AN ABSTRACT OF THE THESIS OF

Ho Seok An for the degree of Master of Science in Electrical and Computer Engineering presented on December 15, 1999. Title: The Multithread Virtual Processor on DSM.

Redacted for Privacy

Abstract approved: _____

Il-Beom Lee

Modern superscalar processors exploit instruction-level parallelism (ILP) by issuing multiple instructions in a single cycle because of increasing demand for higher performance in computing. However, stalls due to cache misses severely degrade the performance by disturbing the exploitation of ILP. Multiprocessors also greatly exacerbate the memory latency problem. In SMPs, contention due to the shared bus located between the processors's L2 cache and the shared memory adds additional delay to the memory latency. In distributed shared memory (DSM) systems, the memory latency problem becomes even more severe because a miss on the local memory requires access to remote memory. This limits the performance because the processor can not spend its time on useful work until the reply from the remote memory is received.

There are a number of techniques that effectively reduce the memory latency. Multithreadings has emerged as one of the most promising and exciting techniques to tolerate memory latency. This thesis aims to realize a simulator that supports software-controlled multithreadings environment on a Distributed Shared Memory and to show preliminary simulation results.

Multithreaded Virtual Processor on DSM

by

Ho Seok An

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 15, 1999
Commencement June 2000

Master of Science thesis of Ho Seok An presented on December 15, 1999

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

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

Redacted for Privacy

Ho Seok An, Author

ACKNOWLEDGMENT

Once again I am indebted to my family for their love, support, and many lost weekends over the past four years. The thesis would have never been completed without their unconditional support.

I am especially thankful to Prof. Il-Beom Lee, my major professor. His guidance and advice have been valuable during my studies at Oregon State University. Also, special thanks to the committee members, Prof. Alexandre F. Tenca, Prof. Kyoung Rok Cho, and Prof. Robert M. Burton. I am also grateful to Dr. Han Tak Kwak for answering many questions about the details of the Pthread Package.

Other friends have helped in many small, but significant ways over the past few years: Yeon Chang Hahm, Chi Young Lim, and Jong Hun Yoon have been great friends through it all. A special thanks to the RSIM people, especially Prof. Sarita Adve in Computer Science at the University of Illinois at Urbana-Champaign.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

MULTITHREADED VIRTUAL PROCESSOR ON DSM

1 INTRODUCTION

1.1 Objectives and Scope of the Thesis

Semiconductor technology advances are giving us increasingly powerful microprocessors, larger memory, large-capacity disks, and faster I/O. But our computational appetite is always growing and therefore the next step is to use parallel architectures. Today, symmetric multiprocessing (SMP) is a commonplace. The NT machines typically ship with 2 to 4 processor per systems, while UNIX systems with even 128 and 256 processors will be common soon.

Lately, multiprocessors such as Cache-Coherent Nonuniform Memory Architectures (CC-NUMA) have become popular. CC-NUMA is a hybrid that retains SMP's shared-memory programming model, while larger systems can be built from SMP nodes. This feat is performed by a special hardware that connects all the node's memory subsystems and acts as a transparent cache. To the processor, all memory is the same, except its local memory's access time is faster than some other node's memory (hence "nonuniform memory"). High-speed interconnect cuts the latency time to access the remote memory.

In CC-NUMA, fetching data from the memory of a remote node involves significant latency. However, a number of latency reducing, or latency-tolerating techniques can decrease the time the processor must stall during the round-trip. There are three techniques for reducing or tolerating latency: *Prefetching* [24], [25], *Relaxed Memory Consistency* [26], and *Multithreading* [27]. Among them, *Multithreading* is a

technique that has emerged the one of the most promising method to tolerate the increasing memory latency.

MVPsim [16] based on SimpleScalar [30] has been built and simulated to see how hardware-controlled multithreading is effective on tolerating memory latency. However, the multithreading effectiveness has not been simulated with multiprocessor environment. The goal of the thesis is to realize a simulator that supports software-controlled *multithreadings* environment (using a POSIX compliant Pthreads package) on a Distributed Shared Memory (i.e., CC-NUMA) for the first step. The thesis includes what has been implemented for supporting the Pthreads package on a CC-NUMA and its preliminary simulated results.

RSIM (Rice Simulator for ILP Multiprocessors) has been chosen as a base simulator for supporting the Pthreads package. In developing a simulator for a new processor architecture, it is often not clear whether it is more efficient to write a new simulator or to modify an existing simulator. Writing a new simulator forces the processor architect to develop all of the related software tools. On the other hand, modifying an existing simulator and related tools, which are usually not well-documented, can be time-consuming and error-prone. RSIM is comparatively well-documented with the modest complexity compared to other simulators, such as SimOS [15]. Also RSIM contains machine components capable of simulating the hardware of CC-NUMA with sufficient speed and detail.

1.2 Outline of the Thesis

Chapter 2 outlines the reasons why parallel machine design may become pervasive and why an accurate multiprocessor simulator is needed. It also examines the application and technology trends that have led to the current state of computer

architecture. Chapter 3 digs deeper into the detailed features of RSIM. The concept and the structure of RSIM are introduced here. Chapter 4 presents how the Pthreads package was ported to RSIM. Discussion of the preliminary simulation results of Matrix Multiplication benchmark is presented in Chapter 5. Finally, Chapter 6 presents a brief conclusion and future work.

2 BACKGROUND

This chapter discusses the advantages of Distributed Shared Memory (DSM) and *Multithreading* in a DSM environment. It also discusses why a multiprocessor simulator is needed and how RSIM was chosen.

2.1 Why Do We Need a Multiprocessor Simulator?

Computer architecture, technology, and applications evolve together and are highly related. Parallel computer architecture is no exception. Researchers believe that parallel machines will definitely have a bigger role in the future. This view is driven by two observations. First, the simplest way to improve performance beyond a single processor is by connecting multiple microprocessors together, especially since the performance of the microprocessor is surpassing the alternatives (see Figure 2.1).

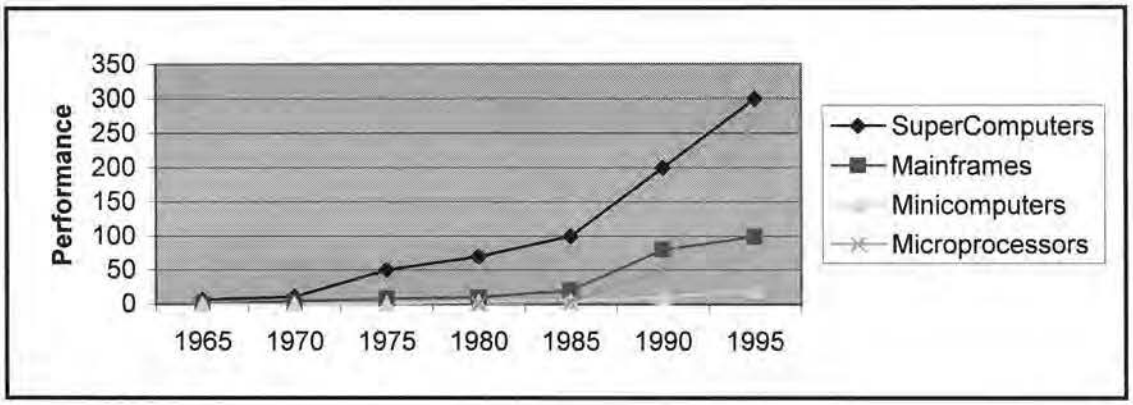


Figure 2.1: Performance trends by Hennessy and Jouppi (1991).

Second, although dramatic changes have been the norm in computer architecture, it is unclear whether the pace of architectural innovation that has contributed to the rapid rate of performance growth starting in 1985 can be sustained indefinitely.

For these reasons, we need to have a simulator for parallel architectures (i.e., multiprocessors). The remainder of this section examines the application demand for increased performance and then the underlying technological trends that strive to meet these demands.

Parallel architectures have become the mainstay of scientific computing, including physics, chemistry, material science, biology, astronomy, earth sciences, and others. The engineering application of these tools for the modeling physical phenomena is now essential to many industries, including petroleum (reservoir modeling), automotive (crash simulation, drag analysis, combustion efficiency), aeronautics (airflow analysis, engine efficiency, structural mechanics, electromagnetism), pharmaceutical (molecular modeling), and others. In almost all of these applications, there is a large demand for visualization of the results, which is a demanding requirement amenable to parallel computing. The visualization component has brought the traditional areas of scientific and engineering computing closer to the entertainment industry. In 1995, the first full-length computer animated motion picture, *Toy Story*, was produced on a parallel computer system composed of hundreds of Sun workstations.

Commercial computing has also come to rely on parallel architectures for its high end. Although the scale of parallelism is typically not as large as in scientific computing, the use of parallelism is even more widespread. Multiprocessors have been the high end machines for the commercial computing market since the mid-1960s. The relationship between a performance and the scale of business enterprise is clearly articulated in the on-line transaction processing (OLTP) benchmarks,

sponsored by the Transaction Processing Performance Council (TPC). These benchmarks rate the performance of a system in terms of its throughput in transactions per minute (*tpm*) on a typical workload. Table 2.1 shows the tpm and the speedup for the Tandem Himalaya and IBM PowerPC systems.

Number of Processors	IBM RS56000 PowerPC		Himalaya K10000	
	Tpm	Speedup	Tpm	speedup
1	735	1		
4	1,438	1.96		
8	3,119	4.24		
16			3,043	1
32			6,067	1.99
64			12,021	3.95
112			20,918	6.87

Table 2.1: The *tpm* and speedup for the Tandem Himalaya and IBM PowerPC.

Several important observations can be drawn from the TPC data. First, the use of parallel architectures is prevalent. Essentially all of the vendors supplying database hardware or software offer multiprocessor systems that provide performance substantially beyond their uniprocessor counterpart. Second, it is not only large-scale parallelism that is important but modest-scale multiprocessor servers with tens of processors, or even small-scale multiprocessors with two or four processors.

The primary technological advance is a steady reduction in the basic VLSI feature size. This makes transistors, gates, and circuits faster and smaller, so more can fit in the same area. Intuitively, clock rate improves in proportion to the improvement in feature size, while the number of transistors grows as the square, or even faster, due to increasing overall die area. Thus, in the long run, the use of many transistors at

once (i.e., parallelism) can be expected to contribute to the need for parallel architectures such as shared-memory multiprocessors.

This intuition is borne out by examination of commercial microprocessors. Clock rates for the leading microprocessors increase by about 30% per year, while the number of transistors increases by about 40% per year. Thus, if we look at the raw computing power of a chip, transistor capacity has contributed an order of magnitude more than the clock rate over the past two decades.

Shared-memory multiprocessors built from commodity microprocessors are expected to provide high performance for a variety of scientific and commercial applications. Current commodity microprocessors improve performance with aggressive techniques to exploit high levels of instruction-level parallelism (ILP). For example, the HP PA-8000, Intel Pentium II, III, and MIPS R10000, R12000 processors use multiple instruction issue, dynamic (out-of-order) scheduling, multiple non-blocking reads, and speculative execution. However, most recent architecture studies of shared-memory systems use direct-execution simulators, which typically assume a processor model with single issue, static (in-order) scheduling, and blocking reads.

Despite the latency-tolerating techniques integrated within ILP processors, multiprocessors built from ILP processors have a greater need for additional memory latency reducing and hiding techniques than previous generation multiprocessors [17]. These techniques include conventional hardware and software techniques, and aggressive compiler techniques to enhance the read miss overlap in applications. For this reason, a multiprocessor simulator is required to exploit advances in uniprocessor technology by studying the next-generation shared-memory architectures.

2.2 What Is the Advantage of Distributed Shared Model?

This subsection reviews the increasingly important area of DSM (Distributed Shared Memory). This is a relatively new concept that combines the advantages of the shared- and distributed-memory approaches. DSM system logically implements the shared-memory model on a physically distributed-memory system. System designers can implement the specific mechanism for achieving the shared-memory abstraction in hardware or software. DSM system hides the remote communication mechanism from the application writer, preserving the programming ease and portability typical of shared-memory systems. DSM systems allow for relatively easy modification and efficient execution of existing shared-memory system applications, which preserves software investments while maximizing the resulting performance. In addition, the scalability and cost-effectiveness of underlying distributed-memory systems are also inherited. Consequently, DSM systems become a viable choice for building efficient, large-scale multiprocessors. The DSM model's ability to provide a transparent interface and a convenient programming environment for distributed and parallel applications have made it the focus of numerous research efforts in recent years. Current DSM system research focuses on the development of general approaches that minimize the average access time to shared data, while maintaining data consistency. Some solutions implement a specific software layer on top of existing message-passing systems. Others extend strategies applied in shared-memory multiprocessors with private caches to multilevel memory systems.

2.3 Why RSIM among MINT, Augmint and SimOS?

Exotic architectural techniques, such as *Prefetching*, *Multithreading*, and *Relaxed Memory Consistency*, can greatly improve the performance of a processor or multiprocessor. Since these features can substantially increase the complexity of a

processor, the study of new architectures is based more and more on detailed simulations. Previous research has shown that execution-driven simulation tools, such as the MINT, Augmint, SimOS, and RSIM provide better evaluations for new designs than traditional trace-driven simulations. Unfortunately, these tools themselves can be very complex to develop. The complexity of both simulator development and processor architecture leads to an interesting trade-off between adapting an existing simulation tool set to current needs and developing a completely new simulator. It would seem more efficient and economical to extend one of these existing tool sets to build a simulator for a newly proposed architecture. This would save the time to develop the auxiliary components, such as the assembler, loader, decoder, execution unit, etc. However, adapting an existing tool set to fit a proposed architecture remains a nontrivial task due to the difficulties of understanding someone else code, and trying to deal with the problems introduced by the extensive use of global variables and macros, in addition to debugging the new codes to simulate the new architecture. This is the reason why RSIM was chosen over other available simulator. RSIM has a modest complexity and contains machine components capable of simulating the hardware of DSM systems with sufficient speed and detail to run application programs. The rest of the section characterizes each simulator more in depth.

2.3.1 MINT

MINT [28] was designed at Rochester University to ease the process of constructing event-driven simulators for multiprocessors. It provides a set of simulated processors that run standard Unix executable files compiled for an old microprocessor MIPS R3000 based multiprocessor. These generate multiple streams of memory reference events that drive a user-provided memory system simulator.

MINT uses a novel hybrid technique that exploits the best aspects of native execution and software interpretation to minimize the overhead of simulation. MINT runs on Silicon Graphics computers and DEC stations and interprets MIPS R3000 instructions. MINT also runs on SPARC machines and interprets MIPS code, not SPARC code.

2.3.2 Augmint

Augmint [14] is a software package on top of which multiprocessor memory hierarchy simulators can be constructed for Intel Architecture specific platforms. The simulation platform runs native on Intel Architecture only. It currently runs on Pentium platforms running Solaris operating system and relies on the GNU gcc compiler tool chain.

There are many simulation environments for studying memory hierarchies that exist in the research domain today. However the focus is on architectures that use RISC processors with no attention being paid to CISC architectures. Augmint was designed to address this gap by providing researchers with a simulation infrastructure that enables the modeling of systems built out of Intel Architecture processors.

Augmint consists of a front end memory event generator, a simulation infrastructure, which manages the scheduling of events and a collection of architectural modes that represent the system under study. Runs on only Intel Architecture platforms and interprets the x86 instruction set.

2.3.3 SimOS

SimOS [15] is a complete machine simulation environment designed for the efficient and accurate study of both uniprocessor and multiprocessor computer

systems. SimOS simulates the computer hardware in enough detail to boot and run commercial operating systems (has maximum complexity). SimOS models hardware similar to that of machines from Silicon Graphics, Inc. and Digital Equipment Corporation. The Key component of such machines is the CPU, and SimOS currently provides modes of the MIPS R4000 and R10000 and Digital Alpha processor families. In addition to the CPU, SimOS simulate caches, multiprocessor memory busses, disk drives, ethernet, consoles, and other devices commonly found on these machines. By simulating the hardware typically found on commercial computer platforms, one can easily port existing operating systems to the SimOS environment. IRIX for SGI machines and Digital UNIX and Linux for Alpha has been ported. SimOS runs on Alpha, and SGI machines and interprets MIPS, Digital, and Linux ISA.

2.3.4 RSIM

RSIM [1] interprets application executables. RSIM chose to use with application executables rather than traces so that interaction between events of different processors during the simulation can affect the course of the simulated execution. This allows a more accurate modeling of the effects of contention and synchronization in simulations of multiprocessors, and more accurate modeling of speculation in simulations of uniprocessors. RSIM is a discrete event-driven simulator. Many of the subsystems within RSIM are activated as events only when they have work to perform. However, the processors (MIPS R10000) and caches are simulated using a single event that is scheduled for execution on every cycle, as these units are likely to have activity on nearly ever cycle. RSIM runs on HP, SGI, and SUN but the *predecode* and *unelf* executable must be created on a SUN platform. *Predecode* translates the SPARC application executable into a form that can be

processed by RSIM. *unelf* must be built and run if the user intends to run RSIM on platforms that do not support the ELF library.

2.4 What Is the advantages of *Multithreading* in a DSM?

In a multiprocessor that uses a private cache on each processor, copies of a given shared-memory block can reside in multiple caches at the same time. A write to this block by one processor requires a mechanism to prevent other processors from reading the old value from their cached copies, a mechanism that guarantees coherency among multiple caches is called *a cache-coherence protocol* [29].

The most commonly used cache-coherence technique is for the hardware to transparently invalidate all other cached copies of the same block when a write occurs. Since the copy in the writing processor's cache is now the only valid copy in the system, that processor can continue to write to the block locally without causing coherence problems. However, access to the block by any of the invalidated processors will result in a coherence or sharing miss. Application programs that write and read to shared variables often will suffer from numerous coherence misses.

In a scalable shared-memory multiprocessor, fetching data from the memory of a remote node involves significant latency. However, a latency-tolerating technique can decrease the time the processor must stall in this situation. Such a techniques is called *Multithreading*.

Hardware-supported *Multithreading* is perhaps the most versatile technique for hiding latency. It has the following conceptual advantages:

- It requires no special software analysis or support (other than having more explicit threads or processes in the parallel program than the number of processors).

- Because it is invoked dynamically, it can handle unpredictable situations, such as cache conflicts and communication misses, just as well as predictable ones.
- *Multithreading* can potentially tolerate any long-latency event just as easily, as long as the event can be detected at run time. This includes synchronization and instruction latency.

There are also a number of other situations where threads can greatly simplify writing elegant and efficient programs. The sorts of problems where threads can be very useful include.

- Blocking Input / Output (I/O).

Programs that perform a lot of I/O have three options. They can either perform the I/O serially, waiting for each to complete before commencing the next. They can use Asynchronous I/O, dealing with all the complexity of asynchronous signals, polling or selects. They can use synchronous I/O, and just spawn a separate thread/process for each I/O call. In this case, *Multithreading* can significantly improve both performance and code complexity.

- Multiple Processors.

If a thread library that supports multiple processors is used, one can gain significant performance improvements by running threads on each processor. This is particularly useful when the program is compute bound.

- User Interface (UI).

By separating the user interface, and the program engine into different threads one can allow the UI to continue to respond to user input even while long operations are in progress.

- Servers.

Servers that serve multiple clients can be made more responsive by the appropriate use of concurrency. This has traditionally been achieved by using the *fork()* system call. However in some cases, especially when dealing with large caches,

threads can help to improve the memory utilization, or even permit concurrent operation where *fork()* was unsuitable.

3 RSIM TOOL

This chapter gives the detailed features of the various subsystems in RSIM, such as application porting, processor, memory hierarchy, interconnection system network, directory-based coherence protocol, and statistics tool.

3.1 Overview of RSIM

Simulation has two major advantages. First, simulation provides the flexibility to modify various architectural parameters and components, and to quickly analyze the benefits of such modifications. Second, it allows for detailed statistics collection, which provides a better understanding of the compromise involved and facilitates performance tuning. An accurate simulation tool is essential for evaluating new ideas in both uniprocessor and multiprocessor architectures. Compared to other simulators, an advantage of RSIM is that it supports a processor model that is more current and state-of-the-art. Currently, available simulators model much simpler processors and can exhibit significant inaccuracies when used to study the behavior of state-of-the-art processors. RSIM provides many features and allows the user to configure a number of architecture parameters. Key features supported by RSIM are [1]:

- Processor features:
    1. Multiple instruction issue
    2. Out-of-order scheduling
    3. Register renaming
    4. Static and dynamic branch prediction support
    5. Non-blocking loads and stores

6. Speculative load execution before address disambiguation of previous stores
7. Simple and optimized memory consistency implementations

- Memory hierarchy features:
  1. Two-level cache (Level 1 (L1), Level 2 (L2)) hierarchy
  2. Multiported and pipelined L1 cache, pipelined L2 cache
  3. Multiple outstanding cache requests
  4. Memory interleaving
  5. Software-controlled non-binding prefetching
- Multiprocessor system features:
  1. CC-NUMA shared memory system with directory based cache coherence protocol
  2. MSI or MESI coherence protocols
  3. Sequential consistency, Processor consistency, and release consistency
  4. 2 dimensional mesh network

Application executables are fed directly into RSIM rather than using traces. This allows for more accurate modeling because events of among processors during the simulation can affect the course of the simulated execution. RSIM is written in C++ and C and supports various platforms. RSIM has been simulated on SUN machines running SunOS 5.6. RSIM version 1.0 is available as a UNIX tar and gzip compressed file rsim-1.0.tar.gz from *http://www-ece.rice.edu/~rsim/dist.html* at no cost. After unpacking the file, users can change to the appropriate subdirectory and compile *rsim* according to a simulation platform in *obj*. Ultra170 and GNU *gcc* complier were use for the simulation study. In addition, the *predecode* executable must be created on a Sun platform. *predecode* translates the instructions of a SPARC application executable into a form that can be processed by RSIM. RSIM application library is located in the directory apps/utils/lib. The user can type *make install* to use

this makefile. The apps directory includes two application ported to RSIM: a quicksort and red-black SOR. These applications are useful for familiarizing oneself with RSIM. There are also some SPLASH and SPLASH-2 applications that have been ported to RSIM. The statistics tools are in the *bin* directory. Names of the scripts are *analyze_misses*, *rsim_analyze*, and *p_rsimanalyze*. These utilities do not require compilation.

3.2 Detailed Features

3.2.1 Porting Application to RSIM

This section describes the issues involved in porting application to RSIM. They are

- Shared memory model and RSIM process creation
- RSIM library and Synchronization

RSIM multiprocessor memory model is depicted in Figure 3.1. The regions below the dividing line are all private memory, while the regions above the dividing line are shared memory allocated with the *void *shmalloc(int size)* function.

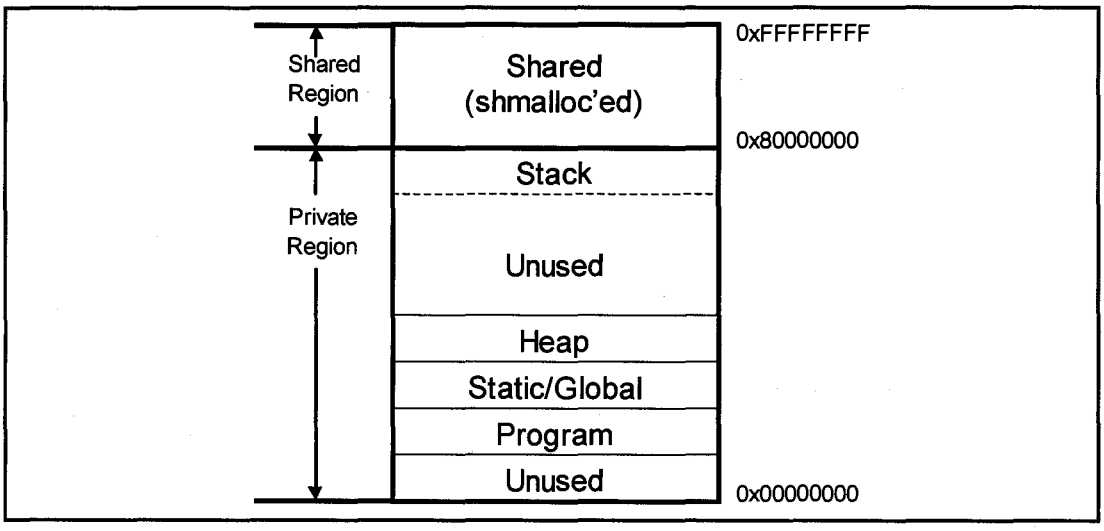


Figure 3.1: RSIM multiprocessor memory model

The stack for each process grows automatically, while the heap and shared region grow only through explicit memory allocation calls. A simple example of shared memory allocation is shown in Figure 3.2.

```
/* main() initialize and distribute shared address space
and the array. Also main() schedule the parallel tasks */
main(int argc, char **argv){
...
    red_=(float **) shmalloc((M+2)*sizeof(float *));
/* Allocate individual rows */
    for (i=0;i<= M+1; i++) {
/* All rows of reds */
        red_[i] = (floate *) shmalloc((N+1)*sizeof(float));
...}}
```

Figure 3.2: Memory Allocation by using *shmalloc* from SOR.

Shared-memory regions can be associated to specific "home nodes" using the *AssociateAddrNode(void *start, void *end, int proc, char *name)* function. Home node (*proc*) provides the directory services for the cache lines in that region. Regions can be associated at the granularity of a cache line. The function partitions the shared address space out and home nodes becomes implicitly with a "first-touch policy" if the *AssociateAddrNode()* is not explicitly specified in applications. An example is shown in Figure 3.3.

```
/* root_prog: handles distribution of shared address space among the different processors.
M: # of rows, N: # of columns. */
void root_prog(void){
    int begin, end, i;
    for(i=0; i<NUM_PROCS; i++){
        begin = (M*i)/NUM_PROCS +1;
        end = (M*(i+1))/NUM_PROCS;
...
AssociateAddrNode(&(reg_[begin][0]),&(red_[end][N]), i, name);
}
/* partitions the shared address space for red_[][] */
```

Figure 3.3: Distribution of shared address space among the processors from SOR.

At the beginning of execution, RSIM starts an application with a single processor in the specified architectural configuration. The application must then use the *fork()* system call to spawn off new processes, each of which is run on its own processor. *fork()* causes the new processor to have its own copy of the application code segment, global and statically allocated variables, private heap, and process stack, but the new processor has the same logical version of the shared portion of the address space. The semantics of *fork()* are identical to those of UNIX *fork()*. Since RSIM currently does not support multitasking, indeterminate results will arise if more

processes are started than the number of processors specified in the configuration file. The simple example for the process creation by using *fork()* is shown in Figure 3.4.

```
for (i=0; i<num_procs-1; i++) {
    if (fork() = = 0) {
        proc_id = getpid();
        break;
    }
}
```

Figure 3.4: Process Creation from SOR.

Additionally, RSIM applications library provides important multiprocessor primitives for synchronization, such as locks, flags, and barriers. Locks are supported through a *test-and-test-and-set* mechanism. The lock function acts on a single integer in a shared region. The macro *GETLOCK(int *lock)* takes a pointer to the lock as an argument and spins with a test-and-test-and-set loop until the lock is available. The macro *FREELOCK(int * lock)* takes a pointer to the lock as an argument and frees the lock. Both of these macros invoke functions that perform the necessary operations. Flags are supported through an ordinary spinning mechanism. An example is shown in Figure 3.5. The processor, which acquires the lock, can access the critical region and push the job onto the TaskStack. The processor releases the lock after the job is pushed onto the stack so that the other processor can have a chance to acquire the lock.

```
/* PushWork(): the procedure that pushes a new task onto the task stack. */
void PushWork(int i, int j){
...
    GETLOCK(&gMem->TaskStackLock);
    a=gMem->TaskStackTop++;          /* Critical Region */
    te->next=gMem->TaskSack; ++;     /* Critical Region */
    gMem->TaskStack=te; ++;          /* Critical Region */
    FREELOCK(&gMem->TaskStackLock);
...}
```

Figure 3.5: The usage of lock from Quicksort.

The only type of barrier supported is a simple binary tree barrier. A tree barrier is a structure of type *TreeBar* and can be declared as an ordinary global variable. The function *TreeBarInit(TreeBar *bar, int numprocs)* is used to initialize the tree barrier to a barrier that requires the specified number of processors. When synchronizing, each processor calls the macro *TREEBAR(TreeBar *bar, int pid)*, where *pid* is the processor's unique ID number (initially determined through *getpid()*). *TreeBarrier* is called after finishing the sorting for correct timing among processors. An example is shown in Figure 3.6.

```
ex) Barrier
main(int argc, char **argv){
...
    TreeBarInit(&tree, NUM_PROCS);
if (fork() == 0) {                    /* forking process */
proc_id = getpid();
break;
}}
    TreeBarrier(&tree, whoami);  /* Barrier after initialization */
...
QuickSort(task->left, task->right); /* call a quicksort */
...
    TreeBarrier(&tree, whoami);  /*Barrier after finishing the sort */
...}
```

Figure 3.6: TreeBarrier.

3.2.2 Processor

The processor microarchitecture RSIM model is MIPS R10000 and is illustrated in Figure 3.7. Specifically, RSIM models the R10000's active list (which holds the currently active instructions that corresponds to reorder buffer or instruction window register map table (which holds the mapping from the logical to physical registers), and shadow mappers (which allow single cycle state recovery on a mispredicted branch). The pipeline parallels the Fetch, Decode, Issue, Execute, and Complete stages of the dynamically scheduled R10000 pipeline. Instructions are fetched, decoded, and graduated in program order. In-order graduation enables precise interrupts.

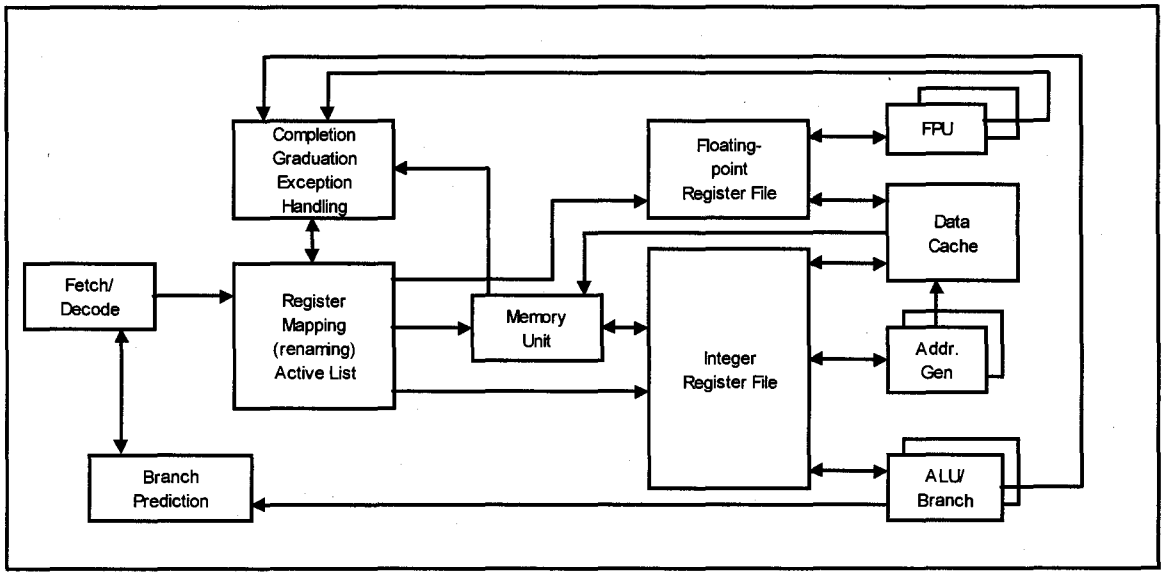


Figure 3.7: The processor microarchitecture of RSIM.

RSIM is organized as a discrete-event-driven simulator. In RSIM, the processors and cache hierarchies are modeled as single event (called *RSIM_EVENT*),

which is scheduled every cycle. However, RSIM is not a pure cycle-by-cycle simulator since events for busses, directories, and network are scheduled only when needed.

The primary subsystems in RSIM are the event-driven simulation library, the out-of-order engine, the processor memory unit, the cache hierarchy, the directory module, and the interconnection system. Each subsystem acts as a largely independent block, interacting with the other units.

RSIM_EVENT loops through all the processors and caches in the system, calling appropriate functions. The overview of *RSIM_EVENT* is shown in Figure 3.8.

```
1. LICacheOutSim()        /* Handle requests in the pipeline LI cache */
2. L2CacheOutSim()        /* Handle requests in the pipeline L2 cache */
3. CompleteMemQueue()     /* Completion of the pipeline */
4. maindecode()           /* main processor pipeline */
      update_cycle()      /*update the result registers/branch */
      graduatel_cycle()   /* Graduation */
      decode_cycle()      /* Instruction Fetch & Decode */
5. IssueQueues()          /* Issue to queues */
6. LICacheInSim()         /* Handle requests coming into LI cache */
7. L2CacheInSim()         /* Handle requests coming into L2 cache */
```

Figure 3.8: Overview of *RSIM_EVENT*.

RSIM_EVENT is scheduled to occur on offset of 0.5 cycles from the processor cycle. This is because the *RSIM_EVENT* function would not be able to pick up the reply until time X+1 if the *RSIM_EVENT* planned for time X had already occurred.

For each processor, *RSIM_EVENT* first calls *L1CacheOutSim* and *L2CacheOutSim,* which are used to bring new messages into each level of the cache. Then, *CompleteMemQueue* and *CompleteQueue* are called to inform the memory unit of any operations that have completed at the caches and process other instructions that have completed at their functional units, respectively. Then, *maindecode* starts out by using *updated_cycle* to update the register files. *graduate_cycle* is next called to remove previously completed instructions in-order from the active list and to commit their architectural state. *maindecode* calls *decode_cycle* to bring new instructions into the activelist before *maindecode* returns control to *RSIM_EVENT*. Then, *RSIM_EVENT* calls *IssueQueues*, which sends ready instructions to their functional units. Finally, the functions *L1CacheInSim* and *L2CacheInSim* are called for the caches to bring in new operations that have been sent to them. After all of this, *RSIM_EVENT* repeats this loop for the next processor.

The instruction *fetch* and *decode* stages are merged because RSIM does not model an instruction cache. The instruction *decode* state handles register renaming and inserts the decoded instructions into the active list. The data structure used in this stage are the register map table, the free list, the active list, and the shadow mappers. These data structures follow the corresponding microarchitectural features of the MIPS 10000.

An instruction is entered into the active list when it is decoded, and it remains in the active list until it graduates. This stage also dispatches memory instructions to the memory unit, which is used to insure that memory operations occur in the appropriate order. For branch instructions, the *decode* stage allocates a shadow mapper to allow for a fast recovery on a misprediction. The prediction of a branch as "taken" stops the RSIM processor from decoding any further instructions in this cycle, as many current processors do not allow the instruction fetch or decode stage to access two different regions of the instruction address space in the same cycle.

RSIM supports 2-bit prediction history scheme. Figure 3.9 shows the states in a 2-bit prediction history scheme. The first bit is the branch prediction bit for future use. The second bit represents that action has really happened. Suppose the initial state is "00". It means that branch will be "not taken" in the future and "not taken" happened the last time. But if the branch prediction is wrong and branch is "taken", then the next state would be "01". This means that the prediction for the future is still "not taken" but the last time branch was "taken".

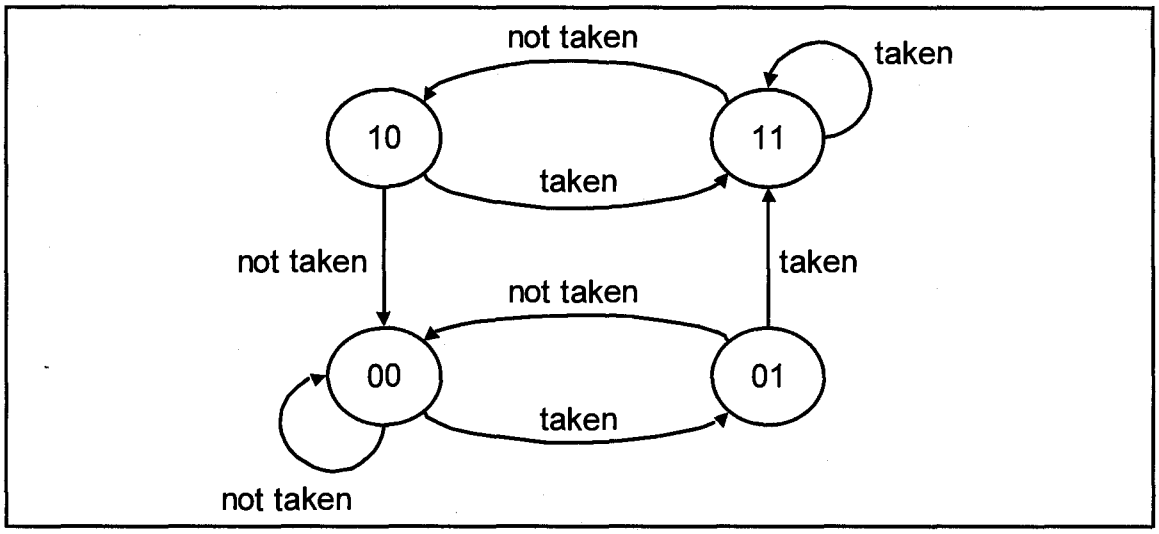


Figure 3.9: The states in a 2-bit prediction scheme.

As MIPS R10000, each predicted branch uses a branch stack (shadow mapper) [8] that stores the state of the register renaming table at the time of a branch prediction. The processor may include multiple predicted branches at the same time, as long as there is at least one shadow mapper for each outstanding branch. When the shadow mapper is full, the processor continues decoding only until it encounters the next

branch instruction. Decoding is then stalled until the resolution of one of the pending branches. These branches may also be resolved out-of-order.

The instruction *issue* stage issues ready instructions. For an instruction to issue, it must have no outstanding data dependencies or structural hazards. With one exception, the only register data dependencies that affect the issuing of an instruction are RAW dependencies. Other register dependencies are eliminated by register renaming. RAW dependencies are detected by observing the "busy bit" of a physical register in the register file. Structural hazards in the issue stage are related to the availability of functional units. Three types of functional units supported in RSIM are arithmetic logical unit (ALU), floating point unit (FPU), and address generation unit (ADDR).

The instruction *execute* state calculates the results of the instruction as it would be generated by its functional unit. These results include the effective addresses of loads and stores for ADDR.

The instruction *complete* state stores the computed results of an instruction into its physical register. This stage also clears that physical register's busy bit in the register file, thus indicating to the issue stage that the instruction stalled on a data dependency. This stage also resolves the proper outcome of predicted branches. If a misprediction is detected, later instructions in the active list are flushed and the processor's program counter is set to the proper target of the branch. The shadow mapper for a branch is also freed also in this stage.

The instruction *graduate* stage ensures that the instructions graduate and commit their values into the architectural state in-order, thereby allowing the processor to maintain precise exceptions. When an instruction is graduated, the processor frees the physical register formerly associated with its destination register. After graduation, the instruction leaves the active list.

User-configurable processor parameters are the number of functional units, the latencies and repeat rates of the functional units, the instruction issue width, the size of the active list, the number of shadow mappers for branch speculation, and the size of the branch prediction structures.

3.2.3 Memory Hierarchy

Figure 3.10 shows the organization for the memory and the network systems. RSIM simulates a hardware cache-coherent non-uniform memory architecture (CC-NUMA) system, with a fully-mapped invalidation directory based coherence protocol. UMA (uniform memory architecture) is not currently supported. Each node consists of one processor, a two level cache hierarchy with a write-buffer (if L1 is write-through), distributed physical memory and its associated directory, and a network interface. A pipelined split-transaction bus connects the L2 cache, the memory and directory modules, and the network interface. Local communication within the node takes place on the bus. The network interface connects the node to an interconnection network for remote communication. Both caches are lockup-free and they store the state of outstanding requests in miss status holding registers (MSHRs) [4].

L1 cache can either be a write-through cache with a no-allocate policy on writes, or a write-back cache with a write-allocate policy. RSIM allows for a multiported and pipelined L1 cache. If L1 cache is used, write-buffer is implemented between L1 and L2 cache. All writes are buffered here and sent to L2 cache as soon as L2 cache is free to accept a new request. L2 cache is always a writeback cache with write-allocate. RSIM allows for a pipelined L2 cache. The memory is interleaved, with multiple modules available on each node. The memory is accessed in parallel with an interleaved directory.

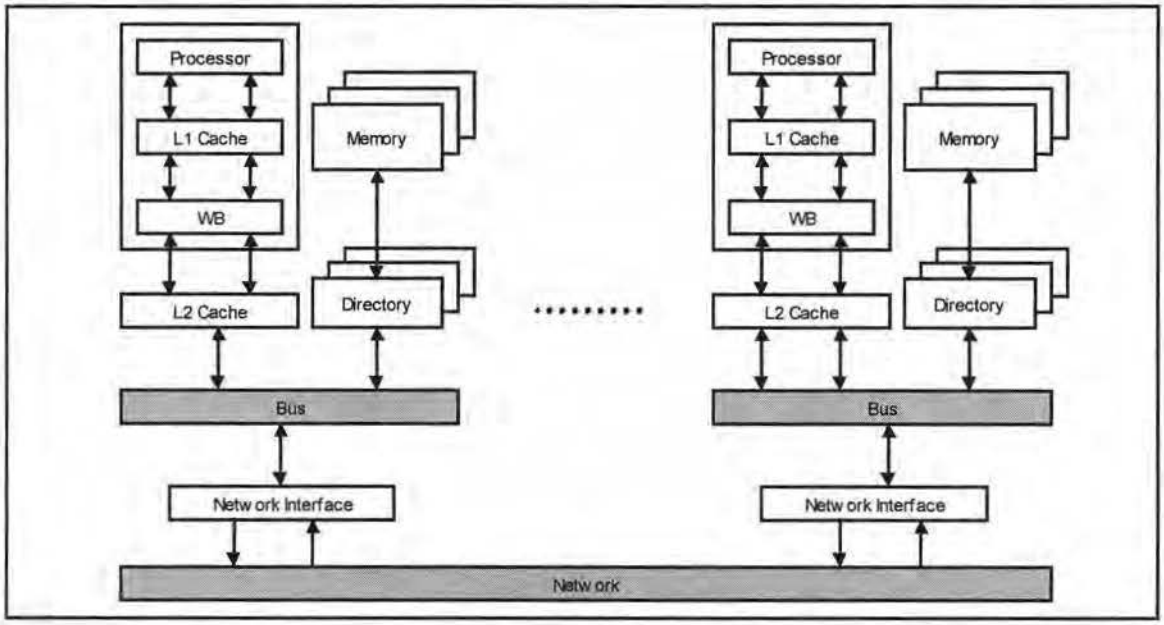


Figure 3.10: The RSIM memory System.

The RSIM bus module simulates a split-transaction bus that imposes no limits on the number of outstanding requests. This bus connects L2 cache, the network interfaces, and the directory/memory modules within a node. Arbitration is round-robin among the bus agents. The function *node_bus* represents the main operation of the bus simulator. This function is subdivided into several stages according to the progress of a request on the bus. The default bus width is 32 bytes. The default delay assumes a bus cycle, which is five times slower than the processor cycle (i.e., bus cycle = 5 * processor cycle).

In the *BUSSTART* stage, the bus has not started processing a transaction. In this case, the bus peeks at the ports in a round-robin fashion starting with the port after the one last accessed for the previous transaction. If none is found, the bus goes to sleep until woken up by a new request. For a new request, if the bus is available, its transaction moves to the *SERVICE* stage.

In the *SERVICE* stage, the routing function is called to determine the output port for this message. If this output port is not available, the bus is delayed for a bus cycle before returning to *BUSSTART*, where it will try to find a different transaction or keep trying this transaction until the output port becomes available. If the port is available, however, the bus transitions to the *BUSDELIVER* stage after stalling for the latency of the transfer based on the bus width, the bus cycle, and the message size.

In the *BUSDELIVER* stage, the bus moves the transaction into the desired output port, after which it will stall for an arbitration delay before allowing *BUSSTART* to continue processing a new request.

The data structure (*SMMODULE*) contains a basic memory framework. This framework includes fields common to all the module types, and is initialized using the *ModuleInit* function. This function sets the node number for the module, along with fields related to the module's input and output ports. These ports, which are of the type *SMPORT*, are used to provide connecting between the various modules in the memory system simulator. *SMPORT* data structures act as queues between the various modules and have memory system simulator messages as their entries. Each queue has a fixed maximum size, initialized by the call to *ModuleInit*. Note that only output ports are actually created; the input ports of one module are set to the same data structures as the output ports of another module using the *ModuleConnect* function.

The various modules and default port connections used in RSIM are shown in Figure 3.11 and 3.12. Input ports shown on this figure are labeled as *iX*, where *X* is the port number. The output ports are labeled as *oX*. The terms Request, Reply, Cohe, and Cohe reply correspond to the types of memory system messages.

If no write-buffer is included, the output port (*o1*) of the L1 cache connects directly to the input port (*i0*) of the L2 cache, while the output port (*o0*) of the L2 cache connects to the input port (*i1*) of the L1 cache. This is shown in Figure 3.12.

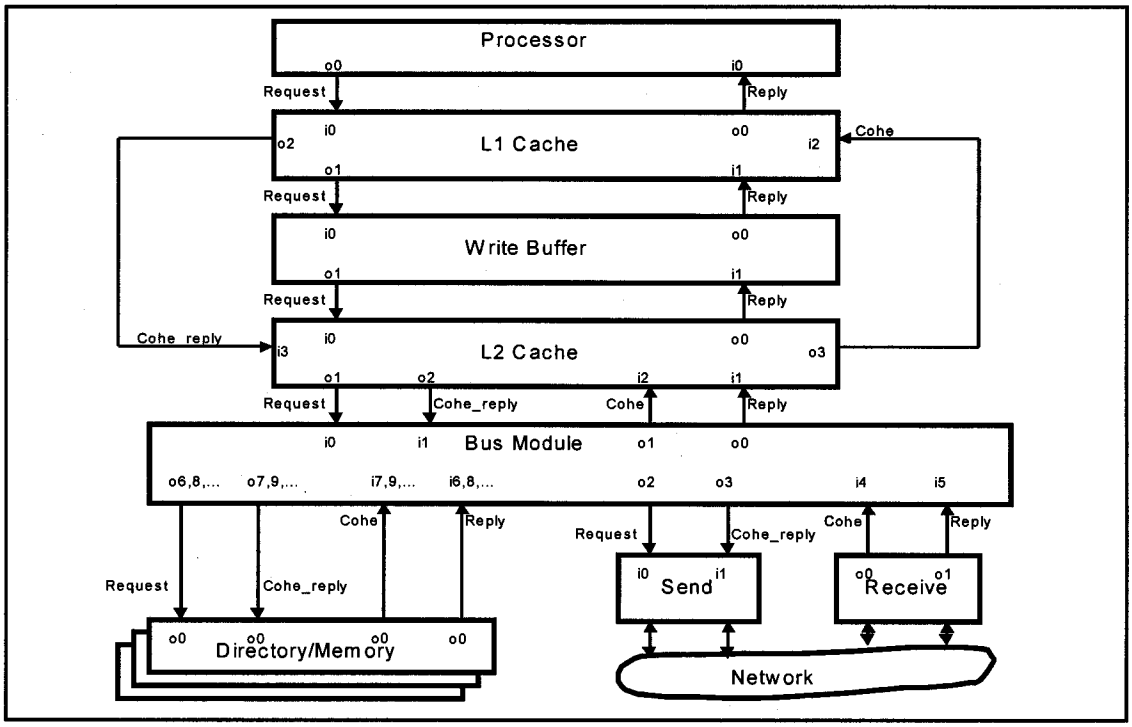


Figure 3.11: Modules and port connections in RSIM with write buffer

The fundamental unit of information exchange among the RSIM memory system simulator modules is referred to as a memory system message. The five most important fields of the message data structure are the *tag* field, the *s.type* field, the *req_type* field, the *s.reply* field, and the *s.nack_st* fieled. The *tag* field specifies the cache line to which the action in question applies. The *s.type* field consists of *REQUEST* (action related to the data request by a processor or a cache), *REPLY* (response to the demands of a *REQUEST* by a cache or a directory), *COHE* (demand to invalidate or change the state of a line by a directory), *COHE_REPLY* (response to the demands of a *COHE* message by a cache).

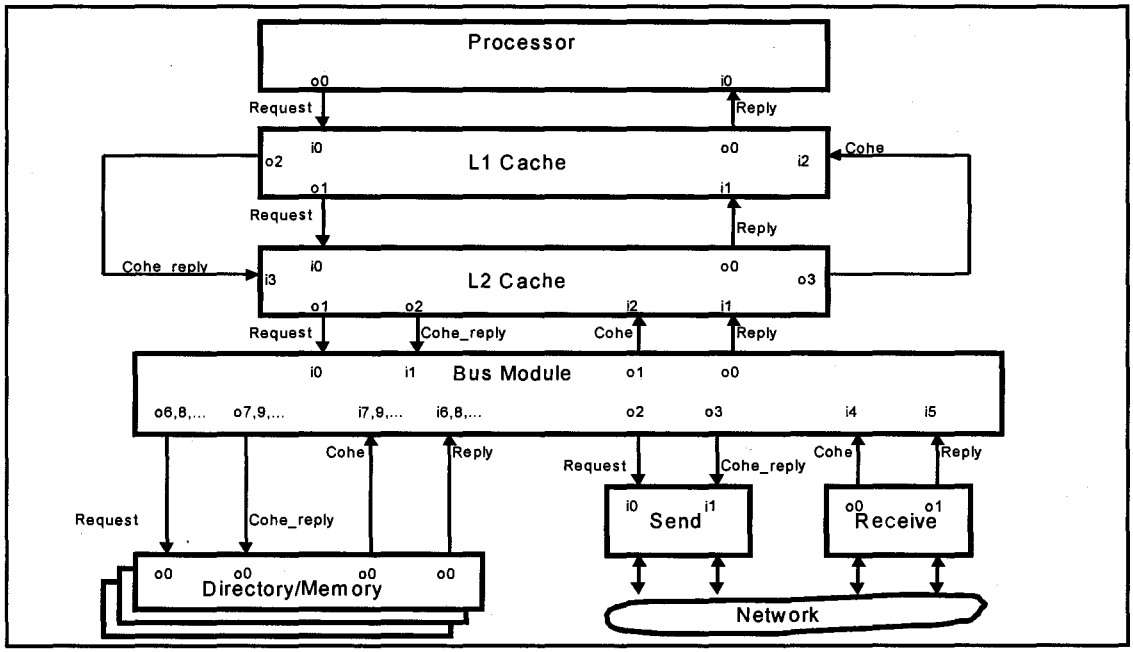


Figure 3.12: Modules and port connections in RSIM without write buffer.

3.2.4 Directory-Based Cache Coherent Protocol

RSIM simulates a hardware cache-coherent distributed shared memory system, using a variations of a full-mapped invalidation based directory coherence protocol, called *Flat Memory-Based Directory Scheme*. The directory is responsible for maintaining the current state of a cache line, serializing accesses to each line, generating and collecting coherence messages, sending replies, and handling race-conditions and network congestion. Subsection 3.2.4.2 (Intranode Coherence) discusses the protocol transactions required to maintain coherence between L1 cache, L2 cache, and the home directory. The protocol transactions required to maintain coherence between different processing nodes will be discussed in Subsection 3.2.4.3 (Internode Coherence).

3.2.4.1 Nomenclature

The coherence protocol classifies the processing nodes into three overlapping categories: *local*, *remote*, and *home*. The *local* node is the node containing the CPU that issues a memory operation. The *home* node, as described above, is the node containing the directory memory for the memory operation. A *remote* node is any node that is not the local node. Note that a node is either a local node or a remote node, and the home node for an arbitrary memory operation can be either a local node or a remote node. The possible states of cache lines in L1 and L2 caches are:

- Invalid (I) - This cache line is not valid.
- Shared (S) - This cache line is valid and readable. Also, the home directory knows that one or more nodes have the shared copy of the cache line.
- Modified (M) - This cache line is valid and readable. Also, the home directory knows that only one node has the valid cache line.

Similarly, the home directory can be in one of the following states:

- Uncached - The cache line is uncached by all nodes.
- Shared - The cache line is held in shared mode by one or more nodes.
- Private - The cache line is held in exclusive mode by only one node.

In the following, the RSIM coherence protocol will be defined through a set of flow diagrams representing protocol transactions required to handle each coherence operation. The flow diagrams depicting the protocol transactions will use the notation given in Figure 3.13.

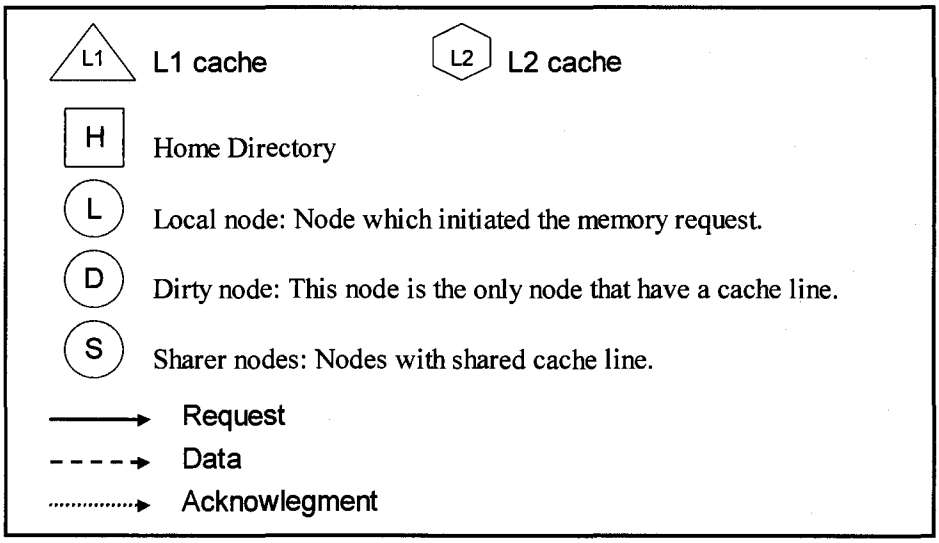


Figure 3.13: Flow graph notations.

Protocol transactions between the home directory and L1 and L2 caches are designed to maintain the subset property of multilevel caches. The subset property states that L1 and L2 caches always contain a subset of states in the home directory. In addition, the access rights to data in L1 and L2 caches are a subset of the access rights to data in the home directory. These restrictions limit the allowable combinations of the home directory and L1 and L2 cache line states. The permitted combinations are shown in Table 3.1.

		L1 and L2 cache State		
		Invalid	Shared	Modified
Home Directory State	Uncached	O		
	Shared	O	O	
	Private	O	O	O

Table 3.1: Allowable home directory and L1 and L2 cache state combinations.

3.2.4.2 Intranode Coherence

The intranode flow diagrams for processor reads are shown in Figure 3.14. The corresponding flow diagrams for processor write requests are shown in Figure 3.15.

a) Read request to a readable cache line in L1 cache
> Processor sends read request (1) to L1 cache
> L1 cache retruns read data (2) to processor

b) Read request to a readable cache line in L2 cache
> Processor sends read request (1) to L1 cache
> L1 cache sends read request (2) to L2 cache
> L2 cache sends read data (3) to L1 cache
> L1 cache state goes to *shared* if L2 cache state is *shared* or *modified* if L2 cache state is *modified*, L1 cache send read data (4) to processor

c) Read to cache line not present in L1, L2 cache
> Processor sends read request (1) to L1 cache
> L1 cache sends read request (2) to L2 cache
> L2 cache sends read request (3) to home directory
> home directory returns read data (4) to L2 cache
> L2 cache state goes to *shared* if home directory state is *shared* or *modified* if home directory state is *private*, L2 cache sends read data (5) to L1 cache
> L1 cache state goes to *shared* if home directory state is *shared* or *modified* if home directory state is *private*, L1 cache sends read data (6) to processor

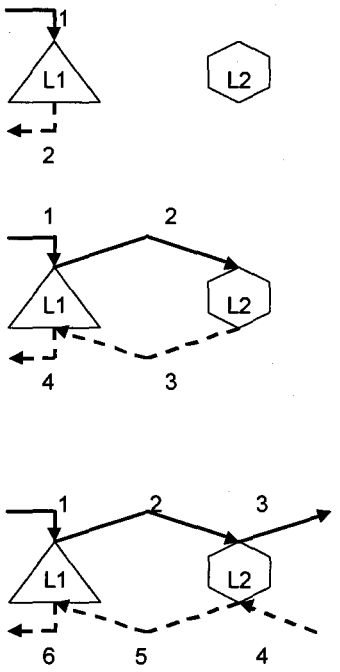


Figure 3.14: Processor read transaction flows.

a) Write request to a *modified* cache line in L1 cache
> Processor sends write request (1) to L1 cache
> L1 cache performs write
> L1 cache sends write data (2) to L2 cache
> L2 cache sends acknowlegement (3) to home directory

b) Write request to a modifed cache line in L2 cache
> Processor sends write request (1) to L1 cache
> L1 cache sends write request (2) to L2 cache
> L2 cache sends acknowlegement (3) to home directory
> L2 cache sends exclusive data (4) to L1 cache
> L1 cache state goes to *modified*, L1 cache performs write

c) Write request to an invalid/shared cache line in L2 cache
> Processor sends write request (1) to L1 cache
> L1 cache sends write request (2) to L2 cache
> L2 cache sends write request (3) to home directory
> home directory returns read data (4) to L2 cache
> L2 cache state goes to *modified*, L2 cache sends exclusive data (5) to L1 cache
> L1 cache state goes to *modified*, L1 cache performs write

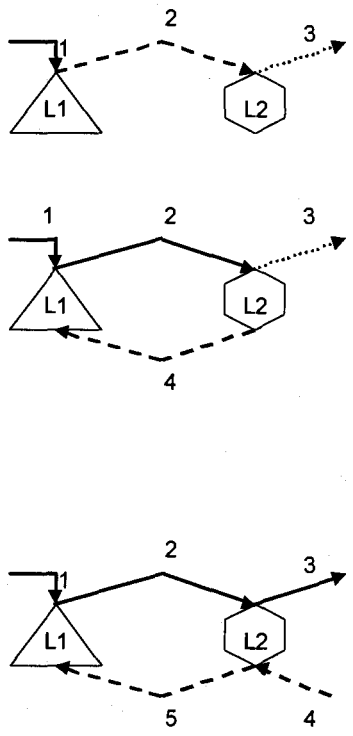


Figure 3.15: Processor write transaction flows.

While intranode coherence transactions are generated when the local processor makes memory requests, intranode coherence transactions are also generated when external processors make memory requests for data held locally. For example, when a processor issues a read request but cannot find a copy of the data in its node, a read request will be sent to the *dirty node* (see definition in Figure 3.13), which must provide a copy of the data to the requestor. The flow diagrams showing intranode coherence transactions generated due to external read requests are shown in Figure 3.16.

Similarly, when a processor issues a write request but cannot find an exclusive copy of the data in its node, a read exclusive request will be sent to the *dirty node*.

The *dirty node* provides a copy of the data to the requestor, and the home directory and L1 and L2 cache state for the block in the *dirty node* will change to invalid. If the requested block is shared among several nodes, the write request will result in sending of invalidation requests to all remote *sharer nodes* (see the definition in Figure 3.13). The remote *sharer nodes* will then invalidate any copies of the block found in their L1 and L2 caches and each of those nodes will reply with an "invalidation acknowledgment" message. The home directory combines all of the invalidation acknowledgments into a single "acknowledgment complete" message, which is then sent to the requestor to indicate the completion of the request. The flow diagrams for external write requests are shown in Figure 3.17.

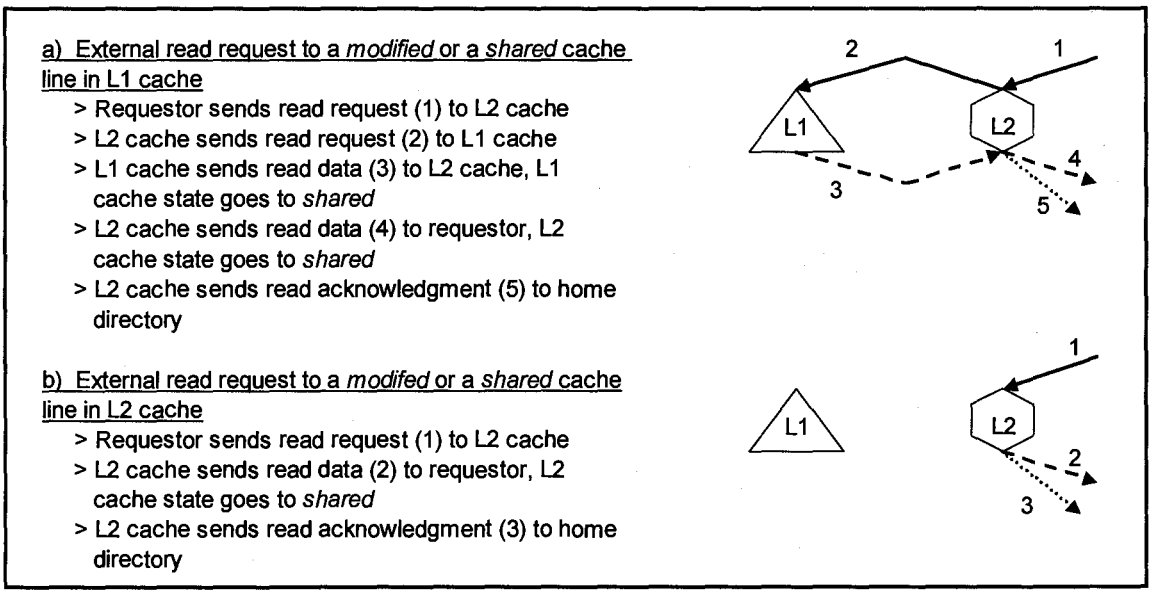
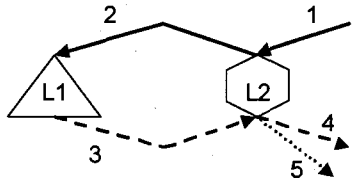
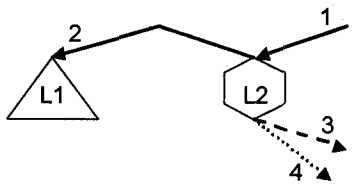


Figure 3.16: External read transaction flows

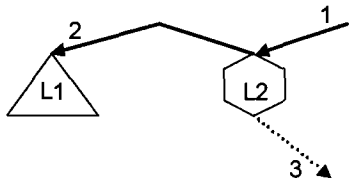
a) External read exclusive request to a *modified* or a *shared* cache line in L1 cache
- > Requestor sends read exclusive request (1) to L2 cache
- > L2 cache sends read request (2) to L1 cache
- > L1 cache sends read exclusive data (3) to L2 cache, L1 cache state goes to *invalid*
- > L2 cache sends read exclusive data (4) to requestor, L2 cache state goes to *invalid*
- > L2 cache sends read exclusive acknowlegement (5) to home directory



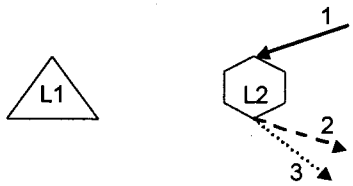
b) External read exclusive request to a *modifed* cache line in L2 cache and a *shared* in L1 cache
- > Requestor sends read request (1) to L2 cache
- > L2 cache sends invalidate(2) to L1 cache, L1 cache state goes to *invalid*
- > L2 cache sends read exclusive data (3) to requestor, L2 cache state goes to *invalid*
- > L2 cache sends read exclusive acknowledgment (4) to home directory



c) External read exclusive request to a *shared* cache line in L2 cache and a *shared* in L1 cache
- > Requestor sends read request (1) to L2 cache
- > L2 cache sends invalidate(2) to L1 cache, L1 cache state goes to *invalid*
- > L2 cache sends read exclusive acknowledgment (3) to home directory, L2 cache state goes to *invalid*



d) External read exclusive request to a *modified* cache line in L2 cache and a *invalid* in L1 cache
- > Requestor sends read request (1) to L2 cache
- > L2 cache sends read exclusive data (2) to requestor, L2 cache state goes to *invalid*
- > L2 cache sends read exclusive acknowledgment (3) to home directory



e) External read exclusive request to a *shared* cache line in L2 cache and a *invalid* in L1 cache
- > Requestor sends read request (1) to L2 cache
- > L2 cache sends read exclusive acknowledgment (2) to home directory, L2 cache state goes to *invalid*

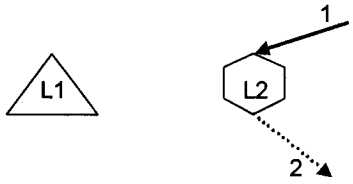


Figure 3.17: External write transaction flows.

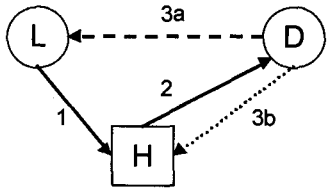
3.2.4.3 Internode Coherence

In most cases, processor reads and writes can be satisfied by its local cache or local memory, thus no internode protocol transactions are required. Requests to data not found in the local cache are sent to the home directory to determine the current location of the dirty copy of the data. Read misses are sent to the directory as read requests. Write misses, however, are translated into read exclusive requests because write requests indicate that the processor wishes to modify the data. To maintain coherence, the cache line must be modified only when it is held in an exclusive state. Both read and read exclusive requests are always sent to the home directory first, to determine where a copy of the requested cache line can be obtained. For a read request, the home directory will forward the request to the *dirty node*, which will supply a copy of the data to the requestor. The *dirty node*'s cache line state will then be changed to shared. When the requestor node receives the data reply, the cache line state will be changed to shared. The possible transaction flows for a read request are shown in Figure 3.18.

For a read exclusive request, the directory forwards the request to the *dirty node*, which replies with data to the requestor. The *dirty node*'s cache line state then becomes invalid. The directory also sends invalidations to all the nodes except for the *dirty node* and/or the requestor node, which currently have copies of the data. When the directory collects all of the invalidation acknowledgements, one summary acknowledgment is sent to the requestor (indicating that at this point no other copies of the requested cache line exist). When the requestor node receives the data reply, the state is changed to exclusive. The possible transaction flows for a read exclusive request are shown in Figure 3.19.

a) Read request to a *shared* cache line

> Local node sends read request (1) to home directory
> Home directory forwards request (2) to dirty node, directory adds local node to sharing list
> Dirty node sends read data (3a) to local node and request complete acknowledgment (3b) to home directory, dirty node state goes to *shared*
> Local node state goes to *shared*



b) Read request to a *modified* cache line

> Local node sends read request (1) to home directory
> Home directory forwards request (2) to dirty node, directory adds local node to sharing list, directory state goes to shared
> Dirty node sends read data (3a) to local node and request complete acknowledgment (3b) to home directory, dirty node state goes to *shared*
> Local node state goes to *shared*

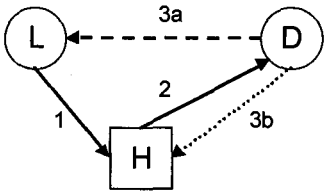
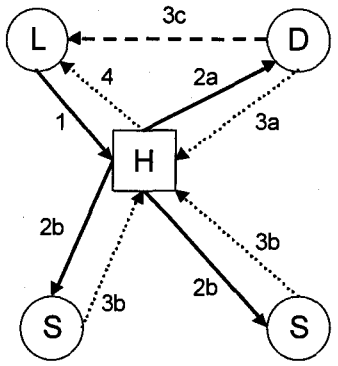
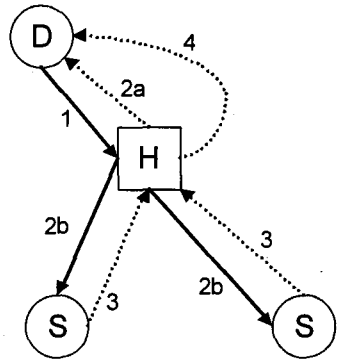


Figure 3.18: Read request flows

a) Read exclusive request to a *shared* cache line
- > Local node sends read exclusive request (1) to home directory
- > Home directory forwards request (2a) to dirty node, sends invalidations (2b) to other sharers
- > Dirty node sends request complete acknowledge (3a) to home directory, dirty node sends read exclusive data (3c) to local node, dirty node state goes to *invalid*
- > Sharing nodes send invalidation acknowledgments (3b) to home directory, sharing node state goes to invalid
- > Home directory sends acknowledgments complete message (4) to local node, home directory state goes to *exclusive*, local node state goes to *exclusive*



b) Read exclusive request to a *shared* cache line and local node is a *modified* node
- > Dirty node sends read exclusive request (1) to home directory
- > Home directory forwards request (2a) back to requestor and sends invalidations (2b) to other sharers
- > Sharing nodes send invalidation acknowledgments (3) to home directory, sharing node state goes to invalid, local node state goes to *modified*
- > Home directory sends acknowledgments complete message (4) to local node, home directory state



c) Read exclusive request to a *modified* cache line
- > Local node sends read exclusive request (1) to home directory
- > Home directory forwards request (2) to dirty node
- > Dirty node send request complete acknowledge (3a) to home directory, sends read exclusive data (3b) to local node, dirty node state goes to *invalid*
- > Home directory sends acknowledgments complete message (4) to local node, home directory state goes to *modified*, local node state goes to *modified*

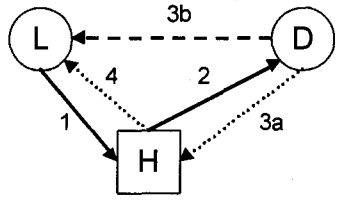


Figure 3.19: Read exclusive request flow.

3.2.5 Interconnection Network System

The SMNET (Shared Memory-Network) interfaces in RSIM are the modules that connect each node's local bus to the interconnection network. The primary functions of the SMNET are given below.

- Receive messages destined for the network from the bus.
- Create the message packets.
- Inject the messages into the appropriate network ports and initiate communication.
- Handle incoming messages from the network by removing them from the network port and delivering them to the bus.

The main procedure for sending packets to the network is *SmnetSend.* This event handles communication between the bus and the network interface. Upon receiving a new message, *SmnetSend* schedules an appropriate event to insert the new message into the request or the reply network. As soon as a message is received, it is forwarded to the appropriate bus port. The bus will actually deliver the message to the caches or the directory.

The base network is a 2-dimensional bidirectional mesh without wraparound connections. Figure 3.20 shows the interconnection network. The interconnection network includes separate request and reply networks for deadlock-avoidance. The base network is taken from the NIETSIM simulator [9]. NETSIM provides various modules which are multiplexers, demultiplexers, and buffers. The basic NETSIM modules are described below:

Multiplexers: These are multi-input, single-output modules use to merge data. A multiplexer is the module that resolves conflicts when two of its input terminals

have data to transfer to its output terminal at the same time. This arbitration is currently implemented as a semaphore with a FIFO queuing discipline.

Demultiplexers: These are single-input, multi-output modules used to route data along one of several paths. This module is the one that implements the routing mechanism of a network. It uses data in the head flit of a packet at its input terminal to select which one of its output terminals the packet will pass through. The user can specify the routing algorithm used to make this choice.

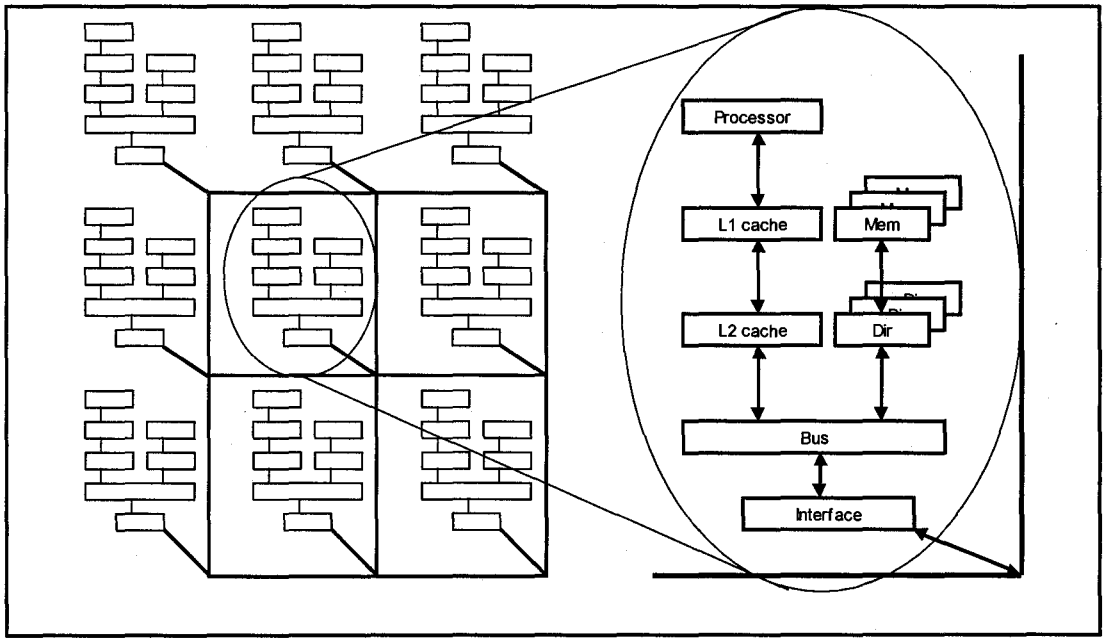


Figure 3.20: Parallel System Architecture [8].

Buffers: These are modules used to provide temporary storage for flits (8 bytes, a subset of a packet) as they move through a network. They are currently implemented as finite FIFO queues. The user can specify the maximum number of flits a buffer can hold.

Network Ports: These are single-input, single-output modules used as interface between a network and its external environment. There are two types of network ports. Input ports provide an interface through which packets are passed into the network, and packets are removed from the network through output ports.

Symbols that can be used to represent these modules in a network diagram are shown in Figure 3.21.

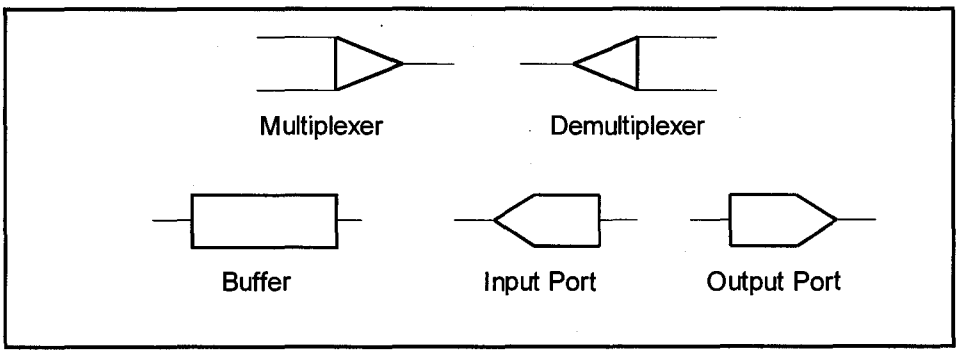


Figure 3.21: Basic network Module Symbols [9].

The processor connection to the network for each node is depicted in Figure 3.22. Similar components connect the node to the interconnection network and neighboring processors along the X and Y axes. Messages are injected into the network using the *SmnetSend* function.

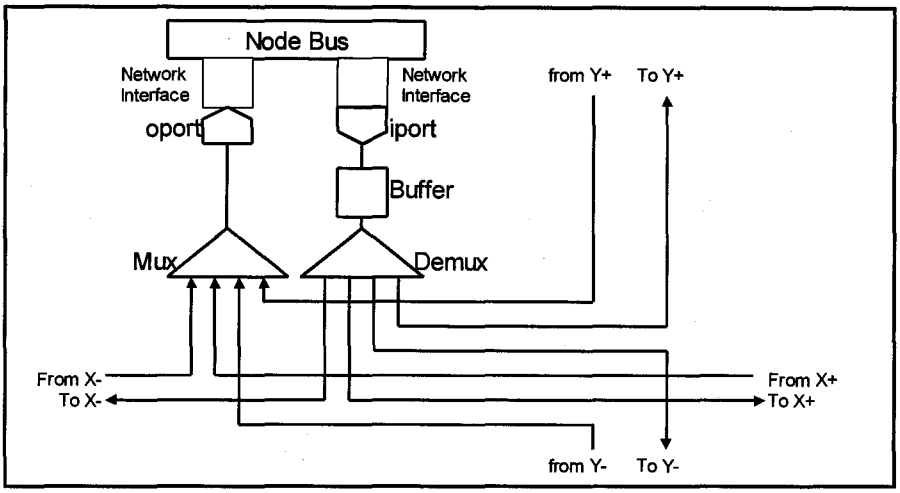


Figure 3.22: Processor side 2D-mesh switch.

The node bus avoids deadlock by accessing subsequent bus agents in a round-robin order. If the desired target of a transaction is not available, the bus does not allow a new transaction to progress beyond the arbitration. Thus, no access is allowed to stall the bus. The interconnection network avoids deadlock through three design decisions. First, replies and requests are sent separately. Second, replies are guaranteed to be processed by the other modules in a finite amount time. Finally, the directory controller can reject the request. This prevents a transaction from starting if the directory controller cannot guarantee that it has buffer space for the reply.

3.2.6 Statistic Tool

RSIM displays the total execution time and the IPC (instruction per cycle) for programs. In order to better characterize the performance, the total execution time is further categorized into busy time and stalls due to various classes of instructions. These classes of instructions include ALU, FPU, data reads, data writes, exceptions,

branches, synchronization, and up to 9 user-defined aggregate classes. Data read and write stalls are further split according to the level of the memory hierarchy at which the memory operations were resolved, i.e., L1 cache, L2 cache, local memory, or remote memory.

For processor statistics, RSIM provides statistics on the branch prediction behavior, the occupancy of the active list, and the utilization of various functional units. RSIM also provides statistics related to the performance of the instruction fetch policy according to the metrics of availability, efficiency, and utility.

For cache, network, and memory statistics, RSIM classifies memory operations at various levels of the memory hierarchy into hits and misses. Misses are further classified into cold, conflict, capacity, and coherence misses. Each data access to the cache calls *StatSet* to specify whether the access hit or missed, and the type of miss in the case of a miss. Statistics on the average latency of various classes of memory operations, MSHR occupancy, and prefetching effectiveness are also provided by default. RSIM also provides statistics on bus utilization, write-buffer utilization, and network contention, traffic, the usage of the network switch buffers, and the types of packet sent in the network. All of the above statistics are generated in *err* and *state* files by RSIM. RSIM also provides shell-scripts and *awk* scripts to process the statistics (*err* and *state* files) produced by the simulator. These are useful for collecting information about the behavior and performance of applications being simulated. The output of the utilities is described below.

- The *stats* and *pstats*

The *stats* gathers the concise statistics from *err* files, which are generated by RSIM. The *stats* takes as input the *phase* of interested section of an application program. The *pstats* gives a more detailed categorization of the memory component of the execution time according to the level of the memory hierarchy at which each

access was resolved. For example, to condense the phase 0 section of an application, e.g., matrix multiplication, one would see the results shown in Figure 3.23.

```
jennfier ~/link/apps/QS/testoutputs 217% pstats ptmm_err 0
Phase 0 STATS
FILE ptmm_err
Total elapsed time of simulation 11.000000
Simulated time 43295.500000
Exec time   IPC     Busy    Acq     Rel     Read    Write   RMW     Spin    Barrier
4.33e+04    0.348   48.01   0.00    0.00    12.84   0.68    31.80   0.00    0.00
USER:   1       2       3       4       5       6       7       8       9
USER:   0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
        Read L1 =       2.09        Write L1 =      0.89        RMW L1 =        0.01
        Read L2 =       0.23        Write L2 =      0.00        RMW L2 =        2.39
        Read local =    0.50        Write local =   0.00        RMW local =     0.00
        Read remote =   10.02       Write remote =  0.00        RMW remote =    29.40
LAT:    READ    WRITE   RMW
Addr:   59.63   92.26   788.24
Issue:  59.08   82.27   128.65
Active: 64.67   98.56   789.89
Busy = 4.92  ALU = 37.34  FPU = 0.25  BRU = 5.50
READ MISS = 10.75  WRITE MISS = 0.00  RMW MISS = 31.79
Avail = 0.537  Eff = 1.205  Util = 0.518
```

Figure 3.23: The *pstate* utility result with *matrix multiplication(ptmm)*.

The statistics displayed is "Exec time" and its various components. All of components are straightforward to understand. "Total elapsed time of simulation time" is actual elapsed time for the simulation in second. "Simulated time" is from the function *GetSimTime()* which returns the current simulation time (total clock cycle). "Exec time" is same as the "Simulated time". Instruction per cycle, "IPC", is calculated with the total number of graduated instruction per cycle, divided by the total clock cycle times number of processors. "Busy" is total of "ALU" time + "FPU" time + "BRU" time (branch prediction unit time) + "Busy" (pipeline stall time (e.g., RAW)). The last three information (Avail, Eff, and Util) are defined as follows:

- Avail: The total number of fetch slots in a given run is the fetch bandwidth times the number of cycles in the run. For example, if the fetch bandwidth is 8, and a benchmark runs for 1,000 cycles, then there are 8,000 fetch slots in the run. Out of these 8 fetch slots each cycle, some cannot be filled because the instruction window is full. The remaining fetch slots are known as the *available fetch slots*. Fetch availability is defined as the ratio of available fetch slots to the total number of fetch slots.
- Eff: Even when it has an available fetch slot, the fetch unit may not be able to fill it. For example, the fetch unit might be stalled waiting for a branch to be resolved. *Fetch efficiency* is defined as the ratio of the number of instructions fetched to the number of available fetch slots.
- Util: Of the fetched instructions, not all of them are necessary to the execution of the program. They might have been fetched speculatively, and never executed. *Fetch utility* measure how appropriate or useful the fetched instructions were. It is defined as the ratio of the number of instructions executed in a non-speculative model to the number of instructions fetched in a given run.

To display all this information into a graph, RSIM provides another sed/awk script, called *plotall*.

- The *plotall*

The *plotall* converts the statistics generated by *pstats* into a form that can be fed to the *splot*[1]. Then, splot creates a plot similar to one shown in Figure 3.24.

---

[1] Plot utility for Encapsulated PostScript and X (ftp://cag.lcs.mit.edu/pub/splot/)

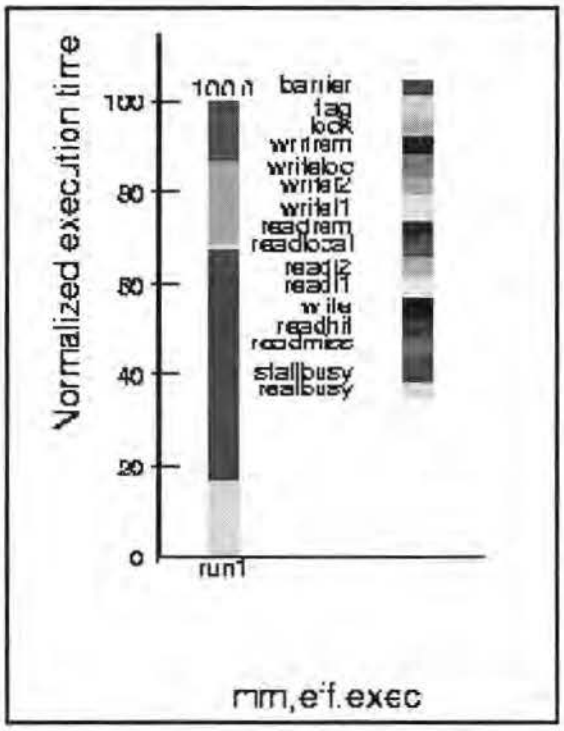


Figure 3.24: The output from *plotall*.

To see statistics of the read misses, RSIM provides the script called *stats_miss*.

- The *stats_miss*

This script generates total number of read misses (Misses), the average absolute read miss latency from the point of address generation (Miss latency), the average read miss latency overlapped by the ILP processor between the point of address generation and graduation (Overlapped miss latency), and the statistics about the number of references to various levels of the cache hierarchy (Refs, L1refs, and L2refs). Also, *stats_miss* includes the average absolute latency beyond the L2 cache seen by read misses in the system (Average network read latency). Figure 3.25 shows an example of an output from *stats_miss* script.

```
Phase 0 STATS
FILE ptmmrun2
Misses: 914
Miss latency: 37.32
Overlapped miss latency: 7.27
Fraction unoverlapped: 78.42\%
Refs: 219736
L1refs: 219736
L2refs: 935
L1hits: 99.50 (218638)
L1misses: 0.43 (935)
L2hits: 78.07 (730)
L2misses: 19.68 (184)
Local mems: 2.17 (4)
Remote mems: 97.83 (180)
Average load queue = 0.529695
Average L1 MSHR = 0.009
Average L2 MSHR = 0.011
Average network read latency = 134.530 (over 149)
```

Figure 3.25: The output from *stats_miss* with *matrix multiplication.*

4 IMPLEMENTIONS OF SOFTWARE-CONTROLLED MULTITHREADING

The implementation of pthread is rather complicated and relies heavily on operating system (OS) support. When a process runs, the OS sets up a location for the potential state of the process on the OS stack. When a interrupt (a software trap) occurs, the state of the process is saved onto the stack, increments the stack pointer and the processor runs the software trap handling routine. When the routine finishes and returns from the software trap, the OS pops the process's state (the one above the stack pointer) back off the stack, decrements the stack pointer, and the process returns to execution. However, RSIM does not have the OS to support software traps (i.e., Tcc), while the Pthreads package needs software traps often. This chapter presents how software trap is implemented. Also, the global variables in the Pthreads package must be allocated in a shared region for other processors to access the global variables. *Shmalloc* function is only way to allocate the global variables into shared area in RSIM. This chapter shows how the global variables are allocated. The lock of RSIM also has a problem when an executable ready thread can not acquire a lock because an idle main thread keep holding the lock and not releasing the lock. This chapter also discusses about newly implemented Lock mechanism (Ticket Lock) to resolve this problem.

4.1 Software Traps

OS is responsible for handling complex operations required during execution of a program. Example of these is memory exception handling or interrupt handling. When a situation requires special handling, a trap occurs. A trap can be caused by an exception brought about by the execution of an instruction, i.e., an explicit trap

instruction in the running software. This is called a software trap. This section discusses the reasons software traps are needed and how this mechanism works in more detail.

When a software trap occurs, *C* library function, *setjmp* and *longjmp*, are used in the Pthreads package to save and restore the state of a thread. *setjmp()* saves its stack environment in a jump buffer argument for later use by *longjmp()*. *longjmp()* restores the environment saved by the last call of *setjmp()* with the corresponding jump buffer argument. Figure 4.1 shows an example of how *setjmp()* and *longjmp()* are used.

These functions generate an interrupt (a software traps). Hardware will trap always (*ta*) whenever they are used. *FLSUHW* (Flush Register Windows) instruction is used to take care of this trap situation. *FLUSHW* causes all active register windows, except the current window, to be flushed to memory.

```
/*  machdep_save_state()  */
int machdep_save_state( struct machdep_pthread *machdep_date)
{
        return setjmp(machdep_data->machdep_state);
}
/*  machdep_restore_state()  */
int machdep_restore_state(struct machdep_pthread *machdep_data)
{
        longjmp(machdep_data->machdep_state, 1);
}
```

Figure 4.1: *setjmp()* and *longjmp()* usage in the Pthread package.

Unfortunately, RSIM dose not support *FLUSHW* instruction. *FLSUHW* acts as a no operation (NOP) if Savable Register Windows (*CANSAVE*) = The Number of Register Windows (*NWINDOWS*) - 2. Otherwise, there is more than one active window, so *FLUSHW* causes an overflow exception (UltraSparc, Version 8 uses the

term "overflow" and the Version 9 uses the term "spill", which are basically the same). RSIM is based on the UltraSparc Version 8 as far as FLUSHW instruction is concerned. The overflow trap handler is invoked with the current window pointer (CWP) set to the window to be overflowed (i.e., (*CWP* - *CANSAVE* -2) mod *NWINDOWS*). Figure 4.2 through 4.6 shows an example of *FLUSHW*.

Initially in Figure 4.2, The *CWP* is 3, CANSAVE is 2, and the next CWP to be flushed is 6 calculated by the given formula. The next CWP's are calculated iteratively until all the used register windows are flushed.

To implement above all operation, *traps.h*, *state.h*, *table.h*, *instance.h*, *except.cc*, *funcs.cc*, and *traptable.cc* files are modified in RSIM.

For example, if Tcc instruction causes a software trap, in other word, *ta* (i.e., trap always) instruction is selected, a software trap is called in the exception handler function. In the exception handler, all instructions before this exception are completed for a precise interrupt. Branch queue, speculative memory operation, and stall queue are flushed. Then, *TrapTableHandle* function is called. In *TrapTableHandle* function, *TRAPTABLE_FLUSHW_RETRY* and *TRAPTABLE_FLUSHW_DONE* are used until all the register windows are flushed.

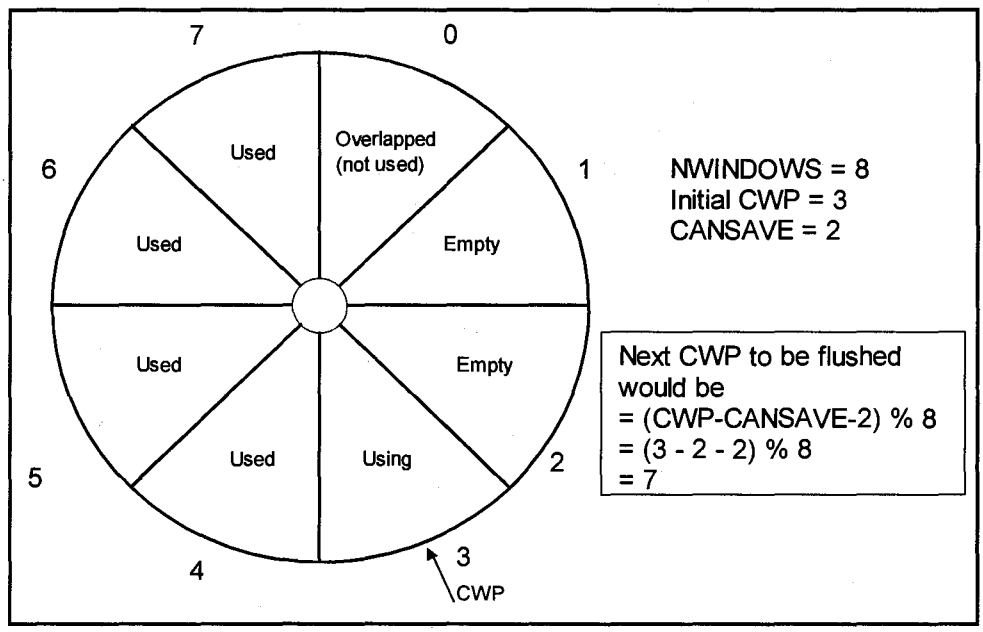


Figure 4.2: The *FLUSHW* operation(STEP 1)

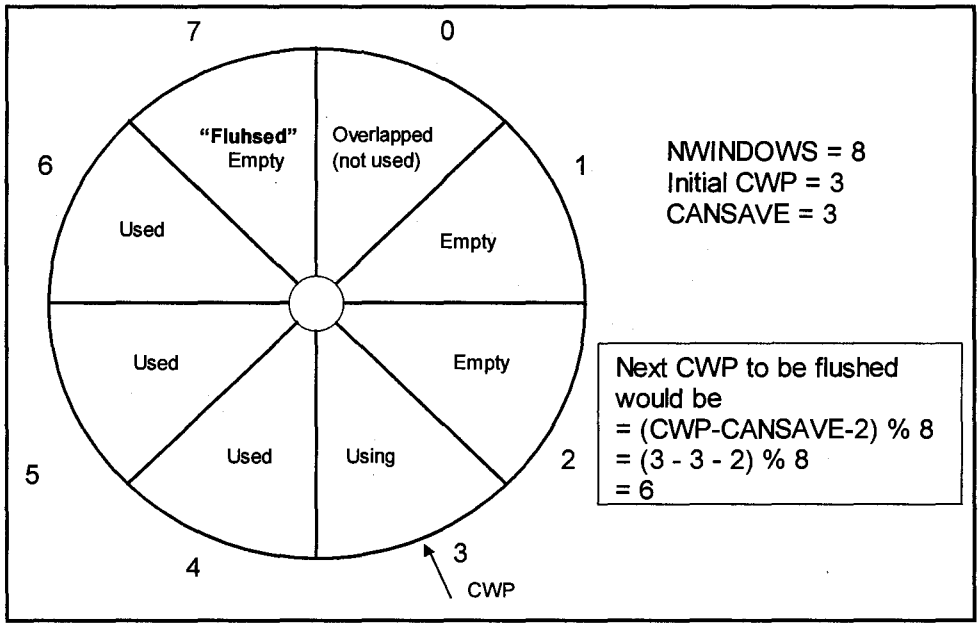


Figure 4.3: The *FLUSHW* operation (STEP 2)

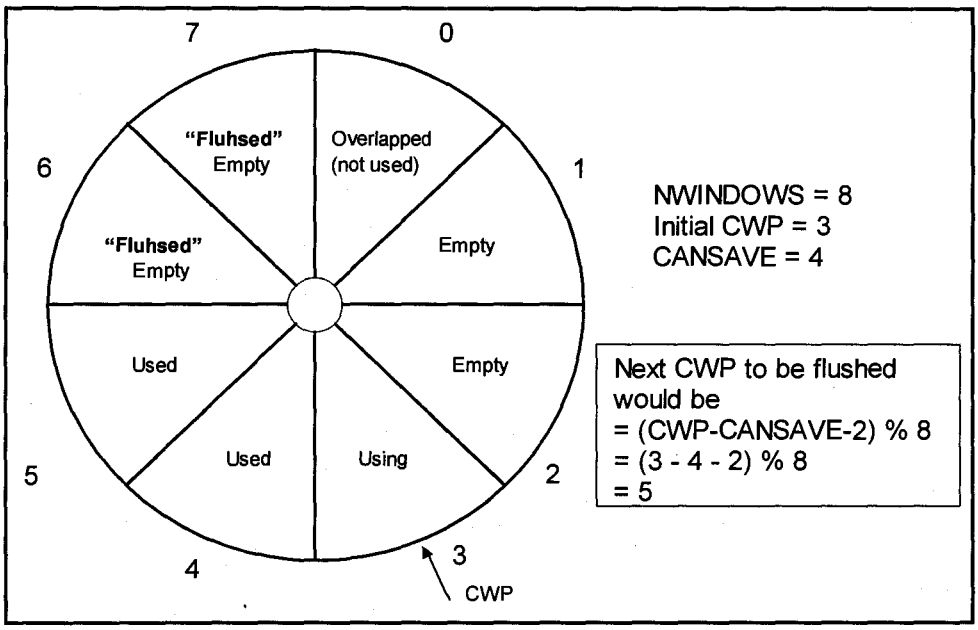


Figure 4.4: The *FLUSHW* operation (STEP 3)

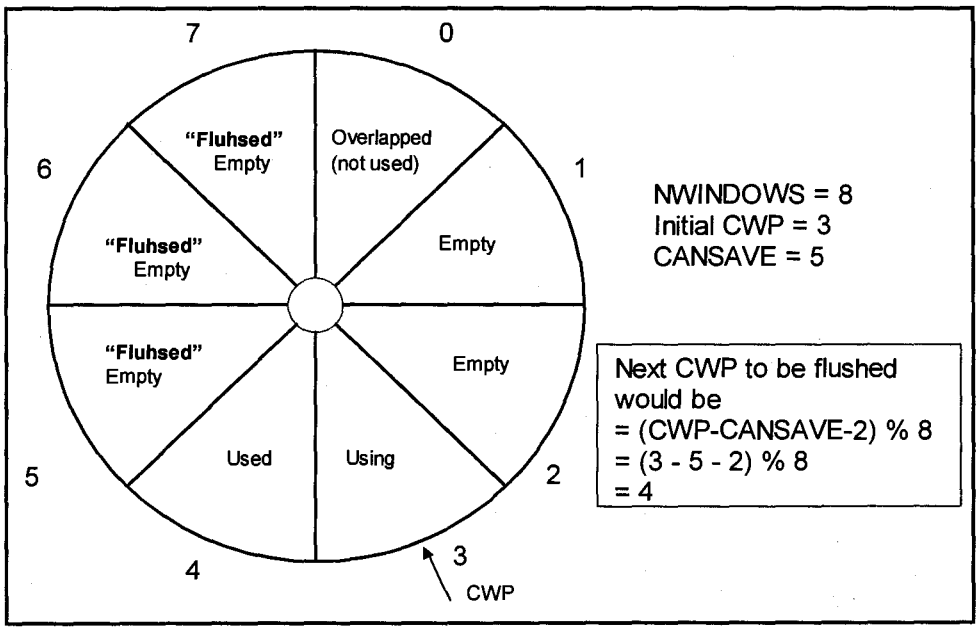


Figure 4.5: The *FLUSHW* operation (Step 4)

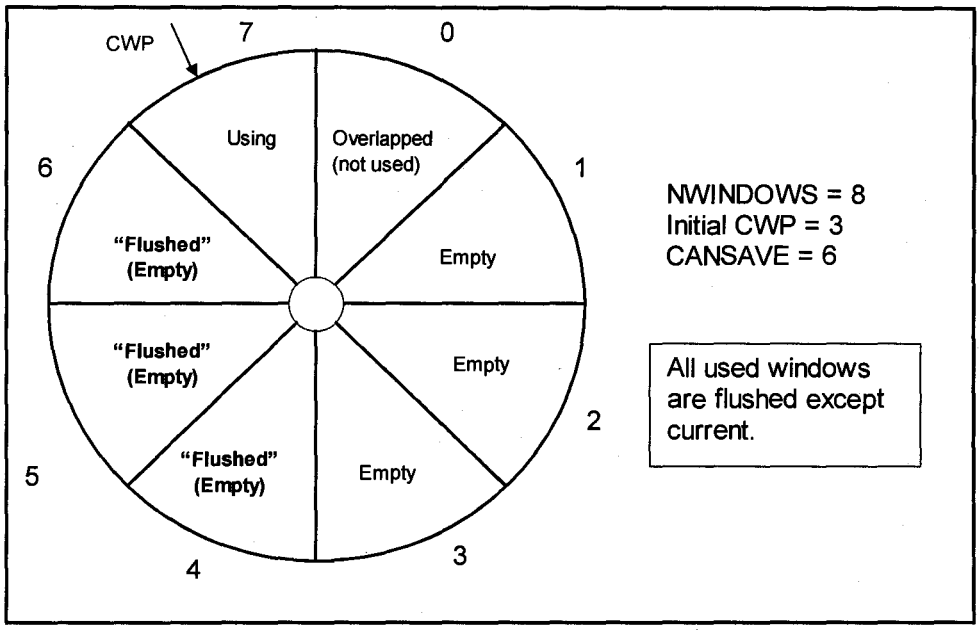


Figure 4.6: The *FLUSHW* (Final Step)

4.2 Shared Memory Allocation

The global variables in the Pthreads package must be allocated in a shared region. Since the only way to allocate shared memory is through the *shmalloc* function, these global variables are *shmalloc*'ed in each application. Figure 4.7 shows what variables are defined in the shared memory region.

Figure 4.8 shows how and where to allocate these global variables. The Global variable pointer, **gMemp,* is *shmalloc*'ed in the last line of Figure 4.8. This procedure must be done in every applications; otherwise, the global variables in the Pthreads package will not be allocated in the shared memory region.

```
Typedef struct gMempthread{
/* Globals */
      struct pthread              *pthread_link_list;
      struct pthread_queue         pthread_dead_queue;
      struct pthread_queue         pthread_alloc_queue;
      struct pthread               initial_thread[MAX_CPUS];
      volatile int                 cur_init_thread;
      int                          pthread_iolock;
      pthread_self_t               pthread_self_list[MAX_CPUS];

/* Globals only the internals should see */
      struct pthread_prio_queue    pthread_current_prio_queue;
      semaphore                    kernel_lock;
      semaphore                    kernel_ticket;
      int                          pthread_fatal;
      volatile int                 pthread_done;
} gMempthread;

      gMempthread *gMemp;
```

Figure 4.7: Global variables in the Pthread package (pthread.h)

```
Int
main(int argc, char **argv){
 extern char *optarg;
 int ch, i, res;
 float trace = 0.0;
      while ((ch = getopt(argc, argv, "n:p:t:voh")) != -1) {
            switch(ch) {
                  case 'n': n = atoi(optarg); break;
                  case 't': T = atoi(optarg); break;
                  case 'p': P = atoi(optarg); break;
                  case 'v': test_result = 1; break;
                  case 'o': doprint = 1; break;
            }}

      printf("Starting Simulator!\n");
      StatClearAll();       /* Clear the stats (RSIM) */

      gMemp = (gMempthread )shmalloc(sizeof(gMempthread)); /* Here gMemp is
                                                              shmalloc'ed */
      Memset(gMemp, 0, sizeof(gMempthread));
```

Figure 4.8: Global variable defined are *shmalloc*'ed in *PTMM* application.

4.3 Lock Mechanism

4.3.1 The test-and-*test_and_set* Lock (RSIM)

The simplest mutual exclusion lock, found in all operating systems and widely used in practice, employs a polling loop to access a Boolean flag that indicates whether the lock is held. Each processor repeatedly executes a *test_and_set* instruction in an attempt to change the flag from false to true, thereby acquiring the lock. A processor releases the lock by setting it to false. The principal shortcoming of the *test_and_set* lock is contention for the flag. Each waiting processor accesses the single shared flag as frequently as possible, using relatively expensive *read-modify-write (RMW)* instructions. This results in degraded performance, not only of the memory bank in which the lock resides, but also from the processor/memory interconnection network side as well. *RMW* instructions can be particularly expensive on cache-coherent multiprocessors, since each execution of such an instruction may cause many remote invalidations. To reduce this overhead, the *test_and_set* lock can be modified to use a *test_and_set* instruction only when a previous read indicates that the *test_and_set* might succeed. This so-called test-and-*test_and_set* technique (RSIM uses this technique) ensures that waiting processors poll with read requests during the time that a lock is held. Once the lock becomes available, some fraction of the waiting processors detect that the lock is free and perform a *test_and_set* operation. When one processor succeeds, it causes remote invalidations on other shared copies.

4.3.2 The Ticket Lock

In a test-and-*test_and_set* lock, the number of *RMW* operations is substantially less than for a simple test_and_set lock, but still results in a large number of *RMW* operations. Specifically, it is possible for every waiting processor to perform a *test_and_set* operation every time the lock becomes available, even though only one can actually acquire the lock. We can reduce the number of *RMW* operations to one per lock acquisition, using a ticket lock [6].

A ticket lock consists of two counters, one containing the number of requests to acquire the lock, and the other the number of times the lock has been released. A processor acquires the lock by performing a **fetch_and_increment** operation on the request counter and waits until the result (i.e., its ticket) is equal to the value of the release counter. Pseudo-code for a ticket lock is shown in Figure 4.9. It assumes that the *new_ticket* and *now_serving* counters are large enough to accommodate the maximum number of simultaneous requests for the lock. The actual assembly language implementation is shown in Figure 4.10.

```
Type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : lock)
    my_ticket : unsigned integer := fetch_and_increment (&L->next_ticket)
    /* returns old value; arithmetic overflow is harmless */
    loop
        pause (my_ticket - L->now_serving)
    /* consume this many units of time */
        if L->now_serving = my_ticket
            return

procedure release_lock (L : lock)
        L->now_serving := L->now_serving +1
```

Figure 4.9: The Ticket Lock Pseudo-code.

```
        .global pthread_aquire
        .type   pthread_aquire,#function
pthread_aquire:
        unimp 0x100c          /*  Start aggregate lock acquire type latency time computation  */
retry:  ld [%o1],%o2          /*   Load a ticket o1 to o2   */
        add %o2,1,%o3         /*   Increment the ticket   */
        cas [%o1],%o2,%o3  /* Load the ticket to o3, "cas" is
                              atomic and guarantee instruction  */
        cmp %o2, %o3          /* Make sure the acquired ticket is one
                              acquired before (blocking race) */
        bne retry             /*   If not, go to retry   */
again:  ld [%o0],%o2          /* Load lock o0 to o2   */
        cmp %o3, %o2          /*  Are o3 (ticket) and o2 (lock) same?  */
        bne  again            /*   If not, try another lock   */
        membar #LoadLoad | #LoadStore   /*   All loads prior to membar must be performed
                                          before the effect of load or store following the
                                          membar is visible to any other processor.   */
        retl                  /*   Return to main program   */
        unimp 0x1000          /*  End aggregation of acquire latency time computation  */

        .global pthread_release
        .type   pthread_release, #function
pthread_release:
        unimp 0x100d                /*   Start aggregate lock release type latency time
computation   */
rep:    ld [%o0], %o1               /*   Load lock o0 to o1   */
        add %o1,1,%o2               /*   Increment lock   */
        cas [%o0],%o1,%o2           /*   Load current lock to o2   */
        cmp %o1,%o2                 /* Is the lock is same as the lock
                                      acquired just before (blocking race ) */
        bne rep
        membar #LoadLoad | #LoadStore
        retl
        unimp 0x1000              /*  End aggregation of acquire latency time computation  */
```

Figure 4.10: The Actual Ticket Lock

5 PRELIMINARY SIMULATION RESULTS

This chapter describes the architecture parameters, the benchmark application, and preliminary results.

5.1 Simulated Architectures

CC-NUMA with 1, 2, and 4 processor connected by a two dimensional mesh has been modeled. The systems use an invalidation coherence protocol and are release-consistent.

Following details the structure of the processors and the memory hierarchy. Table 5.1 summarize the simulated system parameters. L1 and L2 cache size and the cache line size follows the SGI *Origin 2000* (CC-NUMA server) [23].

- Processor Model

The processor resembles the MIPS 10000 processor [5], with 4-way issue, dynamic scheduling, non-blocking reads, register renaming, and speculative execution. Unlike the MIPS 10000, however, release consistency is implemented.

- Memory Hierarchy

Each system node includes a processor with two levels cache, a merging write buffer between the caches, and a portion of the distributed memory and directory. A split-transaction system bus connects the memory, the network interface, and the rest of the system node. L1 cache has 2 request ports, allowing it to serve up to 2 data requests per cycle, and is write-through with a no-write-allocate policy. The L2 cache has 1 request port and is a fully-pipelined, write-back cache with a write-allocate policy. Each cache also has an additional port for incoming coherence messages and

replies. Both L1 and L2 caches have 8 miss status holding register (MSHRs) [4] for prefetching.

Processor	
Processor speed	300MHz
Maximum fetch/retire rate (instruction per cycle)	4
Instruction issue window	64 entries
Functional units	2 integer arithmetic 2 floating point 2 address generation
Branch speculation depth	8
Memory unit size	32 entries
Cache parameters	
Cache line size	128 bytes
L1 cache (on-chip)	32K bytes, 2-way set-associative
L1 request ports	2
L1 hit time	1 cycle
Number of L1 MSHRs	8
L2 cache (off-chip)	4M bytes, 4-way set-associative
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of L2 MSHRs	8
Write buffer entries	8 cache lines
Memory parameters	
Memory access time	75 processor cycles (250ns with 300MHz)
Memory Interleaving	4-way
Network parameters	
Network width	64bits
flitdelay, arbdelay	236 cycles, 4 cycles (300MHz processor)
Network model	100Mbit/s ethernet

Table 5.1: System parameters

5.2 The Benchmark

The Matrix Multiplication (*MM*) program was developed to study the performance of the Multithreaded RSIM. *MM* is manually written to be multithreaded using Pthreads library calls. Also, no optimizations were attempted when converting the serial version to the multithreaded version. *MM* 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.

5.4 Simulation Results

Two sets of simulation runs were performed for the *MM* benchmark. The first set was obtained by running a single thread version of the benchmark and the second set was obtained by running multithreaded version for various number of processor. Approximately 26 - 27 million instructions were simulated with the number of memory references ranging from 6 - 7 million. The number of threads created are four, eight, and twelve. The number of simulated processors are one, two, and four. Figure 5.1 shows the portion of the memory execution time due to the pipeline stalls (realbusy), the processors memory units (ALU, FPU, and BRU) stall time (stallbusy), L1 and L2 access time (readl1 and readl2), the main and remote memory access time (readlocal and readrem), the synchronization time (lock) and finally the context switch cost (cscost). The context switch cost becomes more negligible as the size of the matrix becomes larger. Figure 5.2 shows the weight simulation time with the matrix size of 32×32.

Figure 5.3 shows normalized simulation result with the matrix size of 100 × 100. Except the last graph 12 thread version, the *cscost* and *lock* time become so small that it is hardly visible in the *cscost* and *lock* times in the Figure. Figure 5.4 shows the weighted simulated time with the matrix size of 100 × 100. It can be seen that the context switch cost incurred with the threaded version is about the same as the total execution time with the single thread version. The remote memory access time becomes much larger as the matrix size increases.

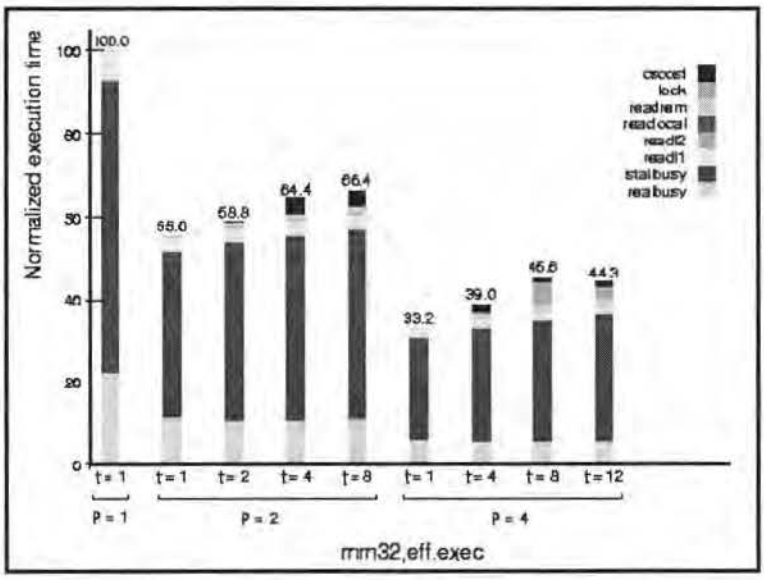


Figure 5.1: The *MM* normalized execution simulation result (32x32 matrix)

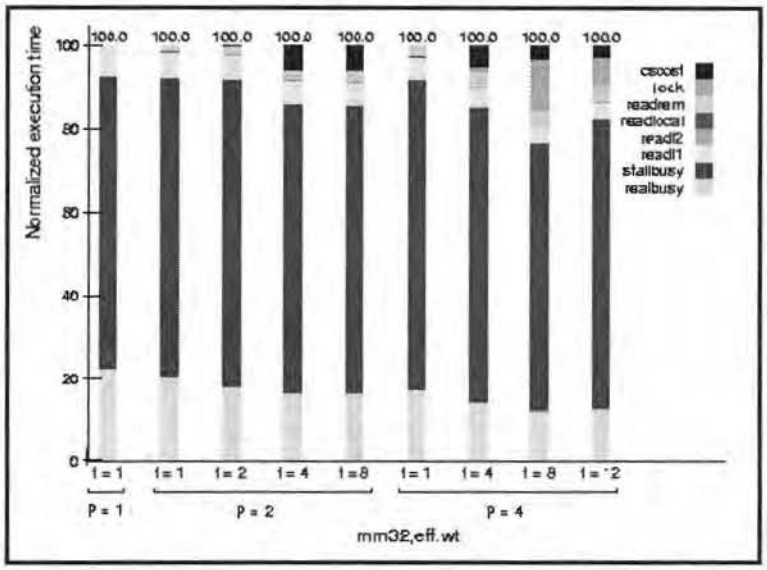


Figure 5.2: The *MM* normalized weighted simulation result (32x32 matrix)

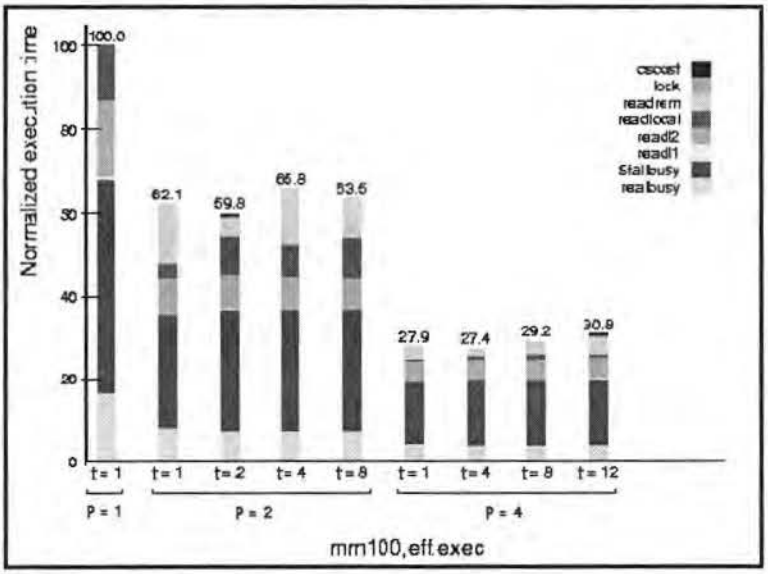


Figure 5.3: The *MM* normalized execution simulation result (100x100 matrix)

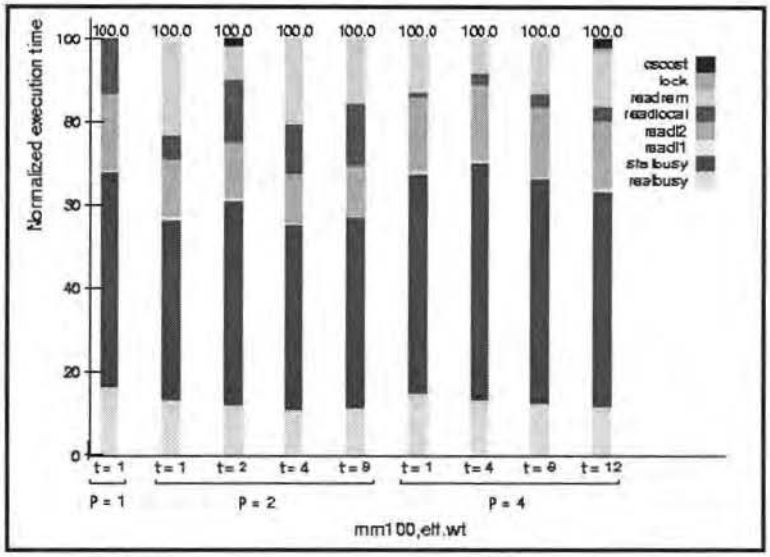


Figure 5.4: The *MM* normalized weighted simulation result (100x100 matrix)

6 CONCLUSION AND FUTURE WORK

As the performance gap between processor and memory grows, memory latency becomes a major bottleneck in achieving high processor utilization. *Multithreading* has emerged as one of the most promising techniques used to tolerate memory latency.

This thesis discussed the simulator implementations for realizing a software-controlled *Multithreading* environment. RSIM was chosen as the base simulator because RSIM has the modest complexity and is relatively well documented. The features of RSIM were introduced. The implementation details for software-controlled *Multithreading* capability was explained in detail.

The software trap for *setjmp()* and *longjmp()* functions in the Pthread package was implemented. How the Pthread global variables must be treated for the shared memory model was also discussed. More scalable lock mechanism (ticket lock) was introduced and implemented for the simulator. The preliminary simulation study with the matrix multiplication showed that context switch overhead compared with the remote memory access overhead (*readrem*) becomes negligible when the size of matrix increases. This implies the hardware-controlled context switch capability when read misses occurs in main memory can be used to tolerate or hide the memory latency.

The complete simulator, called Multithreaded Virtual Processor (MVP), is able to use main memory-misses to switch between up to several threads. The overview of the complete MVP is shown in Figure 6.1.

The proposed hardware for multithreading is based on Prof. Lee's previous research on Multithreaded Virtual Processor (MVP) [16] that consists of the MIPS R10000 processor core augmented with the *Hardware Scheduler* and *Multiple*

Hardware Contexts. Each hardware context includes a valid bit (V), a ready bit (R), a thread identifier (TID), a program counter (PC), and a Register File.

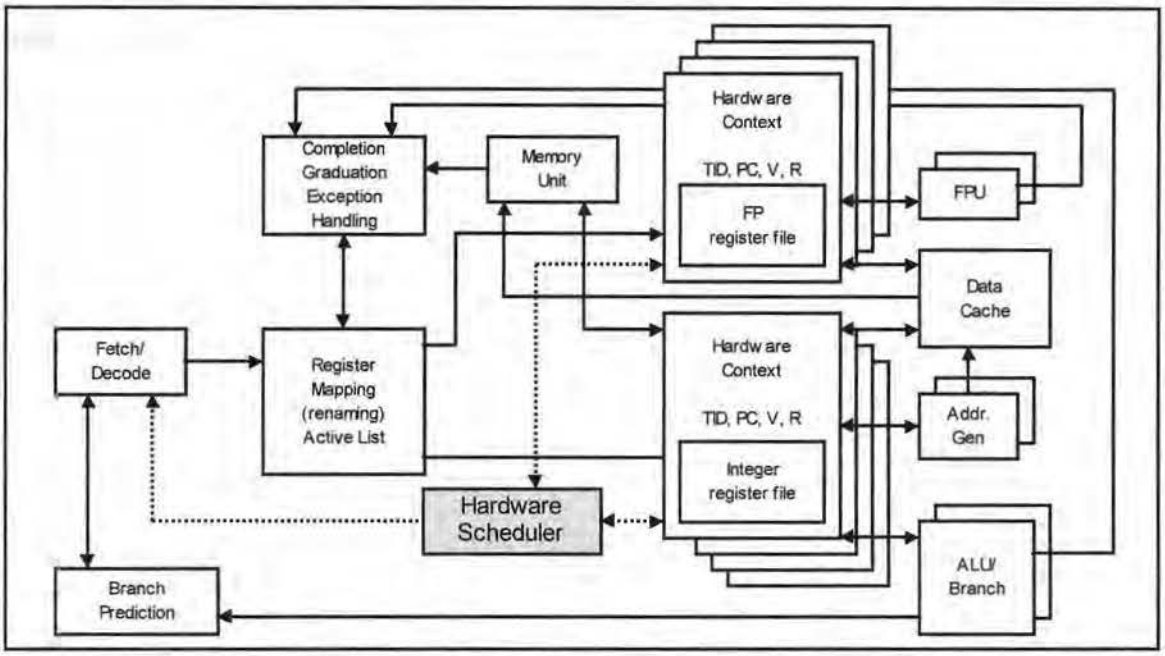


Figure 6.1: An overview of the complete MVP system.

The responsibility of the Hardware Scheduler is to basically maintain the control of thread states that have been scheduled onto MVP and provide fast context switching. When a main memory miss is detected by the memory management unit, MVP initiates a hardware context-switch. After the context-switch is performed, new instructions are fetched from the memory location indicated by PC of the new hardware context. The complete simulator, which has multiple hardware contexts and hardware context switch capability can hide this remote memory by switching to other threads when main memory misses occur. This another thread to be run while the outstanding read miss pending, thereby tolerating long memory latency.

BIBLIOGRAPHY

[1] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. 1997. RSIM Reference Manual. Version 1.0. *Department of Electrical and Computer Engineering, Rice University. Technical Report 9705.*

[2] John L. Hennessy, and David A Patterson. 1996. Computer Architecture a Quantitative Approach. 2d ed. San Francisco: Morgan Kaufmann Publishers, Inc.

[3] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, 1999. Parallel Computer Architecture. San Francisco: Morgan Kaufmann Publishers, Inc.

[4] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. 1981. *In proceedings of the 8th International Symposium on Computer Architecture* 28:1-87.

[5] C. Holt, J. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. *In Proc. 23rd Int'l Symp. On computer Architecture*, pages 134-145, May 1996.

[6] J.M. Mellor-Crummey, and M. L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM transactions on Computer Systems* 9(1).

[7] J. Robert Jump. 1993. *NETSIM Reference Manual.* Rice University Electrical and Computer Engineering Department.

[8] MIPS Technologies, Inc. 1996. *R10000 Microprocessor User's Manual*, Version 2.0.

[9] Sparc International. 1993. *The SPARC Architecture Manual*, Version 9.

[10] Sun Microsystems Inc. January 1991. *The SPARC Architecture Manual*, No. 800-199-12, Version 8.

[11] J. Robert Jump. 1993. *YACSIM Reference Manual.* Rice University Electrical and Computer Engineering Department.

[12] Jim Moore. Sun Microsystems Inc. 1997 SPARC traps under SunOS, Version 1.01. Available at http://sunsite.nus.edu.sg/sun/SparcTrap.

[13] Jian Huang, and David J. Lilja. 1998. An Effiecient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture. HPPC.

[14] Anthony-Trung Nguyen, Maged Michael, Arun Sharma, and Josep Torrellas. 1996. *The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures*. IEEE

[15] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anop Gupta. 1995. *Complete Computer System Simulation: The SimOS Approach*. IEEE Parallel and Distributed Technology.

[16] Ben Lee. 1999. MVPsim. Available at http://www.ece.orst.edu/~benl/Research/MVP/mvp.html

[17] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. 1997. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. *In Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72-83.

[18] Kenneth C. Yeager. 1996. The MIPS R10000 Superscalar Microprocessor. *IEEE micro*, 16(2):28-40.

[19] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. 1996 An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP processors. *In proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operationg Systems*, pages 12-23.

[20] John M. Mellor-Crummey, and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.

[21] Hantak Kwank, Ben Lee, Ali R. Hurson, Suk-Han Yoon, and Woo-Jong Hahn. 1999. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*. Vol 48, No. 2.

[22] Ben Lee, Hantak Kwak, Ryan Carlson, Suk-Han Yoon, and Woo-Jong Han. 1998. Simulation Study of Multithreaded Virtual Processor. *In IASTED International Conference on Parallel and Distributed Systems* (Euro-PDS). Vienna, Austria.

[23] SGI inc. 1999. Datasheet, Scalable Server for Solutions Demanding High Performance and I/O Bandwidth. Origin2000.

[24] T. Mowry. 1994. "*Tolerating Latency through Software-Controlled Data Prefetching*," doctoral dissertation, Computer Systems Lab., Stanford Univ., Stanford, Calif.

[25] F. Dahlgren, and P. Stenstrom. 1996. "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, pp. 385-398.

[26] S.V. Adve, and K. Gharachorloo. 1996. "Shared Memory Consistency Models: A Tutorial," *Computer*, pp. 66-76.

[27] R. Alverson et al. 1990 "The Tera Compute System," Proc. Int'l Conf. Supercomputing, IEEE CS Press, Los Alamitos, Calif. pp. 1-6.

[28] Jack E. Veenstra, and Robert Fowler. 1993. Mint Tutoral and User Manual. Technical Reoprt 452, University of Rochester.

[29] P. Stenstrom, 1990. "A Survery of Cache Coherence Schemes for Multiprocessors," Computer. pp. 12-24.

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