors execute sequential programs following a single flow of control and build a large window of instructions in order to achieve high performance. On the other hand, the Multiscalar executes a single sequential program following multiple flows of control by relying on hardware support for maintaining sequential seman-

tics between those flows of control [41]. This allows the programmer to use a sequential programming style while efficiently executing the program in parallel. The Multiscalar introduces new instructions to support two levels of control speculation: *inter-task* speculation and *intra-task* speculation. First, a program is converted into a task flow graph (TFG). A TFG is directed graph with nodes representing tasks and arcs representing control dependencies, in which each task is a conventional control flow graph (CFG). The Multisclar relies on the compiler for generation of tasks which encapsulate groups of instructions containing arbitrary control flows. The compiler identifies the tasks and inserts task start and task end instructions at the task boundaries. The task start instruction loads the special state register with a task header containing a bit mask that indicates which registers may be updated within the task along with the information about the tasks that may follow. The task end instruction is used to indicate an exit point of the task by setting special bits and transferring the control. These special bits are then used in the speculation of the next task.

The global sequencer in Multiscalar, shown in Figure 2.11, traverses the program's TFG and distributes the tasks to the processing units (PUs) in speculation of the program paths (inter-task speculation). The responsibility of global sequencer is to predict the starting address of the next tasks to be executed using information from the task header of the most recently predicted tasks and the dynamic prediction hardware. At any given time, only one unit can execute the non-speculative task and other units execute speculative tasks. For each task, the processing unit uses traditional control speculation to extract instruction level parallelism (intra-task speculation). The PU ring in Multiscalar operates as a circular queue with a head pointing to the non-speculative task and a tail pointing to the most recently started speculative task. When the task pointed by the head finishes, it informs the global sequencer of its actual non-speculative target address. If the predicted target address followed by the speculative tasks turns out to be incorrect, all tasks following the head task are squashed and execution is redirected to the correct

task. The task misprediction penalty can be large in terms of wasted work. However, this task speculation mechanism, supported by both hardware and software, enables the Multiscalar to effectively build a very large instruction window and increases the performance.

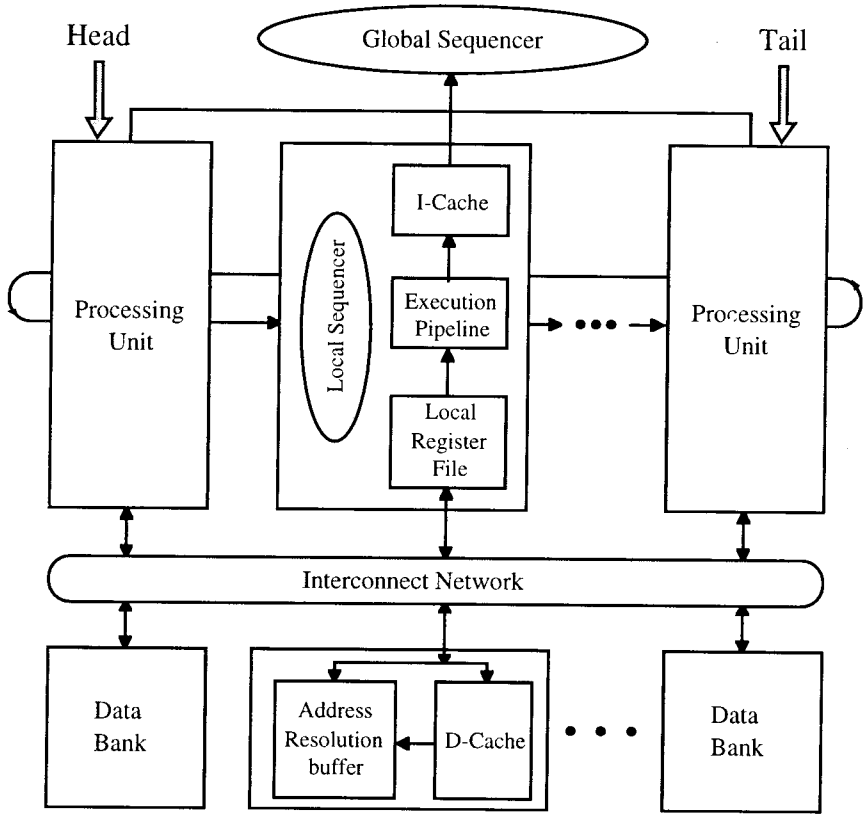


Figure 2.11: The Multiscalar hardware.

2.4.2 I-ACOMA

Researchers at University of Illinois have proposed a software/hardware solution that allows speculative execution of threads using a sequential binary on a clustered SMT architecture, namely Illinois Aggressive Coma (I-ACOMA) [48]. They utilize the

loop iterations to identify threads and multiple threads from the successive iterations of the loop are speculatively spawned onto each SMT-based processor. In order to support this thread speculation, a binary annotator is developed to identify basic blocks in loops and generate threads. The annotation involves the following processes: (1) Identification/annotation of entry and termination points of the loops. (2) Checking register-level dependencies and identifying *looplive* registers. The looplive registers define the inter-thread dependencies that exist at the entry or exit points of the loop. Also, the registers that are modified within the loop are identified. (3) Finally, the registers holding loop-carried variables, called *induction variables*, are identified and the instructions associated with the induction variables are moved close to the entry point of the loop in order to minimize the waiting cycles before the spawning of next iteration.

Speculative execution of threads requires special hardware support for inter-thread register synchronization and memory disambiguation. The inter-thread register synchronization is done by Synchronizing Scoreboard (SS). The SS associates three bits, namely *Busy* (B), *Valid* (V), and *Sync* (S), with each register to handle inter-thread dependencies. The B-bit is set when the register value is being created. The S-bit indicates whether or not the register value is available to the succeeding thread. When a thread is initiated, the S-bits for all the looplive registers are set and later cleared when the release instruction for that register is executed. At this point, the register becomes safe to use by the succeeding thread. The V-bit indicates whether the thread has valid copy of the register. When a given thread need to access a register, the B-bit is checked first. If the B-bit is set, indicating the unavailability of the register, the thread execution is stalled. Otherwise, the thread gets the register value from the predecessor thread if the V-bit set.

To preserve the sequential semantics of memory operation in speculative execution, the Memory Disambiguation Table (MDT) is used to maintain the address of the load/store operations along with *Load* (L) bit and *Store* (S) bit. In addition, each thread

has private write buffer to hold speculative store values. When a speculative thread performs a load, it first checks the MDT for the matching address. If no match is found, a new entry is created for that address with L-bit set. On the other hand, if the matching address is present, the S-bit of that entry is checked to determine the possibility of update by any predecessor threads. If the S-bit is not set, the load operation proceeds in normal fashion. However, if the S-bit is set, the load operation reads the values from the write buffer of the predecessor thread. The store operations performed by the speculative threads are held in the write buffer until they acquire non-speculative status. This prevents the corruption of memory state in case of incorrect speculation.

2.4.3 Dependence Speculative Multithreaded Architecture

The Dependence Speculative Multithreaded Architecture (DeSM), proposed by Marcuello *et al.*, dynamically obtains multiple threads from a conventional sequential binary code without any user/compiler intervention [59]. It relies only on hardware mechanisms to speculate on multiple threads of control obtained from highly predictable branches. This architecture has several distinct advantages: (1) the dynamic behavior of program execution can be used to provide better speculation of the control-flow in the program. (2) Since no compiler support is needed, any existing program can benefit from the performance improvement of the multithreaded processor without any modification.

DeSM, shown in Figure 2.12, extends a SMT architecture with hardware support for speculation of control and data dependencies through register and memory. The goal of control speculation is to generate multiple threads from different iterations of the same loop at run-time. Note that a loop is identified by the target address of a backward branch and the branch itself. The hardware mechanisms, called Loop Table and Loop Stack, are used to keep the loop information, such as the iteration count of the last exe-

cution of the loop, the difference between the number of iterations of the last two execution, the current iteration count, and the nesting level of the loop. Each time a new loop is executed, an entry is created in the loop table for the loop. When a new iteration of the loop is detected by a backward taken branch, the loop table is looked up. If the loop is in the table, the number of iterations left is predicted using the loop information provided in the table and threads are allocated to available contexts.

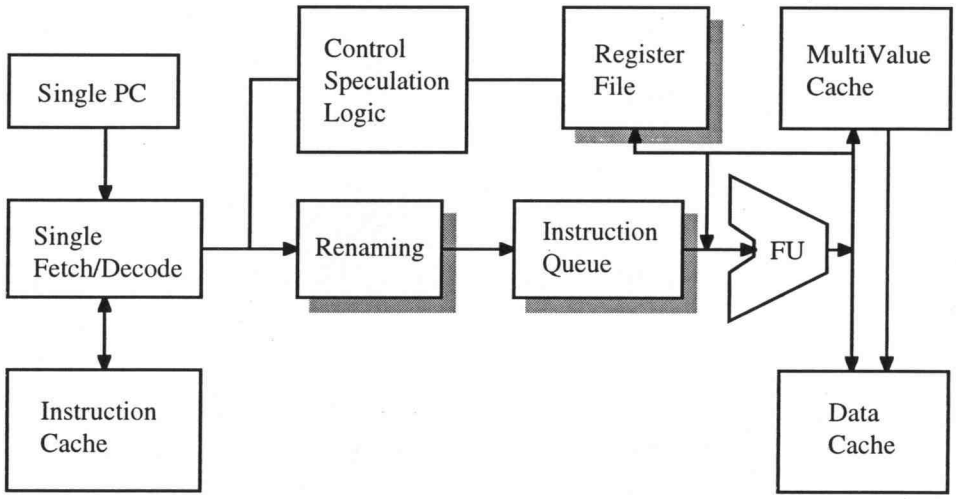


Figure 2.12: The DeSM architecture

The dependencies between threads are predicted based on the history of each loop. At run-time, the hardware-based *Iteration Table* maintains the information such as the number of last/current register writes and the address of last/current stores. The memory dependencies are predicted based on the last effective address of each store instruction and its stride. The register dependencies are resolved by Register Mapping Table (Rmap) and Register Write Table (Rwrite). Rmap table provides the mapping between logical register and physical registers among threads. Rwrite table is used for synchronization by keeping track of the number of writes on each register. Also, Multi-

Value (MV) cache provides temporary memory space until the thread becomes non-speculative. To implement the memory disambiguation mechanism, MV cache replicates every data words of each thread to every context.

3. Viability of Multithreading on Networks of Workstations

Over the past several years, *distributed computing* using networks of workstations (NOWs) has gained a wide acceptance for both scientific and general purpose applications. Distributed computing employs powerful workstations, or nodes, connected by high-speed local area networks (LANs). A software infrastructure system provides the capability to emulate virtual parallel machines with efficiency ranging from moderate to high. Moreover, a virtual global memory space, based on distributed shared memory (DSM) model, is provided to the programmer via software support.

Although DSM provides ease of programming by eliminating the need to explicitly specify the synchronization and communication among nodes, it inevitably leads to performance degradation due to long and unpredictable *memory latency.* Memory latency occurs when a miss in the local memory requires a request/reply to/from the remote node. Multithreading allows the processor to context-switch and execute a new thread of computation rather than waiting for the reply to arrive, thereby effectively tolerating memory latency by overlapping communication and computation. However, in order for multithreading to be effective, a number of interrelated issues must be carefully considered. Issues such as the number of contexts, thread run-length, thread scheduling and granularity of threads have to be considered to provide the best performance for a given architecture. Therefore, this chapter examines the viability and effectiveness of multithreading in a networked computing environment.

Organization of the chapter is as follows: Section 3.1 provides a brief discussion of multithreading. Section 3.2 discusses the basic communication operations involved in row-major distributed matrix multiplication algorithm and presents analytical models that characterize the performance of matrix multiplication using both message-passing

and multithreaded execution models. Section 3.3 provides simulation results of the two execution models. Finally, Section 3.4 provides a brief conclusion.

3.1 Multithreading

As discussed in the previous section, one of the major obstacles in achieving high efficiency in a large multicomputer system is memory latency due to remote load operations [2]. Remote memory latency is the time elapsed between when a remote memory read is requested and when the data is available. In a multithreaded execution, when a processor reaches a point where a remote memory access is necessary, the request is sent out on the network and a context-switch occurs to a new thread of computation. This effectively masks a long and unpredictable latency due to remote loads.

In the multithreaded execution model, each processor maintains a number of ready threads, and a context-switch occurs when a remote memory reference occurs as shown in Figure 3.1.

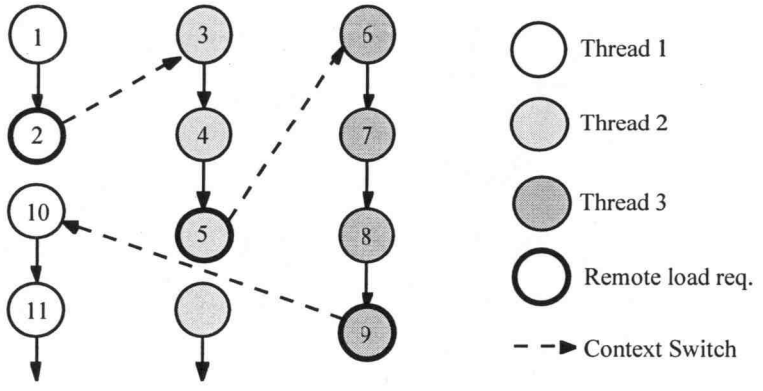


Figure 3.1: An example of multithreading. Numbers indicate the order of execution. It is assumed that the remote load operation at node 2 is completed before the context-switch occurs from node 9 to node 10.

A thread can be either *blocking* or *non-blocking*. A thread is said to be blocking if the thread execution is interrupted due to a remote load and continued later when remote data is available. On the other hand, once initiated a non-blocking thread executes to completion without interruption, and does not require the context to be saved before initiating a new thread. Blocking threads tend to have coarser granularity than non-blocking threads. However, the context switching cost for a blocking thread is more expensive than that of a non-blocking thread since the processor status and the contents of active register set (i.e., context) have to be saved for later execution of the thread.

3.2 Analytical Models for Matrix Multiplication

Matrix multiplication is a simple yet widely used algorithm in many scientific and engineering applications. Matrix multiplication algorithm is well structured in the sense that elements of the matrices can be evenly distributed to the nodes and communications among the nodes have a regular pattern. Therefore, exploiting data-parallelism based on message-passing is more suitable for solving the matrix multiplication problem.

```
Matrix_Multiplication(A,B,C)

for i=0 to n-1
        for j=0 to n-1
                C[i,j]=0;
                for k=0 to n-1
                        C[i,j]=C[i,j]+A[i,k]×B[k,j];
                end
        end
end
```

A sequential version of a $n \times n$ matrix multiplication algorithm consisting of three nested loops is shown above. As can be seen, the complexity of the algorithm is n^3. Thus, assuming each pair of multiplication/addition in the inner-loop takes the time

c, the total sequential execution time is given by $T_s = cn^3$. Our motivation for studying the matrix multiplication algorithm is to see how the multithreaded version of the matrix multiplication algorithm compares to the message-passing counterpart in a highly parallel computing environment. Therefore, the following two subsections derive the analytical models for both message-passing and multithreaded versions of the matrix multiplication algorithm.

3.2.1 Matrix Multiplication using Message-Passing

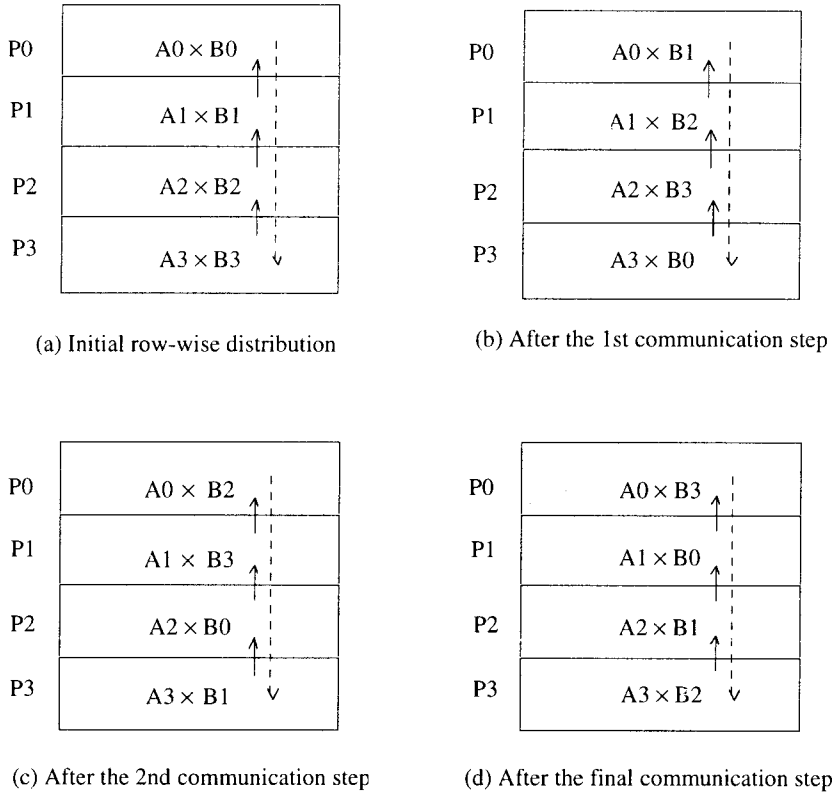
For message-passing, nodes need to communicate data among different parts of the program. For a large parallel system, this exchange of data introduces large communication delays during the execution of a program. Thus, proper implementation of communication operations is important to achieve an efficient execution based on message-passing. There are a few basic communication patterns that frequently appear in various parallel algorithms, e.g., one-to-one, one-to-all broadcast, all-to-all broadcast, and shift with wrap-around. In this section, one-to-one communication used in row-wise stripped matrix multiplication is discussed.

To simplify the development of a communication model, we focus only on two major components of the communication cost between two neighboring nodes: *startup cost*, t_s, and *transmission cost*, t_w. Startup cost consists of the time to setup a network channel between the source and destination nodes, the time to allocate a buffer space, and the time to package the messages. Transmission cost is the time required for a message of unit length to travel from one node to the other (i.e., t_w=1/bandwidth). Based on this, one-to-one communication between two processors takes $t_s + t_w m$, where m is the message length.

Data distribution for matrix multiplication can be divided into three classes: element-wise, row or column major, and block distribution. In this paper, we implement a simple row-major stripped distribution to study the performance of both the message-passing and the multithreaded execution models. We have experimented with other algorithms such as Cannon's and Fox's algorithms [49], but found that their communication requirements do not map well to a LAN environment and thus resulted in inferior performance.

Consider a row-major distribution of the matrices A and B partitioned into p $\frac{n}{p} \times n$ submatrices in Figure 3.2(a), where p is the number of processors. The processors are labeled from P_0 to P_{p-1} and the submatrices A_i and B_i are initially assigned to P_i for $0 \le i \le p-1$. To compute C_i, every processor requires all p submatrices B_k (for $0 \le k \le p-1$), which requires excessive memory if all the submatrices are duplicated in each processor. To avoid this problem, each row of submatrices B_k is systematically shifted so that every processor gets a new B_k from its neighbors on each communication step. The basic communication steps involved in the simple row-wise matrix multiplication is to rotate all B_k (for $0 \le k \le p-1$) by one step up with a wrap-around in each communication step as depicted on four processors shown in Figure 3.2(b-d).

The algorithm proceeds by having every processor perform multiplication/addition on its local submatrices before each communication step. Each processor also has to perform p-1 number of communication steps before completing the matrix multiplication, requiring a total of $(p-1)(t_s + t_w \frac{n^2}{p})$ time for communication. For the computation part, every processor performs multiplication on $\frac{n}{p} \times n$ submatrices, where multiplication/addition of each submatrix takes $\frac{n^3}{p^2}$ time. Therefore, the total computation time for each processor is $c\frac{n^3}{p}$. The overall parallel run-time for the message-passing version is then given by



(a) Initial row-wise distribution

(b) After the 1st communication step

(c) After the 2nd communication step

(d) After the final communication step

Figure 3.2: Data distribution among four processors. Each processor has $\frac{n}{p} \times n$ submatrices of A and B. Submatrices of B are shifted with a wrap-around in each communication step.

$$T_{m-p} = c\frac{n^3}{p} + (p-1)\left(t_s + t_w \frac{n^2}{p}\right). \tag{3.1}$$

3.2.2 Matrix Multiplication using Multithreading

For the multithreaded version, we assume the initial distribution of matrices A and B is the same as in Figure 3.2(a). Each $\frac{n}{p} \times n$ submatrices of A_i and B_i are further partitioned into n_{th} threads per processor, where each thread consists of $\frac{n}{n_{th}p} \times n$ submatrices. In our implementation, each thread performs four basic operations; namely, *request send, request service*, *computation*, and *context-switch*. First, each processor P_i

sends out a request for $B_{j,k}$ (for $0 \le j \le p-1$, $j \ne i$ and $0 \le k \le n_{th}-1$) of size $\frac{n}{n_{th}p} \times n$ to the remote processor P_j, where j is the processor number and k represents the k^{th} thread (i.e., request send). While this remote access is pending, each processor performs the following operations: (1) polls to see if a request from other processors has arrived and if there are any, services it (first request service); (2) performs multiplication/addition on a thread that resides on its local memory, which is non-blocking (computation); (3) once again, polls to see if a remote request has arrived during the computation portion and services it (second request service); (4) polls to determine if its own requested data has arrived (remote receive). Otherwise processor idles until the data arrives; finally, context-switches to the next thread as shown in Figure 3.3.

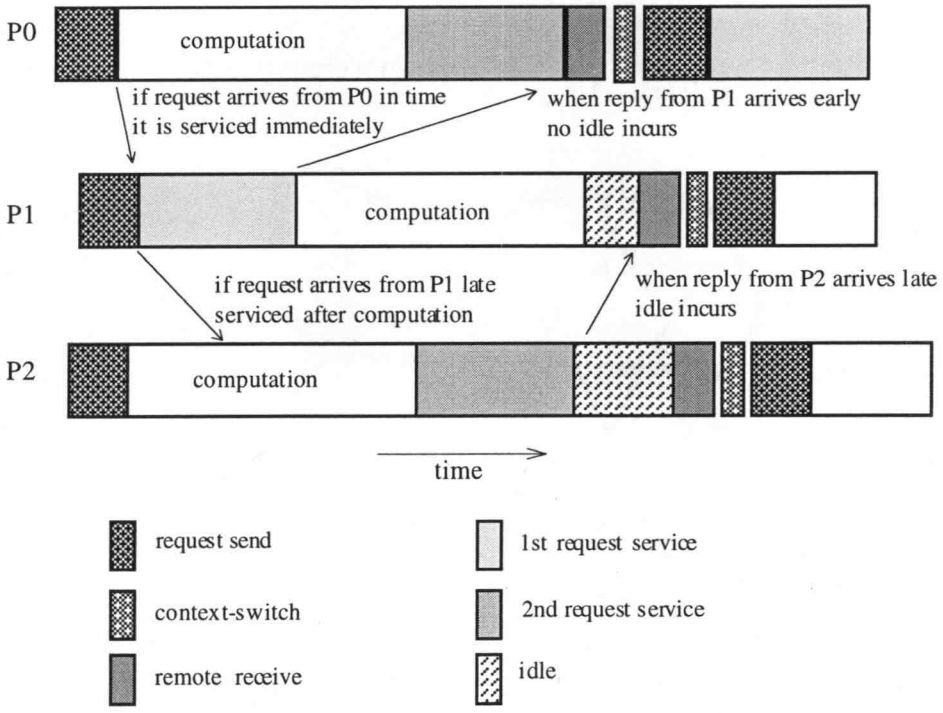


Figure 3.3: An example of multithreaded execution as a function of time. A thread execution is composed of request-send, request-service, computation, remote receive, and context-switch. Note that when remote access takes longer than the computation idling occurs.

For simplicity, assume that t_r is the time required to send out a request plus the time to check for a remote request, and t_{cs} is the context switching cost. We also assume the time to service a request is approximately equal to the startup time t_s, since servicing a remote request involves allocating buffer space, packaging the data, and setting up network interface. Note that the length of computation in a thread is $\frac{n^2}{n_{th}p}$, requiring computation time of $c\frac{n^3}{n_{th}p^2}$. Therefore, the run-length of a thread (i.e., granularity) is given as $t_r + t_s + c\frac{n^3}{n_{th}p} + t_{cs}$. Every processor will eventually execute $n_{th}(p-1)$ (since p^{th} remote request and service are not necessary) before the multiplication completes. Based on this, the run-time of the multithreaded version can be expressed as

$$\begin{aligned} T_{mt}^{ideal} &= n_{th}(p-1)\left(t_r + t_s + c\frac{n^3}{n_{th}p^2} + t_{cs}\right) + n_{th}\left(c\frac{n^3}{n_{th}p^2} + t_{cs}\right) \\ &= c\frac{n^3}{p} + n_{th}pt_{cs} + n_{th}(p-1)(t_r + t_s) \\ T_{mt}^{ideal} &\approx c\frac{n^3}{p} + n_{th}p(t_r + t_s + t_{cs}) \end{aligned} \tag{3.2}$$

The approximation for Equation 3.2 is obtained assuming $p >> 1$. Note that Equation 3.2 is valid only if the granularity is optimal so that the computation time is long enough to mask remote latencies—i.e., the network transfer time is completely hidden by the computation, and thus represents the best possible execution time for multithreaded matrix multiplication.

However, it is more likely that the round-trip time for the remote access takes longer than computation since the network speed is very slow compared to the processor speed and/or servicing of the remote request by the remote processor is delayed (i.e., the request is serviced after the local computation has been completed). This can be mod-

eled by considering two different cases. In the first case, the request is serviced immediately by the remote processor (i.e., the remote processor services the request before performing its local computation), the round-trip time for the remote access can be estimated as $t_r + t_w + t_s + t_w \frac{n^2}{n_{th}p}$, where $t_r + t_w$ is the time to send/detect a request to/from the remote processor, and $t_s + t_w \frac{n^2}{n_{th}p}$ is the time spent by the remote processor to service (including network delay) the request. For this case, the run-length of a thread is composed of the time to send out a request, the round-trip time for the remote access, and the context switching cost, requiring a time of $t_r + t_w + t_s + t_w \frac{n^2}{n_{th}p} + t_{cs}$. Another case is when a remote request is serviced after the computation portion in the remote processor. If we add the computation time to the remote access time, it will require a time of $t_r + t_w + t_s + t_w \frac{n^2}{n_{th}p} + c\frac{n^3}{n_{th}p^2} + t_{cs}$. Assuming $p >> 1$ and considering the fact that the last p^{th} step does not require any remote accesses, the total run-time is given as

$$
\begin{aligned}
T_{mt} &= n_{th}(p-1)\left(t_r + t_w + t_s + t_w\frac{n^2}{n_{th}p} + c\frac{n^3}{n_{th}p^2} + t_{cs}\right) + n_{th}\left(c\frac{n^3}{n_{th}p^2} + t_{cs}\right) \\
&= c\frac{n^3}{p} + n_{th}(p-1)\left(t_r + t_w + t_s + t_w\frac{n^2}{n_{th}p} + t_{cs}\right) + n_{th}t_{cs} \qquad (3.3) \\
T_{mt} &\approx c\frac{n^3}{p} + n_{th}p\left(t_r + t_w + t_s + t_{cs}\right) + t_w n^2
\end{aligned}
$$

Equation 3.3 shows that the overall execution time depends on the communication time rather than the computation time. This situation will occur when the number of threads is too large or the granularity is too small to effectively mask the remote latency. Therefore, the granularity has to be properly determined such that the computation and communication are well balanced to achieve optimum performance.

3.3 Experimental Results

In the previous section, several key characteristics of message-passing and multithreaded versions of the matrix multiplication algorithm were identified using analytical models. This section presents experimental results of the two versions running on four 100MHz Pentium-based LINUX workstations connected via Ethernet. To compare performances, the simulation programs were developed using PVM. Since PVM does not support multithreading, a run-time system was implemented to provide a virtual global memory space and thread scheduling.

Our simulation program for the multithreaded version allows granularity of threads to be varied by assigning an arbitrary number of threads to each node. The scheduling of threads is software-controlled to exploit the locality as much as possible rather than relying on the dynamic behavior of the run-time scheduler provided by PVM or TPVM [29]. The measured execution time of each simulation includes the time taken for PVM processes to initialize as well as the time to execute the matrix multiplication routine. In addition, the thread context switching cost is assumed to be 50 μsec [29].

In the message-passing implementation, each processor proceeds first with the multiplication of submatrices and then communicates among all four nodes. For the multithreading implementation, we simulate the shared-memory abstraction by sending out a remote request for next submatrices before proceeding with the computation part. Also, each processor checks to see if there are any incoming requests before and after the computation step on local submatrices. If a request is detected before the computation, the processor immediately prepares the requested data and sends them out to the requesting nodes. Otherwise, incoming requests are serviced after the computation. After servicing the remote requests and performing computation, each processor checks if its own requested data has arrived. If the remote data has arrived, a context-switch occurs to the next thread. If not, the processor waits for the requested data.

Table 3.1 shows the speedup of the multithreaded version over the sequential version as a function of matrix dimension. These results indicate the speedup increases from 2.18 to 3.45 as the matrix dimension increases. A low speedup was obtained for n=200 because each node experiences a large portion of remote latencies due to its relatively fine granularity. On the other hand, the speedup factor was higher for n=1000, indicating that its coarse granularity allows a larger portion of the remote access time to be tolerated.

Table 3.1: Speedup of multithreaded version vs. sequential version.

Dimension	200	400	600	800	1000
Speedup	2.18	2.81	3.08	3.27	3.45

Figure 3.4 shows the overall execution time of matrix multiplication for the matrices of size 200×200 through 1000×1000. The results for MT were based on five threads per each processor, and the results for MT-opt were obtained when each processor has an optimum number of threads relative to matrix dimensions, i.e., n_{th}=5 for n =200 and 400, n_{th}=20 for n=600, n_{th}=25 for n =800, and n_{th}=50 for n =1000. It can be seen that message-passing version MP gives slightly better performance compared to MT. However, when n_{th} is chosen properly, as in the case of MT-opt, performance is comparable to MP. For example, 50 to 75 threads per processor for 1000×1000 matrix multiplication resulted in very close or even better performance than MP.

We have also experimented with the various distributions of submatrices among nodes for the purpose of investigating the sensitivity of MT or MT-opt to data distribution. The resulting graphs MT-rd and MT-rd-opt show that performance suffered less

than 5 percent compared to MT. This indicates that matrix multiplication algorithm using multithreading is somewhat insensitive to the initial data distribution.

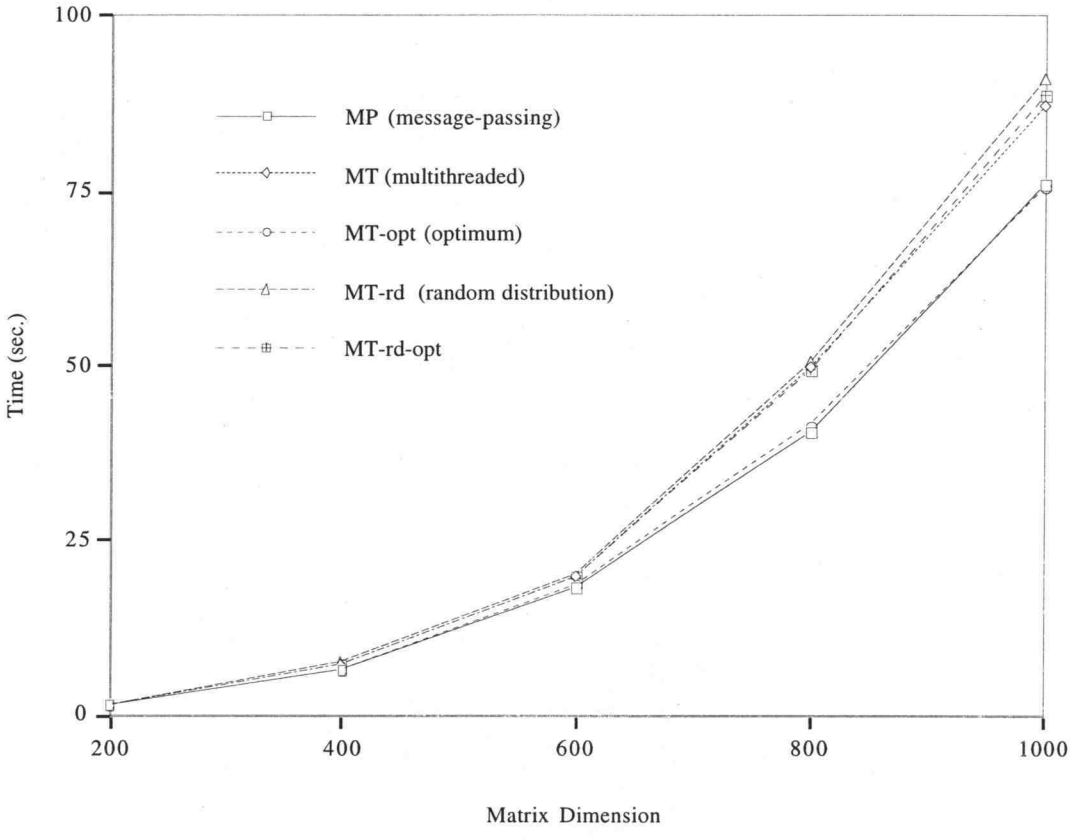


Figure 3.4: Execution times of message-passing and multithreaded versions of the row-wise stripped matrix algorithm.

Figure 3.5 shows the execution time increases when the number of threads is either very small or large. These effects can be explained as follows: When n_{th} is small, the processor has to idle because more data is needed per thread thus requiring more communication time (i.e., $t_w n^2$ term in Equation 3.3 becomes dominant and the computation time cannot mask the remote access delay). As n_{th} becomes large, the number of context-switches required increases thereby degrading the performance as indicated by

the term $n_{th}p(t_r + t_w + t_s + t_{cs})$ in Equation 3.3. The experimental results indicate that depending on the sizes of the matrices, approximately 5 to 25 threads per processor is enough to achieve the best possible performance. For example, we found the following numbers of threads per processor resulted in optimum performance: n_{th}=5 for n=200 and 400, n_{th}=20 for n=600, and n_{th}=25 for n=800. Therefore, a relatively small number of threads per processor is enough to maintain a high processor utilization [77].

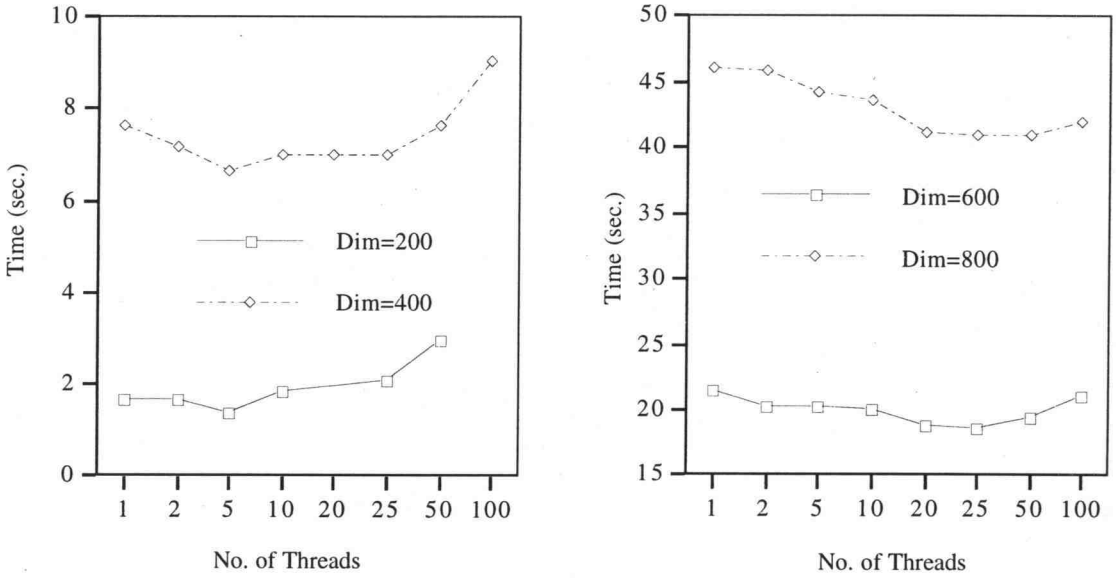
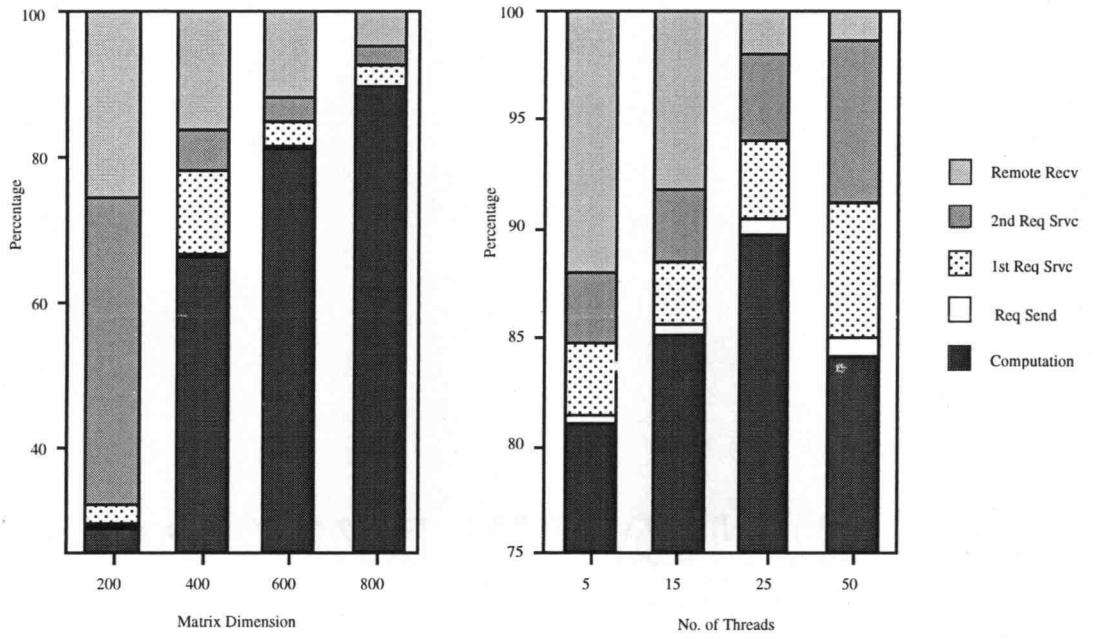


Figure 3.5: Execution time vs. number of threads per processor.

Figure 3.6 (a) shows the percentages of time spent on each component of a thread as a function of n for n_{th}=5. For illustration, a thread execution is subdivided into the following five components: `Computation` is the time spent on multiplication/addition; `Req Send` is the time taken to send a request to a remote processor; `1st Req Srvc` is the time spent to service the remote request before the computation; `2nd Req Srvc` is the time required to service the remote request after the computation (a

remote request will be serviced either in `1st Req Srvc` portion or `2nd Req Srvc` portion, but not both); and `Remote Recv` is the time taken to load the data into the local memory from the network buffer plus, if there are any, the idle time.



(a) as a function of dimension (n_{th}=5) (b) as a function of number of threads (n=600)

Figure 3.6: Percentages of various components of a thread execution.

Figure 3.6 (a) shows `Computation` portion increases as n increases, thereby tolerating longer remote latencies (e.g., this can be seen by the decrease in `Remote Recv` as a function of n). `Req Send` takes almost constant amount of time regardless of matrix dimension because the t_s portion of the communication is much larger than t_w. However, the total amount of time spent on sending out a remote request will increase if the number of threads becomes large. Our experiments also showed there is no definite pattern in terms of whether a remote request is more likely to fall in the `1st Req Srvc` portion or in the `2nd Req Srvc` portion—i.e., the chances of an incoming re-

mote request to be serviced before or after the computation is about the same. Figure 3.6 (b) shows the changes in the various components of a thread execution as a function of n_{th} for n=600. It shows that only the components related to remote operation increases as n_{th} increases. Therefore, each processor has to spend more time servicing remote requests, and the performance will degrade.

3.4 Conclusion

With the availability of powerful microprocessors, high-speed networks, and software systems, it is possible to build a distributed parallel computing environment on NOWs. Moreover, with the availability of software infrastructure system, such as PVM, existing parallel applications can be easily ported to run on NOWs. However, the complexity of programming using the message-passing model still remains to be a major hurdle. Although the DSM system alleviates this problem to some extend by providing a shared-memory abstraction, multithreading technique can further amortize the inefficiency of communication involved in shared-memory accesses. However, multithreading carries the overhead of context switching cost. In addition, granularity and scheduling of threads become very important factors in achieving high performance.

Our findings indicate that the message-passing execution outperforms its multithreaded counterpart when thread computation cannot effectively mask the remote latency. Also, context switching and interruptions within a thread to service remote requests adds to the overhead of implementing a multithreaded system. However, the reverse is true when the granularity of threads is properly chosen so that thread computation effectively masks the remote memory latency. We also found that for the matrix multiplication algorithm the multithreaded execution is relatively insensitive to initial data distribution.

4. Multithreaded Virtual Processor

Current trends in VLSI technology indicates that the performance gap between processors and main memory will continue to increase. However, modern superscalar processors do not provide any mechanism to tolerate memory latency. Consequently, the speed of memory system is likely to be a major limiting factor for improving the performance of a future microprocessor. Multiprocessors and multicomputers also greatly exacerbate the memory latency problem. In SMPs, contention due to the shared bus located between the processor's L2 cache and the shared main memory subsystem adds additional delay to the memory latency. The memory latency problem becomes even more severe for scalable distributed shared memory (DSM) systems because a miss on the local memory requires a request to be issued to the remote memory and a reply to be sent back to the requesting processor. This limits the performance and scalability since the proportion of the processor time actually spent on useful work keeps diminishing as parallel machines become larger.

There are a number of techniques that effectively reduce the memory latency, such as prefetching, compiler optimizations, and multi-level caches, but they do not provide a complete solution. Therefore, the remaining latency must be tolerated. A multithreaded system contains multiple "loci of control" (or threads) within a single program and provides the processor with an ability to switch between the threads in order to tolerate memory latency. Also, the processor's resources may be shared among multiple threads, yielding better processor utilization. There are two types of architectures that support the exploitation of thread-level parallelism (TLP): *multiprocessor* and *multithreaded systems*. Multiprocessors replicate a number of superscalar processors and provide inter-processor communication mechanism via shared-memory. Threads are statically partitioned and executed on a separate processor. Therefore, it is difficult for a

multiprocessor to dynamically exploit TLP among the processors. On the other hand, multithreaded systems provide support for multiple contexts and fast context switching within the processor pipeline. This allows multiple threads to share the processor's resources by dynamically switching between threads.

In light of the aforementioned discussion, this chapter proposes the Multithreaded Virtual Processor (MVP) that exploits the synergy between the multithreaded programming paradigm and the well-designed contemporary microprocessors. MVP is a proof of concept that, by providing an adequate hardware support to an existing superscalar core, we can take full advantage of the increasingly popular and powerful programming tools that exploit thread-level parallelism (TLP). With its fast context switching and hardware scheduling mechanisms, MVP provides the capability to hide cache miss latencies. In order to validate the MVP concept, a simulator was developed that integrates a general purpose POSIX thread package [62] and a multithreaded superscalar processor simulator.

4.1 MVP Architecture

Multithreaded Virtual Processor is a coarse-grain multithreaded system that augments the modern superscalar core with hardware and software support for multithreading. The objective of the proposed MVP is to extend the software-controlled multithreading model with hardware support while providing a transparent view to the programmer. The MVP is based on the block-multithreading paradigm where each thread is constructed from a user-defined function. With its multiple hardware contexts and fast context switching mechanism, MVP can tolerate the long latency operations such as cache misses, synchronization points, and I/O operations.

The MVP architecture is organized into two layers shown in Figure 4.1: *software layer* and *hardware layer*. The software layer provides the facilities for coding

multithreaded applications using Pthreads [15]. Pthreads is a POSIX compliant thread extension that specifies a priority-driven thread model with preemptive scheduling policies. The hardware layer in MVP consists of a conventional superscalar processor core augmented with the *Hardware Scheduler* and *Multiple Hardware Contexts*. The responsibility of the Hardware Scheduler is to basically coordinate the execution of threads scheduled onto MVP in attempt to hide the cache miss latency.

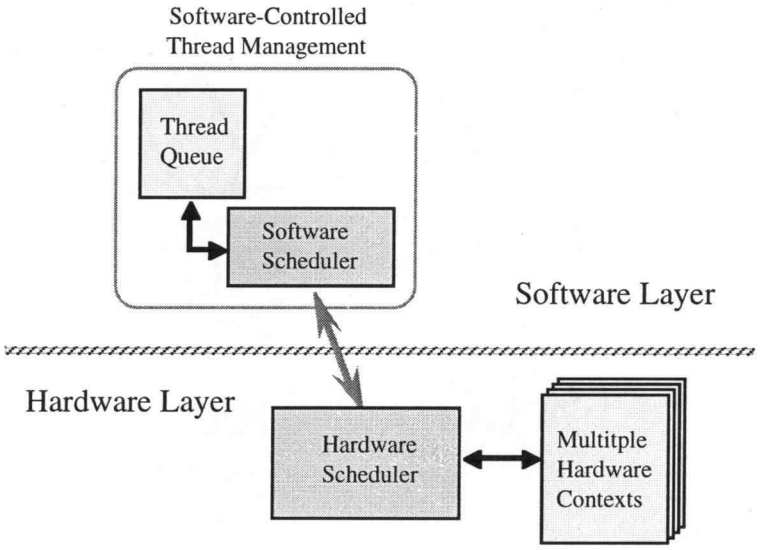


Figure 4.1: An overview of MVP system.

4.1.1 Software Layer of MVP

The software layer of MVP consists of Pthreads, which is based on the POSIX 1003.1c 1995 thread standard. This standard passed the International Standards Organization (ISO) Committee Document balloting in February 1995 and received the IEEE Standards Board approval in June 1995. This POSIX threads extension specifies a pri-

ority-driven thread model with preemptive scheduling policies, signal handling, and primitives to provide mutual exclusion as well as synchronized waiting.

A thread is an independent sequence of code execution (flow of control) within a regular UNIX process. Threads share the global data (global variables, files, etc.) but maintains their own stack, local variables, and program counter. Therefore, the context of thread is much smaller than the context of a process, and the context switching between threads is much cheaper than switching between processes. Thus, the use of threads increases the overall throughput and responsiveness by allowing an efficient overlapping of computation and I/O requests. However, thread programming usually requires more effort due to increased complexity of managing multiple threads.

Although Pthreads' API (Application Programming Interface) is a standard, its implementation varies—Pthreads can be implemented using a combination of kernel and user library [62]. However, our discussion focuses on a library implementation of Pthreads based on Chris Provenzano's Pthreads[1]. There are numerous features available in Pthreads that give programmers the ability to write concurrent applications. For example, Pthreads provides functions for creating, terminating, and joining threads. It also has two thread synchronization primitives, the mutex and the conditional variable, which are used to control access to a shared resource. Table 4.1 lists some of the features available in the POSIX threads standard. A more detailed discussion on the API can be found in [15].

Pthreads also defines the scheduling policies along with mechanisms to control scheduling of threads. There are three scheduling policies: FIFO, round-robin (RR), and user-controlled priority. The thread scheduling in Pthreads is basically just moving

---

[1]Provenzano's Pthreads package can be obtained from
http://www.mit.edu:8001/people/proven/pthreads.html.

threads among various event queues that Pthreads maintains. All 14 queues of Pthreads are listed in Table 4.2 along with their functions.

Table 4.1: Some of the features provided by Pthreads.

Thread Management	
pthread_create()	Creates a new thread
pthread_exit()	Terminate the calling thread
pthread_join()	Synchronize with the termination of a thread
pthread_getschedpara()	Get the scheduling policy and parameters of the specified thread
pthread_setschedpara()	Set the scheduling policy and parameters of the specified thread.
Thread Synchronization	
pthread_mutex_init()	Initialize a mutex
pthread_mutex_lock()	Acquire the indicated mutex
pthread_mutex_unlock()	Release the previously acquired mutex
pthread_mutex_destroy()	Destroy a mutex
pthread_cond_init()	Initialize a condition variable
pthread_cond_destroy()	Destroy a condition variable
pthread_cond_signal()	Unblock one thread currently blocked in the specified condition variable
pthread_cond_broadcast()	Unblock all threads currently blocked in the specified condition variable
pthread_cond_wait()	Block on the specified condition variable.

Figure 4.2 shows the states of a thread as it moves around the queues. When a thread is created, it is initially pushed onto the pthread_current_prio_queue (PQ) and is said to be *runnable*. From this queue, the thread becomes *active* in the order of its priority (i.e., being executed). If the thread has to suspend its execution to wait for some event or signal, it is put on the queue for that type of event, i.e., *blocked.* When the signal or the event for which the thread has been waiting for occurs, the thread becomes runnable again and is moved back to the PQ. Eventually, the thread is *destroyed* when the execution of the thread is done or exits.

Table 4.2: Various queues and their associated functions.

Queue	State	Description and Functions
pthread_current_prio_queue	PS_RUNNING	running
join_queue	PS_JOIN	waiting for a thread to die pthread_join()
mutex.m_queue	PS_MUTEX_WAIT	waiting for a mutex pthread_mutex_lock()
cond.c_queue	PS_COND_WAIT	waiting on a condition variable pthread_cond_wait()
r_queue	PS_FDLR_WAIT	waiting on a fd read lock fd_lock()
w_queue	PS_FDLW_WAIT	waiting on a fd write lock fd_lock()
pthread_sleep	PS_SLEEP_WAIT	waiting on a sleep sleep() usleep() nanosleep()
wait_queue	PS_WAIT_WAIT	waiting for a child to die wait() waitpid() [wait3() wait4()]
pthread_sigwait	PS_SIGWAIT	waiting on a set of signals sigwait()
pthread_dead_queue	PS_DEAD	waiting for a thread to join with it or detach it
pthread_alloc_queue	PS_UNALLOCED	available to use for a new thread
fd_wait_read	PS_FDR_WAIT	waiting on a kernel fd to have data to read read() readv() recv()
fd_wait_write	PS_FDW_WAIT	waiting on a kernel fd to write data write() writev()
fd_wait_select	PS_SELECT_WAIT	waiting for several fds in a select select()

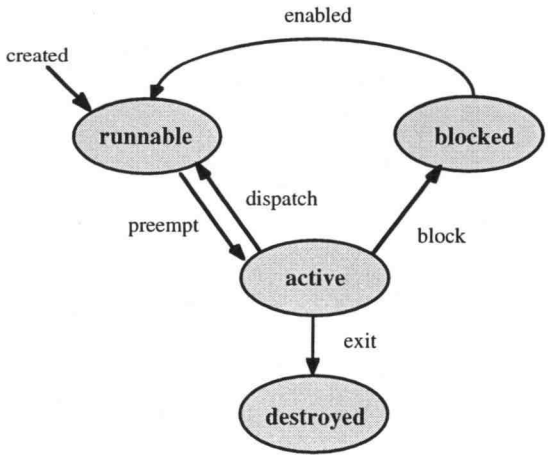


Figure 4.2: A simplified view of the thread state.

Figure 4.3 shows that the PQ of Pthreads is composed of a linked-list of threads for each priority level. If a priority level is empty, it is linked to the tail of the level above. The Pthreads scheduler is responsible for managing the PQ and the context-switches according to the scheduling policy. Threads in a higher priority list are scheduled before threads in a lower priority list. Within each list, where all threads have the same priority, the threads are scheduled according to the policy described next.

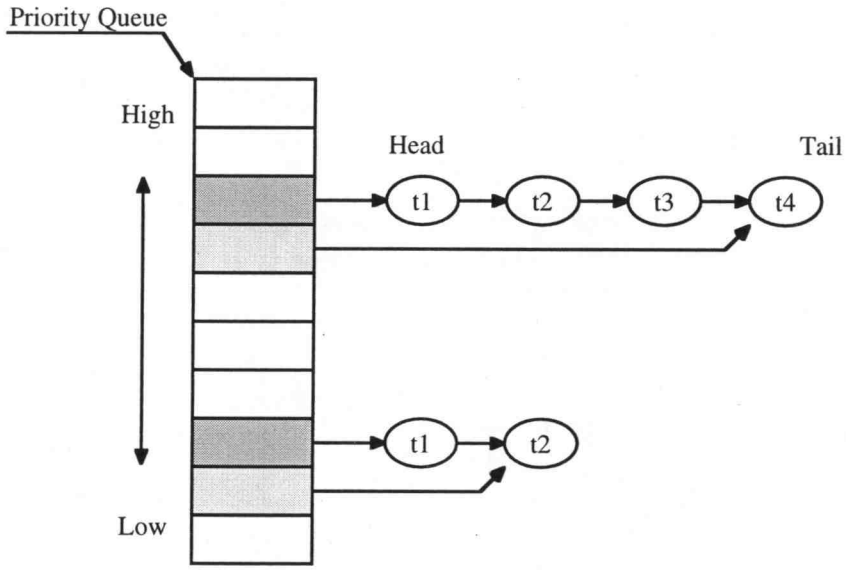


Figure 4.3: A structure of priority queue in Pthreads.

- SCHED_FIFO is a first-in first-out order. When a thread is preempted, it is put back at the head of the list and continues running unless there is a thread with a higher priority waiting. When a thread is blocked, it is put at the tail of the list. Also, if the thread's scheduling policy or priority changes, it is put at the tail of the new priority's list.

- SCHED_RR is like SCHED_FIFO, except that each thread has a timer set. When the timer expires, the thread will go to the tail of the list and another thread is scheduled for the duration of PTHREAD_ITIMER_INTERVAL.
- SCHED_OTHER implements a user-controlled priority scheme that is a combination of FIFO and RR. It will set the timer like RR only when there are no signals pending. Otherwise, it does not set the timer.

Pthreads has a large number of the internal functions and their associated data structures in order to facilitate the process of creating threads, initiating the thread execution, and switching between threads. When a user program is invoked, it first executes Pthreads initialization routines to set up the UNIX *signal* function. The prototype of the signal function, given in the system header file <signal.h>, is defined as void (*signal (int signo, void (*func (int))) (int). Signals are software interrupts, and Pthreads interacts with the operating system (OS) via signals. These interrupts initiate the context switching process in Pthreads.

When the specified signal event (i.e., the signal matching signo) occurs during the execution of a program, Pthreads uses the signal function to transfer the control to the sig_handler() in Pthreads scheduler. There are four major components in Pthreads scheduler: sig_handler(), pthread_resched_resume(), pthread_sched_other_resume(), and context_switch(). The sig_handler() provides an external entry point to the scheduler via OS signals; pthread_resched_resume() and pthread_sched_other_resume() provide an internal path to the scheduler. In particular, pthread_resched_resume() is called whenever the current thread has to reschedule itself after being blocked, and pthread_sched_other_resume() is used to check if the thread to be resumed has a higher priority. If that is the case, it stops the current thread and starts a new thread. All three routines eventually invoke the context_switch() routine to actually perform the context-switch.

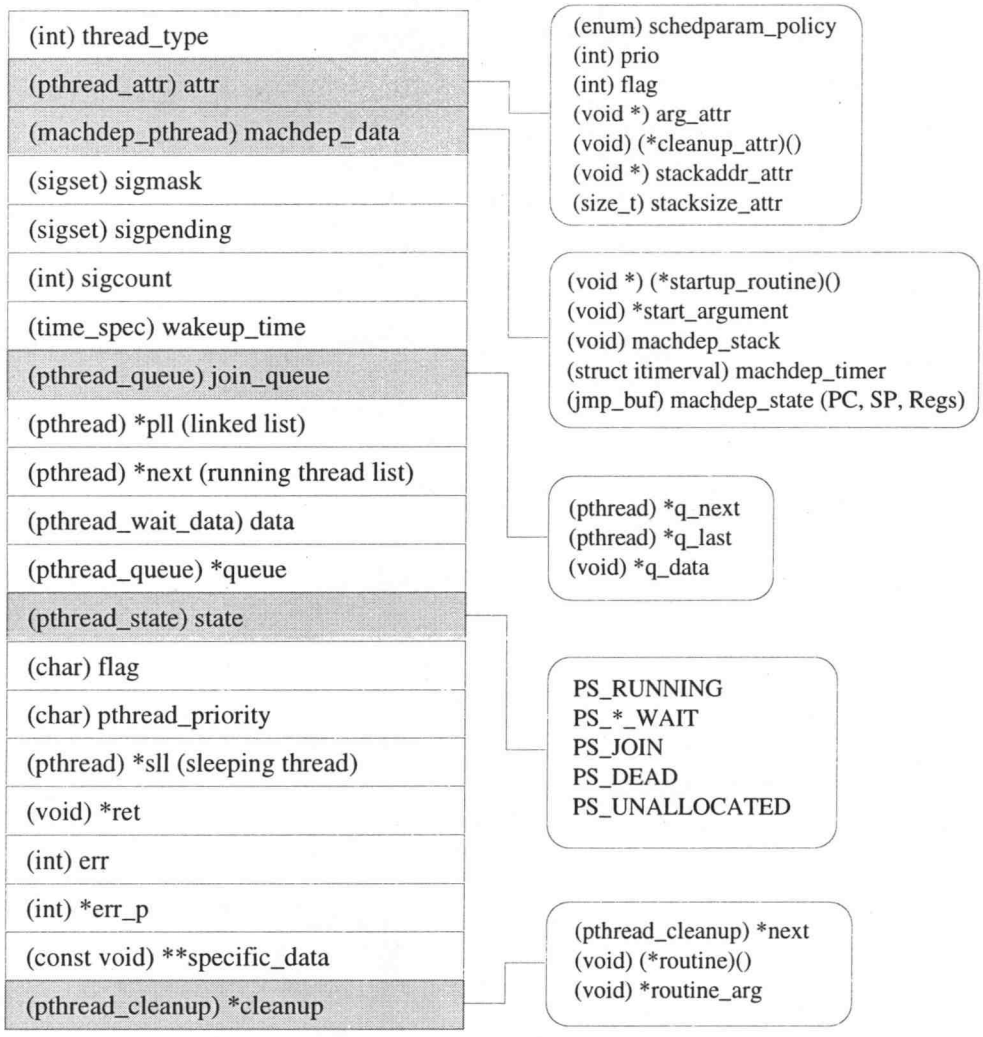


Figure 4.4: The data structure of Pthreads.

In order to understand how a context-switch is performed, let us first examine the thread structure shown in Figure 4.4. Basically this structure is used to hold all the necessary information to manage a thread: thread's attributes (e.g., priority and stack address), signal information (e.g., signal mask), timer value, machine-dependent parameters, etc. Among them, the machine dependent portion of the data structure provides the allocation of private stack space for each thread. This stack space is used as storage not only for function calls, but also for saving and restoring of the thread state. In particular, the jmp_buf is utilized by setjmp() and longjmp() functions to implement

the context switching mechanism. The setjmp() and longjmp() are the standard C library calls that implement non-local jumps (i.e., branching back outside the called frame). The following are the code segments of the setjmp() and longjmp() routines. As can be seen, setjmp() saves the contents of the registers (i.e., the context of the thread), and longjmp() restores the saved state (i.e., callee-saved registers, stack pointer, and program counter) and starts the execution of the thread by jumping to the instruction pointed by the PC.

setjmp()

```
/* Store the floating point callee-saved registers... */
asm volatile ("s.d $f20, %0" : : "m" (env[0].__fpregs[0]));
asm volatile ("s.d $f22, %0" : : "m" (env[0].__fpregs[1]));
asm volatile ("s.d $f24, %0" : : "m" (env[0].__fpregs[2]));
asm volatile ("s.d $f26, %0" : : "m" (env[0].__fpregs[3]));
asm volatile ("s.d $f28, %0" : : "m" (env[0].__fpregs[4]));
asm volatile ("s.d $f30, %0" : : "m" (env[0].__fpregs[5]));
/* .. and the PC; */
asm volatile ("sw $31, %0" : : "m" (env[0].__pc));
/* .. and the stack pointer; */
asm volatile ("sw %1, %0" : : "m" (env[0].__sp), "r" (sp));
/* .. and the FP; it'll be in s8. */
asm volatile ("sw %1, %0" : : "m" (env[0].__fp), "r" (fp));
/* .. and the GP; */
asm volatile ("sw $gp, %0" : : "m" (env[0].__gp));
/* .. and the callee-saved registers; */
asm volatile ("sw $16, %0" : : "m" (env[0].__regs[0]));
asm volatile ("sw $17, %0" : : "m" (env[0].__regs[1]));
asm volatile ("sw $18, %0" : : "m" (env[0].__regs[2]));
asm volatile ("sw $19, %0" : : "m" (env[0].__regs[3]));
asm volatile ("sw $20, %0" : : "m" (env[0].__regs[4]));
asm volatile ("sw $21, %0" : : "m" (env[0].__regs[5]));
asm volatile ("sw $22, %0" : : "m" (env[0].__regs[6]));
asm volatile ("sw $23, %0" : : "m" (env[0].__regs[7]));
```

longjmp()

```
/* Pull back the floating point callee-saved registers. */
asm volatile ("l.d $f20, %0" : : "m" (env[0].__fpregs[0]));
asm volatile ("l.d $f22, %0" : : "m" (env[0].__fpregs[1]));
asm volatile ("l.d $f24, %0" : : "m" (env[0].__fpregs[2]));
asm volatile ("l.d $f26, %0" : : "m" (env[0].__fpregs[3]));
asm volatile ("l.d $f28, %0" : : "m" (env[0].__fpregs[4]));
asm volatile ("l.d $f30, %0" : : "m" (env[0].__fpregs[5]));
/* Restore the stack pointer. */
asm volatile ("lw $29, %0" : : "m" (env[0].__sp));
/* Get and reconstruct the floating point csr. */
asm volatile ("lw $2, %0" : : "m" (env[0].__fpc_csr));
/* Get the callee-saved registers. */
```

```
asm volatile ("lw $16, %0" : : "m" (env[0].__regs[0]));
asm volatile ("lw $17, %0" : : "m" (env[0].__regs[1]));
asm volatile ("lw $18, %0" : : "m" (env[0].__regs[2]));
asm volatile ("lw $19, %0" : : "m" (env[0].__regs[3]));
asm volatile ("lw $20, %0" : : "m" (env[0].__regs[4]));
asm volatile ("lw $21, %0" : : "m" (env[0].__regs[5]));
asm volatile ("lw $22, %0" : : "m" (env[0].__regs[6]));
asm volatile ("lw $23, %0" : : "m" (env[0].__regs[7]));
/* Get the PC.  */
asm volatile ("lw $31, %0" : : "m" (env[0].__pc));
/* Give setjmp 1 if given a 0, or what they gave us if non-zero.  */
if (val == 0)
asm volatile ("li $2, 1");
else
asm volatile ("move $2, %0" : : "r" (val));
asm volatile ("j $31");
```

In the process of a thread creation, PC in the jmp_buf is initialized to the starting point of the thread by setjmp(). Then the longjmp() sets the PC to the starting point of the thread when the thread is scheduled for the first time, and the execution of the thread begins. If the Pthreads scheduler has to perform a context-switch, the context_switch() is called in which setjmp() is executed and the state of the current thread along with the PC are saved in the jmp_buf. After saving the current thread, it first schedule another thread from PQ and restores the context of the thread by calling longjmp(), thereby completing the context-switch.

Now consider how threads and a UNIX process interact during the execution of the threads. When a process runs, the OS sets up a location for the potential state of the process on the process stack. When an interrupt occurs, the state of the process is saved onto the stack, the stack pointer is incremented, and the processor runs the interrupt handling routine (i.e., signal()). When the routine finishes and returns from the interrupt, the OS pops the process's state off the stack, decrements the stack pointer, and the process returns to execution. The following example explains the overall execution sequence of a Pthreads program. Pthreads programs are written in C with Pthreads API and then linked with the Pthreads library to produce the executables. In the example shown below, pthread_create() creates a new thread named f() with attributes

thread_id[i] and arguments pointed to by (void *) &args[i], and pthread_join() joins threads created by pthread_create(). The sequence of overall process is depicted in Figure 4.5.

```
main()
{
        for (i=0;i<=n;i++)
                pthread_create(&thread_id[i],NULL,f,(void *)&args[i]);
        ...
        for (i=0;i<=n;i++)
                pthread_join(thread_id[i], NULL);
}

void *f(void *param)
{
        ...
        do something
        ...
}
```

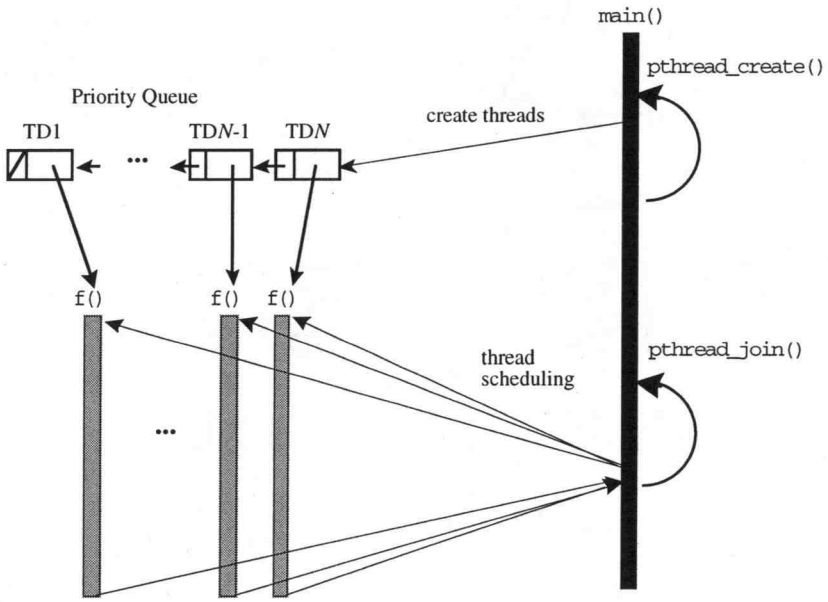
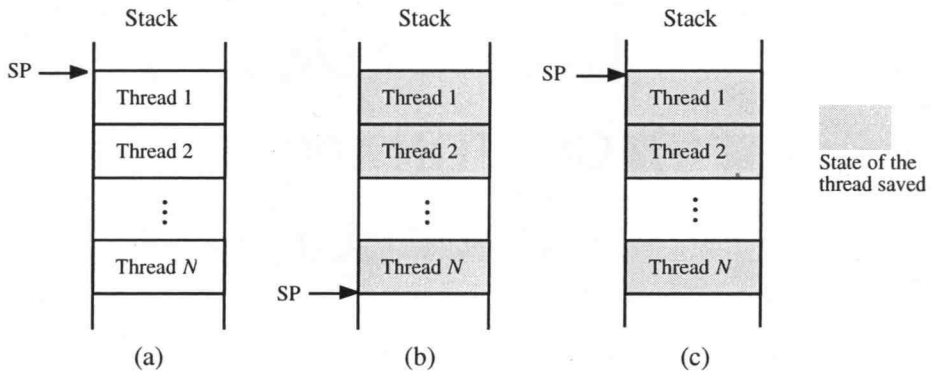


Figure 4.5: The execution of the given sample Pthreads program.

In the first loop, n number of threads are created and their thread identifiers (TIDs) are enqueued onto the PQ. Figure 4.6(b) shows the state of the stack just before the first thread starts its execution and just after the N^{th} thread have been saved onto the PQ. Once all the threads are created, pthreads_join() invokes the Pthreads scheduler, which schedules a thread by dequeuing its TID from PQ. It is most likely that Thread 1 will be removed from the PQ first. Then, a longjmp() call is made by the context_switch() inside Pthreads scheduler, moving the stack pointer from the end of Thread N's frame to the beginning of the Thread 1's frame as shown in Figure 4.6(c), and the Thread 1 starts its execution at f().



(a) Initial state of the stack before allocation.
(b) State of the stack before executing first thread and after saving the state of the thread N.
(c) State of the stack after a `longjmp()` to Thread 1 and after the state of Thread 1 has been restored.

Figure 4.6: A diagram of stack states.

As each thread is executed, Pthreads scheduler starts an OS timer (set by calling setittimer() system call). If the timer expires before the thread finishes its execution, an interrupt (i.e., sigvtalrm) is generated by the OS. Assuming RR scheduling policy, the following sequence of events occur when the timer interrupt occurs:

1. The OS recognizes the interrupt and saves the state of the process onto the frame on the stack given by the stack pointer.
2. The OS loads the interrupt handler, i.e., sig_handler() in Pthreads.
3. sig_handler() eventually runs the context_switch() routine.
4. The TID of the interrupted thread is enqueued onto the PQ and a new thread selected and removed from the PQ (Thread 2 is likely to be the one).
5. To start the new thread running, the stack pointer is set to the end of the new thread's frame by longjmp() and Pthreads returns from sig_handler().
6. The OS then restores the state of the process (which moves SP to the beginning of the new thread's stack frame) and begins the execution of the new thread.

When all the threads on the PQ have completed their execution, the control returns to the main(), and program exits.

4.1.2 Hardware Layer of MVP

The hardware support for MVP model consists of a conventional superscalar processor core augmented with the hardware scheduler and multiple hardware contexts as shown in Figure 4.7. A long latency operation (e.g., L2 cache misses) detected by the memory management unit (MMU) causes a thread to context-switch. The context switching is accomplished by the hardware scheduler where the blocked thread is placed in a hardware context and a new thread is scheduled from another hardware context. Once threads are scheduled onto the hardware from the PQ of Pthreads, threads remain in the processor until the execution of threads is done in order to avoid unnecessary overhead involved in software-controlled thread management. However, in the event of a synchronization, threads may swap between the hardware contexts and the thread queues managed by Pthreads.

In order to manage multiple contexts, each context inside the processor is represented by a tag *TID*, containing a thread ID, a PC, and a pointer to the thread stack. To understand how MVP executes threads, first consider how a context switching is performed. When an L2 cache miss is detected, MVP initiates a hardware context-switch. Since a hardware context in MVP is represented as a register bank, the context switching is a simple process of deactivating one bank and activating another. After the context-switch is done, new instructions are fetched from the memory location indicated by PC of the new hardware context.

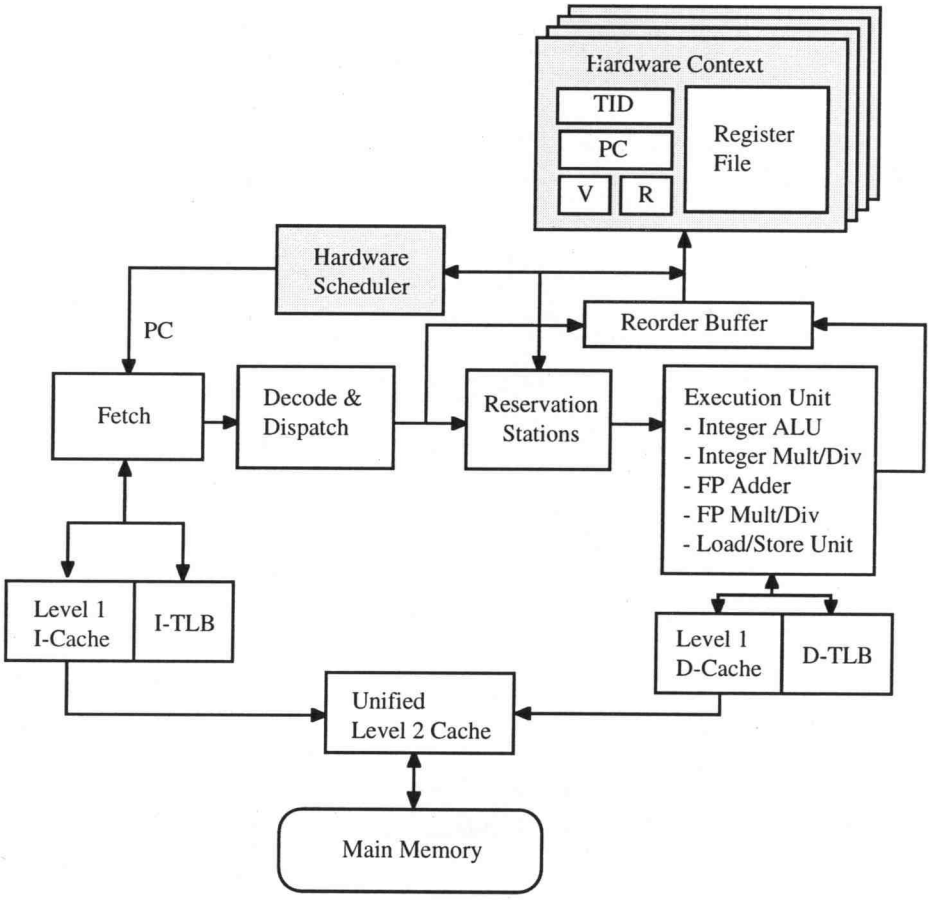


Figure 4.7: An overview of the MVP hardware system.

The interaction among the user program, Pthreads function calls, Pthreads scheduler, and the Hardware Scheduler is shown in Figure 4.8. The functionality of each state is explained below:

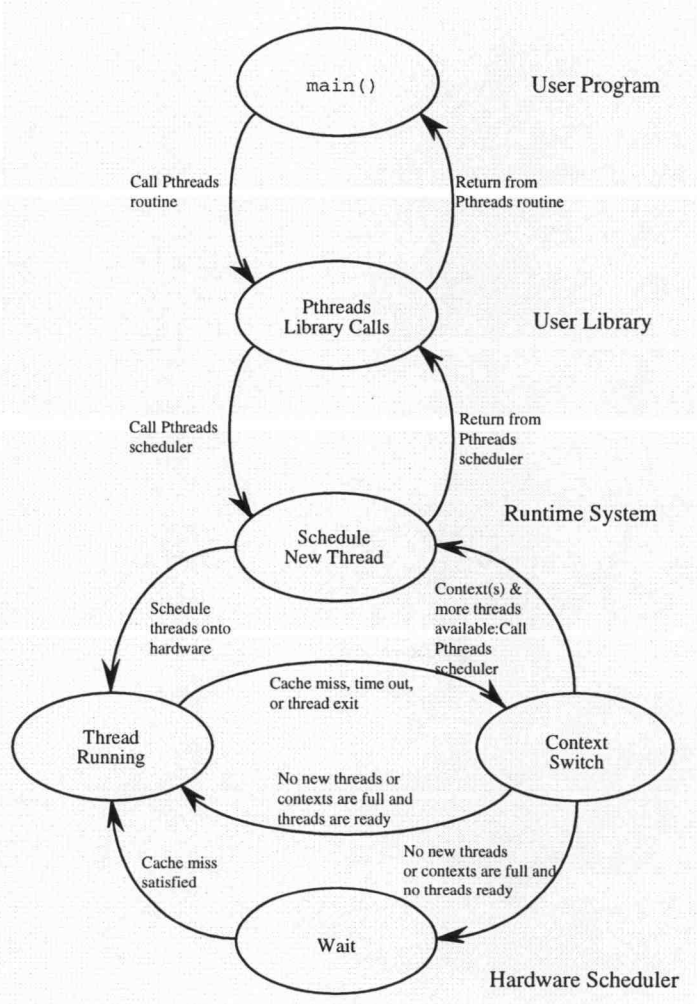


Figure 4.8: The MVP execution model.

- main() - MVP is executing the main user program. When a Pthreads routine is invoked, a state transition is made to Pthreads Library Calls.

- Pthreads Library Calls - Executes a Pthreads routine. After executing a Pthreads routine, it will either return to main() or call the Pthreads scheduler (Schedule New Thread state).
- Schedule New Thread - Depending on which Pthreads routine called the Pthreads scheduler, it will perform either one of the following operations:
  - Checks to see if a thread to be scheduled from the priority queue (PQ), which is maintained by the Pthreads scheduler, has a higher priority than the currently running thread. If so, the thread in the PQ is scheduled onto a hardware context and the V-bit is set; otherwise, returns from the Pthreads scheduler.
  - Selects a thread from the PQ and schedules it to an available hardware context, and the associated V-bit is set.
- Thread Running - Runs a thread that is in a hardware context. A transition to the Context Switch state occurs when a cache miss (R-bit is reset) occurs or a thread exits (V-bit is reset).
- Context Switch - Depending on the state of the machine, it will perform one of the following operations:
  - Context-switch to one of the ready threads in MVP (i.e., Thread Running state).
  - Call the Pthreads scheduler if a hardware context is available and threads are waiting to be scheduled in the PQ (i.e., Schedule New Thread state).
  - Wait if hardware contexts are full but no threads are ready, and no threads are waiting to be schedule from the PQ.

- Wait - Wait for a thread to become runnable (waiting for a long latency memory operation). When a thread becomes runnable (i.e., its cache miss has been satisfied and R-bit is set), a transition is made to Thread Running state.

The MVP maintains a total of 64 regular registers (32 integer and 32 FP) and 3 special registers for each context, and Table 4.3 shows their definitions. All the instructions are 64-bit wide, and the ISA of MVP is derived from the MIPS-IV [43] with the following exceptions: (1) Load, store, and branch instructions do not execute the succeeding instruction, i.e., no delay slot (2) Two additional addressing modes are supported: Indexed Register plus Register and auto-increment/auto-decrement. (3) Single- and double-precision FP square root instructions are introduced. The instructions of MVP can be classified into *register*, *immediate*, and *jump* instruction formats in Figure 4.9. The register format is mainly used for computations instructions. The immediate format supports up to 16-bit values, and the jump format supports the specification of 24-bit branch targets.

Table 4.3: Description of registers in MVP.

Hardware Name	Description
$0	Hardwired to zero
$1	Reserved by assembler
$2-$3	Function return result registers
$4-$7	Function argument value registers
$8-$15	Temp registers, caller saved
$16-$23	Saved registers, callee saved
$24-$25	Temp registers, caller saved
$26-$27	Reserved by OS
$28	Global pointer
$29	Stack pointer
$30	Saved registers, callee saved
$31	Return address register
$hi	High result register
$lo	Low result register
$f0-$f31	FP registers
$fcc	FP conditional register

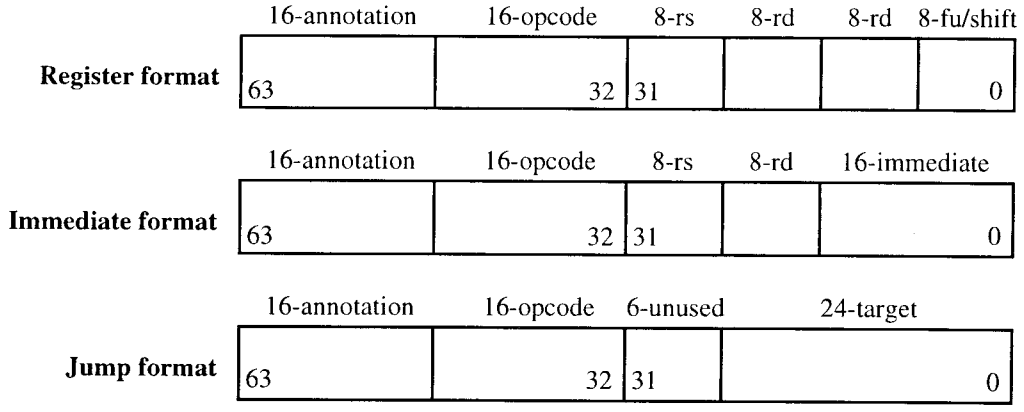


Figure 4.9: Register definition and Instruction format of MVP.

4.2 MVPsim

In order to verify the concept of MVP, we developed a functional simulator, called MVPsim, that integrates a POSIX compliant Pthreads software package and SimpleScalar with support for multithreading [14]. The simulator used for MVPsim is sim-outorder of SimpleScalar, which simulates an *n*-way issue superscalar processor based upon Sohi's Register Update Unit (RUU) [81]. The conventional Reservation Stations and ROB are combined and implemented as RUU. The default number of instructions fetched and decoded is four. Both the fetch and decode bandwidths and the number of entries in the RUU can be varied according to needs of the user. The Execution Unit consists of four integer ALUs, one integer multiply and divide, two load and store, four floating-point adders, and one floating-point multiply and divide, but again the Execution Unit requirements and its latencies can be easily modified.

sim-outorder also has an array of options for setting up branch prediction, caches, and main memory. For branch prediction, a user can specify the number of entries for BTB, where each entry uses 2-bit BPB, as well as a branch prediction penalty. For caches, both L1 and L2 cache parameters can be defined according to the required

number of sets, block size, associativity, and replacement policy. Finally, a user can also specify L1, L2, and main memory latencies. The basic simulator has the following architectural parameters:

- The number of instructions fetched, decoded, and dispatched is 4. The number of entries in the Reservation Stations and ROB were each assumed to be 32.
- Functional unit latencies were based on Table 4.4.
- Cache and main memory organizations and their latencies were based on Table 4.5. We assumed a two-level cache with writeback policy. The main memory latency used in the simulation is rather conservative compared to the current technology, e.g., UltraSparc IIi has a main memory latency of 72 cycles [67]. However, we expect the memory latency to grow in the future. Moreover, for multiprocessor systems, the shared-bus between processors' lower level cache and the main memory adds to the latency [17].
- Context switching to a new thread is initiated when an L2 cache miss is detected. L1 cache misses were not supported since the latency is assumed to 6 cycles and therefore not worth context switching to a new thread. However, a context-switch can be initiated at any level of the memory hierarchy as long as sufficient latency exists.
- The process of switching from one hardware context to another involves (a) simply turning off one register bank and turning on another register bank, (b) flushing the ROB, and (c) fetching from the new context. Assuming this is supported entirely in hardware, this process is very similar to recovering from a miss-predicted branch and requires a penalty of 3 cycles.
- The branch prediction scheme used is a 2K-entry Branch Target Buffer (BTB) with 2-bit branch prediction bits.

Table 4.4: Simulated FUs and their latencies.

Functional Unit	latency	Pipelined
Integer Unit	1	yes
Integer Multiply	3	yes
Integer Division	12	no
Load/Store Unit	2	yes
FP Add	2	yes
FP Multiply	4	yes
FP Division	12	no

Table 4.5: Configuration of L1 and L2 caches.

Cache Memory	L1 I-Cache/L1-D cache	L2 Unified Cache
Size	16 KBytes	256/512 KBytes
Associativity	Direct-mapped	4-way Set Assoc.
Line Size	32 Bytes	64/128 Bytes
Hit Latency	1	6
Miss Latency	6	100

4.3 Benchmark Programs

Five benchmark programs were developed to study the performance of MVP. Matrix Multiplication (MMT) and Gaussian Elimination (GE) programs were manually written to be multithreaded using Pthreads library calls. Other benchmark programs, Fast Fourier Transformation (FFT), MP3D, and Radix Sort (RS) were ported from SPLASH-2 suit [93]. The SPLASH-2 benchmarks used were originally written for shared-memory machines and ANL macros were used to create and manage threads. To port the SPLASH-2 benchmarks to the simulator, the ANL macros were replaced with their Pthreads equivalents. Also, no optimizations were attempted when converting the

serial versions to the multithreaded versions in all the benchmarks. Each benchmark is briefly described below:

- *MMT* parses the matrix data into blocks and assigns them to threads. The data set for the threads is relatively disjoint, but the row by column operation does produce considerable overlapping of data among threads. Moreover, there is no thread intercommunication or synchronization.
- *GE* partitions an *n*-by-*n* matrix into threads by using the row-wise block-cyclic approach. Initially one thread performs the division step with its pivot value and then all other threads perform an elimination step. These two steps are coordinated with barriers. GE threads tend to have very separate and distinct data sets with minimal data sharing besides the pivot value. GE is very similar to LU decomposition in SPLASH-2, except only the upper triangular matrix is generated.
- *FFT* implements a complex 1-D version of the $\sqrt{n}$ six-step FFT algorithm. The algorithm has also been optimized to minimize inter-thread communication. The data set consists of n complex data points and another n complex data points called the roots of unity. Every thread is then responsible for transposing a contiguous submatrix $\sqrt{n}/p \times \sqrt{n}/p$ with every other thread and one submatrix by itself. The data sets between threads are very localized even though the program is optimized against it, thus supports much data sharing between threads.
- *MP3D* is a simple simulator for rarefied gas flow over an object in a wind tunnel. The geometry of the object is created as a data structure in memory at initialization time, thus no initial file read operation is experienced. The algorithm is primary occupied within a loop consisting of three phases where a thread is given particles and proceeds to move them through a single time step. The thread continuously detects any possible collisions of its molecules with other molecules within a defined cell space. In essence, MP3D algorithm contains

data that is very localized and shares much of that data among threads. Also, each phase has to be completed by all the threads before continuing to the next phase, thus the algorithm requires a large amount of synchronization.

- *RS* algorithm passes over its assigned key values, and based on those key values, generates a local histogram. Then, all the local histograms are combined to into a globally shared histogram. Finally, each thread iterates over its assigned array and, by using the global histogram, permutes its keys into a new sorted array. Whenever the global histogram is accessed, the data set is very localized and shared; however, all other data accesses are very disjoint.

4.4 Simulation Results

Two sets of simulation runs were performed for each benchmark described in the previous section. The first set was obtained by running serial versions of the benchmark, and the second set was obtained by running multithreaded versions on MVP with multiple hardware contexts. Approximately 500 million to 1.2 billion instructions were simulated with the number of memory references ranging from 88 million to 300 million. Simulated MVP had 2, 4, and 8 hardware contexts, and the number of threads created for each simulation run was the same as the number of hardware contexts.

First, we examined how much stalls incurred in the serial versions due to memory latency. Figure 4.10 shows the percentage of the execution time that the processor idles due to L2 cache misses. MMT experienced the most stalls among the benchmarks, ranging from 20% to 55% of the total execution time. GE also showed fairly high percentage, about 32%, of stalls. Both GE and MMT programs operate on matrices, thus require a large number of memory accesses. FFT incurred a little over 22% of stalls when the problem sizes were larger than 2^{16}, and RS suffered less than 6% of stalls for all cases. Interestingly, MP3D showed a proportional increase in the stalls as the num-

ber of molecules increased. Therefore, in general, about 20% to 35% of stalls were experienced by the serial versions.

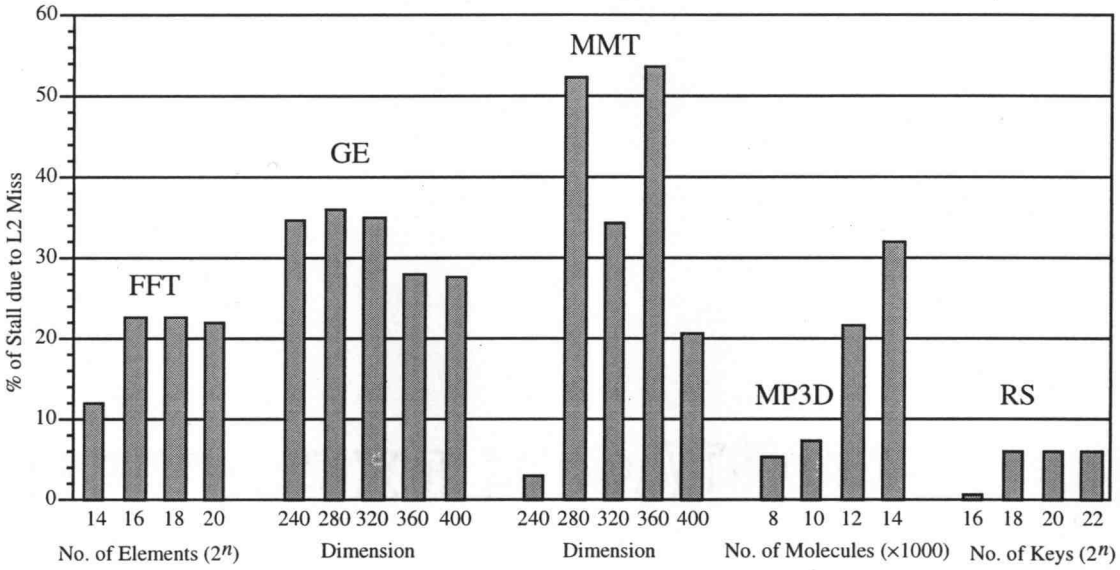


Figure 4.10: Percentage of latency incurred due to L2 cache misses.

Figure 4.11 shows the relative performance of MVP in terms of the execution cycles. The results were normalized relative to the performance of the serial versions. It can be seen that as the data sets become large, MVP begins to overcome its overhead and performs better than the serial cases. An example of this effect is well displayed by MP3D. FFT and GE also show a moderate performance improvement with increasing data size as their algorithms begin to take advantage of the latency tolerance of multi-threading. As expected, MMT showed the best performance improvement due to the fact that a large amount of stalls were incurred for the serial execution and no inter-thread synchronization was necessary among threads. On the other hand, RS showed the minimal performance improvements since the serial execution of RS experienced a small amount of memory latency.

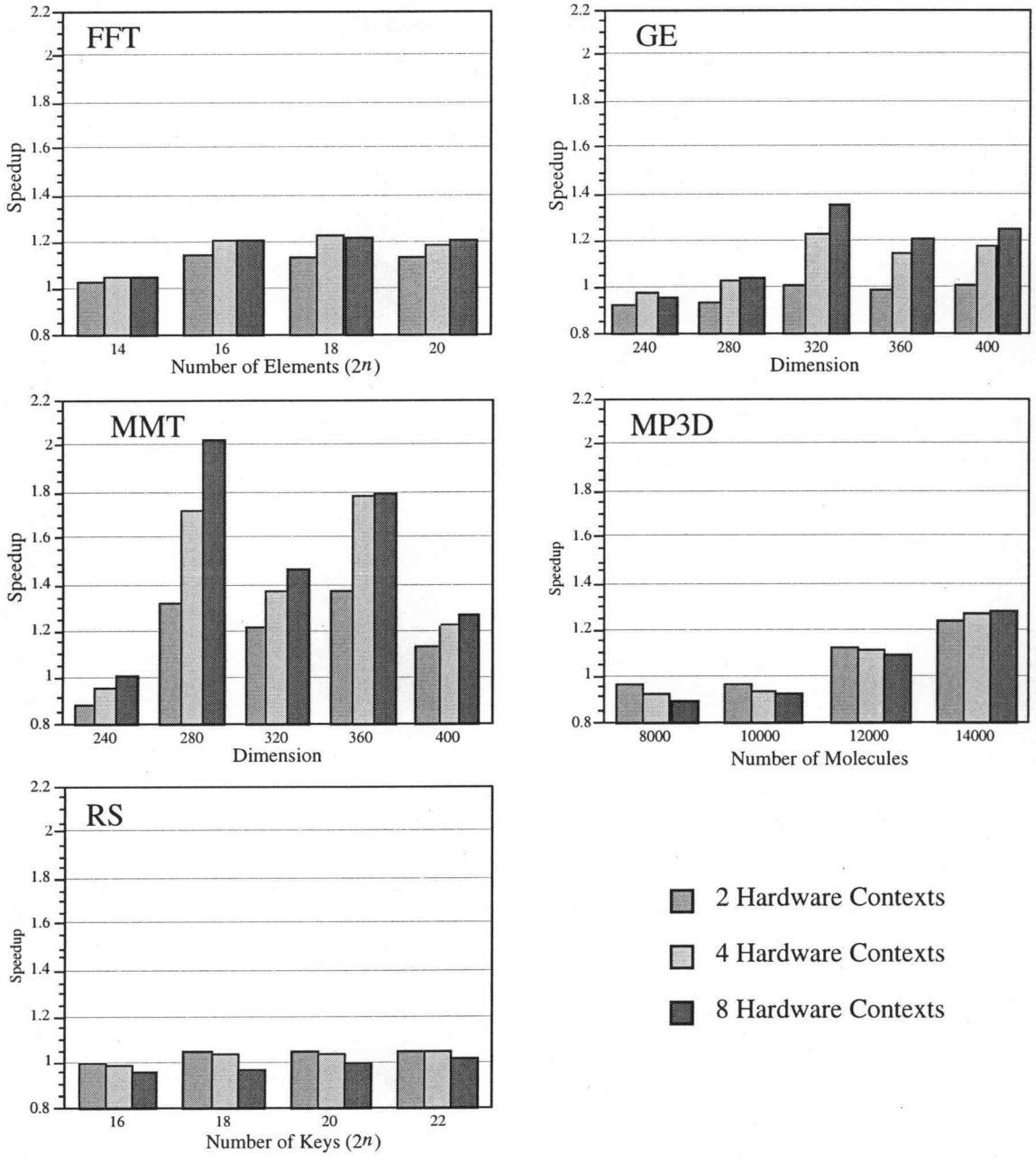


Figure 4.11: Speedup of five benchmarks over serial execution.

Another interesting effect is that the use of more hardware contexts does not necessarily result in improved performance, as seen in both RS and MP3D. This effect is the result of the benchmarks' high synchronization requirements and small parallel portion. Although the performance degrades as the number of contexts increases, the per-

formance margin narrows as the problem size increases. Therefore, it is likely that 4 and 8 hardware contexts would eventually outperform the 2-hardware contexts when the problem becomes large.

4.4.1 Locality in Caches

To gain a good understanding of how the L1 and L2 caches are affected by multithreading, the cache miss rates for the serial versions and MVP were examined. Figure 4.12 shows the miss rates for the L1 D-cache and the L2 cache—the cache miss rates for L1 I-cache were omitted since observed miss rates were below 0.5% for most benchmarks. The results were rather interesting. The L2 miss rates for MVP are lower than the serial versions for most of the cases. This effect is seen in all the benchmarks except GE and RS. Lower L2 miss rates are due to the fact that the data sets used by these programs have high data locality. This locality is exploited when a cache miss caused by a thread brings in the data that other threads may need later. For example, consider the MMT in Figure 4.13 with the rows of both A and B matrices are distributed in blocks. Suppose a cache miss occurs for a column value of B while Thread 1 is performing multiplication on a row of A matrix with a column of B matrix. This cache miss may fill the cache with the column values of B that will be used by Threads 2, 3, and 4 for the computation of matrix C. This data sharing occurs due to the fact that a cache line is essentially part of a row in the matrix. When a thread generates a cache miss while looking for a column value, other values needed by different threads are also retrieved at the same time. Therefore, when another thread looks for its column value, the column may already be in the cache (i.e., prefetching effect). Also, note that matrix B itself is shared among threads by the nature of the MMT algorithm.

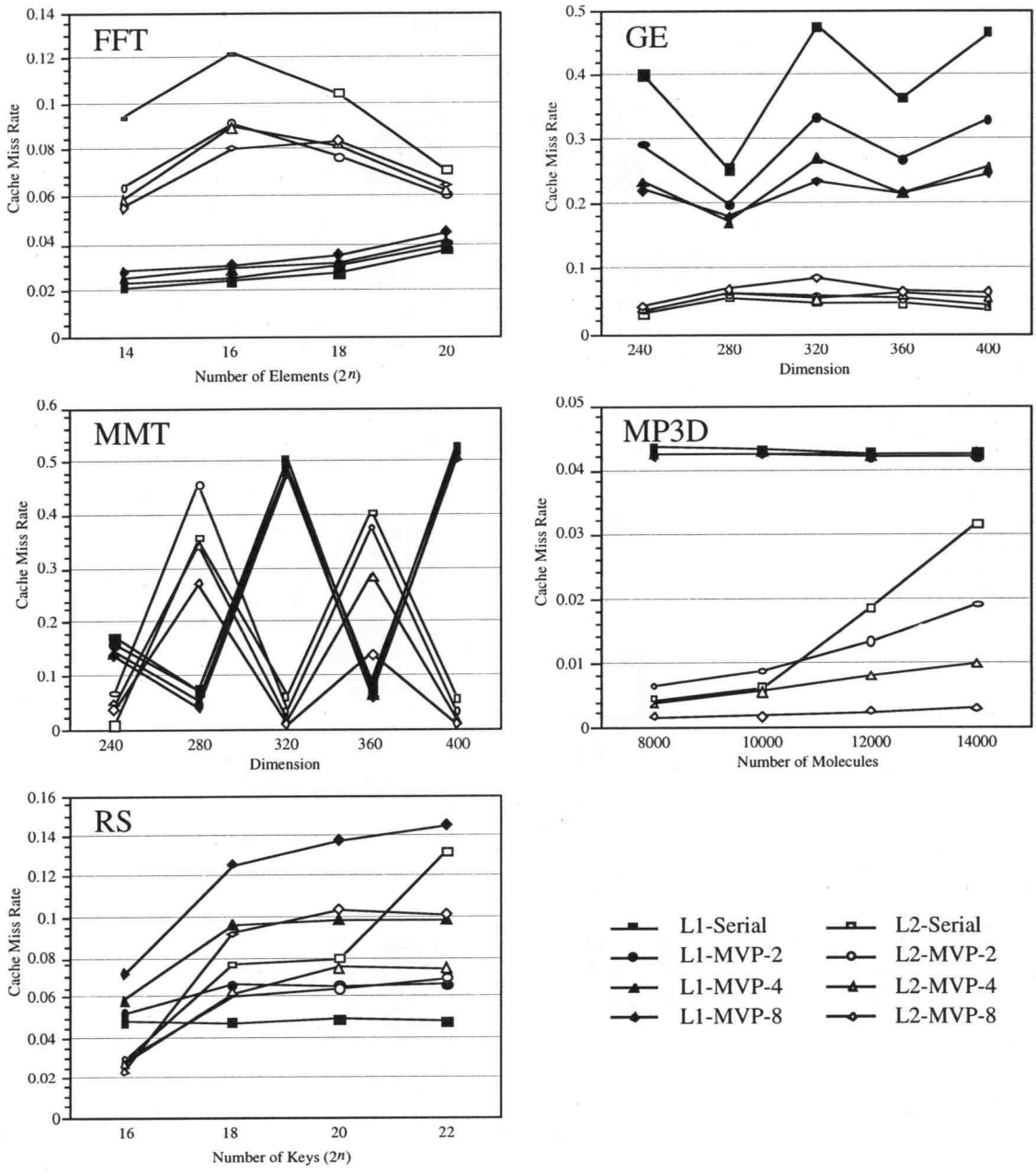


Figure 4.12: The miss rates for L1 D-cache and L2 cache.

Another benchmark that exhibits considerable data sharing is MP3D. In MP3D, the workload is partitioned by molecules, and a molecule is always moved by the thread it belongs to. Also, the partition of molecules changes significantly in each time step. Therefore, access pattern to the space array tends to exhibit low locality for both serial

and MVP versions. However, in the MVP version, it is quite possible for different threads to access the given space at the same time, and this creates the data sharing effect during collision computations. The result is lower cache miss rates for the MVP version.

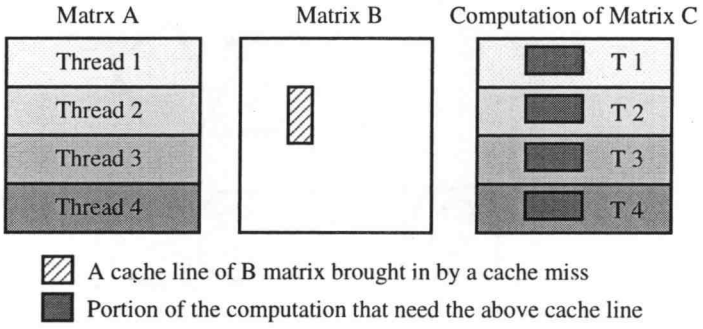


Figure 4.13: An example of data sharing in MMT.

The increased miss rates for both L1 and L2 caches are seen by RS. This effect is caused by the sorting portion of the RS algorithm. When the threads sort their individual keys of the array, the data set becomes very disjoint between threads, and the L2 cache miss rate increases. Similarly, GE also has very distinct data sets with low locality among threads. Therefore, a thread, upon generating a cache miss, would simply bring in more of the rows belonging to the same thread. The result is that the threads compete for space within the L2 and result in a higher L2 miss rate than the serial version.

Our simulation results thus far indicate the importance of the cache behavior, in particular the data sharing effects on the overall performance. However, the cache miss rates shown in Figure 4.12 alone do not accurately reflect the cache behavior since the total number of cache misses also depends on the number of accesses as well as the

cache miss rate. Therefore, we examined the cache behavior in more detail by observing the number of accesses to the cache.

Figure 4.14 monitors the effect of multithreading on L1 and L2 caches in terms of the number of accesses made. The graph shows the relative number of accesses made by the MVP versions using 4 hardware contexts to that of serial versions with 1.0 being the equal number of accesses for both versions (i.e., the number of accesses in the MVP execution divided by the number of accesses in the serial execution). The number of accesses to L1 I-cache for MVP increased 3% to 13% for all benchmarks. The increase in the number of accesses to the L1 I-cache was a result of the MVP versions having to execute more instructions for thread management and synchronization—software overhead involved in multithreading. Also, flushing of the instruction queue on context-switches increases the accesses to L1 I-cache.

The L1 D-cache also sees 1% to 32% increase in the number of accesses. This result signifies that the MVP versions tend to fetch more data than actually needed. There are two main reasons for this behavior. First, the speculative execution tends to fetch extra data which would not have been fetched if it were not for speculation. In other words, speculative execution has a more profound effect on the MVP versions since the multithreading effectively increases the overall amount of the speculative fetches by switching to a new thread and then to refetch the data later (i.e., thrashing). Second, the software overhead of thread management and synchronization inevitably increases the number of data accesses.

FFT and RS show that the MVP versions have a relatively large number of accesses to the L2 cache when compared with the serial versions (50% to 78% for FFT and 32% to 51% for RS). The result suggests that the exploitation of locality in the L1 D-cache is hindered. The conflict between threads causes more access to the L2 cache in order to retrieve the replaced lines in the L1 cache. However, GE and MMT show

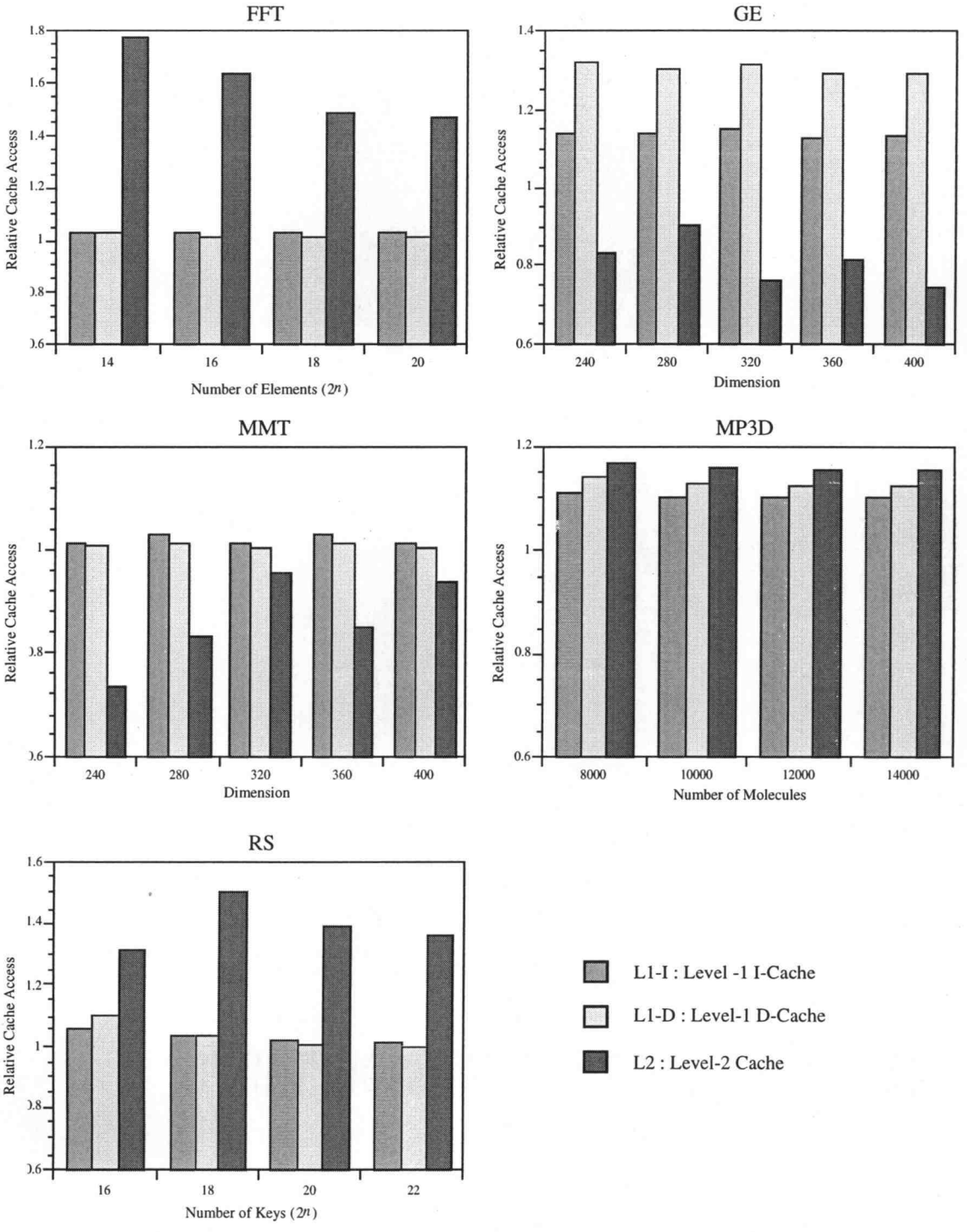


Figure 4.14: The relative cache accesses made by MVP.

less number of accesses to the L2 cache despite the increase in the number of accesses in the L1 caches. This indicates that some portions of the working sets are shared and this shared working sets contribute to the hits among threads, eliminating the need to access the L2 cache.

On the other hand, Figure 4.15 shows the actual number of cache misses in the L1 and the L2 caches for both the serial and the MVP versions. In general, the graphs show that MVP causes more misses in all the caches (GE and MMT are the exceptions). Also, it is apparent that the MVP versions generate a considerable amount of L1 I-cache misses compared to the serial version. This reflects the fact that the dynamic execution that results from context-switching hinders the locality in the I-cache [2]. Also, additional instruction executions for thread scheduling, synchronization, and thread management further contribute to the instruction misses in MVP. For example, both FFT and MP3D have a large amount barrier synchronization, and it is likely that each thread will work on different parts of the program and this tendency will increase as the data size becomes large. This increases the chance of the instruction being replaced when the context-switching occurs.

A counter example is MMT which caused the least amount of additional L1 I-cache misses (about 70% compared to the serial version). Since MMT requires no synchronization, each thread is likely to execute the same instructions with different data sets. Therefore, the instructions tend to remain and are recycled in the L1 I-cache without causing much conflict. Furthermore, MMT program mainly consists of multiplication/addition instructions that have to be repeated many times during the execution, and thus the instructions tend to exhibit strong locality.

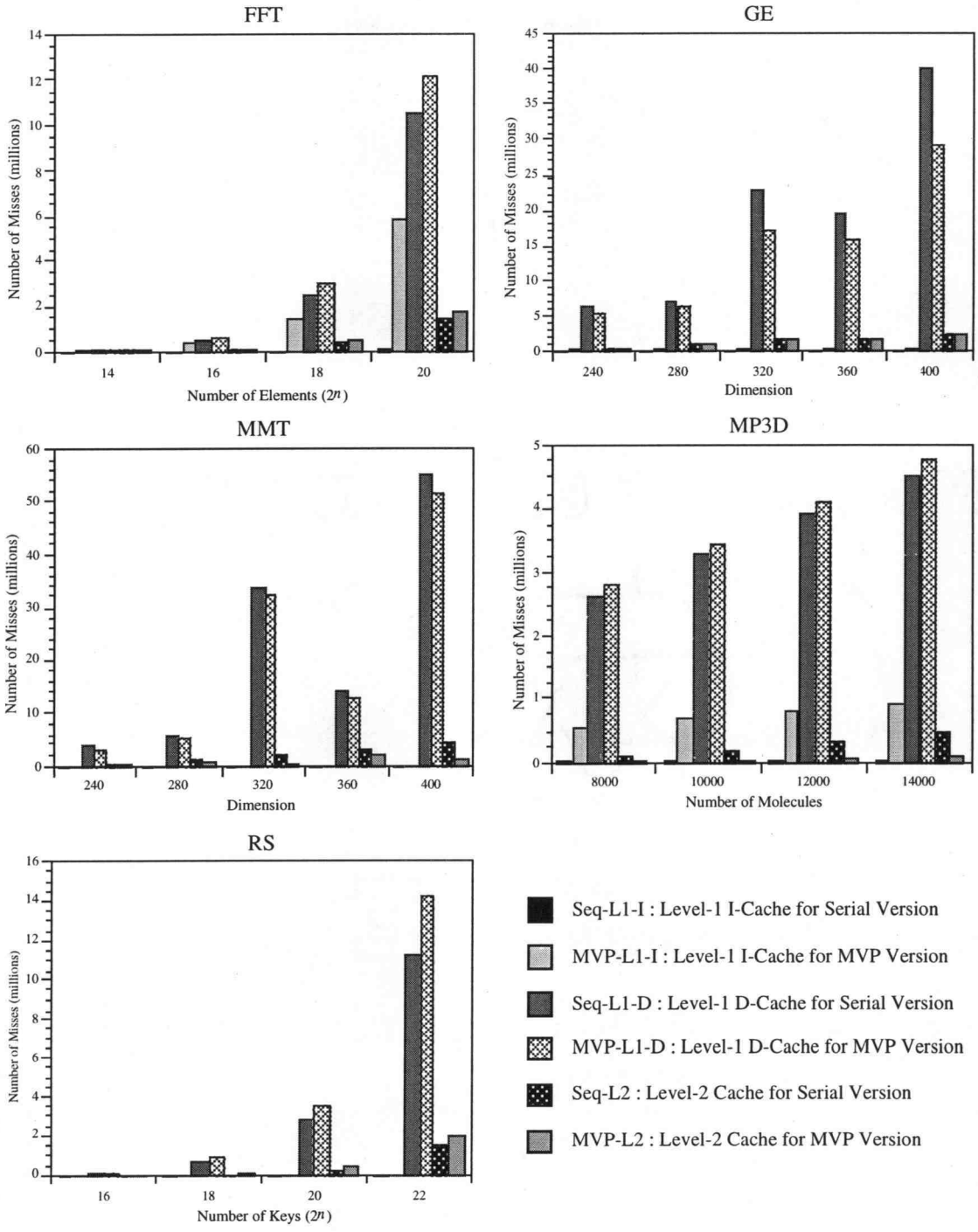


Figure 4.15: The number of misses in L1 and L2 caches.

4.4.2 An Analysis of Multithreaded Execution

In order to provide better understanding of how cache behavior and other components of multithreading contribute to the overall performance, we developed a simple analytical model that compares the serial and the multithreaded execution models. First, consider the serial case. Suppose the processor executes instructions on the average for R_s cycles before a cache miss occurs. Let L denote the average cache miss penalty, which represents the main memory access time and the time to refill the cache line. Assuming M_s number of cache misses occurs during the program execution, the total execution time for serial model, T_s, is given by

$$T_s = (R_s + L)M_s. \tag{4.1}$$

The performance of multithreading depends on two major factors: available parallelism and processor resources. If an application does not exhibit sufficient amount of parallelism (i.e., thread parallelism) for multithreading, the processor utilization will not increase. Even if parallelism exists, the sharing of processor resources (e.g., caches, functional units, etc.) among threads, the context-switching costs, and the overhead of thread management and scheduling may limit the overall performance. For simplicity and convenience, assume that enough parallelism exists for multithreading throughout the program execution. Suppose a multithreaded processor performs a context-switch at an interval of R_{mt} cycles at a cost of C cycles. If the processor performs M_{mt} context switches during its execution, and the overhead of O cycles is spent on thread management and scheduling, the execution time of the multithreaded model, T_{mt}, is given as

$$T_{mt} = O + (R_{mt} + C)M_{mt}. \tag{4.2}$$

Equation 4.2 reflects the ideal case when the cache miss latency is completely masked by the execution of other computational threads. If there are H numbers of hardware contexts, the processor can then execute $H-1$ threads to tolerate the memory latency (i.e., $R_{mt}(H-1) \geq L$). Otherwise, the performance will suffer from the unmasked portion of cache miss latency. In this case, Equation 4.2 can be rewritten as

$$T_{mt} = O + \big((R_{mt} + C) + (L - R_{mt}(H-1))\big)M_{mt}. \tag{4.3}$$

The term $(L - R_{mt}(H-1))M_{mt}$ in the above equation represents the total amount of memory latency that could not be tolerated despite executing other threads. Therefore, the effective improvement of the multithreaded execution over the serial execution, T_{eff}, can be estimated by

$$\begin{aligned} T_{eff} &= T_s - T_m \\ &= L(M_s - M_{mt}) + (R_s M_s - R_{mt} M_{mt}) - O - M_{mt} C + \big(R_{mt}(H-1)\big)M_{mt} \\ &\approx \big(R_{mt}(H-1)\big)M_{mt} + L(M_s - M_{mt}) - O - M_{mt} C. \end{aligned} \tag{4.4}$$

An approximation for Equation 4 can be obtained by assuming that the amount of computation for a given algorithm is about the same for both serial and multithreaded executions (i.e., $R_s M_s \approx R_{mt} M_{mt}$). The first term, $(R_{mt}(H-1))M_{mt}$, represents how much of the memory latency can be tolerated by the multithreading or the amount of overlapping that occurs between memory accesses and computation (*Tolerance*). This term reflects the fact that the tolerance to memory latency increases as the number of hardware context increases. However, if the program does not have sufficient parallelism to fully utilize the hardware contexts, the tolerance will be limited by the available parallelism in the program rather than the hardware contexts.

The term $L(M_s - M_{mt})$, on the other hand, can be used as a measure to reflect the sharing effects of multithreading (*Sharing Eff*). If the multithreaded execution causes less cache misses compared to the serial execution, this term becomes positive, signifying that data locality exists and the threads share a certain portion of the data (i.e., working set) during the execution. Otherwise, the conflict among threads will increase the number of cache misses, thus becomes negative indicating less data locality. The last two terms O and $M_{mt}C$ represent the overhead of multithreading, where O reflects the cycles spent for thread management and scheduling (*Overhead*) and $M_{mt}C$ is the total amount of cycles required for hardware context-switches (*Switching Cost*).

The effects of the four components in Equation 4.4 were studied for MVP with 4 hardware contexts. Figure 4.16 shows the percentage of each component to the total execution time for each benchmark. Each component was computed using the parameter values obtained from the simulation results. The negative values indicate the amount of additional cycles incurred in MVP over serial execution, and positive values signify the amount of benefit obtained from multithreaded execution. Note that the cumulative effect of the four components reflects the performance improvement in MVP and is consistent with the results shown in Figure 4.11.

Figure 4.16 shows that the tolerance increases as the data size increases in all the benchmarks except MMT. Also, the hardware context switching costs of MVP seem to have a minimal effect on the overall execution time. MMT and MP3D have positive sharing effects indicating that the performance of multithreading can also benefit from data sharing if the algorithms can take advantage of the locality. A similar effect is observed for the 320 case in GE. The observed speedup was greater than other cases because the performance benefited not only from its latency tolerance, but also from the positive sharing effect due to the lower L1 miss rates (compared to the serial versions). Also, the performance benefit for GE comes mostly from the tolerance rather than the data sharing.

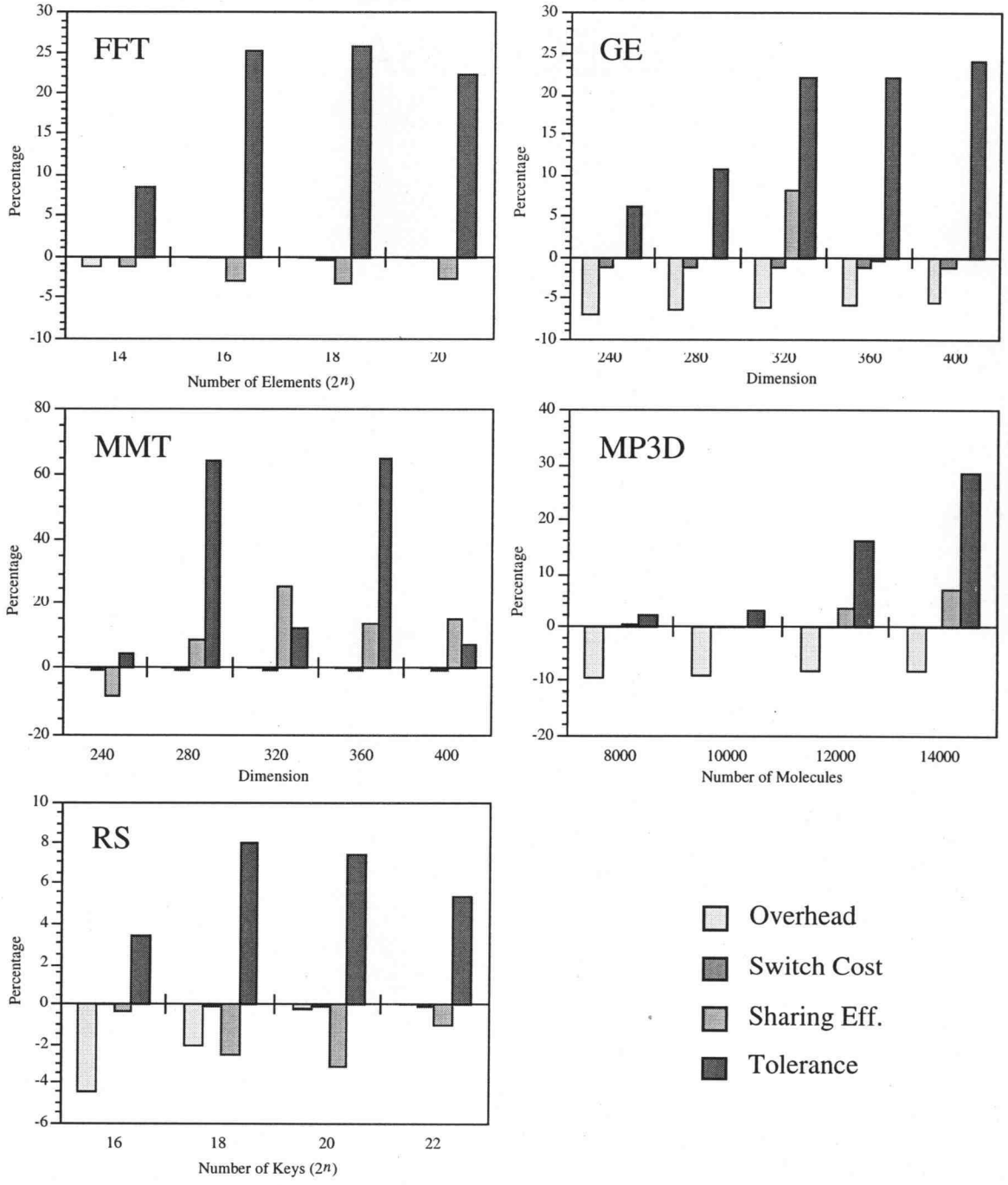


Figure 4.16: Breakdown of the various components of the MVP execution.

As the data size increases, MP3D exhibits a large growth in both its tolerance and the positive sharing effect. Therefore, the speedup increases almost linearly as the data size increases. On the contrary, RS shows an exponential growth for both the tolerance and the negative sharing effect as the data size increases. The result is that the memory latency resulting from additional cache misses offsets the tolerance and thus reduces amount of the effective improvement from multithreading. The graph for FFT is very similar to RS, but the smaller negative sharing effect and the larger tolerance resulted in better performance improvement. For both FFT and RS, the negative sharing effect was greater than the overhead or the switch cost. Therefore, the memory latency due to cache misses had a more significant influence on the performance of MVP for these two benchmarks, whereas the overhead costs had a greater influence on GE and MP3D. GE benchmark has barriers between the division and the elimination steps, and these operations had to be repeated many times before the computation is done, thus a large overhead was incurred for synchronization. MP3D also has a large amount of barrier synchronization and thus resulted in a relatively large overhead.

4.4.3 Pipeline Profile

Frequent context switching in MVP can disrupt the instruction execution and may cause new bottlenecks in the pipeline. Therefore, the average Instructions executed Per Cycle (IPC) of each benchmark were studied as shown in Figure 4.17. IPCs were calculated by dividing the total number of instructions executed by the total number of cycles. The portions of IPC lost were also analyzed from each of Fetch stage, Dispatch stage, and Issue stage. The graphs display the IPC that was lost from the ideal IPC. In other words, the lost IPC that is shown is simply ideal IPC minus the actual IPC. The graphs are further broken down to show what percentage of the total lost IPC was incurred at each of the stages. The pipeline stage bottlenecks that were modeled are: IPC

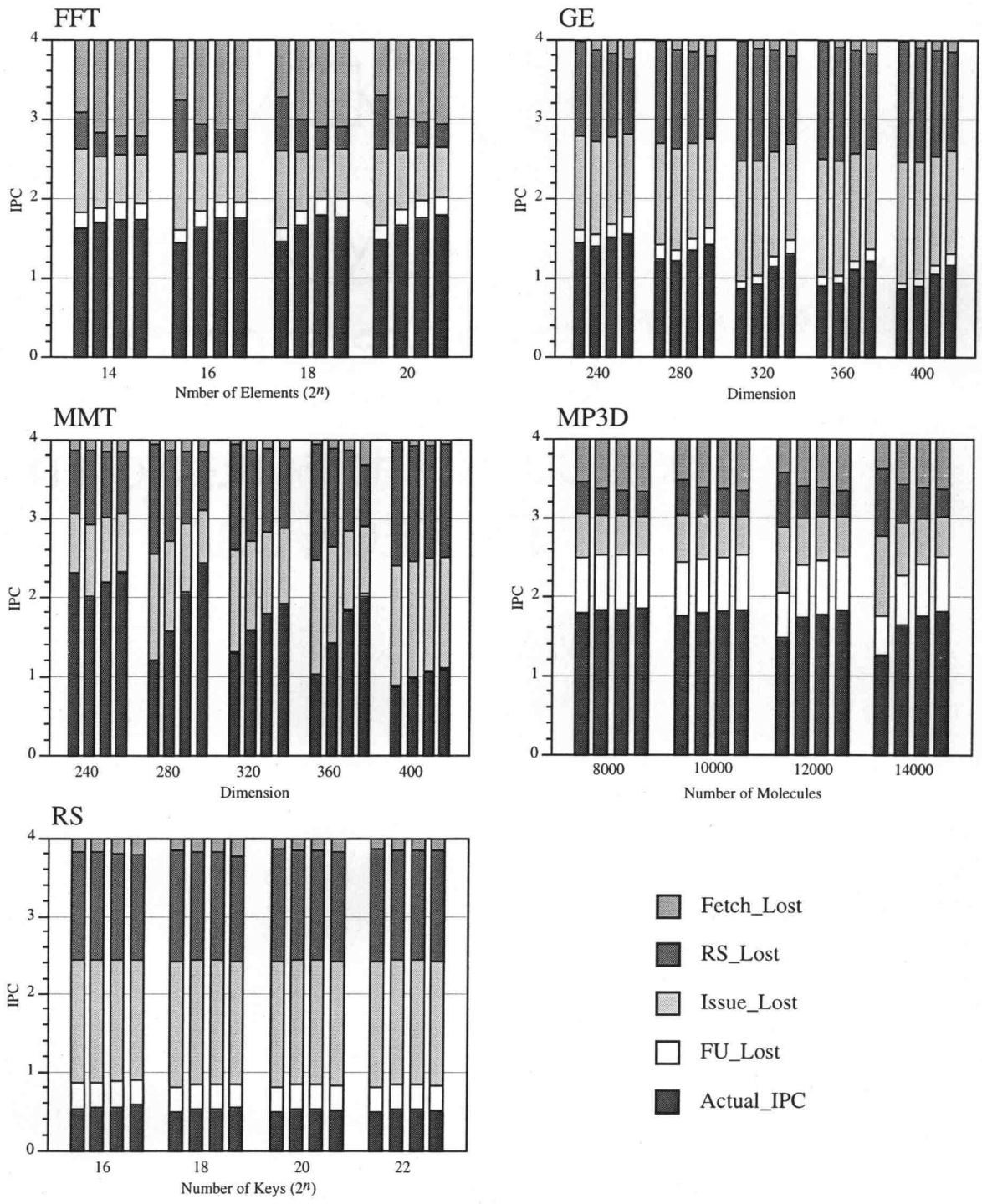


Figure 4.17: IPC for various benchmarks. Four graphs for each point represent, from left to right, serial, 2 HW contexts, 4 HW contexts, and 8 HW contexts.

lost due to fetch bandwidth (Fetch_Lost), reservation stations full (RS_Lost), execution units busy (FU_Lost), and issue bandwidth limitations (Issue_Lost). Decode and Commit stage bandwidths were also observed, but was dropped when it was apparent that bandwidths were never reached in any of the simulations executed.

As can be seen from the graphs, multithreading seems to create a additional stress on the fetch bandwidth. This is due to the fact that program locality is reduced by context switching among threads. Therefore, the fetch bandwidth will have to be improved in order to obtain better performance. Another effect that can be observed is a decrease in IPC lost in the issue stage. By switching threads on a cache miss, long latency data dependencies are avoided and consequently, more instructions are available to be issued. Also seen by the graphs is almost no change by RS and only small amounts of difference experienced by FFT and MP3D. The reason is the level of synchronization and therefore parallelism experienced by the data sets. RS has much synchronization while FFT and MP3D have some and MMT has none. GE exhibits a similar effect as MMT even though it also has high amounts of synchronization. This effect is due to the fact that GE has a very large parallel portion in comparison to the synchronized serial portions.

4.5 Conclusion and Future Research Directions

This paper discussed the simulation study of MVP on the cache performance. The MVP execution model showed 10% to 85% performance improvement over the serial execution model. MVP showed its effectiveness in tolerating memory latency due to L2 cache misses. Also, the performance improvement came from not only tolerating memory latency, but also exploiting data locality. For some benchmarks, the data locality was exploited in the form of sharing, further enhancing the overall performance improvement. On the other hand, when the sharing effect was minimal, it resulted in addi-

tional conflict misses thereby offsetting the advantage of multithreading. Results also showed that the software synchronization requirements among threads could degrade the performance of some of the simulated benchmarks.

Based on our study, there are two major ways MVP can be improved. First, the software synchronization overhead can be mitigated by providing the support in hardware [25]. This improvement will not only reduce the software overhead, but may also reduce the number of conflict misses due to reduced number of context switches. Second, the sharing effect is obviously important and therefore must be encouraged to fully benefit from multithreading. For example, Philbin *et al.* proposed a use of threads for improving the cache locality of serial programs by careful scheduling [34]. The address information associated with each thread is provided to the scheduler, and the threads are scheduled in the order that minimizes the cache misses. Therefore, future study may focus the thread scheduling as a means of improving data locality.

We are also currently working on modifying the MVP to support SMT with dynamic thread creation and speculative execution, called Dynamic Simultaneous Multithreading (DSMT). The idea is to detect and create threads from a serial program without compiler intervention. These dynamic threads will be executed concurrently on a SMT-like machine with special hardware support for register and memory dataflow and speculative execution. The benefits of DSMT are several-fold: (1) eliminates the need for programmers and compilers to generate multithreaded codes, (2) overcomes the technological limitations of arbitrarily increasing the instruction window size to achieve a wide-issue bandwidth, (3) speculative execution can be aggressively applied to across multiples threads, and (4) reduces the required fetch bandwidth by taking advantage of the fact that multiple threads share a common code.

BIBLIOGRAPHY

[1] Aditya, "Normalizing strategies for multithreaded interpretation and compilation of non-strict languages", MIT CSG Memo-374, May 1995.

[2] Agarwal A., "Limits on interconnection network performances," IEEE Transactions on Parallel and Distributed Systems, Vol. 2, Oct. 1991, pp. 398-412.

[3] Agarwal A., Horowitz M., and Hennessy J., "An analytical cache model," ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989, pp. 184-215.

[4] Agarwal, A. *et al.*, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *Proceedings of Workshop on Scalable Shared Memory Multiprocessor*, Kluwer Academic Publishers, 1991.

[5] Agarwal, A. *et al.*, "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 2-13.

[6] Agarwal, A., "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, Sept. 1992, pp. 525-539.

[7] Alkalaj, R. Bopanna. "Performance of multithreaded execution in shared memory multiprocessors", 3rd SPDP, pp. 330-333, Dec. 1991.

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

[9] Ang, B. S., Chiou, D., Rudolph, L., and Arvind, "Message Passing Support in StarT-Voyager," MIT Laboratory for Computer Science, CSG Memo 387, July 1996.

[10] Ang. B. S., Arvind, and Chiou D., "StartT the Next Generation: Integrating Global Caches and Dataflow Architecture," *Proceedings of the 19th international Symposium on Computer Architecture*, 1992, Dataflow workshop.

[11] Blumofe and C. Leiserson. "Scheduling multithreaded computation by work stealing", *Proc. of Foundations of Computer Science*, Nov. 1994, pp. 356-368.

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

[13] Boughton, G. A., "Arctic Routing Chip," Proceedings of the 1st International Workshop, PCRCW, vol. 853 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 310 - 317.

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

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

[16] Callahan, D. *et al.*, "Software Prefetching," *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40-52

[17] Catanzaro, B., Multiprocessor System Architectures, Prentice Hall, 1994.

[18] Chen, T. *et al*, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 223-232.

[19] Chen, W. *et al.*, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Nov. 1991, pp. 69-73.

[20] Chiou, D. *et al.*, "StatT-NG: Delivering Seamless Parallel Computing," *Proceedings of EURO-PAR*, 1995, Stockholm, Sweden.

[21] Coleman, S. *et al.*, "Tile Size Selection Using Cache Organization and Data Layout," *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

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

[23] Cooper, E. and Draves, R., C Threads, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Feb. 1988.

[24] Culler, D. and Singh, J. P., Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1999.

[25] Culler, D. *et al.* "TAM—A Compiler Controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing 18*, 1993, pp. 347-370.

[26] Culler, D. E., Schauser, K.E., and von Eicken, T., "Two Fundamental Limits on Dataflow Multiprocessing," *Proceedings of IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Jan. 1993, pp. 153-164.

[27] Culler, D.E., Sah, A. Schauser, K.E., von Eicken, T., Wawrzynek, J., "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," Fourth International Conferences on Architectural Support for Programming Languages and Operating Systems. April 1991, pp. 164-175.

[28] Eicken, T., *et al.*, "Active Messages: a Mechanism for Integrated Communication and Computation," *IEEE 19th Annual International Symposium on Computer Architecture*, 1992.

[29] Ferrari A., Sunderam V. S., "TPVM: Distributed Concurrent Computing with Lightweight Processes," from http://uvacs.cs.virginia.edu/~ajf2j/tpvm.html.

[30] Fillo, M., *et al.* "The M-Machine Multicomputer," *Proceedings of MICRO-28.*

[31] Fu, J. *et al.*, "Stride directed prefetching in scalar processors," *Proceedings of 25th Annual Conference on Microprogramming and Microarchitectures*, Dec. 1992.

[32] Gonzalez, A. *et al.*, "A data cache with multiple caching strategies tuned to different types of locality," *Proceedings of International Conference on Supercomputing*, July 1995, pp. 338-347.

[33] Gulati, M. and Bagherzadeh, N., "Performance Study of a Multithreaded Superscalar Microprocessor," *The 2nd International Symposium on High-Performance Computer Architecture*, Jan. 1995, pp. 298-307.

[34] Gunther, B., "Multithreading with Distributed Functional Units," *IEEE Transactions on Computers*, Vol. 46, No. 4, April 1997, pp. 399-411.

[35] Haines M., Bohm W., "An Evaluation of Software Multithreading in a Conventional Distributed Memory Multiprocessor," 1993, pp. 106-113.

[36] Halstead, "Multilisp: A language for concurrent symbolic computation", ACM *Transactions on Programming Languages and Systems*, Oct. 1985, pp. 501-538.

[37] Heinlein J., *et al.*, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct., 1994.

[38] Hennessy, J. and Patterson, D., Computer Architecture-A Quantitative Approach, 2nd Edition, Morgan Kaufmann.

[39] Hilly, S. and Seznec, A., "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading," *The 4th International Symposium on High Performance Computer Architecture*, MTEAC 98, Jan. 1998.

[40] Hwang, K., "Advanced Computer Architecture: Parallelism, Scalability, Programmability," McGraw Hill, Inc., 1933.

[41] Jacobson, Q. *et al.*, "Control Flow Speculation in Multiscalar Processors," The 3rd International Symposium on High-Performance Computer Architecture (HPCA-3), Feb. 1997.

[42] Johnson, T. *et al.*, "Runtime Spatial Locality Detection and Optimization," *Proceedings of Micro-30*, Dec. 1997.

[43] Kane, G. and Heinrich J., MIPS RISC Architecture, Prentice Hall.

[44] Kavi K., *et al.*, "Design of Cache Memories for Multi-Threaded Dataflow Architecture," *IEEE International Symposium on Computer Architecture*, 1995, pp. 253-264.

[45] Klaiber, A. and Levy, H., "A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs," *IEEE 21st Annual International Symposium on Computer Architecture*, 1994, pp. 94-105.

[46] Klaiber, A. and Levy, H., "An Architecture for Software-Controlled Data Prefetching," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 43-53.

[47] Kodama, Y., H. Sakane, H., Sato, M., H. Yamana, Sakai, S., Yamaguchi, Y., "The EM-X Parallel Computer: Architecture and Basic Performance," *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 14-23

[48] Krishnan, V. and Torrellas, J., "Executing Sequential Binaries on a Multithreaded Architecture with Speculation Support," *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (MTEAC '98), January 1998.

[49] Kumar, V. *et al.*, "Introduction to Parallel Computing," The Benjamine/Cummings Publishing Company, Inc. 1994.

[50] Kurpanek, G. *et al.*, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *Digest of Papers, Spring COMPCON 94*, 1994, pp. 375-382.

[51] Kuskin J., *et al.*, "The Stanford FLASH Muitiprocessor," *Proceedings of the 21st International Symposium on Computer Architecture*, 1994, pp. 302-313.

[52] Lam, M. *et al.*, "The Cache Performance and Optimizations of Blocked Algorithms," *Proceedings of the 4th International Conference on Architectural Support for Programming Language and Operating Systems*, April 1991, pp. 63-74.

[53] Lee B., Kavi K., "Program Partitioning for Multithreaded Dataflow Computers."

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

[55] Lee, B. and Hurson, A. R., "Issues in Dataflow Computing," *Advances in Computers*, Vol. 37, 1993, pp. 285-333.

[56] Lee, B., "An Analysis of Data Distribution Methods for Gaussian Elimination in Distributed-Memory Multicomputers," IEEE International Symposium on Parallel and Distributed Processing, 1994, pp. 152-159.

[57] Lo, J. *et al.*, "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Transactions on Computer Systems*, Aug. 1997, pp. 322-354.

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

[59] 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.

[60] Milutinovic, M. *et al.*, "The Split Temporal/Spatial Cache: Initial Performance Analysis," *Proceedings of the SCIzzL-5*, Mar. 1996.

[61] Mowry, T. *et al.*, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the 5th International Conference on Architectural Support for Programming Language and Operating Systems*, Oct. 1992, pp. 62-73.

[62] Mueller, F., "A Library Implementation of POSIX Threads under UNIX," *1993 Winter USENIX*, Jan. 1993, pp. 29-42.

[63] Najjar W. A., *et al.*, "A Quantitative Analysis of Dataflow Program Execution-Preliminaries to a Hybrid Design," *Journal of Parallel and Distributed Computing*, 1994, PP. 314-326.

[64] Nikhil, R. S., "Cid: A Parallel Shared-memory C for Distributed-memory Machines," *Proceeding of 7th Ann. Workshop on Language and Compilers for Parallel Computing*, Aug. 1994, pp 376-390.

[65] Nikhil, R. S., "Id World Reference Manual," MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 1985.

[66] Nikhil, R. S., Papadopoulos, G. M., and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

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

[68] Olukotun, K. *et al.*, "The Case for a Single-Chip Multiprocessor," *Architectural Support for Programming Languages and Operating Systems*, 1996.

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

[70] Palacharla, S. and Kessler, R., "Evaluating Stream Buffers as a Second Cache Replacement," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 24-33.

[71] Papadopoulos, G. M. and Culler, D. E., "Monsoon: An Explicit Token Store Architecture," *IEEE 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 82-91.

[72] Papadopoulos, G. M. and Traub, K. R., "Multithreading: A Revisionist view of Dataflow Architectures," *IEEE 18th Annual International Symposium on Computer Architecture*, 1991, pp. 342-351.

[73] Papadopoulos, G. M., *et al.*, "*T: Integrated Building Blocks for Parallel Computing," ACM, 1993, pp. 624-635.

[74] Philbin, J. *et al.*, "Thread Scheduling for Cache Locality," *Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 60-71.

[75] Rinker, R. *et al.*, "Evaluation of Cache Assisted Multithreaded Architecture," *The 4th International Symposium on High Performance Computer Architecture*, MTEAC 98, Jan. 1998.

[76] 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.

[77] Sakane H., *et al.*, "Dynamic Characteristics of Multithreaded Execution in the EM-X Multiprocessor," *Proceedings of 1995 International Workshop on Computer Performance Measurement and Analysis* (PERMEAN '95), Beppu Ohita Japan, pp. 14-22.

[78] Sato, M. *et al.*, "Thread-Based Programming for EM4 Hybrid Dataflow Machine," *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[79] Schauser, K.E., Culler, D.E., von Eicken, T., "Compiler-Controlled Multithreading for Lenient Parallel Languages," Proceedings of FPCA '91 Conference on Functional Programming languages and Computer Architecture, Aug. 1991, pp. 50-72.

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

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

[82] Sohi, G. *et al.*, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proceedings of ASPLOS-IV*, April 1991, pp. 53-62.

[83] Sohn, A., Kim, C., and Sato, M., "Multithreading with the EM-4 Distributed-Memory Multiprocessor," Technical Report NJIT-CIS-31-94.

[84] Sunderam V. S., *et al.*, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends," can be obtained from http://www.ornl.gov:80/pvm.

[85] Thekkath, R., and Eggers, S. J., "Impact of Sharing-Based Thread Placement on Multithreaded Architectures," *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[86] 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.

[87] Theobald, K. B., "Panel Session of the 1991 Workshop on Multithreaded Computers," ACAPS Technical Memo 30, April 1, 1993. McGill University.

[88] Tsai, J., *et al.*, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1996.

[89] Tsai, Z. Jiang, E. Ness, and P. C. Yew "Performance Study of a Concurrent Multithreaded Processor," Technical Report #97-034, Dept. of Computer Science, Univ. of Minnesota, July 1997.

[90] Tucker, L.W., and Mainwaring, A., "CMMD: Active messages on the CM-5," *Parallel Computing* 20, 1994, pp. 481-496.

[91] Tullsen, D. M, *et al.*, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.

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

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