URSIM Reference Manual

Version 1.0

Lixin Zhang

UUCS-00-015

Deparment of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

July 25, 1999

Abstract

This document describes URSIM — the **U**tah **RSIM**. URSIM is an execution-driven simulator derived from the Rice Simulator for ILP Multiprocessors and built for the Impulse Adaptive Memory System Project.

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

Contents

Chapter 0

Overview

This document describes URSIM — the **U**tah **RSIM**. URSIM is an execution-driven simulator derived from RSIM (the **R**ice **S**imulator for **I**LP **M**ultiprocessors) [9]. As a result, this document is an extended version of the RSIM manual [9].

URSIM is built for the Impulse Adaptive Memory System Project. This document assumes that the users have a fair understanding of the Impulse technology. Impulse documents [3, 4, 18, 23] give a good overview of the Impulse technology and terminology.

0.1 Key features of simulated systems

URSIM provides many configuration parameters to allow the user to simulate a variety of multiprocessor and uniprocessor architectures. Key features supported by URSIM are:

Processor features:

- Multiple instruction issue
- Out-of-order (dynamic) scheduling
- Register renaming
- Static and dynamic branch prediction support
- Non-blocking loads and stores

- Speculative load execution before address disambiguation of previous stores
- Superpage-supporting TLB

Memory hierarchy features:

- Two-level cache hierarchy
- Multi-ported and pipelined L1 cache, pipelined L2 cache
- Split-transaction and time-multiplexed system bus
- Impulse main memory controller
- Impulse DRAM backend

0.2 Simulation technique

URSIM is a discrete event-driven simulator based on the YACSIM library [5]. Many of the subsystems within URSIM are activated as events only when they have work to perform. However, the processors 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 every cycle. On every cycle, this event appropriately changes the state of processor's pipeline and processes outstanding cache requests.

0.3 Platforms supported

Currently, URSIM has been tested on the following platforms:

- HP PA-RISC running HP-UX version 9 or 10
- HP PA-RISC running 4.3 BSD
- SGI Power Challenge running IRIX64 6.5
- SunSparc workstations running Solaris 2.5 or above

0.4 Organization of this manual

The remainder of this manual is split into two parts. Part I is the *URSIM User's Guide* and provides information of interest to all users of URSIM. Part II is the *URSIM Developer's Guide* and provides information about the implementation of URSIM for users interested in modifying URSIM. Part II assumes the reader is familiar with Part I.

Within Part I, Chapter 1 explains the simulated architectural model to give the users an overview about what exactly is simulated by URSIM. Users who only use the URSIM to simulate applications and do not care about the details inside the simulator can skip this chapter. Chapter 2 describes how to use URSIM, including checking out and compiling URSIM and porting applications to URSIM. Chapter 3 explains how to configure URSIM to model the system desired, including command line arguments, configuration file, and compile-time options.

Within Part II, Chapter 4 gives an overview of the various subsystems in the URSIM implementation. Chapter 5 explains the YACSIM event-driven simulation library functions used by URSIMĊhapter 6 describes the functions used for initializing and configuring the simulator. Chapter 7 gives an overview of the processor out-of-order execution engine, along with the event that controls processor and cache simulation. Chapter 8 describes the processor memory unit, which interfaces the processor pipelines with the memory hierarchy. Chapter 9 examines the cache hierarchy implementation. Chapter 10 discusses the system bus. Chapter 11 explains the Impulse main memory controller. Chapter 12 describes the Impulse DRAM backend.

Part I

USER'S GUIDE

Chapter 1

Architectural Model

The chapter describes the instruction set architecture, the processor microarchitecture, and the memory system supported by URSIM.

1.1 Instruction set architecture

URSIM simulates applications compiled and linked for SPARC V8Plus/Solaris using ordinary SPARC compilers and linkers, with the following exceptions.

Although URSIM supports most important user-mode SPARC V8Plus instructions, there are a few unsupported instructions. More specifically, all instructions generated by current Sun C compilers for the UltraSPARC-I or UltraSPARC-II with Solaris 2.5 or 2.6 are supported. Unsupported instructions that may be important on other SPARC systems include 64-bit integer register operations and quadruple-precision floating-point instructions. The other unsupported instructions are flush, flushw, and tagged add and subtract [16].

1.2 Processor microarchitecture

URSIM models a processor microarchitecture close to the MIPS R10000 microprocessor [8] and is illustrated in Figure 1.1. The default processor features include:

- Superscalar execution — multiple instructions issued per cycle.

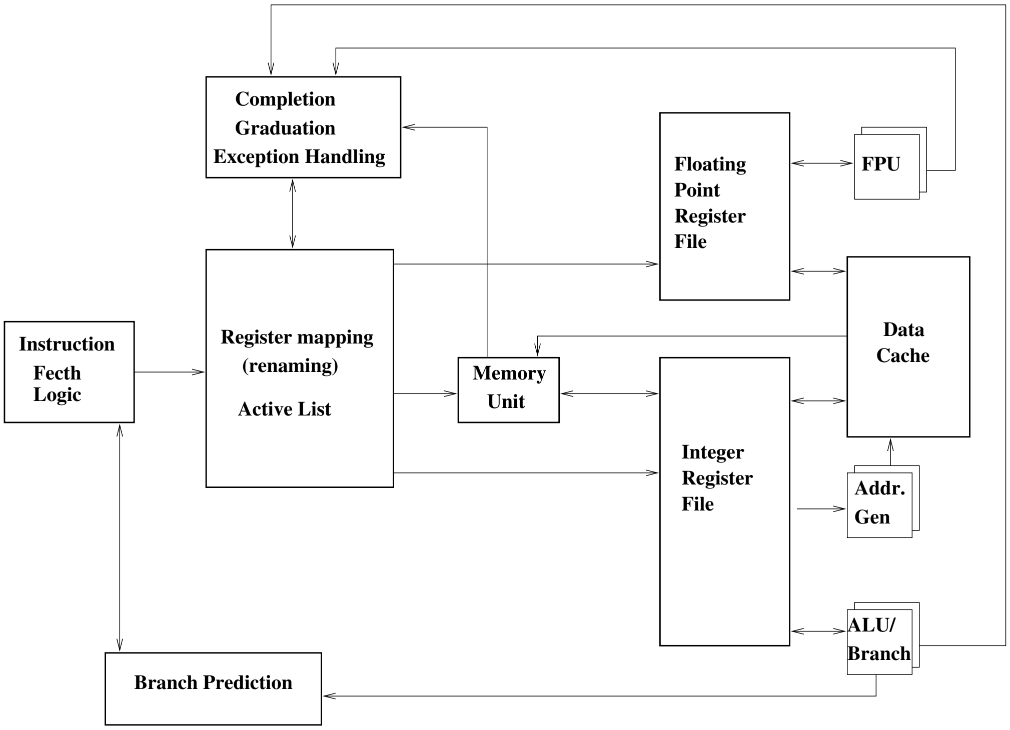

Figure 1.1: URSIM Processor Microarchitecture.

- Out-of-order (dynamic) scheduling
- Register renaming
- Static and dynamic branch prediction support
- Non-blocking memory load and store operations
- Speculative load execution before address disambiguation of previous stores
- Superpage-supporting TLB
- Precise exceptions
- Register windows

In particular, URSIM models the R10000's *active list* (which holds the currently active instructions, corresponding to the *reorder buffer* or *instruction window* of other processors), *register map table* (which holds the mapping from logical registers to physical registers), and *shadow mappers* (which store register map table information on branch prediction to allow single-cycle state recovery on branch misprediction). The instruction pipeline parallels the *Fetch*, *Decode*, *Issue*, *Execute*, and *Complete* stages of the dynamically scheduled R10000 pipeline. Instructions are *graduated* (i.e., *retired*, *committed*, or removed from the *active* *list*) after passing through this pipeline. Instructions are fetched, decoded, and graduated in program order; however, instructions can issue, execute, and complete out-of-order. In-order graduation allows URSIM to implement precise exceptions.

Most processor parameters are configurable at run-time. These parameters are listed in Chapter 3.

1.2.1 Instruction pipeline

Fetch stage

The **instruction fetch stage** reads instructions from the instruction cache. The maximum number of instructions brought into the processor per cycle is a configurable parameter.

Decode stage

The **instruction decode stage** handles register renaming and inserts the decoded instruction into the active list. The key data structures used in this stage are the register map table, the free list, the active list, and the shadow mapper. These data structures closely follow the corresponding micro-architectural features of the MIPS R10000. The URSIM processor follows the MIPS R10000 convention for maintaining registers, in which both architectural register state and speculative register state are kept in a unified physical register file [8]. The register map table keeps track of the logical to physical register mapping, and the free list indicates the physical registers available for use. A logical register is mapped to a new physical register whenever the logical register is the destination of an instruction being decoded. The new physical register (taken from the free list) is marked busy until the instruction completes. This physical register is not returned to the free list until the instruction with the new mapping has graduated. The previous value of the logical register remains in the physical register to which it was formerly mapped. Integer and floating-point registers are mapped independently. The size of the active list is configurable.

This stage also dispatches memory instructions to the memory unit, which is used to insure that memory operations occur in the appropriate order, as detailed in Section 1.2.3. The maximum number of outstanding memory instructions allowed by the memory unit is configurable.

For branch instructions, the decode stage allocates a shadow mapper to allow a fast recovery on a misprediction, as discussed in Section 1.2.2. The "taken" prediction of a branch stops the URSIM 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. The number of shadow mappers is configurable.

Issue stage

The **instruction issue stage** issues ready instructions. For an instruction to issue, it must have no outstanding data dependences or structural hazards. The only register data dependences that affect the issue of an instruction in URSIM are RAW (read-after-write, or true) dependences; other register dependences are eliminated by register renaming. RAW dependences are detected by observing the "busy bit" of a source physical register in the register file.

Structural hazards in the issue stage relate to the availability of functional units. There are 3 basic types of functional units supported in URSIM: ALU (arithmetic/logical unit), FPU (floating-point unit), and ADDR (address generation unit). If no functional unit is available, the processor simulator notes a structural hazards and refrains from issuing the instruction. The number of each type of functional unit is configurable. A memory instruction issues to the cache only if a cache port is available and the address of the instruction has already been generated. Additional constraints for memory issue are described in Section 1.2.3.

Execute stage

The **instruction execute stage** calculates the results of the instruction as it would be generated by its functional unit. These results include the addresses of loads and stores at the address generation unit. The latencies and repeat rates of the ALU and FPU instructions for this stage are configurable.

Complete stage

The **instruction complete stage** stores the computed results of an instruction into its destination physical register. This stage also clears that physical register's "busy bit" in the register file, thus indicating to the issue stage that instructions stalled for data dependences on this register may progress. This stage does not affect memory store operations, which have no destination register.

The complete 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 program counter is set to the proper target of the branch. The shadow mapper for a branch is freed in this stage.

Graduate stage

The **instruction graduate stage** ensures that the instructions graduate and commit their values into 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 when this instruction was decoded. With the exception of stores, the graduation of an instruction marks the end of its life in the system; stores are discussed separately in Section 1.2.3. After graduation, the instruction leaves the active list. The number of instructions that can graduate in a single cycle is configurable.

The URSIM processor also detects exceptions at the point of graduation. Section 1.2.4 describes how the processor simulator handles exceptions.

1.2.2 Branch prediction

The URSIM processor includes static and dynamic branch prediction, as well as prediction of return instructions (other jumps are not currently predicted). As in the MIPS R10000, each predicted branch uses a shadow mapper which stores the state of the register renaming table at the time of branch prediction. The shadow mapper for an ordinary delayed branch is associated with its delay slot; the shadow mapper for an annulling branch or a non-delayed branch is associated with the branch itself. If a branch is later discovered to have been mispredicted, the shadow mapper is used to recover the state of the register map in a single cycle, after which the processor continues fetching instructions from the actual target of the branch. A shadow mappers is freed upon resolution of a branch instruction at the complete stage of the pipeline. The processor may include multiple predicted branches at a time, as long as there is at least one shadow mapper for each outstanding branch. These branches may also be resolved out-of-order.

URSIM currently supports three branch prediction schemes: dynamic branch predictors using a two-bit history scheme [15], dynamic branch predictors using a two-bit agree predictor [17], and a static branch predictor using only compiler-generated predictions. Return addresses are predicted using a return address stack. Each of the schemes supported uses only a single level of prediction hardware.

The instruction fetch and decode stages initiate branch speculation; the instruction complete stage resolves speculated branches and initiates recovery from mispredictions.

1.2.3 Processor memory unit

The processor memory unit is the interface between the processor and the caches. The most important responsibility of the processor memory unit is to insure that memory instructions occur in the correct order. There are three types of ordering constraints that must be upheld:

1. Constraints to guarantee precise exceptions;
2. Constraints to satisfy uniprocessor data dependences;
3. Constraints due to the multiprocessor memory consistency model[1].

Constraints for precise exceptions

The URSIM memory system supports non-blocking loads and stores. To maintain precise exceptions, a store cannot issue before it is ready to be graduated; namely, it must be one of the instructions to graduate in

[1] Since Impulse is focusing on uniprocessor systems at this moment, discussions about memory consistency are not included in this document.

the next cycle and all previous instructions must have completed successfully. When a store can be allowed to graduate, it does not need to maintain a space in the active list for any later dependences. However, if it is not able to issue to the cache before graduating, it must hold a slot in the memory unit until it is actually sent to the cache. The store can leave the memory unit as soon as it has issued to the cache.

Loads always wait until completion before leaving the memory unit or graduating, as loads must write a destination register. Prefetches can leave the memory unit as soon as they are issued to the cache, as these instructions have no destination register. Furthermore, there are no additional constraints on the graduation of prefetches.

Constraints for uniprocessor data dependences

These constraints require that a processor's conflicting loads and stores (to the same address) appear to execute in program order. The precise exception constraint ensures that this condition holds for two stores and for a load followed by a store. For a store followed by a load, since we do not want the load to wait until the store has graduated, the processor may need to maintains this kind of data dependence by enforcing additional constraints on the execution of the load. When the load has generated its address, the state of the store address determines whether or not the load can issue. Specifically, the prior store must be in one of the following three categories:

1. address is known, non-conflicting;
2. address is known, conflicting;
3. address is unknown.

In the first case, there is no data dependence from the store to the load. As a result, the load can issue to the cache in all configuration options, as long as the multiprocessor ordering constraints allow the load to proceed.

In the second case, the processor knows that there is a data dependence from the store to the load. If the store matches the load address exactly, the load can forward its return value from the value of the store in the memory unit without ever having to issue to cache. If the load address and the store address only partially overlap, the load may have to stall until the store has completed at the caches; such a stall is called a partial overlap, and is discussed further in Chapter 8.

In the third case, however, the load may or may not have a data dependence on the previous store. The behavior of the URSIM memory unit in this situation depends on the configuration options. In the default URSIM configuration, the load is allowed to issue to the cache. When the load data returns from the cache, the load will be allowed to complete unless there is still a prior store with an unknown or conflicting address. If a prior store is now known to have a conflicting address, the load must either attempt to reissue or forward a value from the store as appropriate. If a prior store still has an unknown address, the load remains in the memory unit, but clears the busy bit of its destination register, allowing further instructions to use the value of the load. However, if a prior store is later disambiguated and is found to conflict with a later completed load, the load is marked with a soft exception, which flushes the value of that load and all subsequent instructions. Soft exception handling is discussed in Section 1.2.4.

There are two less aggressive variations provided on this default policy for handling the third case. The first scheme is similar to the default policy; however, the busy bit of the load is not cleared until all prior stores have completed. Thus, if a prior store is later found to have a conflicting address, the instruction must only be forced to reissue, rather than to take a soft exception. However, later instructions cannot use the value of the load until all prior stores have been disambiguated. The second variation stalls the issue of the load whenever a prior store has an unknown address.

The second most important responsibility of processor memory unit is to map virtual addresses to physical addresses, which is done by the MMU/TLB. If a virtual address matches in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address. If an TLB miss occurs, an exception is taken and software refills the TLB from the page table residing in memory.

1.2.4 Exception handling

URSIM supports precise exceptions[2] by prohibiting instructions from committing their effects into the processor architectural state until the point of graduation. Excepting instructions are recognized at the time of graduation.

URSIM supports the following categories of exceptions: *division by zero*, *floating-point errors*, *segmentation faults*, *bus errors*, *system traps*, *window traps*, *TLB misses*, *soft exceptions*, *serializing instructions*, *privileged instructions*, *illegal or unsupported instructions*, and *illegal program counter value*. URSIM simply either emulates the effects of the trap handlers or traps into kernel to actually have the trap handlers simulated. However, *soft exceptions* are handled entirely in the hardware and do not have any associated trap handler.

A *division by zero* exception is triggered only on integer division by zero. *Floating point exceptions* can arise from illegal floating-point operands, such as attempts to take the square root of a negative number or to perform certain comparisons of an "NaN" with a number. Both of these exception types are non-recoverable.

A *Segmentation fault* is generated when the processor attempts to access a page of memory that has not been allocated and is not within the limits of the stack. This type of exception is non-recoverable.

A *bus error* occurs whenever a memory access is not aligned according to its length. Generally, these exceptions are non-recoverable. However, the SPARC architecture allows double-precision floating-point

[2]We use the terms exception and trap interchangeably.

loads and stores to be aligned only to a word boundary, rather than to a double-word boundary[3]. URSIM currently traps these accesses and emulates their behavior.

System traps are triggered by operating system calls in applications. The system traps supported are listed in Section 2.3.5. URSIM supports some important system calls, such as I/O, memory allocation, and process management. Some operating system calls are currently not supported; consequently, functions using these system calls (such as strftime and signal) may not currently be used in applications to be simulated with URSIM.

A *window trap* occurs when the call-depth of a window-save chain exceeds the maximum allowed by URSIM (called an overflow), forcing an old window to be saved to the stack to make room for the new window, or when a RESTORE operation allows a previously saved window to once again receive a register window (called an underflow) [16]. The number of register windows is configurable, and can range from 4 to 32 (in all cases, 1 window is reserved for the system). A *TLB miss* occurs when a memory instruction misses in the TLB. Both window traps and TLB misses are handled in the kernel.

Soft exceptions are distinguished from other exception types in that even a regular system would not need to trap to any operating system code to handle these exceptions; the exception is handled entirely in hardware. The active list is flushed, and execution restarts at the excepting instruction. These are used for recovering from loads incorrectly issued past stores with unknown addresses or from consistency violations caused by speculative load execution (as described in Section 1.2.3).

URSIM uses *serializing traps* to implement certain instructions that either modify system-wide status registers (e.g., LDFSR, STFSR) or are outdated instructions with data-paths that are too complex for a processor with the aggressive features simulated in URSIM (e.g., MULScc). This can lead to significant performance degradation in code that uses old libraries.

Privileged instructions include instructions that are valid only in system supervisor mode, and lead to an exception if present in user code. *Illegal instruction* traps are invalid instruction encodings and instructions unsupported by URSIM, such as **flush**, **flushw**, and tagged addition and subtraction. An *illegal program counter value* exception occurs whenever a control transfer instruction makes the program counter invalid for the instruction address region. These three exception types are all non-recoverable.

1.3 Overview of memory hierarchy

Figure 1.2 shows the memory system in URSIM. URSIM simulates a two level cache hierarchy (with a coalescing write buffer if the first-level cache is write-through), a multiplexed split-transaction system cluster bus, an Impulse main memory controller, and an Impulse DRAM backend. The cluster bus connects the secondary caches and the memory controller.

[3]The SPARC architecture also allows word-alignment for quadruple-precision floating-point loads and stores, but URSIM does not support such instructions.

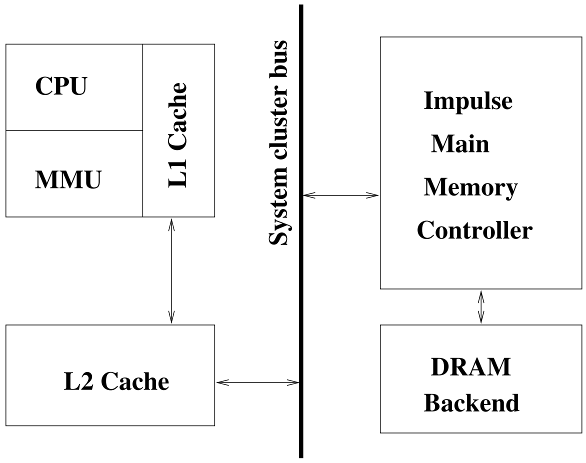

Figure 1.2: URSIM Memory System

1.4 Cache hierarchy

Both cache levels are lockup-free and store the state of outstanding requests using miss status holding registers (MSHRs).

The first-level 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. URSIM allows a multi-ported and pipelined first-level cache. The size, line size, set associativity, cache latency, number of ports, number of MSHRs, and lots of other configuration parameters can be varied.

The coalescing write buffer is implemented as a buffer with cache-line-sized entries. All writes sent to the next memory hierarchy level by L1 cache are buffered here and sent to the second level cache as soon as the second level cache is free to accept a new request. The number of entries in the write buffer is configurable.

The second-level cache is a write-back cache with write-allocate. URSIM simulates a pipelined secondary cache. Lines are replaced only on incoming replies; more details of the protocol implementation are given in Chapter 9. The secondary cache maintains *inclusion property* with respect to the first-level cache. The size, line size, set associativity, cache latency, and number of MSHRs can be varied. The simulator also allows user to completely disable the second-level cache. In that case, the cache hierarchy contains only the first-level cache, which must be write-back with write-allocate.

1.5 System cluster bus

The system bus is a time-multiplexed split-transaction MIPS R10000 cluster bus. It supports snoopy cache coherence protocol and a simple pipelined arbitration scheme. More details of implementation are given in Chapter 10. The bus speed, bus width, and bus arbitration delay are all configurable.

1.6 Main memory controller

The MMC (Main Memory Controller) is the core of the Impulse memory system. It communicates with the processors and I/O adapters over the cluster bus, translates shadow addresses into physical DRAM addresses, generates DRAM accesses, and performs MC-based prefetching.

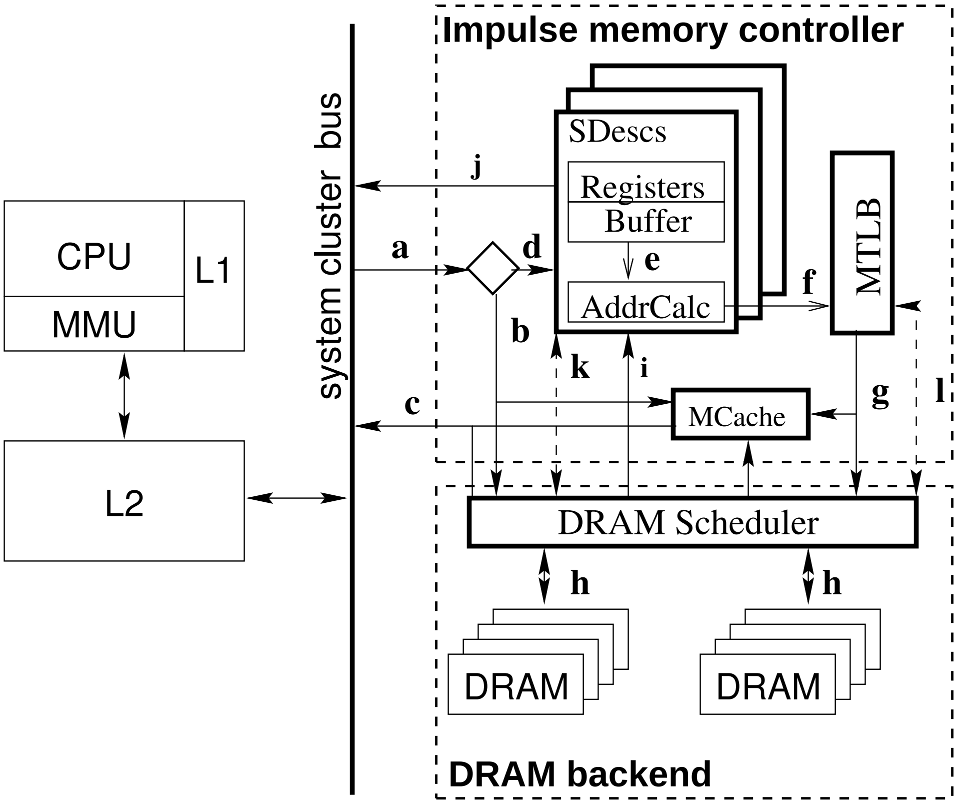

Figure 1.3: The Impulse memory architecture. The arrows indicate how data flow within an Impulse memory system.

Figure 1.3 illustrates the Impulse memory architecture. The Impulse memory controller includes following components:

- a small number of *Shadow Descriptors (SDescs)*, each of which contains some *registers* to store remapping information, a small *SRAM buffer* to hold remapped data prefetched from DRAMs, the logic to assemble sparse data retrieved from DRAMs into dense cache lines, and a simple *ALU unit (AddrCalc)* to translate shadow addresses to pseudo-virtual addresses;
- a small number of *Memory Controller TLBs (MTLB)*, each of which is backed up by main memory and maps pseudo-virtual addresses to physical DRAM addresses, along with a small number of buffers to hold prefetched page table entries;
- a *Memory Controller Cache (MCache)*, which buffers non-remapped data prefetched from DRAMs;
- a *DRAM Scheduler*, which contains circuitry that orders and issues accesses to the DRAMs;
- DRAM chips, which constitute the main memory.

The extra level of address translation at the memory controller is optional, so an address appearing on the system cluster bus may be a real physical or a shadow address (**a**). A real physical address passes untranslated to the MCache/DRAM scheduler (**b**). A shadow address has to go through the matching shadow descriptor (**d**). The AddrCalc unit in the shadow descriptor translates the shadow address into a set of pseudo-virtual addresses using the remapping information stored in control registers (**e**). These pseudo-virtual addresses are translated into real physical addresses by the MTLB (**f**). Then the real physical addresses pass to the DRAM scheduler (**g**). The DRAM scheduler orders and issues the DRAM accesses (**h**) and sends the data back to the matching shadow descriptor (**i**). Finally, the appropriate shadow descriptor assembles the data into a cache line and sends it over the system cluster bus (**j**).

1.6.1 Memory controller TLB

The MMC first translates shadow addresses to pseudo-virtual addresses, and then translates pseudo-virtual addresses to physical addresses. The mapping from pseudo-virtual addresses to physical addresses is like the one from virtual addresses to physical addresses performed by the processor's MMU. Just as the CPU uses a TLB (Translation Lookaside Buffer), the MMC uses a memory controller TLB (MTLB) to accelerate the pseudo-virtual to physical translations.

When an application issues a system call to remap data, the operating system creates a dense, flat page table to store the pseudo-virtual-to-physical translations of the data structure being remapped. The OS also sends the starting physical address of this page table to the MMC so that the MMC can access this page table without interrupting the OS. We refer to this page table as the memory controller page table. Since the memory controller page table is dense and flat, it can be indexed by the virtual offset of the original data structure; and each memory controller page table entry does not need to store the pseudo-virtual page number.

The MTLB is currently using a Least Recently Used (LRU) replacement policy, and has a one-memory-cycle access latency.

A small SRAM buffer inside the MTLB is used to cache the memory controller page table entries loaded from physical memory. Each MTLB miss checks the buffer before sending a fill request to DRAM. If an MTLB miss hits in the buffer, it only takes one extra cycle to load the translation into the MTLB. If it misses in the buffer, the MTLB will generate a fill request to load a cache line from physical memory into the buffer, and then load the relevant translation from the buffer to the MTLB.

Memory controller cache

The MCache includes a modest-sized buffer to store prefetched, non-shadow data and a tiny buffer (two to four cache lines) for each shadow descriptor to store prefetched, shadow data. The tiny buffer for each shadow descriptor is fully associative. Its design is trivial. Except where stated otherwise, the MCache represents the modest-size buffer for non-shadow data in the following discussion.

The MCache is physically indexed, physically tagged, and has configurable associativity and size. Its line size equals the size of an L2 cache line. The MCache uses a write-invalidate protocol, i.e., any write memory transaction invalidates the matched data in the MCache. So the data in the MCache can never be dirty, which means that a victim line can simply be discarded when a conflict occurs.

1.7 DRAM backend

The DRAM backend[4] is another important component of the Impulse memory system. The section describes the DRAM backend currently simulated by URSIM. The DRAM backend[5] is constructed from three major components: the DRAM Dispatcher, Slave Memory Controller (SMC), and plug-in memory modules — DRAM chips. The DRAM dispatcher, SMCs, and the connecting wires between them – RAM Address bus (RA bus) – constitute the DRAM scheduler shown in Figure 1.4. One DRAM backend contains one DRAM dispatcher only, but can have multiple SMCs, multiple RA busses, and multiple plug-in memory modules. Figure 1.4 shows a simple configuration with four SMCs, four DRAM chips (each of them has two banks), and two RA busses. Bear in mind that the DRAM dispatcher and SMCs don't have to be in different chips. Figure 1.4 just shows them in a way easy to understand. Whether or not to implement the DRAM scheduler in a single chip is an open question.

A DRAM access can be either a shadow access or a non-shadow access. The MMC sends requests to the DRAM backend via Slave Address busses (SA bus) and passes data from or to the DRAM backend via Slave Data busses (SD bus). In the simulator, the number of SA busses/SD busses can vary from one to one plus the number of total shadow descriptors. If there is only one SA bus or SD bus, both non-shadow accesses and shadow accesses will share it. If there are two SA busses or SD busses, non-shadow accesses

[4]The current design was simulated based on Lixin's initial CS676 project. Reader should expect major changes in a near future.

[5]Since the Impulse memory system was designed based on the HP Kitt-Hawk memory system, this manual follows the terminology of Kitt-Hawk memory system.

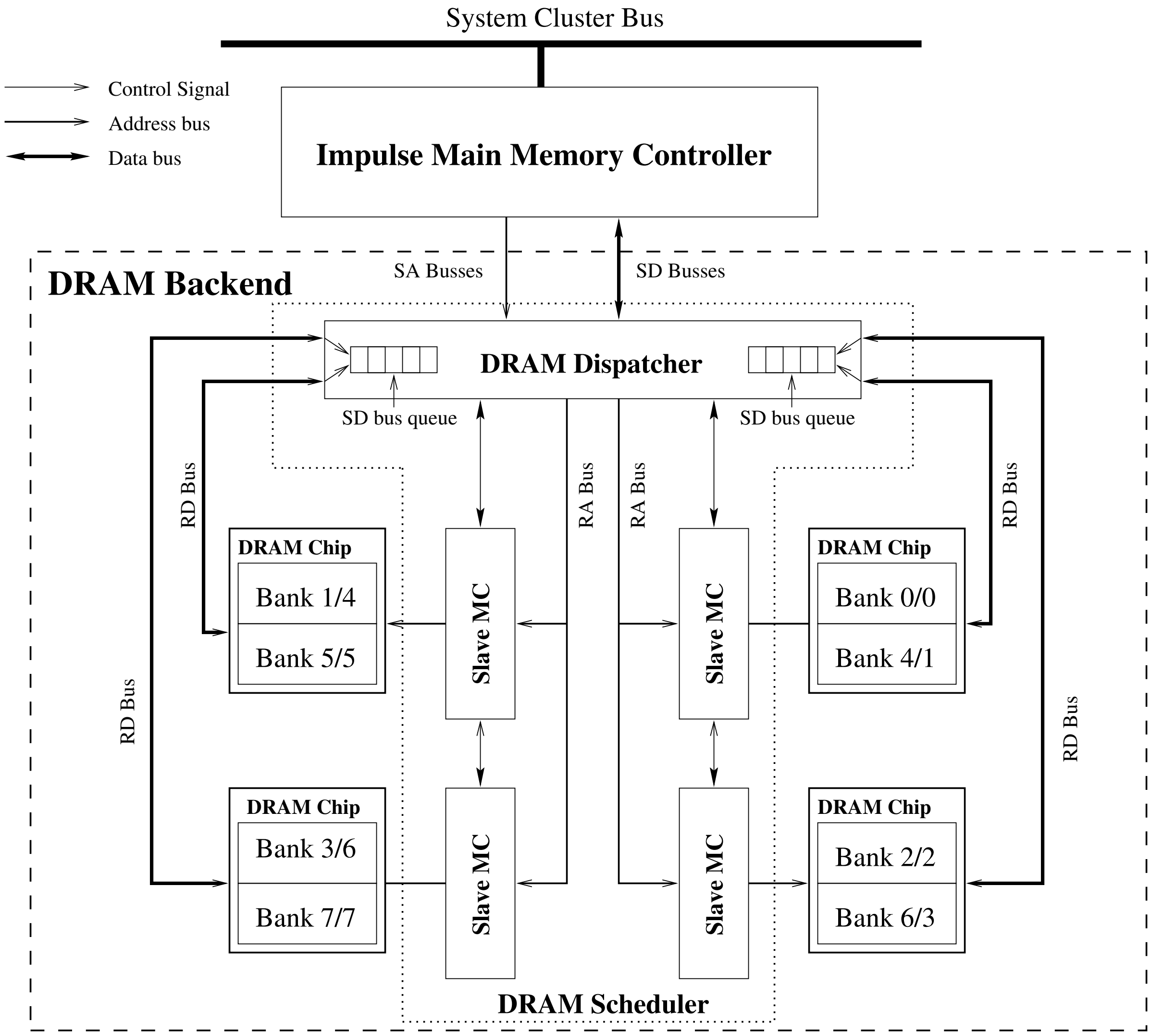

Figure 1.4: Impulse DRAM Backend Block Diagram

will use one exclusively and shadow accesses will use the other one exclusively. If there are more than two SA busses or SD busses, one will be exclusively used by non-shadow accesses and each of the rest will be used by a subset of the shadow descriptors. The contention on SA busses is resolved by the MMC and the contention on SD busses is resolved by the DRAM dispatcher.

1.7.1 DRAM dispatcher

The DRAM dispatcher is responsible for sending memory accesses coming from SA busses to the relevant SMC via RA busses and passing data between SD busses and RAM Data busses (RD bus). If there is more than one SA bus, contention on RA bus occurs when two accesses from two different SA busses simultaneously need the same RA bus. For the same reason, contention on SD busses or RD busses will occur if there is more than one RD bus or more than one SD bus. The DRAM dispatcher resolves all the contentions by picking a winner according to a designated algorithm and queuing the others. If a waiting queues becomes full, the DRAM dispatcher will stop the sender (either MMC or SMC) from sending more requests. No sophisticated scheduling algorithms have been applied on the waiting queues. All waiting queues work in First-Come-First-Serve (FSFC) order. Normally, most waiting transactions are on RD busses, so the DRAM dispatcher holds a configurable number of waiting queues (called *SD bus queue*[6]) for transactions coming off RD busses. The SD bus queue has two uses: to buffer transactions coming from RD busses so that the RD busses can be freed up for other transactions; to resolve contention on SD busses by queuing all other contenders except the winner. Each SD bus queue is connected to an exclusive subset of RD busses.

1.7.2 Slave memory controller

Each Slave Memory Controller controls one RD bus and several DRAM chips sharing the RD bus. The SMC has independent control signals for each DRAM chip. The basic unit of memory is a *memory bank*. Each memory bank has its own page buffer and can be accessed independently from all other banks. Some RDRAM (Direct Rambus DRAM) chips let each page buffer to be shared between two adjacent banks, which introduces the restriction that adjacent banks may not be simultaneously accessed. We approximately model this type of RDRAM by making the effective independent banks be half of its physical number of banks. How many banks each DRAM chip has depends on the DRAM type. Typically, each SDRAM (Synchronous DRAM) chip contains two to four banks and each RDRAM chip contains eight to 16 banks.

SMC is responsible for several important tasks. First, SMC keeps track of each memory bank's page buffer and decides whether or not to leave page buffer open after an access. Second, SMC controls an independent waiting queue for each bank and schedules the transactions in the waiting queue with the intention of reducing the average memory latency. Third, SMC manages the interleaving of memory banks. When an access is broadcasted on a RA bus, all SMCs on the RA bus will see it, but only the SMC that controls the memory bank for the access will respond. The interleaving scheme determines which SMC should respond to a specified physical address. Fourth, SMC is responsible for DRAM timing and refreshing DRAM chips periodically.

The following sections illustrate several algorithms implemented in the SMC simulator: **hot row policy** which decides whether or not to leave a hot row open at the end of an access; **bank queue reordering algorithm** which reorders the transactions in order to minimize the average memory latency perceived by the processor; **interleaving scheme** which determines how the physical DRAM addresses are distributed among DRAM banks.

[6]Conceptionally, *SD bus queue* equals to the Jetway in old technical report [20].

1.7.3 Hot row policy

In order to save the RAS signals, the Impulse DRAM backend allows hot rows to remain active after being accessed. The size and number of the hot rows vary with the type of DRAM chips and the number of memory banks. The collection of hot rows can be regarded as a cache. Proper management is necessary to make this “cache” profitable. The benefit to leave a row open is that the DRAM access latency is reduced due to eliminating the RAS signals if a DRAM access hits the open row. However, a DRAM access has to pay the penalty of closing the open row if it misses the open row. URSIM supports three precharge policies: *close-page* policy, where the active row is always closed after an access; *open-page* policy, where the active row is always left open after an access; *use-predictor* policy, where predictors are used to guess whether the next access to an open row will be a hit or a miss.

The *use-predictor* policy was initially designed by R.C. Schumann [13]. In this policy, a separate predictor is used for each potential open row. Each predictor records the hit/miss results for the previous several accesses to the associated memory bank. If an access to the bank goes to the same row as the previous access to the same bank, it is recorded as a hit no matter whether the row was kept open or not. Otherwise, it is recorded as a miss. The predictor then uses the multiple-bit history to predict whether the next access will be a hit or a miss. If the previous recorded accesses are all hits, it predicts a hit. If the previous recorded accesses are all misses, it predicts a miss. Since the optimum policy is not obvious for the other cases, a software-controlled precharge policy register is provided to define the policy for each of all the possible cases. Application can set this register to specify the desired policy or can disable the hot row scheme altogether by setting the register to zeros. In our experiment, the precharge policy register is set “open” when there are more hits than misses in the history and “close” when there are more misses than hits or the same misses as hits in the history. For example, if the history has four-bit, the precharge policy register is set to be 1110 1000 1000 0000 upon initialization, which keeps the row open whenever three of the preceding four accesses are page hits.

We expanded the original use-predictor policy with one more feature: when the bank waiting queue is not empty, use the first transaction in the waiting queue instead of the predictor to make decision. If next transaction goes to the same row as current transaction does, the row is left open after current transaction. Otherwise, the row is closed.

1.7.4 Bank waiting queue reordering

There are many different types of DRAM accesses that the MMC can send to the DRAM backend. Figure 1.3 shows flows to the DRAM backend from different units. Based on the issuer and the nature of a DRAM access, each DRAM access is classified as one of following four types.

- *Direct access* is generated by real physical address directly coming off from system memory bus (arrow **b** in Figure 1.3). Since each normal memory request is for a cache line, each direct access requests a cache line from the DRAM backend.

- *Indirection vector access* is generated by the shadow descriptors to fetch the indirection vector during the translation for *scatter/gather using an indirection vector* [3] (**k**). Each indirection vector access is for a cache line and the return data are sent back to the relevant shadow descriptor.
- *MTLB access* is generated by the MTLB to fetch page table entries from DRAMs into the MTLB buffers (**l**). To save total number of MTLB accesses, each MTLB access requests a whole cache line, not just a single entry.
- *Shadow access* is any DRAM access generated by the Impulse memory controller to fetch remapped data (**g**). The size of each shadow access varies with application-specific mappings. The data of shadow accesses are returned back to the remapping controller for further processing.

This document also uses another definition – *non-shadow access*, which includes direct access, indirection vector access, and MTLB access. Normally, most of DRAM accesses are either direct accesses or shadow accesses, with a few or none being MTLB accesses and indirection vector accesses. Intuitively, different types of DRAM access should be treated differently in order to reduce the average memory latency. For example, an indirection vector access is depended on by a bunch of shadow accesses and its waiting cycles directly contribute to the latency of the associated memory request, so it had better be taken care of as early as possible. Any delay on a prefetching access will not likely increase the average memory latency as long as the data are prefetched early enough, which is easy to accomplish in most situations, so a prefetching access does not have to complete as early as possible and it can give away its memory bank to more important accesses like indirection vector accesses and MTLB accesses. After having taken consider of those facts, we propose a reordering algorithm with following rules.

1. Ensure the consistency. Make sure no violation of data dependencies — read after write, write after read, and write after write.
2. Once an access is used to make decision that whether or not to leave a row open at the end of the preceding access (see Section 1.7.3), no any other accesses can get ahead of it and it's guaranteed to access the relevant memory bank right after the preceding one.
3. Give any normal (non-prefetching) access higher priority over any prefetching access.
4. Give MTLB access and indirection vector access the highest priority so that these accesses can be finished as early as possible, therefore releasing their dependent shadow accesses as early as possible.
5. It's hard to determine by imagination whether the direct access or the shadow access should be given higher priority over each other. For experimental purpose, URSIM supports two opposite choices: giving direct access higher priority over shadow access; or giving shadow access higher priority over direct access.
6. Increase each access' priority along with its increasing waiting time. This rule guarantees no access would stay in bank waiting queue "forever". This rule is optional in the simulator.
7. If there is a conflict among some rules, always use the rule that is the earliest in above sequence among those conflicting rules.

1.7.5 Memory bank interleaving schemes

The interleaving of memory banks controls the mapping from physical DRAM addresses to memory banks. Figure 1.4 shows two interleaving schemes. The first one numbers the physically-adjacent memory banks as separately as possible. Its mapping function from the memory banks to the SMCs is (*SMC-id* = *bank-id* MOD *number of SMCs*). The second one numbers the physically-adjacent memory banks with consecutive numbers. Its mapping function is (*SMC-id* = *bank-id* / *banks-per-chip*). We call the first one *modulo-interleaving* and the second one *sequential-interleaving*. Each scheme can be either page-level or cache-line-level. So there are total four interleaving schemes modeled in the simulator: page-level modulo-interleaving, page-level sequential-interleaving, cache-line-level modulo-interleaving, and cache-line-level sequential-interleaving. In the simulator, the page size is the size of the page buffer in each bank and the cache-line size equals to the size of a second level cache line. When we say the banks are interleaved at the page-level, it means bank 0 has all pages whose address modulo page-size is 0, bank 1 has all pages whose address modulo page-size is 1, and so on.

Chapter 2

Using URSIM

URSIM is located at
`snowball:/usr/lsrc/impulse`
which is mounted at the following point on department facility machines
`/home/css/impsrc`
Because the simulator is a large and complex system, we use CVS (Concurrent Versions System) to manage URSIM codes. For a quick CVS tutorial, please look at
`http://www2.cs.utah.edu/impulse/cvs/index.html`
There is also a distributed URSIM version located in
`snowball:/usr/lsrc/impusle/dist/simpulse`
People who do not want to compile the simulator by their own can find simulator executables, compiled application library, commonly-used benchmark executables, and many other things there.

This chapter and the next chapter describes how to use URSIM. Section 2.1 lists the steps to check out the source codes. Section 2.2 describes how to build the simulator executables, as well as other executables needed to simulate applications with URSIM. Section 2.3 explains how to port applications to URSIM. Section 2.4 shows the steps to build applications to run on URSIM. Section 2.5 briefly describes the statistics collected by URSIM. Section 2.6 talks about the debugging support of URSIM. Because URSIM is an extremely flexible simulator and supports hundreds of configurable parameters, we use a separate chapter (Chapter 3) to illustrate how to configure the simulator.

2.1 Checking out the source codes

Checking out the simulator for the first time.

- Set environment variable CVSROOT to be /home/css/impsrc/CVS, for example, run the following command in C shell:
  `setenv CVSROOT /home/css/impsrc/CVS`
- Go to a directory where you want to place the simulator, for example, put it under your directory in Impulse filesystem:
  `cd /home/css/impsrc/@userid`
- Run the following command to fetch the entire tree.
  `cvs checkout simpulse`
  Be sure that you have sufficient disk space before you start. It needs about 100Mbytes.

This will produce a directory **simpulse**, containing the following subdirectories.

app-examples Example applications ported to URSIM (including source, makefile, executables, and output files)

app-lib The kernel library to be linked by applications, and generic application makefiles

bin Makefiles for compiling URSIM for all supported platforms, and simulator executables

kernel C source files of the kernel library

share Makefiles containing common definitions and generic rules

src C/C++ source files of the simulator

2.2 Building the simulator executables

The top-level makefile for compiling URSIM are located under subdirectory **bin**. To compile it, change to **bin** and run command "`make all`". You can expect a lot of informational messages to appear when running `make` for the first time because some dependency files and directories are constructed as well as compiling all of the source files. The makefile can handle four common platforms and puts the generated simulator executables under the relevant subdirectory:

SunOS Sun SPARCstations with Solaris 2.5 or above. Makefiles assumes GNU C and C++ compilers, version 2.8.0 (porter, mashie, qattusa, birdie, ursa10[1]).

4.3bsd Platforms with HP PA-RISC processor and 4.3BSD. Makefile assumes GNU C and C++ compilers, version 2.7.2 (snowball, cans, swoosh, rum).

[1]Names of the machines that we usually use and belong to this category.

HP-UX Platforms with HP PA-RISC processor and HP-UX version 9 or 10. Makefile assumes GNU C and C++ compiler with version 2.7.2 or HP C and C++ compiler with version 20.32 (roadkill, slush).

IRIX64 SGI PowerChallenge platforms with MIPS R10000 and IRIX64 6.5. Makefile assumes MIPSpro C and C++ compiler, version 7.2.1 (burn, raptor, prospero).

The makefile creates an optimized executable called **rsim** and a debugging executable called **rsim_d**. The makefile automatically updates the dependency files, so it can efficiently recompile the executables whenever some source files have been changed. The makefile also supports some other functionality, for example, generating testing executables. Interested readers should refer to the comments in the head of the makefile for more details.

Additionally, on the Sun platforms, the **predecode** executable will be created. **predecode** translates the instructions of a SPARC application executable into an intermediate form that can be processed by URSIM.

2.3 Porting applications to the URSIM

When porting applications to run under URSIM, the user must link applications with URSIM kernel library, which includes the following three parts:

1. Startup routine, which initiates kernel data structures, sets up and initiates the user data segment and user stack, writes trap table base address register, etc.;
2. Virtual memory system, including TLB miss handler and the implementation of system call **sbrk()**;
3. Impulse system calls [21];
4. Interface to use URSIM traps.

This section describes each of the above issues and the effects they have on porting applications to URSIM. Assembly programmers will also need to account for the unsupported instructions discussed in Section 1.1. The reader is encouraged to see the example applications included in the **app-examples** directory as illustrations of the concepts discussed in this section.

2.3.1 Startup routine

The application must set the startup routine in kernel library as the entry point of application executables to run on URSIM (using **ld** option "-emystart"). The startup routine first initiates kernel data structures necessary to make the kernel work.

Then, it sets up data segment and user stack for the application. In real operating system, the kernel runs as an independent process and can load application executables by itself. In URSIM kernel, as an application library, is integrated with applications, so it can not load application executables. Instead, the simulator loads the application executables, passing the necessary information, namely the starting address and size of data segment and the size of stack, to the kernel startup routine.

Then, the startup routine writes the base address of trap table into trap-table base address register (TBA). The TBA is used by the simulator to jump to the relevant trap handler, whenever the simulator detects a trap supported by the kernel.

Finally, the startup routine calls the main function in application code.

2.3.2 Virtual memory model

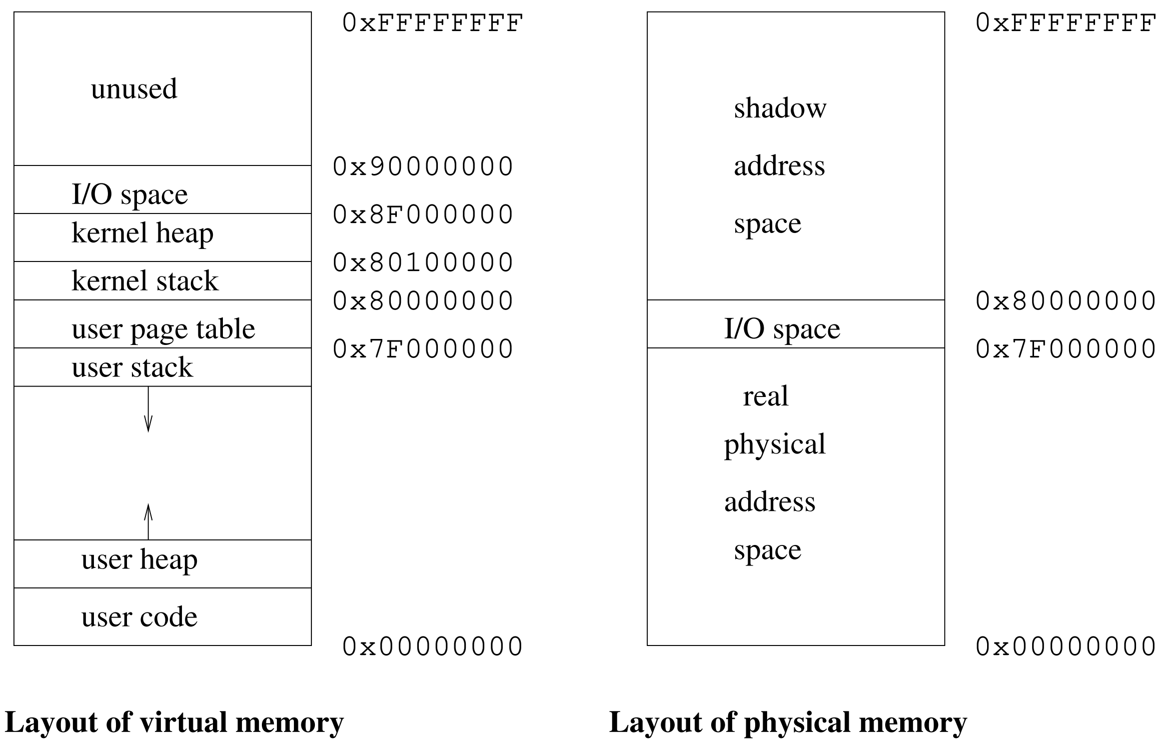

Figure 2.1: URSIM memory model.

The URSIM memory model is depicted in Figure 2.1. The left half of this figure shows the layout of virtual memory. The current design assumes 32-bit virtual address. User code is put at the bottom of virtual memory. The user heap (including user data) follows user code and grows upward. The user stack starts from 0x7F000000 (exclusively) and grows downward. The virtual memory region [0x7F000000, 0x80000000) is reserved for user page table, which maps user code, user heap, and user stack. The page translations for user page table are held in the kernel heap. Reserving a special region for user page table can speed up the TLB miss handler because the miss handler does not need to trace down multi-layer page tables. Since a user process uses both ends of user space contiguously, real physical memory space is not needed for the parts of the page table that map "holes" in the process's address map. In addition, using kernel heap to map the page table region avoids recursive invocations of TLB miss handlers. Kernel stack [0x80000000, 0x80100000) and kernel heap [0x80100000, 0x8F000000) are unmapped but cached. The physical address in kernel space is selected by subtracting 0x80000000 from the virtual address. To simplify design, we use a pre-allocated 1-megabyte space for the kernel stack. If the kernel runs out of stack for some simulated applications, we will have to adjust the kernel stack. The I/O space is unmapped and uncached. It is located in the region [0x8F000000, 0x8FFFFFFF), which is mapped to physical region [0x7f000000, 0x7FFFFFFF). Both kernel heap and user heap region grow through explicit memory allocation calls; while both kernel stack and user stack grow automatically in TLB miss handlers.

The right half of Figure 2.1 shows the layout of physical memory. URSIM supports 32-bit physical address. The top half of physical memory [0x80000000, 0xFFFFFFFF] is taken as shadow address space. The region [0x7F000000, 0x7FFFFFFF] is reserved for I/O addresses. The rest physical memory [0x00000000, 0x7EFFFFFF] contains real physical addresses. So the maximum DRAM memory that URSIM can simulate is $(2G - 16M)$ bytes.

2.3.3 Impulse system calls

Please see Impulse system calls reference manual [21].

2.3.4 URSIM traps

URSIM provides a bunch of trap that applications can use to control and communicate with the simulator.

Statistics-related traps

URSIM automatically generates statistics for many important characteristics of the simulated system. URSIM has special functions and macros that can be used to subdivide these statistics according to the phases of an application.

The user can add the **newphase(phaseid)** and **endphase()** functions to indicate the start and end of an application phase. The **newphase()** function takes a single integer argument that represents the new phase number (the simulation starts in phase "0"). This function also clears out all current processor simulation statistics. The **endphase()** function takes no arguments. This function prints out both a concise summary and a detailed set of processor simulation statistics.

The functions **StatReportAll()** and **StatClearAll()** handle the statistics associated with the memory system, including cache, bus, memory controller, and DRAM backend. **StatReportAll()** prints out a detailed set of statistics associated with the memory system, while **StatClearAll()** clears all the statistics gathered.

Avoiding memory system simulation

In cases, where a detailed memory system simulation is not important (for example, initialization and testing phases), the macro **MEMSYS_OFF** can be used to speed up the simulation. This macro turns off the memory system simulation and instead assume a perfect cache hit rate. This macro can be useful for initialization and cleanup phases; the macro **MEMSYS_ON** is used to restart full memory system simulation.

Other useful traps

- **abort(int code)** forces all processors to stop immediately; this differs from **exit(int code)** which only terminates the calling process. Further, **exit** calls the cleanup functions provided through the **atexit** library functions, whereas **abort** does not.
- **GET_L2CACHELINE_SIZE()** returns the cache-line size of the secondary cache, which is the system's coherence granularity. This can be useful for padding out array accesses to avoid false-sharing.
- **sysclocks()** returns the number of simulated processor clock cycles since the start of the simulation.

2.3.5 Supported system calls

Only a limited number of system calls are either handled by the kernel or emulated by the URSIM. URSIM handles the other unsupported system calls by returning 0 without actually doing anything to fake that the system call has been successfully processed. This method may fool some applications, but may cause invalid results for some applications. So applications using any unsupported system calls may or may not run on URSIM. We have implemented a set of important system calls that are enough to simulate our testing benchmarks. If a reader finds that some other system calls are important and should be supported by URSIM, please contact the URSIM develop team ({retrac,lizhang,impulse}@cs.utah.edu).

Currently, URSIM supports the following system calls: *exit*, *fork*, *read*, *write*, *open*, *close*, *time*, *stat*, *lseek*, *getpid*, *fstat*, *dup*, *times*, *ioctl*, *fcntl*, *sysconfig*, and *sbrk*.

Although *time* and *times* have the same semantics as in Unix, they report the simulated time of the application execution starting at the beginning of simulation rather than the beginning of 1970.

Note that, although all of these functions have the same behavior as UNIX functions on success and return the same values on failure, these functions do not set the **errno** variable on failure.

2.4 Building applications

The URSIM kernel library is located in the subdirectory **app-lib**. This directory includes a makefile used for actually building the library on a SPARC Solaris platform.

The directory **app-examples** includes several example applications ported to URSIM: a parallel red-black SOR (in the subdirectory **SOR**), the conjugate gradient benchmark from NAS benchmarks suite 2.3 (in the subdirectory **CG**), and several testing programs using Impulse system calls (in the subdirectory **amstest**). These applications are provided primarily for instructional purposes. They are useful for familiarizing oneself with the URSIM command line options, configuration file, and the Impulse system calls.

There are two generic makefiles in directory **app-lib** that can be used for building applications: *makefile_fortran* is for applications containing at least one Fortran source file; *makefile_generic* is for applications containing C code only.

Makefiles for a specific application can include a generic makefile, so long as they define the `SRC`, `HEADERS`, and `TARGET` variables. The generic makefile assumes that the application-specific makefile is located in the top-level directory for a given application, that all source and header files are located in the **src** directory of the application, that the relocatable object files will be placed in the **obj** directory, and that the linked and predecoded executables will be in the **execs** directory of the application. For example, if an application consisted of the source files **src/source1.c** and **src/source2.c**, the header files **src/header1.h** and **src/header2.h**, the application makefile should simply read:

```
SRC = source1.c source2.c
HEADERS = header1.h header2.h
TARGET = app

include /home/css/impsrc/dist/simpulse/app-lib/makefile_generic
```

The generic makefile assumes applications' source files contains both non-Impulse codes and Impulse codes, which are separated by directives about macro "IMPULSE", for example, "`#ifdef IMPULSE`" or "`#ifndef IMPULSE`". Each generic makefile seeks to produce four executables:

execs/app Optimized executable (compiled with "-xO4" option) to run on hardware.

execs/app_d Debugging Executable (compiled with "-g" option) to run on hardware.

execs/app.rsim_dec Predecoded, optimized executable to run on URSIM, without Impulse optimization (compiled without "-DIMPULSE" option).

execs/app_i.rsim_dec Predecoded, optimized executable to run on URSIM, with Impulse optimization (compiled with "-DIMPULSE" option).

Note that with the generic makefile, all source files will be recompiled if any of the source or header files change. Thus, the user may wish to modify the generic makefile before using it for large applications that will change frequently.

For best performance, the user should invoke the compiler with full optimization flags as specified in the generic makefile. With such options, the compiler will not generate code using outdated instructions such as MULScc [19], which lead to poor performance with URSIM (see Section 1.2.4). The user should also inform the compiler to assume that accesses are aligned, as this will avoid unnecessary single-precision floating-point loads. The generic makefiles provided has this option set for the Sun C compiler.

2.5 Statistics collection

URSIM provides a wide variety of statistics related to the processors, the caches, the bus, the memory controller, and the DRAM backend. URSIM prints a concise summary of the most important processor statistics on the standard error file and a detailed set of statistics on the simulation output file; both can be redirected through command line options. An application can use the phase-related and statistics-reporting URSIM traps described in Section 2.3 to print statistics for relevant portions of the application separately, rather than for the entire application at once.

2.5.1 Processor statistics

URSIM provides statistics on the branch prediction behavior, the occupancy of the active list, and the utilization of various functional units. URSIM also provides statistics related to the performance of the instruction fetching policy according to the metrics of availability, efficiency, and utility [1]. Deficiencies in each of these metrics are categorized according to the type of instruction or event that prevented peak performance.

2.5.2 Memory system statistics

URSIM classifies memory operations at various levels of the memory hierarchy into hits and misses. Misses are further classified into conflict, capacity, and coherence misses. The method for distinguishing between conflict and capacity misses is discussed in Section 9.8. Statistics on the average latency of various classes of memory operations and prefetching effectiveness are also provided by default. The simulator also provides statistics on write-buffer utilization and bus utilization. Statistics on the memory controller module contains numbers (count, average, and total cycles) of all kind of memory accesses (read/write/writeback/copyback, shadow/nonshadow, etc.), effectiveness of MC-based prefetching, performance numbers of the MCache and MTLB. The DRAM module provides the detailed statistics about each component of DRAM backend for each type of DRAM accesses.

2.6 Debugging

2.6.1 Support for debugging URSIM

Generating debug version

URSIM provides compile-time options to enable some self-checking codes and to print debugging and diagnostic messages to the standard output or to predefined files. Such tracing information is likely to be very important to anyone seeking to modify URSIM. The following table lists the currently available compile debug options.

DEBUG_PROC	debug the processor module
DEBUG_MMC	debug the main memory controller module
DEBUG_EVENT	debug the YACSIM event library
TRACE_MMC	trace each memory transaction (see Section 3.2.8 for more detail)

Trace execution of application program

A useful debugging trick is to trace the instructions being executed. URSIM can trace instructions in graduate order (a.k.a program order). Please Section 3.1.2 for details on command line options controlling trace.

Check-point

URSIM can create a new executable at a specified simulation cycle (so-called *check-point*). The basic idea is to replace the old data area with the current data area in the original simulator executable. When the new executable is executed, the process will see the same data structures and data values that the original process had when check-point was generated. Therefore, the new executable starts simulation right at the cycle it was created. This is *extremely* useful when a bug comes out after a long time of simulation. It is also needed for simulator uses to report bugs to simulator developers. Creating a check-point right before a bug occurs allows us to restart the simulator at a specified cycle, therefore saving the time of displaying or regenerating the bug (thus, speeding up debugging). The check-point is controlled by command line option `-d int,file[,]`, which means create a new executable to `file` at cycle `int`. The `,` controls whether or not the simulation continues: with it, simulation stops; without it, simulation continues.

Dumping functions

- **DumpProcMemQueue(pid, count)**: Dump the memory queue in the processor memory unit. **pid** indicated the processor id. **count** represents how many entries should be printed out. “0” means printing all.
- **TLB_dump(pid)**: Dump the current state of CPU TLB.
- **Cache_dump(pid)**: Dump the current state of the cache hierarchy.
- **Bus_dump(pid)**: Dump the current state of the system bus.
- **MMC_dump(pid)**: Dump the current state of the main memory controller.
- **DRAM_dump(pid)**: Dump the current state of the DRAM backend.
- **MemoryDump(pid)**: Call all the six functions described above.

These functions have been proved to be very helpful. You can either check the output of one dump to see if there are any unmatched operations (e.g., if processor shows a memory operation is accessing the cache while the cache does not have this operation, there must be something wrong.), or compare multiple dumps to see if the simulator has been making any progress. Each of the these functions calls finer dump functions to print out information about smaller component. Interested users should read the source code for details.

Other useful functions or variables

- **PC(inst)** prints the program counter for the specified instance.
- **GR(proc, r)** prints the current content of the specified register.
- **GR(inst, r)** prints the content of the specified register when the specified instruction is executed.
- **DumpP()** calls DumpProcMemQueue(0, 0).
- **DumpInstruction(inst)** dumps out everything about an instruciton.
- **DumpP()** calls DumpProcMemQueue(0, 0).
- **useless_i**, **useless_f**, **useless_d**, and **useless_ll** are global integer, float, double, and long long variables used when debugger (gdb or dbx) calls functions with “pass-by-reference” argument(s).

2.6.2 Debugging applications

URSIM does not currently include support for debugging application programs with a debugger like gdb or dbx, as URSIM does not expose information about the application being simulated to such a debugger. If URSIM encounters a non-recoverable exception (such as a segmentation fault or bus error), the simulator halts immediately and a termination message is printed on the standard error file. Application errors can be debugged either by running the applications natively, or by inserting printf calls into the application. If the latter option is chosen, the debugging code should include an fflush(stdout) after each printf, as stdio streams are not guaranteed to be flushed on an abnormal exit in URSIM.

Chapter 3

Configuring URSIM

This chapter discusses the various run-time and compile-time options available to configure URSIM, and specifies the default values for the parameters. The parameters most frequently modified in our experience are available to the user on the URSIM command line; most other parameters are presented to URSIM via a *configuration file* (or called *parameter file*[1]). Different configuration files can be used for different simulation runs, as the name of the configuration file is passed to URSIM on the command line. Except mentioned otherwise, the parameter values in configuration file override the command line options.

3.1 Command line options

Command line options contain three parts: arguments for the URSIM; arguments for the kernel; and arguments for the application. Command line arguments to kernel are given after a double-dash and command line arguments for the application being simulated are given after another double-dash. If there is only one double-dash, all the arguments after it are passed into application. For example, to simulate the application program `sor` with an active list of size 64 and with the kernel parameters "`-a 10`" and the application parameters "`-p1 -m32 -n16 -i4`", one would use the command line:

```
rsim -a64 -f sort -- -a 10 -- -p1 -m32 -n16 -i4
```

The remainder of this section describes the command line parameters. In each case, **num** specifies a non-negative integer and **file** represents a file name on the host file system (may be relatively or absolutely specified). Other option specifiers are explained as needed below.

[1]We use the terms configuration file and parameter file interchangeably.

3.1.1 Parameters related to simulation input/output

In this section, we distinguish between "simulation" input and output and "standard" input and output. "Standard" input and output refer to the standard input and output streams provided to the application being simulated. By default, these are the same as the input and output streams used by the simulator. However, the simulator input and output streams can be redirected separately from the application input and output, as described below. Note that simulator and application share the same standard error file.

-0 file Redirects standard input to **file**. Defaults to stdin.

-1 file Redirects standard output to **file**. Defaults to stdout.

-2 file Redirects standard error to **file**. The simulator outputs its error messages and concise statistics to this **file**. Defaults to stderr.

-3 file Redirects simulator detailed statistics to **file**. This option can be used (either alone or in conjunction with "-1") to redirect detailed statistics separately from the output produced by the application. If "-1" is used without "-3", both detailed statistics and application output are written to the same file. Defaults to stdout.

-d dir Directory for output files. This option can only be used in conjunction with the "-s" option. Unused by default.

-s subj Subject to use in output filenames. This option overrides "-1", "-2", and "-3", and can only be used in conjunction with "-d". When this option is used, URSIM redirects application standard output to a file titled "dir/**subj**_out", redirects application standard error and simulator concise statistics to "dir/**subj**_err", and redirects simulator detailed statistics to "dir/**subj**_stat". Unused by default.

-E emailaddr Send an email notification to the specified address upon completion of this simulation. The notification tells the user the location of the various output files and is sent using the subject specified in "-s". Unused by default.

-z file Redirects configuration file to **file**. Defaults to *rsim_params* under current directory.

3.1.2 Simulator control and debugging parameters

-f file Name of application **file** to interpret with URSIM, without the **.rsim_dec** suffix. No default.

-A num Every **num** minutes, the simulator will print out partial statistics, which simply provide the number of cycles since each processor last graduated an instruction. These are typically used to determine if incorrect application synchronization or simulator source code modification has caused a deadlock. Defaults to 60. Used only when **procstaton**, which is specified in the configuration file, is non-zero.

-T num,count[,file] Starting trace instruction graduation after **num** instructions have been graduated, for number of **count** instructions. Tracing instruction graduation mean printing out the detail information about each instruction executed in program order. If no **file** is specified, standard output will be used.

-C num,file[,] Create check-point at **num** cycle to **file**. **[,]**:with it, simulation stops after check-point is created; without it, simulation continues.

-c num Maximum number of cycles to simulate. Unused by default.

3.1.3 Processor parameters

-a num Active list size. Defaults to 64.

-i num Number of instructions to fetch in a cycle. Defaults to 4.

-g num Maximum number of instructions to graduate per cycle. If the value 0 is given, then the processor will be able to graduate an unbounded number of instructions per cycle. Defaults to the same value as the instruction fetch width (specified in "-i", or 4 if no "-i" is given).

-e num Number of instructions to flush per cycle (from the active list) on an exception. If the value of 0 is given, the processor will flush all instructions immediately on an exception. Defaults to the same value as the graduation rate.

-r num Number of register windows.

-u Simulate fast functional units — all ALU and FPU instructions have single cycle latency. This option overrides any latencies specified in the configuration file.

-q num,num Many processors include one or more issue windows (corresponding to different sets of functional units) separate from the active list. These issue windows only hold instructions that have not yet issued to the corresponding functional units (or, in the case of memory instructions, instructions that have not completed all of their ordering constraints). Thus, the issue logic only needs to examine instructions in the corresponding windows for outstanding dependences. The "-q" option supports a processor that has separate issue windows for memory and non-memory instructions, and stalls further instruction decoding when a new instruction cannot obtain a space in its issue window. The first number specified with this option represents the size of the issue window for non-memory operations. The second number represents the size of the memory unit, and overrides any earlier use of the "-m" option below). Note that when "-q" is not used, the processor still supports a memory unit, but does not stall if the memory unit is full. This option has not yet been extensively tested. Unused by default.

-X Static scheduling. The static scheduling supported in URSIM includes register renaming and out-of-order completion. Memory instructions are considered issued once they have been sent to their address generation units; memory fences and structural hazards beyond that point may cause additional delays. (Do not use this for Impulse experiments.)

3.1.4 Memory unit parameters

-M Turn off memory system.

-l num Latency of each memory transaction when memory system is turned off using '-M'.

-m num Maximum number of operations in the processor memory unit, described in Section 1.2.3. Defaults to 32.

-L num Represents the memory ordering constraint for uniprocessor data dependences in the situation of a load past a prior store with an unknown address (as described in Section 1.2.3). The following table specifies the policies supported:

- **0** — Stall load until all previous store addresses known.
- **1** — Issue load, but do not let other instructions use load value until all previous store addresses known.
- **2** — Issue load and let other instructions use load value even when addresses of previous stores are unknown. If prior store later discovered to have conflicting address, cause soft exception. This is the default.

-K Enable speculative load execution.

-P num Turn on hardware-controlled prefetching

- **1** —- bring all hardware prefetches to L1 cache.
- **2** —- bring all hardware prefetches to L2 cache.
- **3** —- Same as **1**, but brings write prefetches to L2 cache.

-d Discriminate prefetching. If a hardware or software prefetch is stalled for resource constraints at the L1 cache, it will be dropped (to make place for later demand accesses that may also be stalled).

-x Drop all software prefetches. Useful only for measuring instruction overhead of prefetching.

-N Store buffering in SC: allows stores to graduate before completion [7].

-6 Processor consistency, if URSIM is compiled with -DSTORE_ORDERING. URSIM compiled with -DSTORE_ORDERING provides sequential consistency by default.

3.1.5 Approximate simulation models

-s Turn off ILP simulation.

This parameter allows URSIM to simulate simple processors with single instruction issue, static scheduling, and blocking reads, possibly with increased processor and/or cache clock rates. This approximate simulation model is not intended to speed up the performance of URSIM, but is provided only for purposes of comparison.

3.2 Configuration file

Most configuration inputs are passed to URSIM through the configuration file (which can be redirected using “-z” above). Sample configuration files are provided in the **bin** directory.

The format of the configuration file is very simple. Blank lines and lines beginning with a “#” are ignored. In each line, characters after “#” are takes as comments and ignored. Each parameter in the input file will be followed by either an integer, or a float, or a string, as specified below. The parameters are case sensitive. If any parameter is listed multiple times in the configuration file, the first one specifies the actual value used. The parameters that can be specified in the configuration file are given below.

3.2.1 Overall system parameters

numnodes The number given with this parameter specifies the number of nodes in the system. Defaults to 1.

cpus_per_node The number given with this parameter specifies the number of processors that each node in the system contains. Defaults to 1.

Critical_word_first Whether or not the memory system sends cirtical word back first.

3.2.2 Processor parameters

procstaton Whether or not collect statistics about processor module. Defaults to 0.

numalus This number specifies the number of ALU functional units in the processor. Defaults to 2.

numfpus This number specifies the number of FPU functional units in the processor. Defaults to 2.

numaddrs This number specifies the number of address generation units in the processor. Defaults to 2.

regwindowsi This number gives the number of register windows in the processor (one of these is always reserved for the system). Must be a power of 2 between 4 and 32, inclusive. Defaults to 8.

bpbtype Type of branch predictor included in the processor. The argument is a string, and is specified as follows:

2bit	2-bit history predictor. This is the default.
2bitagree	2-bit agree predictor
static	static branch prediction using compiler hints

bpbsize This number specifies the number of counters in the branch prediction buffer (unused with static branch prediction). Defaults to 512.

rassize The number provided here sets the number of entries in the return address stack. Defaults to 4.

shadowmappers This number controls the number of shadow mappers provided for branch prediction. Defaults to 8.

The following pairs of parameters specify the latencies and repeat delays of ALU and FPU instructions. In each pair, the first element specifies the latency, while the second specifies the repeat delay. The latency is the number of cycles after instruction issue that the calculated value can be used by other instructions. The repeat delay is the number of cycles after the issue of an instruction that the functional unit type used is able to accept a new instruction (a value of 1 indicates fully-pipelined units). Each parameter below is expected to be followed by a positive integer used to specify the value of the corresponding parameter.

latint,repint Latency and repeat delay for common ALU operations — addition, subtraction, move, and logical operations. Default latency is 1 cycle and repeat delay is 1 cycle.

latmul,repmul Latency and repeat delay for integer multiply operations. Default latency is 3 cycles and repeat delay is 1 cycle.

latdiv,repdiv Latency and repeat delay for integer divide operations. Default latency is 9 cycles and repeat delay is 1 cycle.

latshift,repshift Latency and repeat delay for integer shift operations. Default latency is 1 cycle and repeat delay is 1 cycle.

latflt,repflt Latency and repeat delay for common FP operations (e.g., add, subtract, multiply). Default latency is 3 cycles and repeat delay is 1 cycle.

latfmov,repfmov Latency and repeat delay for simple FP operations (e.g., move, negate, absolute value). Default latency is 1 cycle and repeat delay is 1 cycle.

latfconv,repfconv Latency and repeat delay for FP conversions (e.g., int-fp, fp-int, float-double). Default latency is 5 cycles and repeat delay is 2 cycle.

latfdiv,repfdiv Latency and repeat delay for FP divide. Default latency is 10 cycles and repeat delay is 6 cycle

latfsqrt,repfsqrt Latency and repeat delay for FP square-root. Default latency is 10 cycles and repeat delay is 6 cycle.

3.2.3 TLB parameters

TLB_on Turns on/off the TLB.

TLB_fakemiss Set only when there is no virtual memory system. If it's set, the TLB simulator implements each TLB miss by delaying it a fixed number of cycles and the physical address is equal to the virtual address.

TLB_fakemiss_psize Specifies the page size used by fakemiss.

TLB_debug Enables TLB debugging support.

TLB_num_entries Specifies the number of entries in the TLB.

TLB_assoc Specifies the associativity of the TLB.

TLB_io_miss_cycles Specifies the number of cycles needed for an I/O TLB miss, used only when **fakemiss** is set.

TLB_dmiss_cycles Specifies the number of cycles needed for a data TLB miss, used only when **fakemiss** is set.

TLB_imiss_cycles Specifies the number of cycles needed for an instruction TLB miss, used only when **fakemiss** is set.

3.2.4 Cache hierarchy parameters

Cache_collect_stats Specifies whether or not collect statistics for the cache hierarchy. Defaults to 1.

Cache_mshr_coal This number specifies the maximum number of requests that can coalesce into a cache MSHR or a write buffer line. Defaults to 16 (64 is the maximum allowable).

Cache_frequency Decrease cache access speed by the specified factor. Defaults to 1.

L1C_prefetch Turn on/off hardware L1 cache prefetching. Defaults to 0.

L1C_perfect Simulate a perfect L1 cache. Perfect L1 cache means every cache access is a hit. It is used to test the best performance that an application can possibly achieve.

L2C_prefetch Turn on/off hardware L2 cache prefetching. Defaults to 0.

L2C_perfect Simulate a perfect L2 cache.

L1C_writeback This parameter specifies the L1 cache type. If "0" is chosen, a write-through cache with no write-allocate is used. If "1" is chosen, a write-back cache with write-allocate is used. With a write-through cache, the system will also have a coalescing write-buffer. (In either case, the secondary cache is write-back with write-allocate.) The default is 0.

L1C_size This number specifies the size of the L1 cache in kilobytes. Defaults to 32.

L1C_assoc This number specifies the set associativity of the L1 cache. The cache uses LRU replacement policy within each set. Defaults to 2.

L1C_line_size The number given here specifies the cache-line size of L1 cache in bytes. Defaults to 32.

L1C_ports Specifies the number of cache request ports at the L1 cache. Defaults to 2.

L1C_tag_latency Specifies the cache access latency at the L1 cache (for both tag and data access). Defaults to 1.

L1C_tag_repeat Specifies the number of cycles must elapse before the L1 cache can serve the next access. Defaults to 1.

L1C_wbuf_size If a write-through L1 cache is used, this parameter specifies the number of cache lines in the coalescing write-buffer. With a write-back L1 cache, this parameter is ignored. Defaults to 8.

L1C_mshr_num Specifies the number of MSHRs (miss status hold register) for L1 cache.

L2C_size This number specifies the size of the L2 cache in kilobytes. Defaults to 256.

L2C_assoc This number specifies the set associativity of the L2 cache. The cache uses LRU replacement policy within each set. Defaults to 4.

L2C_line_size The number given here specifies the cache-line size of L2 cache in bytes. Defaults to 64.

L2C_tag_latency Specifies the cache access latency of the L2 cache tag array. Defaults to 3.

L2C_tag_repeat Specifies the repeat rate of the L2 cache tag array. Defaults to 3.

L2C_data_latency Specifies the cache access latency of the L2 cache data array. Defaults to 4.

L2C_data_repeat Specifies the repeat rate of the L2 cache data array. Defaults to 2.

L1C_mshr_num Specifies the number of MSHRs (miss status hold register) for L1 cache.

3.2.5 Bus parameters

BUS_width Bus width. Defaults to 16 bytes.

BUS_arbdelay Delay of arbitration. Defaults to 3.

BUS_frequency Bus clock rate relative to processor's clock rate. The frequency of the bus is the frequency of the processor divided by the number given. Defaults to 1.

BUS_addr_cycles The number of cycles needed to transfer an address on the bus. Defaults to 1.

BUS_data_cycles The number of cycles needed to transfer a cache line on the bus. Defaults to L2 cache line size divided by bus width.

3.2.6 Memory controller parameters

MMC_sim_on Turns on/off detailed memory controller simulator. When its value is 0, each memory access is satisfied by a fixed number of cycles specified by **MMC_latency**. Defaults to 1.

MMC_latency Used in conjunction with *MMC_sim_on* to specify the number of cycles that each memory access takes. Defaults to 20.

MMC_frequency Specifies how many times main memory controller clock is slower than the processor's. Defaults to 1.

MMC_debug Enables debugging support. Defaults to 0.

MMC_collect_stats Specifies whether or not to collect statistics about MMC module. Defaults to 1.

MMC_prefetch_on Turns on/off MC-based prefetching.

- **1,4** — prefetch non-shadow data only
- **2,5** — prefetch shadow data only
- **3,6** — prefetch both non-shadow and shadow data
- **1,2,3** — to issue prefetch right after the triggered transaction
- **4,5,6** — to issue prefetch when the MMC is free

MMC_cache_size The size of the MCache in cache-lines. Defaults to 64.

MMC_cache_assoc The associativity of the MCache. Defaults to 4.

MMC_writeupdate Use write-update protocol in the MCache instead of write-invalidate. Defaults to 0.

MMC_replacement Specifies the replacement policy of the MCache. It can be "FIFO", or "NRU", or "LRU". Defaults to "FIFO".

MMC_perfect_cache This parameter is set when we want to test the best potential performance of the MCache. Defaults to 0.

MMC_saddr_check Number of cycles to check whether a physical address is a DRAM address or a shadow address. Defaults to 1.

MMC_buffer_size The size of the buffer inside each shadow descriptor in cache-lines. Defaults to 4.

MMC_buffer_assoc The associativity of the buffer inside each shadow descriptor. Defaults to 4.

The following parameters are related the memory controller TLB.

MTLB_num How many banks that the MTLB has. Defaults to one for each shadow descriptor.

MTLB_numentries Number of entries in each MTLB bank. Defaults to 16.

MTLB_associativity Associativity of each MTLB bank. Defaults to 1.

MTLB_buffer_num The size of the MTLB prefetch buffer in number of cache-line. Defaults to 2.

MTLB_prefetch_on Turns on/off the MTLB prefetch. When it's set, the MTLB will prefetch ahead the memory controller table entry sequentially. Defaults to 1.

MTLB_collect_stats 0 means no statistics; 1 means generic statistics; 2 means detailed statistics. Defaults to 1.

MTLB_debug Enable debugging support. Defaults to 0.

3.2.7 DRAM backend parameters

Please consult document [21] for terminologies used below.

DRAM_sim_on Turns on/off DRAM simulator. When the DRAM simulator is off, each DRAM access takes a fixed latency to satisfy.

DRAM_latency Used to specify the latency of each DRAM access when **DRAM_sim_on** is "0".

DRAM_frequency Specifies the DRAM clock rate relative to the processor's. Defaults to 1 for RDRAM and 3 for SDRAM.

DRAM_scheduler_on Enables aggressive reordering algorithm.

DRAM_debug_on Enables debugging support.

DRAM_collect_stats Collects statistics in the DRAM simulator.

DRAM_trace_on Trace DRAM accesses.

DRAM_trace_maximun The maximum number of DRAM accesses should be traced.

DRAM_trace_file The file to store trace information. Defaults to stdin.

Parameters related the configuration of the memory system.

DRAM_sa_busses The number of SA busses. Defaults to 1.

DRAM_sa_bus_cycles SA bus frequency relative to the processor's. Defaults to DRAM_frequency.

DRAM_sd_busses The number of SD busses. Defaults to 1.

DRAM_sd_bus_cycles SD bus frequency relative to the processor's. Default value is DRAM_frequency.

DRAM_sd_bus_width SD bus width. Default value is 16.

DRAM_num_smcs The number of slave memory controllers. Default value is 4.

DRAM_num_jetways The number of jetways. Default value is 2.

DRAM_num_banks The number of memory banks. Default value is 16.

DRAM_banks_per_chip How many banks each chip has. Default value is 2 for SDRAM and 8 for RDRAM.

DRAM_bank_depth Depth of each bank. Default value is 8.

DRAM_interleaving Specifies interleaving scheme described in Section 1.7.5. The default value is 0.

- **0** — Cache-line-level modulo-interleaving.
- **1** — Page-level modulo-interleaving
- **2** — Cache-line-level sequential-interleaving
- **3** — Page-level sequential-interleaving

DRAM_bqueue_policy Specifies the bank queue reordering algorithm described in Section 1.7.4. Options "1" to "5" are the alternatives of the algorithm described in that section. The default value is 0.

- **0** — No reordering, or say it's FCFS.
- **1** — Giving direct access priority over shadow access, but without priority updating – step 6 in the original algorithm.
- **2** — Giving shadow access priority over direct access, without priority updating.
- **3** — Giving shadow access and direct access the same priority, without priority updating.
- **4** — Same as option **1**, except it allows priority updating, i.e., increasing priority with increased waiting time.
- **5** — Same as option **2**, except it allows priority updating.

DRAM_hot_row_policy Specifies the how row policy described in Section 1.7.3. The default value is 0.

- **0** — Always close the hot row.
- **1** — Always leave the hot row open.
- **2** — Use predictor.

DRAM_max_bwaiters The maximum number of DRAM accesses can be sent to DRAM backend by the MMC. Default value is 256.

DRAM_type DRAM type, either "SDRAM" or "RDRAM". Default is "SDRAM".

DRAM_width Width of each DRAM bank. Default is 16.

DRAM_mini_access Minimum DRAM access in bytes. Default is 16.

DRAM_block_size The block size in bytes. Default is 128.

SDRAM parameters:

SDRAM_tCCD CAS to CAS delay time. Defaults to 1.

SDRAM_tRRD Bank to bank delay time. Defaults to 2.

SDRAM_tRP Precharge time. Defaults to 3.

SDRAM_tRAS Minimum bank active time. Defaults to 7.

SDRAM_tRCD RAS to CAS delay time. Defaults to 3.

SDRAM_tAA CAS latency. Defaults to 3.

SDRAM_tDAL Data in to precharge time. Defaults to 5.

SDRAM_tDPL Data in to active/refresh time. Defaults to 2.

SDRAM_row_size The size of an active row in bytes.

SDRAM_row_hold_time How long can a hot row be active.

SDRAM_refresh_delay Delay of a refresh operation.

SDRAM_refresh_period Refresh period.

RDRAM parameters:

RDRAM_tPACKET Length of each command. Defaults to 4.

RDRAM_tRC The minimum delay from the first ACT command to the second ACT command. Defaults to 28.

RDRAM_tRR Delay from a RD command to next RD command. Defaults to 8.

RDRAM_tRP The minimum delay from a PRER command to an ACT command. Defaults to 8.

RDRAM_tCBUB1 Bubble between a RD and a WR command. Defaults to 4.

RDRAM_tCBUB2 Bubble between a WR and a RD command to the same device. Defaults to 8.

RDRAM_tRCD RAS to CAS delay. Defaults to 7.

RDRAM_tCAC Delay from a RD command to its associated data out. Defaults to 8.

RDRAM_tCWD CAS write delay. Defaults to 6.

RDRAM_row_size The size of an active row in bytes.

RDRAM_row_hold_time How long can a hot row be active.

RDRAM_refresh_delay Delay of a refresh operation.

RDRAM_refresh_period Refresh period.

3.2.8 Tracing and debugging parameters

MemTraceOn Turn on/off memory request tracing. The parameter, together with **MemTraceSample**, is used to print out some reminding message that shows the simulator is making progress.

MemTraceSample For how many memory requests, a reminding message is printed.

MemTraceMax When to stop tracing memory requests.

MMC_trace_start Trace the timeline of each memory transaction and print out the times that it enters and leaves each component. This parameter set which memory transaction to start tracing.

MMC_trace_end The parameter specifies when to stop tracing.

MMC_trace_sonly If this parameter is non-zero, only shadow access will be traced.

MMC_trace_file Specifies which file the tracing information is sent to.

TraceLevel Set system trace level. Any warning messages whose level is less than or equal to the number given is sent to simulator error file. This parameter decides what kind warning messages should be printed out. URSIM ranks the warning messages from level 0 (serious warning messages, may cause fatal problems, and must be printed out at any circumstance) to level 5 (the least important warning messages). In debugging mode, the trace level is usually set to be 1 or 2.

Part II

DEVELOPER'S GUIDE

Chapter 4

Overview of URSIM Implementation

The remainder of this manual describes the implementation of URSIM. It is intended for users interested in modifying URSIM, and assumes an understanding of Part I of this manual.

URSIM is organized as a discrete-event-driven simulator. The central data structure of such a simulator is an event list consisting of events that are scheduled for the future in simulation time. Typically, an event for a hardware module is scheduled for a given time only when it is known that the module will need to perform some action at that time. Thus, discrete-event-driven simulators typically do not perform an action for every cycle. In URSIM, however, the processors and cache hierarchies are modeled as a single event (called **RSIM_EVENT**), which is scheduled every cycle. This is because we expect that some activity will be required of the processor and caches every cycle. Note that URSIM is not a pure cycle-by-cycle simulator since events for the bus, memory controller, and DRAM backend are scheduled only when needed. The underlying event library and the processor module original came from RSIM (**R**ice **S**imulator for **I**LP **M**ultiprocessors), which models a MIPS R10000-like microprocessor [8]. We profiled and applied some optimizations on the event library and processor module without losing any accuracy of the processor module. The optimized version is two–five times faster than the original one. The cache hierarchy and cluster bus are also modeled based on MIPS R10000 microarchitecture. The main memory controller and DRAM backend are simulated based on the current Impulse design and opt to change whenever a new Impulse design comes out.

URSIM is implemented in a modular fashion for ease of development and maintenance. The primary subsystems in URSIM are the event-driven simulation library, the processor out-of-order execution engine, the processor memory unit, the cache hierarchy, the main memory controller, and the DRAM backend. These modules perform the following roles:

Event-driven simulation library Steers the course of the simulation. This subsystem is based on the YACSIM event-driven simulation library [5, 12].

Processor out-of-order execution engine Maintains the processor pipelines described in Section 1.2.

Processor memory unit Interfaces between the processor pipelines and the caches, maintaining the various ordering constraints described in Section 1.2.3 and implementing CPU TLB functionality.

Cache hierarchy Processes requests to the caches, including both demands from the processor and demands from bus.

Main memory controller module Processes requests from processors, maintaining the cache coherence protocol of the system. This module includes the core of the Impulse Adaptive Memory System — Impulse Remapping Controller.

DRAM backend Processes DRAM access requests from main memory controller. It optimizes the dynamic ordering of accesses to the actual DRAM chips.

Each of the above subsystems acts as a largely independent block, interacting with the other units through a small number of predefined mechanisms. Thus, we expect most modifications to URSIM to be quite focused, and affecting only the desired functionality. However, each type of simulator change does require detailed knowledge of the subsystem being modified.

The remaining chapters in this part describe the above subsystems in detail and provide other additional information needed to understand the implementation of URSIM. Chapter 5 gives a brief explanation of the event-driven simulation library underlying URSIM and the manner in which URSIM uses it. Chapter 6 describes the initialization routines in URSIM. Chapter 7 gives an overview of RSIM_EVENT and describes the details of each stage in the processor out-of-order execution engine. Chapter 8 explains the implementation of the processor memory unit. Chapter 9 explains the key functions within the simulation of the cache hierarchy. Chapter 10 illustrates the bus module simulated based on R10000 system interface. Chapter 11 describes the implementation of Impulse main memory controller. Chapter 12 shows the current Impulse DRAM backend. Chapter **??** gives information about other important functions provided by URSIM, including some useful utility functions, statistics collection library, and **predecode** utility.

Chapter 5

Event-driven Simulation Library

The event-driven simulation library underlies the entire URSIM simulation system, guiding the course of the various subsystems. The event-driven simulation library used in URSIM is a subset of the YACSIM library distributed with the Rice Parallel Processing Testbed [5, 12].

Section 5.1 describes the YACSIM event-manipulation functions used by URSIM. Section 5.2 describes the fast memory-allocation pools used by URSIM.

5.1 Event-manipulation functions

Source file: **sim_main/evlst.c**[1]

Header file: **sim_main/evlst.h**

All actions that take place during the course of a URSIM simulation occur as part of YACSIM **event**s. Each event has a function for its body, an argument for use on invocation, and a state used for subsequent invocations of the same event. Each time an event is scheduled, the body function is invoked. The event is not deactivated until the body function returns control to the simulator (through a return statement or the end of the function). Thus, an event can be thought of as a function call scheduled to occur at a specific point in simulated time, possibly using a previously-set argument and state value and/or setting a new state value and argument for use on a future invocation.

The following functions or macros are used for manipulating events in URSIM.

[1] All the source files and header files are stored under directory **simpulse/src**. Suffixes convention: `.cc` — C++ source file; `.c` — C source file; `.h` —- header file; `.hh` — inline functions.

EVENT *NewEvent(char *ename, void (*bodyname)(), int delflg, int etype)

This function constructs a new event and returns its pointer. The state of the event is initialized to 0. The ename argument specifies the name of the event and is used for debugging purpose only. bodyname is a pointer to a function that will be invoked on each activation of the event. The function must take no arguments and must have no return value; the argument actually used by the event is passed in through EventSetArg() described below and is read with EventGetArg(). delflg can be either DELETE or NODELETE, and specifies whether the storage for the event can be freed at the end of its invocation. Events specified with DELETE can only be scheduled once, whereas NODELETE events can reschedule themselves or be rescheduled multiple times. However, URSIM never uses any event with this field set to DELETE. The etype argument is available for any use by the user of the event-driven simulation library. URSIM events always have this field set to 0.

int EventSetState(int stval)

This function can only be called within the body function of an event, and it sets the state value of the event to stval.

int EventGetState()

This function returns the state value of the calling event, and can be used at the beginning of its body function to determine the current state of the event.

void EventSetArg(EVENT *vptr, char *argptr, int argsize)

This function sets the argument of the event pointed to by vptr to the value of argptr, with argsize indicating the size of the argument structure in bytes. Note that the argument is passed in by pointer; consequently, the value of the argument structure at the time of event invocation may differ from the value of the argument structure at the time when the argument is set, if intervening operations reset the value of the structure.

char *EventGetArg(EVENT *vptr)

This function returns the argument pointer for a given event; if this function is called with a NULL pointer or the predefined value ME, the function returns the argument pointer for the calling event.

void EventSchedTime(EVENT *vptr, double timeinc, int blkflg)

This operation schedules the event pointed to be vptr for timeinc cycles in the simulated future. Block flag blkflg indicates how the event is schedule and can be INDEPENDENT, BLOCK, or FORK. The only valid value for the current URSIM events is INDEPENDENT.

double YS_EventListHeadval()

This function returns the time value (i.e., the time that the event's body function is scheduled to be called.) of the first event in event-list. If the event list is empty, it return -1.

schedule_event(EVENT *vptr, double activetime)

This macro schedules the event pointed by vptr to be activated at time activetime. Note that the user must ensure the activetime is larger than the current simulation time. Otherwise, the behavior would be undetermined.

The YACSIM event-list is implemented either as a calendar queue [2] or as a straightforward linear queue. The calendar queue is effective only when there are big number of events in the system. Since URSIM uses linear queue because there are only few active events existing in the system during most of the simulation time. Event-list processing in YACSIM is controlled by the function DriverRun(double period), which processes the event list for period cycles if period is larger than 0, or until the event list has no more events scheduled if the value of period given is less than or equal to 0. URSIM uses “0” since we always run applications to the end.

5.2 Memory allocation utility functions

Source file: **sim_main/pool.c**

Header file: **sim_main/pool.h**

Many of the objects used in the event-driven simulation library and memory system simulator are allocated using the YACSIM pool functions, which seek to minimize the number of calls to malloc() and free(). Each structure in the pool must begin with the following two fields:

```
char *pnxt char *pfnxt
```

These fields maintain the pool and the free list for the pool. The pool functions supported in URSIM are:

void YS__PoolInit(POOL *pptr, char *name, int objs, int objsz)

This function initializes the pool pointed to by pptr, setting the name of the pool to name and declaring that this pool will allocate structures of size objsz (this size includes the pnxt and pfnxt fields). Whenever the pool runs out of available objects, it will allocate objs structures of size objsz from the system memory allocator.

char *YS__PoolGetObj(POOL *pptr)

This function returns an object from the given pool. If this pool does not have any objects, it should allocate the number of objects specified on the original call to YS__PoolInit.

YS__PoolReturnObj(POOL *pptr, void *optr)

This function returns the object pointed to by optr back to the pool pointed to by pptr, from which the object was allocated. Indeterminate results will occur if an object is returned to a different pool than the one from which it was allocated.

YS__PoolStats(POOL *pptr)

This function prints the number of objects allocated from and returned to a given pool. This function can be used to detect memory leaks in certain cases.

Users further interested in the YACSIM simulation library should consult the YACSIM reference manual [12].

Chapter 6

Initialization and Configuration Routines

Source files: **sim_main/main.c, Processor/mainsim.cc, Processor/config.cc, Processor/funcsunits.cc, Processor/funcs.cc, Processor/tlb.cc, Caches/system.c**

Header files: **Processor/mainsim.h, Processor/simio.h, Processor/funcunits.h, Caches/system.h**

URSIM execution starts with the main function provided by YACSIM. This function takes the arguments passed in on the command line and passes them to the UserMain function in **Processor/mainsim.cc**, which performs the following tasks.

Parsing command-line arguments

The first purpose of the UserMain function is to parse the command-line arguments. The appropriate global variables (e.g., the size of the active list, the number of register windows) are set based on the options described in Chapter 3.

Setting up input and output files

The various input and output files used by the simulator and application are redirected according to the command-line options. The FILE data structure called simout, which defaults to stdout, is specified through option "-3". The FILE data structure called simerr, which defaults to stderr, is specified through option "-2". simerr is used to print out warning or error messages generated by URSIM. simout is used to output simulator statistics. Three int data structures called appstdin, appstdout, and appstderr, which defaults to stdin, stdout, and stderr and behave as application's standard input, output, and error file, are specified through options "-0", "-1", and "-2". The default configuration file is rsim_params, which can be redirected through option "-z".

Reading processor configuration from parameter file

Next, the function ParseConfigFile is invoked to read in the options from the configuration file (described in Chapter 3) and set global simulation variables related to processor module. Each parameter recognized by ParseConfigFile is associated with a global variable and a parsing function in the table called configparams. The parsing function is used to convert the operand given for a parameter into an acceptable input value. For example, the ConfigureInt function merely calls atoi to read a string into an integer variable. The parameter names and values are currently case-sensitive.

Reading in the simulated application

The application to be simulated is chosen based on the command line option "-f". The predecoded version of the application executable is read through the read_instructions function. This sets the num_instructions variable according to the number of instructions in the application.

Assigning emulated functions to each instruction

UserMain calls the UnitArraySetup function, which defines the functional unit used by each of the instruction types. For memory instructions, this function also specifies the type of memory access and the amount of data read or written by each memory instruction, as well as the address alignment needed. The FuncTableSetup function is called next to assign each instruction type to the function that emulates its behavior at the functional units.

Initiating TLB

TLB_init function is called to read in TLB-related parameters from the configuration file and to set up necessary data structures to simulate a superpage-supported TLB.

Setting up the memory system

Next, the SystemInit function is called to set up the URSIM memory system. It calls functions Cache_init, Bus_init, MMC_init, and DRAM_init to initiate cache, bus, main memory controller, and DRAM backend module. These functions will be described in their respective chapters.

Creating the first processor

After this point, the first processor data structure (ProcState) is created. The constructor for this structure sets up fundamental state parameters and initializes the auxiliary data structures used in the processor pipeline (described in Section 7.9).

Each ProcState has two important page tables: VPageTable and PPageTable. They map the simulated virtual addresses and physical addresses to URSIM UNIX addresses respectively. VPageTable is necessary to emulate some system traps because the parameters of system calls pass virtual addresses, not physical addresses, into the URSIM. However, VPageTable can not handle virtual addresses mapped to shadow addresses because such a virtual page may not correspond to one and only one real physical page. That is why another page table – PPageTable – is needed for mapping shadow regions.

Now that the first processor data structure has been created, the system must load the application executable and data segment into the processor's address space, and must initialize the processor's kernel stack, user stack, user data segment, and register set. The startup function performs each of these actions. The kernel stack is set up to hold the starting address and size of user data segment, the size of initial user stack, and the command line arguments passed to the kernel. The user stack is set up to hold the command line arguments passed to the application, and the registers %o0 and %o1 are set up to hold the corresponding values of argc and argv. (Environment variables are not currently supported.) The PC (program counter) is set to the entry point of the application executable, while the NPC (next program counter) points to the subsequent instruction.

Creating and starting RSIM_EVENT

After this point, the RSIM_EVENT is scheduled for execution, and the event-driven simulator is started.

Chapter 7

RSIM_EVENT and Processor Engine

Section 7.1 gives an overview of RSIM_EVENT, the event corresponding to the processors and cache hierarchies. The rest of this chapter focuses on the processor out-of-order execution engine. The other subsystems handled by RSIM_EVENT are the processor memory unit and the cache hierarchy, and are discussed in Chapters 8 and 9.

The out-of-order execution engine is responsible for bringing instructions into the processor, decoding instructions, renaming registers, issuing instructions to the functional units, executing instructions at the functional units, updating the register file, and graduating instructions.

7.1 Overview of RSIM_EVENT

Source files: **Processor/mainsim.cc, Processor/exec.hh**

RSIM_EVENT is responsible for the processors and cache hierarchies of the simulated system. It is scheduled every cycle as described in Chapter 4. On every invocation, RSIM_EVENT loops through all the processors in the system, calling the functions described below.

1. RSIM_EVENT first calls TLB_sim to see if there are any returns of previous tlb_fill requests waiting for processing (which happens only when a fake TLB (described in Section 3.2.3) is simulated).
2. Then, it calls L1CacheOutSim, L1CacheWBufferSim (only when L1 cache is write-through), and L2CacheOutSim (described in Chapter 9), which are used to process cache accesses.
3. Next, it checks if there exists an exception detected at a previous cycle but having not been processed. It such an exception exists, PreExceptionHandler is called to process this exception.

4. Next, CompleteMemQueue is called to inform the memory unit of any operations that have completed at the caches.
5. Next, CompleteQueues is called to process instructions that have completed at their functional units.
6. Next, CompleteFreeingUnit is called to free functional units that have completed their functional unit delay.
7. Next, It calls maindecode. This function starts out by using update_cycle to update the register file and handle other issues involved with the complete stage of instruction pipeline. Next, graduate_cycle is called to remove previously completed instructions in-order from the active list and to commit their architectural state. Then, maindecode calls decode_cycle to bring new instructions into the active list. After this, maindecode returns control to RSIM_EVENT.
8. RSIM_EVENT then calls IssueQueues, which sends ready instructions to their functional units, and IssueMem, which issues new memory accesses to the processor memory unit.
9. After that, the functions L1CacheInSim and L2CacheInSim are called for the caches to bring in new operations that have been sent to them.

Each of the functions mentioned above is more thoroughly discussed in the chapter related to its phase of execution. In particular, CompleteQueues, update_cycle, graduate_cycle, maindecode, decode_cycle, and IssueQueues are part of the out-of-order execution engine, which is discussed in the next several sections.

7.2 Instruction fetch and decode

Source files: **Processor/exec.cc, Processor/exec.hh, Processor/tagcvt.hh, Processor/active.hh, Processor/active.cc, Processor/stallq.hh**

Header files: **Processor/ProcState.h, Processor/exec.h, Processor/instruction.h, Processor/tagcvt.h, Processor/active.h, Processor/stallq.h**

Since URSIM currently does not model an instruction cache, the instruction fetch and decode pipeline stages are merged. This stage starts with the function decode_cycle, which is called from maindecode and performs the following actions.

step 1a: Checking the stall queue

The function decode_cycle starts out by looking in the processor stall queue, which consists of instructions that were decoded in a previous cycle but could not be added to the processor active list, either because of insufficient renaming registers or insufficient active list size. The processor will stop decoding new instructions by setting the processor field stall_the_rest after the first stall of this sort, so the stall queue should have at most one element. If there is an instruction in the stall queue, check_dependencies is called for it (described below). If this function succeeds, the instruction is removed from the processor stall queue. Otherwise, the processor continues to stall instruction decoding.

step 1b-i: Picking up the right instruction

After processing the stall queue, the processor will decode the instructions for the current cycle. If the program counter is valid for the application instruction region, the processor will read the instruction at that program counter, and convert the static instr data structure to a dynamic instance data structure through the function decode_instruction. The instance is the fundamental dynamic form of the instruction that is passed among the various functions in URSIM. If the program counter is not valid for the application, the processor generates a single invalid instruction that will cause an illegal PC exception. Such a PC can arise through either an illegal branch or jump, or through speculation (in which case the invalid instruction will be flushed before it causes a trap).

step 1b-ii: Decoding the instruction

The decode_instruction function sets a variety of fields in the instance data structure.

First, the tag field of the instance is set to hold the value of the processor instruction counter. The tag field is the unique instruction ID of the instance; currently, this field is set to be unique for each processor throughout the course of a simulation. Then, the functional unit type field and the win_num field of the instance is set. win_num represents the processor's register window pointer (cwp or current window pointer) at the time of decoding this instruction.

Then, the various fields associated with the memory unit are cleared, and some fields associated with instruction registers and results are cleared. The relevant statistics fields are also initialized.

decode_instruction then initializes dependence fields for this instance. Additionally, the stall_the_rest field of the processor is cleared; since a new instruction is being decoded, it is now up to the progress of this instruction to determine whether or not the processor will stall.

At this point, the instance must determine its logical source registers and the physical registers to which they are mapped. In the case of integer registers (which may be windowed), the function convert_to_logical is called to convert from a window number and architectural register number to an integer register identifier that identifies the logical register number used to index into the register map table (which does not account for register windows). If an invalid source register number is specified, the instruction will be marked with an illegal instruction trap.

Then, the instance must handle the case where it is an instruction that will change the processor's register window pointer (such as SAVE or RESTORE). The processor provides two fields (CANSAVE and CANRESTORE) that identify the number of windowing operations that can be allowed to proceed [16]. If the processor can not handle the current windowing operation, this instance must be marked with a register window trap, which will later be processed by the appropriate trap handler. Otherwise, the instance will change its win_num to reflect the new register window number.

The instance will now determine its logical destination register numbers, which will later be used in the renaming stage. If the previous instruction was a delayed branch, it would have set the processor's copymappernext field (as described below). If the copymappernext field is set, then this instruction is the delay slot of the previous delayed branch and must try to allocate a shadow mapper. The branchdep field of the instance is set to indicate this.

Now the processor PC and NPC are stored with each created instance. We store program counters with each instruction not to imitate the actual behavior of a system, but rather as a simulator abstraction. If the instance is a branch instruction, the function decode_branch_instruction is called to predict or set the new program counter values (Section 7.2.1); otherwise, the PC is updated to the NPC, and the NPC is incremented. decode_branch_instruction may also set the branchdep field of the instance (for predicted branches that may annul the delay slot), the copymappernext field of the processor (for predicted, delayed branches), or the unpredbranch field of the processor (for unpredicted branches).

If the instance is predicted as a taken branch, then the processor will temporarily set the stall_the_rest field to prevent any further instructions from being decoded this cycle, as we currently assume that the processor cannot decode instruction from different regions of the address space in the same cycle.

After this point, control returns to decode_cycle. This function now adds the decoded instruction to the tag converter, a structure used to convert from the tag of the instance into the instance data structure pointer. This structure is used internally for communication among the modules of the simulator.

step 2: Checking dependencies

Now the check_dependencies function is called for the dynamic instruction. If URSIM was invoked with the "-q" option and there are too many unissued instructions to allow this one into the issue window, this function will stall further decoding and return. If URSIM was invoked with the "-X" option for static scheduling and even one prior instruction is still waiting to issue (to the ALU, FPU, or address generation unit), further decoding is stopped and this function returns. Otherwise, this function will attempt to provide renaming registers for each of the destination registers of this instruction, stalling if there are none available. As each register is remapped in this fashion, the old mapping is added to the active list (so that the appropriate register will be freed when this instruction graduates), again stalling if the active list has filled up. It is only after this point that a windowing instruction actually changes the register window pointer of the processor, updating the CANSAVE and CANRESTORE fields appropriately. Note that single-precision floating-point registers (referred to as REG_FPHALF) are mapped and renamed according to double-precision boundaries to account for the register-pairing present in the SPARC architecture [16]. As a result, single-precision floating-point codes are likely to experience significantly poorer performance than double-precision codes, actually experiencing the negative effects of anti-dependences and output-dependences which are otherwise resolved by register renaming.

If a resource was not available at any point above, check_dependencies will set stall_the_rest and return an error code, allowing the instance to be added to the stall queue.

After the instance has received its renaming registers and active list space, check_dependencies continues with further processing. If the instruction requires a shadow mapper (has branchdep set to

2, as described above), the processor tries to allocate a shadow mapper by calling AddBranchQ. If a shadow mapper is available, the branchdep field is cleared. Otherwise, the stall_the_rest field of the processor is set and the instance is added to the queue of instructions waiting for shadow mappers. If the processor had its unpredbranch field set, the stall_the_rest field is set, either at the branch itself (on an annulling branch), or at the delay slot (for a non-annulling delayed branch).

The instance now checks for outstanding register dependences. The instance checks the busy bit of each source register (for single-precision floating-point operations, this includes the destination register as well). For each busy bit that is set, the instruction is put on a distributed stall queue for the appropriate register. If any busy bit is set, the truedep field is set to 1. If the busy bit of rs2 or rscc is set, the addrdep field is set to 1 (this field is used to allow memory operations to generate their addresses while the source registers for their value might still be outstanding).

If the instruction is a memory operation, it is now dispatched to the memory unit, if there is space for it. If there is no space, either the operation is attached to a queue of instructions waiting for the memory unit (if the processor has dynamic scheduling and "-q" was not used to invoke URSIM), or the processor is stalled until space is available (if the processor has static scheduling, or has dynamic scheduling with the "-q" option to URSIM).

step 3: Issuing the instruction to the next stage

If the instruction has no true dependences, the SendToFU function is called to allow this function to issue in the next stage (see Section 7.3 for details).

step 4: Decoding the next instruction

decode_cycle continues looping until it decodes all the instructions allowed by the architectural specifications in a given cycle or the instruction pipeline stalls for various reasons explained above.

7.2.1 Branch prediction

Source files: **Processor/branchpred.hh, Processor/branchpred.cc**

Header files: **Processor/branchpred.h**

Although branch prediction can be considered part of instruction fetching and decoding, it is sufficiently important to be discussed separately. The decode_branch_instruction calls StartCtlXfer to determine the prediction for the branch.

If the branch is an unconditional transfer with a known address (either a call instruction or any variety of ba (branch always) instruction), StartCtlXfer returns -1 to indicate that the branch is taken non-speculatively. On call instructions, this function also adds the current PC to the return address stack. For other types of branches, this function either predicts them using the return address stack (for procedure returns) or the branch prediction buffer (for ordinary branches), or does not attempt to predict their targets (for calculated jumps).

Based on the return value of StartCtlXfer and the category of branch (conditional vs. unconditional, annulling vs. non-annulling), decode_branch_instruction sets the processor PC and NPC appropriately, as well as setting processor fields such as copymappernext (for speculative branches which always have a delay slot) and unpredbranch (for branches that are not predicted). Additionally, this function may set the branchdep of the instance for unpredicted branches or branches that may be annulling and thus need to associate a shadow mapper with the branch itself (rather than with a delay slot).

The function AddBranchQ is called by check_dependencies to allocate a shadow mapper for a speculative branch. If a mapper is available, this function copies the current integer and floating-point register map tables into the shadow mapper data structure. The shadow mapper is used to resume the processor state in case the speculation is wrong.

7.3 Instruction issue

Source files: **Processor/exec.hh, Processor/exec.cc, Processor/execfuncs.cc**

Header files: **Processor/exec.h**

This stage actually sends instructions to their functional units. The SendToFU function is called whenever an instruction has no outstanding true dependencies. This function reads the values of the various source registers from the register file and holds those values with the instance data structure. This mechanism is not meant to imitate actual processor behavior, but rather to provide a straightforward simulator abstraction. At the end of this function, the issue function is called if there is a functional unit available; otherwise, this instance is placed on a waiting queue for the specified functional unit.

The issue function places the specified instance in the ReadyQueue data structure and occupies the appropriate functional unit (for memory operations, this function is used for address generation).

The function IssueQueues processes instructions inserted in the ReadyQueues by issue. This function then inserts the appropriate functional unit onto the FreeingUnits data structure, specifying that that unit will be free a number of cycles later, according to the repeat delay of the instruction. This function places the instance itself on the Running heap structure of the processor, which is used to revive the instruction for completion after its functional unit latency has passed.

This stage assumes no limit on register file ports. In real processors, port contention may cause additional stalls that are not considered here.

7.4 Instruction execution

Source files: **Processor/funcs.cc, Processor/prostate.hh, Processor/branchpred.cc**

Header files: **Processor/funcs.h**

The actual execution of instructions at their functional units is simulated through the functions in the file **Processor/funcs.cc**. These functions use the source register values previously set in the SendToFU function and fill in the destination register values of the instance structure correspondingly.

Two instruction classes are significant with regard to their execution: branches and memory instructions. For each branch instruction executed at the functional units, the branch-prediction buffer state is updated appropriately to indicate the actual result of the branch. For memory instructions, the GetMap function is used to map from the simulated address of the reference to the corresponding address in the simulator's UNIX address space.

GetMap starts by checking if the address misses in the TLB.

- If it is a TLB miss, there is no need to perform translation at this time because TLB miss will cause memory instructions to be reissued after TLB miss handler returns.
- If it is an I/O address, which does not correspondent to any real physical pages, no needs to work on them neither.
- Shadow addresses can not be mapped to simulator addresses directly. They have to be translated into real physical addresses. In real hardware, the MMC gathers the data into cache lines and sends them back the cache. However, the URSIM only simulates the timing model; there is no real data flowing around the memory system. So a shadow address must be mapped to a real physical address in the URSIM processor. Since the MMC is the only place that knows how to map shadow addresses to real physical addresses, GetMap calls MMC_saddr2paddr provided by the MMC simulator to get the real physical address.
- At this point, we must have a real physical address. The final step of GetMap is to look up page table PPageTable, which stores the mapping from the simulated physical addresses to simulator UNIX addresses.

7.5 Instruction completion

Source files: **Processor/exec.cc, Processor/exec.hh, Processor/branchpred.cc, Processor/stallq.hh, Processor/active.cc**

Header files: **Processor/exec.h, Processor/ProcState.h**

The instruction complete stage performs the following steps.

Move from `Running` heap to `DoneHeap` heap

The CompleteQueues function processes instructions from the Running heap that have completed in a given cycle. For all non-memory instructions, this function calls the appropriate emulation function from **Processor/funcs.cc** and then inserts the instance into the processor's DoneHeap. For memory instructions, this function marks the completion of address generation (set addr_ready field to 1), and thus calls the Disambiguate function (described in Chapter 8).

Issue next instruction to the functional unit

The function CompleteFreeingUnit is called to free functional units that have completed their functional unit delay, as determined from the FreeingUnits data structure. As each functional unit is freed, the processor checks to see if a queue of ready instructions has built up waiting for that unit. If so, one instruction is revived, and the issue function is invoked.

Remove from `DoneHeap` and flag `done` in active list

The function update_cycle processes instructions from the DoneHeap data structure. For each instruction removed from the DoneHeap in a given cycle, update_cycle first sees if the completion of this instruction will allow a stalled processor to continue decoding instructions.

Next, update_cycle resolves completed branches. If the branch was unpredicted, update_cycle sets the processor PC and NPC appropriately and allows execution to continue. On a correct prediction, the GoodPrediction function is called. If this branch had already allocated a shadow mapper, this function calls RemoveFromBranchQ to free the shadow mapper, possibly yielding that shadow mapper to a later stalled branch. If the branch had not yet received a shadow mapper, it is no longer considered to be stalled for a mapper.

On the other hand, the BadPrediction function is called to resolve a mispredicted branch. If the branch (or its delay slot, as appropriate) had allocated a shadow mapper, CopyBranchQ is used to revive the correct register mapping table. After that, FlushBranchQ is used to remove the shadow mapper associated with the current branch and all later branches. Then, FlushMems is invoked to remove all instructions from the memory unit after the branch or delay slot in question. FlushStallQ removes any possible item in the processor stall queue, and is followed by FlushActiveList, which removes all instructions after the branch or delay slot from the active list. FlushActiveList also removes entries from the tag-converter data structure, frees the registers renamed as destinations for the instructions being flushed, and negates the effects of any register windowing operations being flushed. After BadPrediction returns control to update_cycle, the processor sets its PC and NPC appropriately.

update_cycle then updates the physical register file with the results of the completed instruction and marks the instruction in the active list as having completed. The busy-bits of the destination registers are cleared, and the instructions in the distributed stall queue for these registers are checked. If a waiting instruction now has no more true dependencies, the function SendToFU is called to provide the register values to that instruction and possibly allow it to issue. If a memory instruction in the memory unit had been waiting on a destination register for an address dependence which is now cleared, the CalculateAddress function (described in Chapter 8) is used to send the instruction to the address generation unit.

7.6 Graduation

Source files: **Processor/exec.hh, Processor/active.cc**

Header files: **Processor/exec.h**

The graduate_cycle function controls the handling associated with instruction graduation. First, the remove_from_active_list function is called. In this function, the processor looks at the head of the active list. If this operation completed in the previous cycle (and thus, has already had time to write its result into its register) and is not stalled for consistency constraints, the instruction is allowed to graduate from the active list. If an exception is detected, graduation is stopped and control is returned to graduate_cycle. If the instruction has no exception, then the old physical registers for its destinations are freed and the operation is graduated. As a simulator abstraction, URSIM also maintains a "logical register file", which stores committed values. This file is also updated at this time. The active list element is removed, and the instance is also freed for later use. remove_from_active_list repeats until the first operation in the active list is not ready to graduate, or an exception is detected, or the processor's maximum graduation rate is reached. At that point, control is returned to graduate_cycle.

However, when the exception code shows that the instruction is a URSIM trap (Section 7.7), remove_from_active_list will call the real URSIM trap handler RsimTrapHandler to perform associated operation and treat this instruction as a normal non-exception one.

If remove_from_active_list returned an exception, the processor is put into exception mode and will handle the exception as soon as possible, without decoding or graduating any further instructions in the meantime.

graduate_cycle also calls mark_stores_ready, in which stores are marked ready to send data to the data cache if they are within the next set of instructions to graduate. Namely, the store must be no further from the head of the active list than the processor graduation rate, and all previous instructions must be completed and guaranteed free of exceptions. The store itself must also have its address ready and must not cause any exceptions; the only exception type currently detected at the time of mark stores ready is a segmentation fault (other exceptions would have already been detected). Note that this function considers stores primarily with regard to their effect on precise exceptions; even after being marked ready in this fashion, a store may still have to wait many cycles to issue due to store ordering constraints. In any system with non-blocking stores (with the "-N" option), a store is considered ready to graduate as soon as it has been marked; it need not wait for issue or completion in the external memory system.

7.7 URSIM traps

URSIM provides some additional traps to allow kernel and application to control and communicate with the simulator. These traps usually do not correspond to any UNIX-like system calls. and are triggered by special instruction "`illtrap <trap_number>`".

Some of the URSIM traps are used for debug or statistics. Some of them are used to emulate some instructions that are included in MIPS-3 instruction set but are not included in SPARC V8Plus instruction set. Some of them are used to get the configuration of URSIM. Some of them are used to improve the simulation speed. The following table lists the supported URSIM traps.

Trap 1 reports fatal kernel error.

Trap 3 indicates a return of TLB fill request.

Trap 4 indicates a return of TLB access protection miss request.

Trap 5 is used by kernel to inform the simulator a new mapping from a virtual page to a physical page. URSIM must know the virtual-physical mapping of the simulated application so that it knows how to map a virtual page and a physical page of application to the same URSIM page. This trap is used only if the physical page is a real simulated physical page, not a shadow page.

Trap 6 is the reverse of trap 5. It's used when kernel deletes an existing virtual-physical mapping.

Trap 8 purges tlb entry for a specified virtual address.

Trap 18 returns the simulation time in cycles.

Trap 19 gets the L2 cache line size.

Trap 20 starts a new statistics collection phase.

Trap 21 ends a statistics collection phase.

Trap 22 clear all the statistics.

Trap 23 reports the current statistics.

Trap 25 does a disgraceful stop.

Trap 33 turns off the memory system simulation.

Trap 34 turns on the memory system simulation.

Trap 49 turns off ILP simulation.

Trap 50 turns on ILP simulation.

Whenever a return value is required, URSIM will set the %o0 register to the result of the URSIM trap. For the interface to URSIM traps, please check Section 2.3.4.

7.8 Exception handling

Source files: **Processor/except.cc, Processor/trap.cc, Processor/syscall.cc**

Header files: **Processor/instance.h, Processor/trap.h, Processor/syscall.h**

When the processor is first set into exception mode by the graduation functions, it stops decoding new instructions and instead calls the function PreExceptionHandler each cycle until the exception has been processed. This function makes sure that all stores in the memory unit prior to the excepting instruction have issued to the caches before allowing any exceptions to trap into the kernel. This step is important if a kernel trap can eventually result in context termination or paging, as the pages needed for the store to take place may no longer be present in the system after such an exception. Soft exceptions (described in Section 1.2.4) may be processed immediately, as these are resolved entirely in hardware.

After the above conditions have completed, PreExceptionHandler calls ExceptionHandler, which starts by flushing the branch queue, the memory unit, the processor stall queue, and the active list, just as in the case of a branch misprediction (Section 7.2.1). Although a real processor would also need to reverse process the register mappings in order to obtain the correct register mapping for continuing execution, the URSIM processor uses its abstraction of logical register files to reload the physical register file and restart with a clean mapping table.

ExceptionHandler then processes exceptions based on the exception type. ExceptionHandler handles soft exceptions with the bare minimum amount of processing for an exception. Specifically, the processor PC and NPC are reset, and normal instruction processing occurs as before starting with the instruction in question.

In the case of an alignment error, this handler first checks to see if the alignment used is actually acceptable according to the ISA (this can arise as double-precision and quadruple-precision floating-point loads and stores must only be aligned to single-word boundaries in the SPARC architecture). In such cases, the simulator must seek to emulate the effect of these instructions and then continue. As we expect these occurrences to be rare, URSIM currently does not simulate cache behavior for these accesses, instead calling the corresponding functions in **Processor/funcs.cc** immediately. In cases of genuine alignment failures, the exception is considered non-recoverable, and function FatalException is called. FatalException prints out a fatal error message and the exception instruction and stops the simulator.

In the case of system trap, the function SysTrapHandle is called. For the system traps supported by the kernel (sbrk), it saves certain processor state and sets PC and NPC appropriately. SysTrapHandle also handles the emulation of some system traps not supported in the kernel, which include some functions with UNIX-like semantics (exit, fork, read, write, open, close, time, stat, lseek, getpid, fstat, dup, times, ioctl, fcntl, and sysconfig), These functions are emulated by actually calling the corresponding functions in the simulator and setting the %o0 register value of the simulated processor to indicate the return value. Note that these accesses are processed by the host filesystem: as a result, simulated programs can actually overwrite system files. The time and times functions use the simulated cycle time to set the appropriate return values and structure fields. Although all of these emulated functions have the same behavior as UNIX functions on success and return the same values on failure, these functions do not set the errno variable on failure.

In the case of window trap, TLB miss, and TLB access protection miss, ExceptionHandler first saves certain processor state and sets PC and NPC to the appropriate trap handler (e.g., TLB miss handler) in the kernel.

URSIM also uses exceptions to implement certain instructions that either modify system-wide status registers (e.g., LDFSR, STFSR), or are outdated instructions with data-paths too complex for a processor with the aggressive features simulated in URSIM (e.g., MULScc), or deal with traps and must have their effects observed in a serial, non-speculative manner (e.g., SAVED and RESTORED, which are invoked just before the end of a window trap to indicate that the processor can modify its CANRESTORE and CANSAVE fields; and DONE and RETRY, which are used to return from a trap back to regular processing [16]). All of these instructions types are marked with *serializing traps*, and are handled in the function ProcessSerializedINstructions. In case of DONE and RETRY, control is transferred back to the trappc and trapnpc fields saved aside before entering the trap mode. Other serialized instructions continue with normal execution starting from the instruction after the serialized one.

For the remaining non-recoverable exceptions (division by zero, floating point error, illegal instruction, privileged instruction, and illegal program counter value), the function FatalException is called.

7.9 Principal data structures

Source files: **Processor/procstate.hh, Processor/procstate.cc, Processor/active.hh, Processor/active.cc, Processor/tagcvt.hh, Processor/stallq.hh, Processor/branchpred.hh, Processor/branchpred.cc**

Header files: **Processor/procstate.h, Processor/active.h, Processor/freelist.h, Processor/tagcvt.h, Processor/stallq.h, Processor/circq.h, Processor/branchpred.h, Processor/heap.h, Processor/hash.h, Processor/memq.h, Processor/fastnews.h, Processor/allocator.h**

The majority of the data structures used in the out-of-order execution engine are associated with the processor's ProcState data structure. ProcState represents the state of an individual processor and includes the following classes of data structures.

- The first class of data structures are concerned with instruction fetching, decoding, and graduation. These structures include the register freelist class, the activelist class, the tag converter (tag_cvt), the register mapping tables (fpmapper, intmapper, and activemaptable), the busy-bit arrays (fpregbusy and intregbusy), and the processor stall queue (stallq).
- The second class of data structures include those associated with branch prediction. These include the branchq structure, which holds the shadow mappers; the BranchDepQ, which holds branches waiting for shadow mappers (in our system, only one branch can be in this queue at a time); and the actual branch prediction tables (BranchPred and PrevPred) and the return address stack fields (ReturnAddressStack and rasptr).
- The third class of data structures deal with instruction issue, execution, and completion. Several time-based heaps are included in this class: FreeingUnits, Running, DoneHeap, and MemDoneHeap. Several MiniStallQ structures are also used in this class. These include the UnitQ structures, which include instructions waiting for functional units; and the dist_stallq (distributed register stall queue) structures, which include instructions stalling for register dependences.
- The final important class of data structures used in the out-of-order execution engine deal with simulator memory allocation and are provided to speed up memory allocation for common data structures which have an upper bound on their number of instantiations. These Allocator data structures include elements for instance structures, bqes for elements of the branch queue, mappers for shadow mappers, stallqs for elements of the processor stall queue, ministallqs for elements of the functional unit and register stall queues, actives for active list elements, and tagcvts for elements of the tag converter. Structures are dynamically allocated from and returned to these structures through the inline functions provided in **Processor/fastnews.h**.

Chapter 8

Processor Memory Unit

The processor memory unit includes nearly as much complexity as the rest of the processor, which was discussed in Chapter 7. The functions provided include adding new memory instructions to the memory unit, generating virtual addresses, translating virtual addresses to physical addresses, issuing memory instructions to the memory hierarchy, and completing memory instructions in the memory hierarchy [1]. Throughout this entire process, the memory unit must consider the ordering constraints described in Section 1.2.3: constraints for precise exceptions, constraints for uniprocessor data dependences, and constraints for multiprocessor memory consistency models.

The remainder of this section discusses the various tasks of the memory unit in the context of the above requirements. Note that the code for implementing sequential consistency (SC) or processor consistency (PC) is chosen by defining the preprocessor macro STORE_ORDERING, whereas the code for release consistency (RC) is selected by leaving that macro undefined. Impulse users should always have STORE_ORDERING defined. Variable `Processor_Consistency` (set through command line option "-6" described in Section 3.1.4) controls the selection of SC ("0") or PC ("1").

8.1 Adding new instructions to the memory unit

Source files: **Processor/memunit.hh**

Header files: **Processor/memunit.h**

The function `AddToMemorySystem` is called to add new instructions to the memory unit. This function

[1] Note that in this chapter, the terms *issue* and *complete* usually refer to issuing to the memory hierarchy and completion at the memory hierarchy. These are different from the *issue* and *completion* stages of the instruction pipeline.

first sets memory-unit-related fields of the instance: in_memunit to 1, limbo to 0, and kill to 0. It then adds this instruction to the unified memory unit queue MemQueue. If this instance does not have any outstanding address dependences (i.e., addrdep is clear, as discussed in Chapter 7), it is sent on to the address generation unit by calling function CalculateAddress.

8.2 Address generation

Source files: **Processor/memunit.hh, Processor/memunit.cc, Processor/exec.cc**

Header files: **Processor/memunit.h**

The CalculateAddress function is the first function called when an instruction in the memory unit no longer has address dependences. In this function, the vaddr and finish_addr fields of the instance are filled by using the GetAddr function. Additionally, the instance will be marked with a bus error (misalignment exception) if it is not aligned to an address boundary corresponding with its length[2]. GetAddr also marks serialization exceptions for stores of the floating-point status register (STFSR, STXFSR).

Next, function TLB_V2P is called to translate the virtual address to a physical address. Accessing TLB happens after address generation in real hardware, we put this function here just to mark the exceptions of TLB misses. However, we do simulate the TLB timing model after address generation, ensuring the accuracy of TLB simulator.

Next, the GenerateAddress function is called. If an address generation unit is free, the issue function sends this instruction to an address generation unit. Otherwise, the instruction is added to a queue of instructions stalling on an address generation unit. The instruction will be revived when a unit frees up, just as described in Chapter 7.

After the instruction has passed through the address generation unit, the Disambiguate function is called. In this function, the addr_ready field of the instance is set, indicating to the memory issue stage that this instruction may be ready to issue. No additional processing occurs for loads. However, address generation for a store may allow the processor to detect violations of the uniprocessor constraints discussed above. In particular, the processor can determine if a load that occurred later in program order than the given store was allowed to issue to the memory system and thereby obtain an incorrect value. This situation can arise based on the policy chosen with the "-L" command-line option (described in Chapter 3). Loads that have obtained values in this fashion are marked with the limbo field. If this store has an address that conflicts with any of the later limbo loads, the load is either forced to reissue (if "-L1" was used) or is marked with

[2]For some operations, the minimum alignment requirement specified in the ISA is smaller than the actual length of data transferred. However, we simulate a processor that traps and emulates instructions that are not aligned on a boundary equal to their length, as these seem more appropriate for high-performance implementation. That is, the possibility of having multiple cache line accesses and multiple page faults for a single instruction seems to be an undesirably difficult problem.

an exception (if "-L2" or the default policy was specified). On the other hand, if this store is the last prior store with an ambiguous address and does not conflict with a given load, that load is allowed to have its limbo field cleared and possibly leave the memory unit as a result. The memory unit must also check all loads that have issued to the memory hierarchy but not yet completed; if any of these loads has an address that conflicts with the newly disambiguated store, it must be forced to reissue.

8.3 Issuing memory instructions to the memory hierarchy

Source files:**Processor/memunit.hh, Processor/memunit.cc, Processor/memprocess.cc, Processor/tlb.cc, Caches/cache_cpu.c**

Header files: **Processor/memunit.h, Processor/hash.h, Processor/tlb.h, Caches/cache.h**

Every cycle, the simulator calls the IssueMem function, which seeks to allow the issue of actual loads and stores in the memory system by scanning the appropriate part of the memory unit. At a bare minimum, the instruction must have passed through address generation and there must be a cache port available for the instruction (or the TLB must have slots available if we chose to simulate fake TLB. The rest of this manual assumes real TLB miss handler is simulated). The following description focuses on the additional requirements for issuing each type of instruction.

Steps 1a-1c below refer to the various types of instructions that may be considered available for issue. **Step 2** is required for each instruction that actually issues. **Step 3** is used only with consistency implementations that include hardware-controlled non-binding prefetching from the instruction window.

Step 1a: I/O operations

An I/O load or write can issue only when it is at the head of the memory unit or there are only I/O operations ahead of it in program order in the memory queue. If an I/O operation is at the head of memory queue, no non-I/O loads or stores behind it in the memory queue are allowed to issue. I/O operations (usually writing Impulse remapping controller registers) behaves like a **membar** in a multiprocessor system: any previous memory operations must have completed; any following memory operations must wait until they have completed. Those constraints are critical for shadow descriptor reconfiguration support because we want memory accesses before reconfiguration use the old setting and memory accesses after reconfiguration use the new setting.

Step 1b: Stores

If the instruction under consideration is a store, it must be the oldest instruction in the memory unit and must have been marked ready in the graduate stage (as described in Section 7.6) before it can issue to the TLB and cache. If the processor supports hardware prefetching from the instruction window, then the system can mark a store for a possible hardware prefetch even if it is not ready to issue as a demand access to the caches.

Step 1c: Loads

A load instruction in sequential consistency can only issue non-speculatively if it is at the head of the memory unit. If hardware prefetching is enabled, later marked for possible prefetching. If speculative load execution is present, later loads can be issued to the caches. Before issuing such a load, however, the memory unit is checked for any previous stores with an overlapping address. If a store exactly matches the addresses needed by the load, the load value can be forwarded directly from the store. However, if a store address only partially overlaps with the load address, the load will be stalled in order to guarantee that it reads a correct value when it issues to the caches.

Loads issue in processor consistency (variable Processor_Consistency was set) under circumstances similar to those of sequential consistency. However, a load can issue non-speculatively whenever it is preceded only by store operations. A load that is preceded by store operations must check previous stores for possible forwarding or stalling before it is allowed to issue.

Step 2: Issuing to the memory hierarchy

For both stores and loads, the IssueOp function actually initiates an access. First, the memprogress field is set to -1 to indicate that this instance is being issued. (In the case of forwards, the memprogress field would have been set to a negative value). This function then consumes a cache port for the access (cache ports are denoted as functional units of type uMEM). The memory_rep function is then called. This function prepares the cache port to free again in the next cycle if this access is not going to be sent to the cache (i.e., if the processor has issued a MEMSYS_OFF directive). Otherwise, the cache is responsible for freeing the cache port explicitly.

Next, the memory_latency function is called. This function starts by calling GetMap, which checks the processor page table PPageTable to determine if this access is a segmentation fault (alignment errors would have already been detected by GetAddr). If the access has a segmentation fault or bus error, its cache port is freed up and the access is considered completed, as the access will not be sent to cache.

If the access does not have any of the previous exceptions, it will now be issued. Prefetch instructions are considered complete and removed from the memory unit as soon as they are issued. If the access is an ordinary load or store and is not simulated (i.e., if the processor has turned MEMSYS_OFF), it is set to complete in a single cycle. If the access is simulated, it is sent to the memory hierarchy by calling StartUpMemRef, which passes the access to the L1 cache through the TLB_lookup.

Function TLB_lookup simulates the TLB access and calls Cache_recv_cpureq to begin the simulation of an memory access. If the access is missed in the TLB, the instance will be associated with a TLB_MISS or TLB_ACCESS_PROTECTION exception which will eventually lead to a trap into TLB miss handler in the kernel.

Cache_recv_cpureq and the other functions in **Processor/memprocess.cc and Caches/cache_cpu.c** are responsible for interfacing between the processor memory unit and the memory hierarchy itself. It starts by initializing a memory system request data structure REQ for this memory access. (This data structure type is described in Section 9.1.) Next, the request is inserted into its cache port. If this request fills up the cache ports, then the L1Q_FULL field is set to inform the processor not to issue further requests (this is later cleared by the cache when it processes a request from its ports). After this point, the memory system simulator is responsible for processing this access.

Step 3: Issuing any possible prefetches

After the functions that issue instructions have completed, the memory unit checks to see if any of the possible hardware prefetch opportunities marked in this cycle can be utilized. If there are cache ports available, prefetches are issued for those instructions using IssuePrefetch. These prefetches are sent to the appropriate level of the cache hierarchy, according to the command line option used.

8.4 Completing memory instructions in the memory hierarchy

Source files: **Processor/memprocess.cc, Processor/memunit.hh, Processor/memunit.cc, Processor/funcs.cc**

Header files: **Processor/memunit.h**

GlobalPerform and MemDoneHeapInsert

Completion of memory references takes place in two parts. First, the GlobalPerform function is called at the level of the memory hierarchy which responds to the reference. This function calls the function associated with this instruction (as specified in **Processor/funcs.cc**) to actually read a value from or write a value into the UNIX address space of the simulator environment. In the case of virtual store-buffer forwards, the value taken by the load is the value forwarded from the buffer rather than that in the address space. In the case of accesses which are not simulated, this behavior takes place as part of the CompleteMemOp function (described below).

Then, when a reference is ready to return from the caches, the MemDoneHeapInsert function is called to mark the instruction for completion. In the case of non-simulated accesses, the access is put into the MemDoneHeap by the memory_latency function invoked at the time of issue.

CompleteMemQueue and CompleteMemOp

The function CompleteMemQueue processes instructions from the MemDoneHeap of the processor by calling function CompleteMemOp for each instruction to complete in a given cycle. For loads, this function first checks whether or not a soft exception has been marked on the load for either address disambiguation or consistency constraints while it was outstanding. If this has occurred, this load must be forced to reissue, but does not actually need to take an exception. Otherwise, this function checks to see whether the limbo field for the load must be set (that is, if any previous stores still have not generated their addresses), or whether the load must be redone (if a previous store disambiguated to an address that overlaps with the load). If the load does not need to be redone and either does not have a limbo set or has a processor in which values can be passed down from limbo loads (as discussed above), the function PerformMemOp is called to note that the value produced by this instruction is ready for use. The function PerformMemOp is called for all stores to reach CompleteMemOp.

PerformMemOp

PerformMemOp has two functions: removing instructions from the memory unit and passing values down from limbo loads. In the case of SC, memory operations must leave the memory unit strictly in order. The constraints for PC are identical to those for SC, except that loads may leave the memory unit past outstanding stores. In no memory model, may limbo loads leave the memory unit before all previous stores have disambiguated. If the memory unit policy allows values to be passed down from limbo loads, PerformMemOp fulfills some of the duties otherwise associated with the update_cycle function (filling in physical register values and clearing the busy bit and distributed stall queues for the destination register). Note that PerformMemOp will be called again for the same instruction when the limbo flag is cleared.

SpecLoadBufCohe

If the system supports speculative load execution to improve the performance of its consistency model (with the ”-K” option), the constraints enforced by PerformMemOp will be sufficient to guarantee that no speculative load leaves the memory unit. Each coherence message received at the L1 cache because of an external invalidation or a replacement from L2 cache must be sent to the processor memory unit through the SpecLoadBufCohe function. If such a message invalidates or updates a cache line accessed by any outstanding or completed speculative load access, that access is marked with a soft exception. If the access is still outstanding, the soft exception will be ignored and the load will be forced to reissue; if the access has completed, the exception must be taken in order to guarantee that the load or any later operations do not commit incorrect values into the architectural state of the processor [6, 8].

Chapter 9

Cache Hierarchy

URSIM simulates a two-level cache hierarchy. The first-level of cache can be either write-through with no-write-allocate or write-back with write-allocate. The second-level cache is write-back with write-allocate and maintains inclusion of the first-level cache. Each cache supports multiple outstanding misses and is pipelined. The first-level cache may also be multi-ported. If the configuration uses a write-through L1 cache, a write-buffer is also included between the two levels of cache. The L1 cache tag and data access is modeled as a single access to a unified SRAM array, while an L2 cache access is modeled as an SRAM tag array access followed by an SRAM data array access. These arrays themselves are modeled as pipelines, processed by the functions in **Caches/pipeline.c**.

Both levels of cache support cache-initiated prefetch. Currently, only sequential prefetching algorithm is supported. It contains two rules: when a miss occurs, fetch the missed line and prefetch the next line; when a prefetched line is hit, prefetch the next line. The prefetch in each level is controlled independently; user can use any combination that he/she prefers. Also, both caches can be configured to be perfect, which is usually used to test the best performance that each cache can possibly achieve. The user also can completely disable the L2 cache by setting the size of L2 cache to 0, making one-level cache hierarchy.

Like the processor, the cache hierarchy is activated by RSIM_EVENT function, which is scheduled to occur every cycle. RSIM_EVENT calls the functions L1CacheInSim, L1CacheWBufferSim, L2CacheInSim, L1CacheOutSim, and L2CacheOutSim for each cache, as mentioned in Section 7.1. Each of these functions, as well as the functions called by those functions, are described in this chapter.

9.1 Data structure of basic cache message

Header files: **Cache/req.h**

This section describes the essentials of the message data structure REQ used to convey information within the cache hierarchy. This data structure conveys essential information about the cache access being simulated, just as the instance structure acts as the basic unit of information exchange among the processor simulator. The two most important fields of the message data structure are the type field and the req_type field. The following sections describe how each of the two fields is used to distinguish the types of messages in the memory system simulator.

9.1.1 The type field

Memory system messages come in five basic varieties, as conveyed by the type field:

REQUEST [1] Sent by a processor or cache to request some action related to the data requirements of the processor; may demand a data transfer.

REPLY Sent by a cache or memory in response to the demands of a REQUEST; may include a data transfer.

COHE Sent by a cache to other caches with a demand to invalidate or change the state of a line; may demand a data transfer.

COHE_REPLY Sent by a cache in response to the demands of a COHE, or a replacement message; may include a data transfer.

WRITEBACK Sent by a cache to write some dirty data back to memory due to flushes or victims of cache line replacement.

9.1.2 The req_type field

The req_type field can take on several values. The req_types that are supported in URSIM are split into the following categories:

1. Those seen only at the processors and caches:

   READ Reads data from cache.

   WRITE Writes data to cache.

   RMW (read-modify-write) Reads a value from cache, modifies it, and writes it back to cache.

   FLUSHC Flushes the matched cache lines in both L1 and L2 cache.

   PURGEC Purges the matched cache lines in both L1 and L2 cache.

[1] Do not confuse REQUEST with the REQ described earlier, which is data structure.

WBACK Writes data from L1 cache to L2 cache. This is used only when L1 cache is write-back.

RWRITE Writes to a remote node. Reserved for future expansion.

2. System-visible transaction requests:

   READ_SH Reads a cache line without demanding ownership. Issued for read misses and read prefetches.

   READ_OWN Reads a cache line and demands ownership. Issued for write misses, read-modify-write misses, and exclusive (write) prefetches.

   UPGRADE Demands ownership for a cache line (without reading the line). Issued for writes, read-modify-writes, or exclusive prefetches that are to lines that hit in the cache, but are held in shared state.

3. Replies

   REPLY_SH Brings a line to cache in shared state. Valid response to **READ_SH**.

   REPLY_EXCL Brings a line to cache in exclusive state. Valid response to all requests: READ_SH, READ_OWN, or UPGRADE.

   REPLY_UPGRADE Acknowledges ownership of a cache line. Valid response to **UPGRADE**.

4. Coherence actions:

   INVALIDATE Changes a line from exclusive or modified state to invalid state.

   SHARED Changes a line from exclusive to shared state.

9.2 Bringing in messages

Source files: **Caches/l1cache.c, Caches/l2cache.c, Caches/pipeline.c**

Header files: **Caches/cache.h, Caches/pipeline.h**

Two functions are used to bring new messages into each level of cache — L1CacheInSim and L2CacheInSim. Each of these functions checks incoming messages from the ports of the module, and then attempts to insert each incoming message into the appropriate tag-array pipeline, according to its type field. If the message can be added to its pipeline, it is removed from its input port; otherwise, it remains on the input port for processing in a future cycle.

9.3 Processing the cache pipelines

Source files: **Caches/l1cache.c, Caches/l2cache.c, Caches/pipeline.c, Caches/cache_help.c**

Header files: **Caches/cache.h, Caches/pipeline.h**

For each cycle in which there are accesses in the cache pipelines, the functions L1CacheOutSim, L1CacheWBufferSim, and L2CacheOutSim are called. These functions start out by considering the current state of their pipelines. If a message has reached the head of its pipeline (in other words, has experienced all its expected latency), the cache calls one of the functions to process messages according to the type field of the message: L1ProcessTagRequest, L1ProcessTagReply, L1ProcessTagCohe, L2ProcessTagRequest, L2ProcessTagReply, L2ProcessTagCohe, L2ProcessTagCoheReply, or L2ProcessDataReq. If the corresponding function returns successfully, the element is removed from the pipeline. The following sections describe these functions.

9.4 Processing L1 cache accesses

Source files: **Caches/l1cache.c, Caches/cache.c, Caches/cache_help.c**

Header files: **Caches/cache.h**

9.4.1 Handling REQUEST type

Function L1ProcessTagRequest processes REQUEST from processor. If the REQUEST is a flush or purge operation and L1 cache is write-through, this function sends the REQUEST to L2 cache directly without doing anything. If the REQUEST hits in L2 cache, L2 cache will send an invalidation message back to L1 cache. If it misses in L2 cache, nothing will happen thus avoiding searching L1 cache. Since L2 cache maintains inclusion of L1 cache, this way guarantee the L1 cache will be appropriately flushed/purged while avoiding unnecessary flush/purge operations on L1 cache. If the REQUEST is a flush or purge and L1 cache is write-back, function L1ProcessFlushPurge is called to perform required actions. Note that each cache flush or purge operation affects the matched L2 cache line, which may contain multiple L1 cache lines. L1ProcessFlushPurge must consider flushing/purging a region as big as the size of an L2 cache line instead of an L1 cache line.

For other types of REQUEST, L1ProcessTagRequest performs the following actions.

Step 1: Calling L1Cache_check_mshr

L1Cache_check_mshr first checks to see if the REQUEST has a tag that matches any of the outstanding MSHRs. Next, it determines if the desired line is available in the cache. For L1-initiated prefetch, this function returns MSHR_USELESS_PFETCH if the REQUEST matches either an MSHR or a cache line.

For the other REQUEST, the operation depends on the req_type of the REQUEST, the state of matching cache line, and the previous accesses to the matching MSHR.

The REQUEST being processed does not match an outstanding MSHR.

- If the line being accessed hits a cache line with an acceptable state, this request will not require a request to a lower level. As a result, the cache will return NOMSHR, indicating that no MSHR is involved or needed in this request.
- If the request goes to the next level of cache without taking an MSHR at this cache level (either by being a write in a write-through cache or an L2 prefetch), the value NOMSHR_FWD is returned to indicate that no MSHR is required, but that the request must be sent forward.
- If the request needs a new MSHR, but none are available, the value NOMSHR_STALL is returned.
- Otherwise, the cache books an MSHR and returns a response based on whether this access is a complete miss (MSHR_NEW) or an upgrade request (MSHR_FWD).
- In all cases where the line is present in cache, the Cache_hit_update function is called to update the ages of the lines in the set (for LRU replacement).

The REQUEST being processed matches an outstanding MSHR.

- If the access is a shared or an exclusive processor-generated prefetch[2] matching an MSHR returning in exclusive state, the prefetch is not necessary because a fetch is already in progress. In this case, this function returns MSHR_USELESS_FETCH to indicate that the request should be dropped.
- If the access is an L2 or an exclusive processor-generated prefetch in the case of a write-through L1 cache, the request should be forwarded around the cache. In this case, the function returns NOMSHR_FWD. If the request is an exclusive prefetch, it is converted to an L2 exclusive prefetch before being forwarded to the L2 cache.
- In certain cases, the request may need to be stalled. Possible scenarios that can result in stalls and the values that they return are as follows. If the MSHR is marked with an unacceptable pending flush message, the function returns MSHR_STALL_FLUSH. If the MSHR already has the maximum number of coalesced requests for an MSHR, the return value is MSHR_STALL_COAL. The maximum number of coalesced accesses is a configurable parameter. Finally, when a write (or exclusive prefetch) request comes to the same line as an MSHR held for a read (or shared prefetch) request, the value MSHR_STALL_WAR is returned. This last case can significantly affect the performance of hardware store-prefetching, and is called a WAR stall. The impact of WAR stalls can be reduced through software prefetching, as an exclusive prefetch can be sent before either the read or write accesses [10, 11].

[2] Processor-generated prefetch results from speculative loads/stores of processor. We will use the shorter-term speculation with this term interchangeably in the rest of this manual.

- If the access is not dropped, forwarded, or stalled, it is a valid access that can be processed by merging with the current MSHR. In this circumstance, the cache merges the request with the current MSHR and returns MSHR_COAL.

Step 2: Processing based on the results of L1Cache_check_mshr

The processing of REQUEST continues based on the return value of L1Cache_check_mshr. For a hit (NOMSHR), Cache_global_perform function is called to collect statistics and to inform the processor memory unit using functions described in Section 8.4.

For new misses (MSHR_NEW), upgrades (MSHR_FWD), or write-through (NOMSHR_FWD), the cache attempts to send the request down, returning a successful value if the request is sent successfully.

For MSHR_COAL, the element is considered to have been processed successfully.

On MSHR_USELESS_FETCH, the request is dropped.

For each of the other cases, the processing is generally considered incomplete, and the function returns 0 to indicate this. However, if the stalled request is a prefetch and DISCRIMINATE_PREFETCH has been set with the ”-T” option, the request is dropped and processing is considered successful. Note that DISCRIMINATE_PREFETCH cannot be used to drop prefetches at the L2 cache, as these prefetches may already hold MSHRs with other coalesced requests at the L1 cache.

9.4.2 Handling REPLY type

For REPLY type, function L1ProcessTagReply is called to perform as follows.

Step 1a: Process upgrade replies

For upgrade reply, it changes the final state of the cache line to PR_DY[3].

Step 1b: Process cache miss replies

For cache miss REPLYs, the handler calls either Cache_pmiss_update or Cache_miss_update based on whether or not the line is a ”present miss” (a line whose tag remains in cache after a COHE, but in an INVALID state). If the line is not a ”present miss”, the Cache_miss_update function tries to find a possible

[3] Each cache line can be one of the following states: INVALID — invalid; SH_CL — shared clean; PR_CL — private clean; PR_DY — private dirty.

replacement candidate. If any set entry is INVALID, this line is used so as to avoid replacement. If a line must be replaced, then the least-recently used SH_CL line is used; if none is available, the least-recently used PR_CL or, finally, PR_DY line is used.

After having found a space in which to insert the new line, the REPLY handler must determine the state for the new line being brought in. If this line replaces a current line with modified state and L1 cache is write-back, a write-back must be sent as a result of this replacement. In this case, the Cache_make_req function must be called to create a write-back message. This function sets up all the fields for a new message that writes back the line being replaced. This new write-back message is sent out in next step.

The cache simulator also classifies each cache miss into capacity miss, conflict miss, and coherence miss using a data class called CapConfDetector (described in detail in Section 9.8).

Step 2: Returning replies to processor

Regardless of REPLY type processed, the system now prepares to remove the corresponding entry from the MSHRs. Function L1Cache_uncoalesce_mshr is called to release all accesses coalesced into the MSHR. It also will calls statistics collection functions and appropriate functions described in Section 8.4 to inform the processor memory unit that the certain memory access has completed.

If this reply does not cause a write-back, its MSHR is freed. Otherwise, the MSHR is temporarily used as storage space for the write-back, and will be thus held until the write-back is able to issue from the MSHR to the next level of cache.

9.4.3 Handling COHE type

The req_types of incoming COHE transactions understood by the L1 cache are INVALIDATE and SHARED sent by L2 cache. Since L2 cache maintains inclusion of L1 cache, a COHE access is sent to L1 cache only if it hits in L2 cache. L2 cache uses INVALIDATE message to inform L1 cache to invalidate the L1 cache lines matched with a specified L2 cache line, and uses SHARED message to inform L1 cache to change the states of matching L1 cache lines to SH_CL. Theoretically, each COHE transaction needs access L1 cache (*L2_cache_line_size / L1_cache_line_size*) times. However, in the current version, we assume it can be done in one L1 tag access.

By the end of the COHE handler, it sends a COHE_REPLY message back to L2 cache.

9.5 Processing L2 tag array accesses

Source files: **Caches/l2cache.c, Caches/cache.c, Caches/cache_help.c**

Header files: **Caches/cache.h**

L2ProcessTagRequest, L2ProcessTagReply, L2ProcessTagCohe, and L2ProcessTagCoheReply are called for accesses that have reached the head of an L2 tag array pipeline. The first three functions are largely similar to the counterparts of L1 cache. But they have some key differences, described below.

Difference 1: Presence of a data array

The data array is only presented in L2 cache. In L2 cache, REQUESTs that hit L2 cache, REPLYs, write-backs of replaced lines, as well as the copy-backs of COHE messages all require data array accesses.

Difference 2: Servicing COHE messages

For COHE messages, the L2 cache marks the line in question with a "pending-cohe" bit and then forwards possible actions to the L1 cache first. The actual actions for the message are processed at the time of receiving the COHE_REPLY from L1 cache; The "pending-cohe" bit is also cleared upon receiving the COHE_REPLY.

Additionally, the L2 cache is responsible for resolving cache-to-cache transfer requests. On a successful cache-to-cache transfer, the L2 cache not only sends a copy-back to the memory, but also sends a REPLY to the requesting processor with the desired data.

Difference 3: Conditions for stalling REQUEST messages

The L2Cache_check_mshr behaves slightly different than L1Cache_check_mshr.

First, L2Cache_check_mshr can not return NOMSHR_FWD and MSHR_FWD because there is no next level cache to forward. Second, if the matching MSHR is being released by the owner, the REQUEST must stall until the release has done. Because the MSHR may have coalesced write REQUESTs, flush/purge REQUESTs, or COHE transactions, releasing MSRH may involve sending INVALIDATE or SHARED messages to L1 cache, which can not guarantee to be done in one-cycle. The REQUEST must wait all of the coalesced REQUESTs has been released.

Difference 4: Replies that replace a valid line

If the REPLY replaces a line, L2 cache sends an enforcement INVALIDATE COHE message to the L1 cache. If the line is in modified state and L1 cache is write-back, L2 cache has to wait its correspondent COHE_REPLY from L1 cache.

Difference 5: Interface with system bus

L2 cache simulator contains the system interface communicating with the system cluster bus. The implementation of the system interface is straightforward. Before processing an access, L2 cache simulator must check the system interface to make sure that resources that the processing of this access may generate are available.

9.6 Processing L2 data array accesses

Source files: **Caches/l2cache.c, Caches/cache_help.c**

Header files: **Caches/cache.h**

L2ProcessDataReq is a short function that handles the L2 cache data-array access of each type.

REQUEST type

The REQUEST is either a write-back from L1 cache, or a L2 cache hit. For the former case, this function does nothing. For the latter case, this function just changes the REQUEST to a REPLY and tries to send it to L1 cache. If the REPLY can not be sent out, it will be awaken in REPLY case in the future.

REPLY type

If the REPLY is a stalled L2 hit (described above), this function tries to send it to L1 cache. Otherwise, it must be a REPLY from the memory. This function first returns the REPLY to L1 cache, then releases coalesced requests in the relevant MSHR, then handles pending flush/purge operations if they exist. Finally, it frees the MSHR.

COHE type

This must be a cache-to-cache copy initiated a COHE message matching a L2 cache line. This function sends a cache-to-cache copy message to the system interface if no previous cache-to-cache copy is in the system interface. Otherwise, it stalls and tries again at the next cycle.

WRITEBACK type

This must be a write-back transaction to the memory. L2ProcessTagReq sends this transaction to the outgoing buffer in system interface. If the outgoing buffer is full, this transaction must wait and try again at the next cycle.

9.7 Coalescing write buffer

Source files: **Caches/cache_wb.c**

Header files: **Caches/cache.h**

The coalescing write buffer is used in systems with a write-through L1 cache. Although the write buffer is conceptually in parallel with the L1 cache, the simulated module sits between the two caches. To provide the semblance of parallel access, the write buffer has zero delay.

The L1CacheWBufferSim function implements the write buffer and is called from RSIM_EVENT. It first checks for a message on its input queues. If one is available, the function jumps to the appropriate case to handle it.

If the incoming message is a REQUEST, it is handled according to the req_type field of the message. If the request is a read type that does not match any write in the buffer, it is immediately sent on to the L2 cache. If the request is a read that does match a write in the buffer, the read is stalled until the matching write issues from the write buffer. This scheme follows the policy used in the Alpha 21164, referred to as "Flush-Partial" in other studies [14][4].

If the incoming REQUEST is a write, the write-buffer attempts to add it to the queue of outstanding write buffer entries. If the request matches the line of another outstanding write, it is coalesced with the previous write access. Each line conceptually includes a bit-vector to account for such coalescing. If there is no space for the write in the buffer, it is stalled until space becomes available. Writes are sent out of the write buffer and to the L2 cache as soon as space is available in the L2 input ports. As soon as a write is added to an L2 port, its entry is freed from the write-buffer.

9.8 Cache initialization and statistics

Source files: **Caches/cache_init.c, Caches/cache.c, Cache/cache_stat.c, Processor/capconf.cc**

Header files: **Caches/cache.h, Caches/cache_param.h, Caches/cache_stat.h, Processor/capconf.h**

[4]Note that our architecture does permit "forwarding" of values in the processor memory unit.

The functions Cache_init first calls Cache_read_params to read in cache-related parameters from configuration file. Next, it allocates memory for the general cache data structures. Then, it calls L1Cache_init, L2Cache_init, and L2Cache_wbuffer_init to initialize their respective data structures, including the cache-line state structures, the MSHR array, the write-back buffer, the cache pipelines, and the statistics structures.

When a data access has completed, the simulator calls Cache_stat_set to record whether the access hit or missed, and the type of miss in the case of misses. Each cache module classifies misses into conflict, capacity, and coherence misses. Capacity and conflict misses are distinguished using a structure called the CapConfDetector. The detector consists of a hash table combined with a fixed-size circular queue, both of which start empty.

Conceptually, new lines brought into the cache are put into the circular queue, which has a size equal to the number of cache lines. When the circular queue has filled up, a new insertion replace the oldest element in the queue. However, before inserting a new line into the detector, the detector must first be checked to make sure that the line is not already in the queue. If it is, then a line has been brought back into the cache after being replaced in less time than a full-associative cache would evict it; consequently, it is a conflict miss. We consider a miss a capacity miss if it is not already in the queue when it is brought into the cache, as this indicates that at least as many lines as in the cache have been brought into the cache since the last time this line was present. The hash table is used to provide a fast check of the entries available; the entries in the hash table are always the same as those in the circular queue.

Chapter 10

System Cluster Bus

10.1 Overview of the cluster bus

The system bus of URSIM was simulated based upon the MIPS R10000 cluster bus[1]. The cluster bus connects processors and cluster coordinator (i.e., main memory controller) together. It is time-multiplexed, split-transaction bus supporting a simple pipelined arbitration scheme and a snoopy cache coherence protocol.

Each bus module[2] is either in *master* or *slave* state. In *master* state, it drives the system bus and is permitted to issue requests. In *slave* state, it accepts external requests from other bus modules. Only one bus module is in *master* state in a give time. The simulated system supports a simple arbitration protocol, which relies on the cluster coordinator to select arbitration winner. The arbitration protocol gives the cluster coordinator higher priority on sending data responses back to the processors, and implements a round-robin priority scheme for the processors. The protocol also supports overlapped arbitration which allows arbitration to occur in parallel with requests and responses.

Each cluster bus uses a unique request number to identify each processor request. The system allows a maximum of eight outstanding requests for the whole system through a 3-bit request number. An individual processor supports a maximum of four outstanding processor requests at any given time.

[1] *cluster bus* and *system bus* are used interchangeably in this manual.

[2] In the manual, a bus module refers to either a processor or a cluster coordinator.

10.2 Implementation

Source files: **Bus/bus.c, Cache/cache_bus.c, MMC/mmc_bus.c**

Header files: **Bus/bus.h**

In the simulator, the bus is responsible for sending transactions from one bus module to other bus modules, managing the free request numbers, and picking up arbitration winner. Though the arbitration is handled by the cluster coordinator in real hardware, we move the arbitration handlers into the bus simulator just for better understandability and modularity. This move neither changes the architecture of simulated system nor loses any accuracy because it is essentially a matter of moving some functions from one source file to another.

In order to access the bus, different operation (e.g., request and response) has to meet different conditions. For example, some operations need a request number; most operations need the issuer in master state. If a request needs a request number, it has to call Find_free_reqnum to reserve a free request number. If no free request number is available, the request enters request number waiting list.

If an operation can be issued only in master state and the issuer is not in the master state, the issuer must start arbitrating for the bus via function Bus_recv_arb_req. Note that only one outstanding arbitration request is allowed for each bus module at any given time. Each processor or the cluster coordinator must ensure not to issue a second arbitration request before the first one has been granted.

Each cluster bus has two events — one controls the arbitration and another one controls the transfer. Whenever there exist outstanding arbitration requests, the arbitration event will be schedule to call Bus_arbitrator at an appropriate future time. Bus_arbitrator picks up an arbitration winner according to the rule of "coordinator first and round-robin policy among processors". Then, it schedules the arbitration event to activate function Bus_wakeup_winner at the first cycle that the new winner starts driving the bus. Bus_wakeup_winner calls function MMC_in_master_state (if the winner is an MMC) or Cache_in_master_state (if the winner is a processor), and reschedules the arbitration event if there exist outstanding arbitration requests. Each of MMC_in_master_state and Cache_in_master_state starts by sending a transaction on the bus. Then, it updates the architectural state accordingly. At the end, it schedules the transfer event to call its body function Bus_issuer at the cycle that the ongoing transaction is coming off the cluster bus. Bus_issuer calls the function associated with the type of the transaction to copy the essential information of the transaction from its source bus module to the destination bus modules.

10.3 Initialization and statistics

Both initialization and statistics are simple and straightforward. They are not worth describing. Interested readers should easily understand short functions Bus_init and Bus_stat_report_all.

Chapter 11

Main Memory Controller

The MMC (main memory controller, also called "bus coordinator") is the most important piece of the Impulse adaptive memory system. The MMC coordinates the system bus (thus the name "bus coordinator"), maintains cache coherence protocol[1], translating shadow addresses to DRAM addresses, issuing accesses to the DRAM backend, and performing MC-based prefetching. This chapter describes the implementation of each component of the MMC in detail. Well understanding of Impulse technology [3, 22] would be a plus to understand the MMC simulator.

11.1 Lifetime of a memory transaction

Source files: **MMC/mmc_remap.c**

Header files: **MMC/mmc_ams.h**

This section describes the timing model that each memory access need go through. Each MMC is associated with one YACSIM control event, which is scheduled only when necessary (different with cycle-by-cycle RSIM_EVENT used by processor and cache simulator). The fundamental unit of information exchange among the MMC is data structure mmc_trans_t, which conveys essential information about each memory request: physical address, type, time being issued, size, etc.

Step 1: Receive memory access requests

A memory access comes either from the processor through the system bus or from the MTLB. Function

[1] Note that the coherence issues are omitted in this version.

MMC_recv_trans sends a processor memory request to the MMC; while functions MMC_post_fastread and MMC_post_fastwrite (described in Section 11.3) send in MTLB-initiated memory requests.

All of these three functions first get a mmc_trans_t and assign the essential information to this data structure, then call MMC_trans_enqueue. MMC_trans_enqueue does three things:

- Searching the MCache if the MC-based prefetching is enabled; (If a line being fetched is matched, add the request to MCache waiting queue and return.)
- Adding the request into a relevant waiting queue — processor requests go to normal waiting queue (waitlist) and MTLB requests go to the high priority queue (hipriq); (Since each MTLB request is likely depended on by some shadow accesses, the memory controller gives MTLB requests higher priority than normal memory requests from the processors.)
- Scheduling the control event to be active at a right future (must consider the time spent on search MCache).

Step 2: Start processing transactions

The control event calls its body function MMC_process_wait when active. This function picks up the transaction at the head of hipriq, or at the head of waitlist if hipriq is empty and call MMC_process_trans to process the transaction. After the transaction has been processed, it checks the hipriq and waitlist. If either of them is not empty, the control event is rescheduled.

Step 3: Processing transactions

Transaction processing is done by function MMC_process_trans. It first checks if the transaction hits in the MCache.

- If the transaction hits in the MCache and is a read, the requested data is returned without going through DRAM access. To "retire" this transaction from the memory controller, MMC_process_trans collects the necessary statistics and calls MMC_data_fetch_done (if it came from the processor) to send data back to the processor or MTLB_trans_returned (if it came from the MTLB) to send data back to the MTLB. Finally, the data structure mmc_trans_t is released.
- If the transaction hits in the MCache and is a write the matching MCache line is either invalidated (if the MCache is write-invalidate[2]) or overwritten (if the MCache is write-update). In either case, the cache line's waiting queue must be disbanded. The function then continues as the transaction misses the MCache.

[2]Write-update means dirty data is written to both the MCache and the DRAM; Write-invalidate means dirty data invalidates matching line of the MCache and is written to DRAM only.

- Otherwise, it must be an MCache miss. If it is an access to shadow address, MMC_start_translation (explained in Section 11.2) is called to translate the shadow address to DRAM addresses. Otherwise, a DRAM access is sent to the DRAM backend through function DRAM_recv_request.

Step 4: Address translation and DRAM access

Please see Section 11.2 and 11.3 for address translation and Chapter 12 for DRAM access.

Step 5: Processing returns from DRAM backend

The DRAM backend uses MMC_dram_done to inform the MMC simulator that a DRAM access has completed. If the DRAM access is for a shadow address, it decreases 1 from the count readcount that records how many DRAM accesses are still needed to gather a cache line for the shadow address. If the readcount equals 0 or the DRAM access is for a non-shadow address, this functions behaves as an MCache hit occurs (see item 1 of **step 3** above).

11.2 Remapping controller

Source files: **MMC/mmc_remap.c**

Header files: **MMC/mmc_ams.h**

The remapping controller contains a configurable number of shadow descriptors. Each shadow descriptor has its own event to control the address translation from shadow to pseudo-virtual. Before the translation starts, function MMC_start_translation must be called. It sets shadowcount field of the transaction which indicates how many pseudo-virtual addresses that the specified shadow address will generate. Then the event of the matching descriptor is scheduled to be active at the next cycle. The event's body function MMC_process_remap calls the relevant translation functions according to the mapping type. It generates one pseudo-virtual address on each invocation and decreases shadowcount by 1. If shadowcount is still larger than 0, the event will be rescheduled to the next cycle. Whenever a pseudo-virtual address is generated, the remapping controller calls MMC_got_pvaddr function, which simply calls MTLB_lookup described in Section 11.3.

Currently, the simulator can handle the following types of remapping.

- **Direct mapping:** MMC_superpage_remap() and MMC_pagecolor_remap()

  Direct mapping maps one contiguous cache line in the shadow address space to one contiguous cache line in real physical memory. The relationship between a shadow address *saddr* and its pseudo-virtual address *pvaddr* is ($pvaddr = pvsaddr + (saddr - ssaddr)$), where *pvsaddr/ssaddr* is the starting address (assigned by the OS) of the data structure's *pseudo-virtual address space image/shadow address space image*. Uses of this mapping include recoloring physical pages without copying [3] (handled by function MMC_pagecolor_remap), and constructing superpages from non-contiguous physical pages without copying [18] (handled by function MMC_superpage_remap).

- **Strided mapping:** MMC_basestride_remap()

  Strided mapping creates dense cache lines from data items whose virtual addresses are distributed in uniform stride. Depending on the size of each data item, the mapping function maps the cache line addressed by a shadow address *saddr* to multiple pseudo-virtual addresses: ($pvsaddr + stride \times (saddr - ssaddr) + stride \times i$), where i ranges from 0 to (*(size of cache line) / (size of data item)* -1). Each of the multiple pseudo-virtual addresses is mapped to exactly one real physical address.

- **Scatter/Gather mapping using an indirection vector:** MMC_indirvector_remap()

  Scatter/gather mapping using an indirection vector packs dense cache lines from array elements according to an indirection vector. The mapping function first computes the offset of a shadow address *saddr* in shadow address space $soffset = saddr - ssaddr$, then uses the indirection vector *vector* to map the cache line addressed by the shadow address *saddr* to several pseudo-virtual addresses: ($pvsaddr + vector[soffset + i]$), where i ranges from 0 to (*(size of cache line) / (size of array element)* -1). When the OS sets up this type of remapping, it moves the indirection vector into a contiguous physical memory region so that the address translation for the indirection vector is not needed.

- **Transpose mapping:** MMC_transpose_remap()

  Transpose mapping creates the transpose of a two-dimensional matrix by mapping the element $transposed\_matrix[j][i]$ of the transposed matrix to the element $original\_matrix[i][j]$ of the original matrix. This mapping can be used wherever a matrix is accessed along an axis different from how it is stored. This mapping also can be easily expanded to support higher-dimension matrices. For example, taking each vector as an array element allows the same mapping algorithm to be applied to three-dimensional matrices.

Obviously, in *direct mapping*, each shadow address generates exactly one DRAM access; in other three mappings, each shadow address generates (*cache-line-size / sizeof(data item)*) DRAM accesses if (*cache-line-size* $>$ *sizeof(data item)*), or one DRAM access if (*cache-line-size* $\leq$ *sizeof(data item)*). Note that the translation procedure for *scatter/gather through indirection vector* is a little bit more complicated since this mapping also needs access the indirection vector. Please see the comments in the source file for details.

11.3 Memory controller TLB

Source files: **MMC/mmc_tlb.c**

Header files: **MMC/mmc_ams.h**

When the remapping controller generates a pseudo-virtual address, it passes the pseudo-virtual address into the MTLB simulator for its relevant physical address. Generally, the MTLB processes accesses as follows.

1. If the access hits in the MTLB, the MTLB generates the real physical address in one cycle. Otherwise, it looks up the MTLB buffer, which holds previously fetched page table entries and contains a configurable number of cache-line-sized entries.
2. If it hits an entry in the buffer with valid state, the MTLB takes an extra cycle to load the relevant translation and then performs as the access hits in the MTLB.
3. If it hits a buffer entry being fetched, it will enter the buffer waiting queue.
4. If the access misses in the buffer, it must reserves an entry in the buffer, issues a load request to memory controller, and waiting for the reply (so-called miss reply in this section).

Function MTLB_lookup is the interface to pass in pseudo-virtual addresses from outside. It first checks if the MTLB can process this access. If any of the following conditions is true, it adds the access into the waiting queue:

- If the MTLB is busy, (The MTLB can not handle two accesses in parallel.)
- If there is a pending miss waiting for resource, (The MTLB can handle only one outstanding miss. The access can not be issued before it might generate another miss.)
- If the waiting queue is not empty. (This means somebody else has already been waiting.)

Otherwise, the request enters the MTLB's state machine described below. The number of MTLBs is configurable; the maximum allowable number is the number of shadow descriptors. Whether to have a unified MTLB for all the shadow descriptors or to have an independent MTLB for each shadow descriptors is currently an open question. The state machine of each MTLB is controlled by an event. Properly scheduling the event is necessary to keep the state machine running[3]

State 0

This state means nothing is really happening in the MTLB. So the simulator tries to find a waiting access to process, if such a waiting access exists. It picks up the next access according to the following sequence: miss reply alway gets the first priority; miss waiter has the second priority; accesses in the buffer waiting queue are the next group to search; accesses in the MTLB waiting queue have the lowest priority.

[3]Previous experiments showed that was extremely difficulty and error-prone.

State 1: MTLB_state_1()

This state is the starting point of the state machine, meaning start a MTLB lookup. Every new access starts here. First, it looks for a match in the MTLB. If a match is found, go to **state 2**. Otherwise, it checks the buffer for a match. If an entry is matched and its state is valid, the control event is scheduled to go to **state 3** at the next cycle. If the state of matched entry shows that this entry was reserved for an ongoing fetch, the access enters the buffer waiting queue and the state machine sets state **state 0** to allow the MTLB to continue servicing accesses.

If the access misses in the buffer, it tries to reserve a buffer entry and issues a load request to the memory controller by using function MMC_post_fastread. If no entry is available (i.e., all of them are reserved for ongoing fetches), the MTLB sets the miss waiter and stops processing new accesses and the state is set back to **state 0**.

State 2: MTLB_state_2()

This state is entered after an access hits in the MTLB. It first generates a physical address and calls MMC_mtlb_return to send the new physical address to the memory controller. Next, it sets the ref and modify bit of the matching page table entry accordingly. If either one of these two bits is changed, MTLB_state_2 schedules the control event to go to **State WB** at the next cycle. Otherwise, it schedules the control event to go to **State DONE** at the next cycle. This state also updates the reference count of MTLB entries, which is used to implement NRU (Not Recently Used) replacement policy.

State 3: MTLB_state_3()

This state means that a miss has returned. It first allocates an entry in the MTLB for the new entry: either a free entry or the one with least count (NRU replacement). Then, it goes to **state 2**.

State WB: MTLB_state_wb()

This state writes a modified entry back to memory by calling function MMC_post_fastwrite. After that, it goes to **state done**.

State DONE: MTLB_state_done()

An access has completed. The busy field (cur_trans) of the MTLB is cleared and the state machine goes back to **state 0**.

MTLB_trans_returned()

This function is used by the memory controller to inform the MTLB a certain MTLB request has completed. If that certain request is a write, this function does nothing. If that is a load request, this function's behavior depends on the current state. It schedules the control event to the next cycle with **state 3** if the MTLB is not being accessed. Otherwise, it adds the reply into miss reply waiting queue, which will be serviced right after the current access releases the MTLB.

MTLB_start_prefetch

The MTLB can speculatively load page table entries before they are accessed. This function allocates an entry in the MTLB buffer and sends a prefetch request to the memory controller. If there is no free entry available in the buffer, a prefetch is simply dropped. This function is called in two situations during **state 1**: when a prefetched line is being hit; when a miss is detected.

11.4 MC-based prefetching

Source files: **MMC/mmc_prefetch.c**

The prefetching algorithm currently supported by URSIM is next-one sequential prefetch: prefetch the next line when a request misses in the MCache; prefetch the next line when a prefetched line is hit by a non-prefetch transaction. The simulator supports several different prefetching options. They are controlled by the global variable mparam.prefetch_on: the first bit indicates whether or not prefetch non-shadow data; the second bit indicates whether or not prefetch shadow data; the third bit selects when to send prefetch accesses to DRAM — either when there is no outstanding transaction in the MMC ("0") or whenever a prefetch access is generated by algorithm ("1"). If the former is selected, a prefetch is saved into a prefetch queue. The size of the prefetch queue is configurable and works in first-in-first-out order. When the prefetch queue is full, the old one is popped out and the new one is pushed in. When the MMC does not have outstanding non-prefetch transactions, it will issue the prefetch access at the head of prefetch queue.

11.5 Initialization and statistics

Source files: **MMC/mmc_init.c, MMC/mmc_cache.c, MMC/mmc_remap.c, MMC/mmc_tlb.c, MMC/mmc_stat.c**

Header files: **MMC/mmc.h, MMC/mmc_ams.c, MMC/mmc_stat.h**

The function MMC_init is called by SystemInit. It first reads configuration parameters from the configuration file. Then, it allocates and initializes global data structures in the MMC simulator. Next, MMC_cache_init is called to initiate memory controller cache, MMC_remap_init is called to set up the initial states of the Impulse remapping controller, and MTLB_init is called to initialize the memory controller TLB.

The MMC simulator uses a lot of simple counters to collect statistics and the statistics collection code spreads around source files. The first class of statistics is collected for each type of memory access (namely, read, write, and writeback): count, average latency, total cycles, and percentage of hitting MCache. The second class of statistics is about shadow access: count, average latency, total cycles, and cycles stall for indirection vector (only occurs during scatter/gather through indirection vector). The third class of statistics is about MC-based prefetching (statistics for shadow and nonshadow are reported separately): total number of issues, hits, invalidates, conflicts, etc. The fourth class is on the MTLB: count of accesses, hits, misses, hitumiss (hit under miss), evictions, stall time, waiting time, and page table entry prefetch numbers.

Chapter 12

DRAM Backend

The DRAM backend module was simulated based on Lixin Zhang's initial design for the CS676 class project. It is likely to change in the a near future, so we will wait until then to write this chapter. Those who are really interested in the current DRAM backend simulator can read the document [21].

Bibliography

[1] J. E. Bennett and M. J. Flynn. Performance factors for superscalar processors. Technical Report CSL-TR-95-661, Stanford University, Feb 1995.

[2] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communication of the ACM*, 31(10):1220–1227, October 1988.

[3] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth IEEE Symposium on High Performance Computer Architecture*, pages 70–79, Orlando, FL USA, January 1999.

[4] J. B. Carter, W. C. Hsieh, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, L. B. Stoller, and T. Tateyama. Memory system support for irregular applications. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 17–26. Pittsburgh, PA, May 1998.

[5] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, January 1991.

[6] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

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

[9] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual, version 1.0. Technical Report 9705, Rice University, 1997. Available from `http://www-ece.rice.edu/~rsim`.

[10] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. 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 Operating Systems*, pages 12–23, Cambridge, MA USA, October 1996.

[11] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. The interaction of software prefetching with ilp processors in shared-memory systems. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 144–156, Denver, CO USA, June 1997.

[12] Rice University. *YACSIM Reference Manual*, March 1993. Available at `http://www-ece.rice.edu/~rsim/`.

[13] R. Schumann. Design of the 21174 memory controller for digital personal workstations. *Digital Technical Journal*, 9(2), November 1997.

[14] K. Skadron and D. W.Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third IEEE Symposium on High Performance Computer Architecture*, pages 144–155, San Antonio, TX USA, February 1997.

[15] J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Honolulu, Hawaii, May 1981.

[16] SPARC International, Inc., Menlo Park, CA. *The SPARC Architecture Manual*, 8th edition, 1992.

[17] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 284–291, Denver, CO USA, June 1997.

[18] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 204–213, Barcelona, Spain, June 1998.

[19] D. L. Weaver and T. Germond. *The SPARC Architecture Manual, version 8*. SPARC International, Inc., Menlo Park, CA, 1994.

[20] L. Zhang. ISIM: The simulator for the Impulse adaptable memory system. Technical Report UUCS-99-017, University of Utah, September 1999.

[21] L. Zhang. Design a DRAM backend for the Impulse memory system. Technical Report UUCS-00-002, University of Utah, January 2000.

[22] L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee. Memory system support for imaging processing. In *Proceedings of the 1999 International Conference on Parallel Architecture and Compilation Techniques*, pages 98–107, Newport Beach, CA USA, October 1999.

[23] L. Zhang and L. Stoller. Reference manual of Impulse system calls. Technical Report UUCS-99-018, University of Utah, July 1999.