RSIM Reference Manual

Version 1.0

Vijay S. Pai
Parthasarathy Ranganathan
Sarita V. Adve

Dept. of Electrical and Computer Engineering
Rice University
6100 South Main
Houston, Texas 77005

Email: rsim@ece.rice.edu
URL: http://www-ece.rice.edu/~rsim

Technical Report 9705

August 1997

Acknowledgments

We thank other past and present members of the RSIM group for their contributions. Hazim Abdel-Shafi contributed parts of the memory system simulator code and documentation. Murthy Durbhakula provided valuable support in setting up the RSIM distribution. Jonathan Hall worked on initial versions of the memory system simulator. Tracy Harton supported our development effort over the last two years.

We are also grateful to the Rice Parallel Processing Testbed (RPPT) group. Significant parts of the RSIM memory and network system are based on code from RPPT, a project led by Prof. J. R. Jump and Prof. J. B. Sinclair and involving several graduate students.

The development of RSIM was funded in part by the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, the Texas Advanced Technology Program under Grant No. 003604016, and funds from Rice University. Vijay S. Pai is also supported by a Fannie and John Hertz Foundation Fellowship.

Contents

Chapter 1

Overview

Simulation has emerged as an important method for evaluating new ideas in both uniprocessor and multiprocessor architecture. Compared to building real hardware, simulation provides at least two advantages. First, it provides the flexibility to modify various architectural parameters and components and to analyze the benefits of such modifications. Second, simulation allows for detailed statistics collection, providing a better understanding of the tradeoffs involved and facilitating further performance tuning.

This document describes RSIM – the **R**ice **S**imulator for **I**LP **M**ultiprocessors (Version 1.0). RSIM is an execution-driven simulator primarily designed to study shared-memory multiprocessor architectures built from state-of-the-art processors. Compared to other current publicly available shared-memory simulators, the key advantage of RSIM is that it supports a processor model that aggressively exploits instruction-level parallelism (ILP) and is more representative of current and near-future processors. Currently available shared-memory simulators assume a much simpler processor model, and can exhibit significant inaccuracies when used to study the behavior of shared-memory multiprocessors built from state-of-the-art ILP processors [14]. A cost of the increased accuracy and detail of RSIM is that it is slower than simulators that do not model the processor.

We have used RSIM at Rice for our research in computer architecture [14, 15, 16, 17], as well as for undergraduate and graduate architecture courses covering both uniprocessor and multiprocessor architectures.

1.1 Key features of simulated systems

RSIM provides many configuration parameters to allow the user to simulate a variety of shared-memory multiprocessor and uniprocessor architectures. Key features supported by RSIM 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
- Simple and optimized memory consistency implementations

Memory hierarchy features:

- Two-level cache hierarchy
- Multiported and pipelined L1 cache, pipelined L2 cache
- Multiple outstanding cache requests

- Memory interleaving
- Software-controlled non-binding prefetching

Multiprocessor system features:

- CC-NUMA shared-memory system with directory-based cache-coherence protocol
- Support for MSI or MESI coherence protocols
- Support for sequential consistency, processor consistency, and release consistency
- Wormhole-routed mesh network

RSIM models contention at all resources in the processor, caches, memory banks, processor-memory bus, and network.

1.2 Simulation technique

RSIM interprets application executables. We chose to drive RSIM with application executables rather than traces so that interaction between events of different processors during the simulation can affect the course of the simulated execution. This allows more accurate modeling of the effects of contention and synchronization in simulations of multiprocessors, and more accurate modeling of speculation in simulations of multiprocessors and uniprocessors. We chose to interpret application executables rather than use direct execution because modeling ILP processors accurately with direct execution is currently an open problem.

RSIM is a discrete event-driven simulator based on the YACSIM library [3, 8]. Many of the subsystems within RSIM 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 each processor's pipelines and processes outstanding cache requests.

1.3 Platforms supported

The RSIM simulator is written in a modular fashion using C++ and C to allow ease of extensibility and portability. Currently, the simulator has been tested on:

- Convex Exemplar running HP-UX version 10
- SGI Power Challenge running IRIX 6.2
- SUN machines running Solaris 2.5 or above

Section 2.2 discusses the possibility of porting the simulator to other systems.

Requirements for application executables interpreted by RSIM are described in the next section.

1.4 Applications interface

RSIM simulates applications compiled and linked for SPARC V9/Solaris using ordinary SPARC compilers and linkers, with the following exceptions.

First, although RSIM supports most important user-mode SPARC V9 instructions, there are a few unsupported instructions. More specifically, all instructions generated by current C compilers for the UltraSPARC-I or UltraSPARC-II with Solaris 2.5 or 2.6 are supported. Unsupported instructions that may be most important on other SPARC systems include 64-bit integer register instructions and quadruple-precision floating-point instructions.

Second, the system trap convention supported by RSIM differs from that of Solaris or any other operating system. Therefore, standard libraries and functions that rely on such traps cannot be directly used. We provide an RSIM applications library to support such commonly used libraries and functions; all applications must be linked with this library. Nevertheless, there are some unsupported traps and related functions (e.g., `strftime`), and our library has only been tested for application programs written in C. More details are given in Chapter 5.

For faster processing and portability, the RSIM simulator actually interprets applications in an expanded, loosely encoded instruction set format. A predecoder is provided to convert SPARC application executables into this internal format, which is then fed into the simulator.

1.5 Future announcements and user feedback

We are currently engaged in various additions to the features supported by RSIM as well as improvements in implementation organization and efficiency. Users interested in announcements on the current and future releases of RSIM may send us email at `rsim@ece.rice.edu`. We also welcome feedback and contributions for future releases at the above mailing address.

1.6 Organization of this manual

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

Within Part I, Chapter 2 describes how to obtain and install RSIM version 1.0. Chapter 3 explains the architectural model simulated by RSIM. Chapter 4 explains how to configure RSIM to model the system desired, including command line, configuration file, and compile-time options. Chapter 5 explains how to port applications for simulation with RSIM. Chapter 6 describes the statistics collection and debugging support provided by RSIM.

Within Part II, Chapter 7 gives an overview of the various subsystems in the RSIM implementation. Chapter 8 explains the YACSIM event-driven simulation library functions used in RSIM. Chapter 9 describes the functions used for initializing and configuring RSIM. Chapter 10 gives an overview of the processor out-of-order execution engine, along with the event that controls processor and cache simulation. Chapter 11 describes the processor memory unit, which interfaces the processor pipelines with the memory hierarchy. Chapter 12 explains the fundamentals of the memory hierarchy and interconnection system simulator provided by RSIM. Chapter 13 examines the cache hierarchy implementation. Chapter 14 explains the implementation of the directory and memory module. Chapter 15 describes the interconnection systems simulated in RSIM. Chapter 16 gives information about the statistics and debugging support provided by RSIM. Chapter 17 describes the implementation of the `predecode` and `unelf` utilities.

Part I

RSIM USER'S GUIDE

Chapter 2

Installing RSIM

RSIM version 1.0 is available from `http://www-ece.rice.edu/~rsim/dist.html` at no cost with a non-commercial license agreement (Appendix A reproduces the license terms and conditions). This chapter describes how to install the software on the local system.

Section 2.1 describes how to unpack the software distribution on the local system. Section 2.2 explains the steps to build the RSIM executable, as well as other executables needed to simulate applications with RSIM. Section 2.3 describes the steps to build the RSIM applications library, which provides some of the library functions needed for building applications to run under RSIM. Section 2.4 describes the steps to build the sample applications provided with RSIM. Section 2.5 explains the requirements for using the statistics processing utilities provided with RSIM.

2.1 Unpacking the RSIM software distribution

RSIM version 1.0 is available as a UNIX `tar` file `rsim-1.0.tar` (uncompressed format) or `rsim-1.0.tar.gz` (compressed with `gzip`). Unpacking the file requires the UNIX `tar` utility. The compressed format also requires the `gunzip` utility (available from the Free Software Foundation at `http://www.gnu.org`).

For the compressed format, first uncompress the file (some web browsers will uncompress the file automatically) with:

prompt% `gunzip rsim-1.0.tar.gz`

This will create the file `rsim-1.0.tar`. Now unpack this file with:

prompt% `tar -xf rsim-1.0.tar`

This will produce a directory `rsim-1.0`, containing the following subdirectories:

apps Example applications ported to RSIM (including source, executable, and output files), the RSIM applications library, header files, and generic application makefile

bin RSIM utilities for statistics processing

docs Documentation related to RSIM, including this manual

incl Header files for RSIM

inputs Sample configuration files for simulated systems, a Javascript utility to construct new configuration files

obj Makefiles for compiling RSIM for all supported platforms

src C and C++ source files for the RSIM simulator

2.2 Building the RSIM simulator

Each of the following directories in `rsim-1.0` contains a Makefile for a specific simulation host platform and simulation target store-ordering policy. RSIM versions with store-ordering support the memory consistency models of sequential consistency and processor consistency, while versions without store-ordering support release consistency. (See Chapter 3 for more details.) Except where noted otherwise, the makefiles in the directories below compile a simulator for a release-consistency target system. Debugging makefiles are also provided to compile RSIM with debugging and tracing options. All makefiles are provided for the standard `make` utility for each system.

obj/hp Convex Exemplar platform with HP PA-8000 processors and HP-UX version 10

Makefile assumes GNU C and C++ compilers, version 2.5.8 or above

obj/sgi SGI PowerChallenge platform with MIPS R8000 processors and IRIX 6.2

Makefile assumes SGI C and C++ compilers, version 6.2 or above

obj/ss10 Sun SPARCstation-10 with Solaris 2.5 or above

Makefile assumes SUN C and C++ compilers, version 4.0 or above

obj/ultra140 Sun Ultra-I/140 workstation with Solaris 2.5 or above

Makefile assumes SUN C and C++ compilers, version 4.0 or above

obj/ultra170 Sub Ultra-I/170 workstation with Solaris 2.5 or above

Makefile assumes SUN C and C++ compilers, version 4.0 or above

obj/SC_hp,SC_ultra,SC_sgi As above, except compiles a simulator for a target system with store-ordering (to simulate systems with sequential consistency or processor consistency). `SC_ultra` is optimized for Sun Ultra-I/140 workstations, but can also be used on Ultra-I/170 workstations.

obj/dbg,obj/SC_dbg Debugging makefiles for Sun workstations with Solaris 2.5 or above

Makefile assumes GNU C and C++ compilers, version 2.5.8 or above

These makefiles create an executable called `rsim`. For example, to make an `rsim` executable that will run on a Convex Exemplar target to simulate a system with processor consistency, the user should type the following in the `rsim-1.0` directory:

prompt% `cd obj/SC_hp`
prompt% `make rsim`

This sequence should cause the system to start compiling all of the C and C++ source files of RSIM into relocatable object files. After this, the make utility links these object files to form the `rsim` executable. The compile-time parameters specified in the Makefile are listed in Section 4.3.

Additionally, the `predecode` executable must be created on any of the Sun platforms listed above by typing the commands below from the `rsim-1.0` directory (any of the Sun directories listed above may be used in place of `ss10`):

prompt% `cd obj/ss10`
prompt% `make predecode`

`predecode` translates the instructions of a SPARC application executable into a form that can be processed by RSIM, as explained in Section 1.

`unelf` must be built and run only if the user intends to run RSIM on platforms that do not support the ELF library functions[1]. `unelf` must be built on a platform that supports ELF (such as a Sun platform with

[1] Currently, the only tested platform that does not use ELF is the Convex Exemplar with HP-UX.

Solaris). For example, `unelf` can be built on a SPARCstation 10 by typing the following command sequence from the `rsim-1.0` directory:

prompt% `cd obj/ss10`
prompt% `make unelf`

It should be relatively simple to compile `rsim`, `predecode`, and `unelf` for related systems in the same processor and operating system families as specified above. The optimization options in the makefile may need to be changed to represent a processor of a different generation, and some C preprocessor flags may need to be defined or left undefined according to the characteristics of the specified system. Additionally, it should also be straightforward to change the Makefiles to run with another make utility on a supported platform, such as `gnumake`. The file `obj/README` documents the possible changes needed.

Porting RSIM to other architectures may or may not be straightforward. In particular, porting to 64-bit platforms or little-endian platforms may require additional effort.

2.3 Building the RSIM applications library

The RSIM applications library provides some of the library functions needed for linking the applications to be simulated, as described in Chapters 1 and 5.

The RSIM applications library is located in the directory `apps/utils/lib`. This directory includes a makefile used for actually building the library on a SPARC Solaris platform. The user should type `make install` to use this makefile. This will create the RSIM libraries. Additionally, it will also produce a separate modified version of the standard system C library in the directory `apps/utils/lib`. This modified C library excludes many of the functions used internally by other functions in the C library to insure that the linker properly resolves these references to the RSIM applications library, rather than to standard functions in the system C library.

Details on supported and unsupported system library functions appear in Chapter 5.

2.4 Building applications ported to RSIM

The `apps` directory includes two example applications ported to RSIM: a parallel red-black SOR (in the directory `apps/SOR`) and a parallel quicksort algorithm (in the directory `apps/QS`). The inputs taken by these applications and the outputs provided are described in the `README` files in each of the application directories. These applications are provided primarily for instructional purposes. Both programs can be meaningfully run with inputs small enough to fully trace a large part of their execution. Additionally, these applications are useful for familiarizing oneself with the RSIM command line and configuration file.

To facilitate use of the SPLASH [20] and SPLASH-2 [27] applications, the RSIM distribution includes a set of PARMACS macros appropriate for RSIM in the file `apps/utils/lib/rsim_macros`.

2.4.1 Using the generic makefile

The RSIM distribution contains a generic makefile that can be used for building applications. This generic makefile includes all the needed steps for compiling, linking, and predecoding applications to be simulated. If needed, the generic makefile also expands the RSIM PARMACS macros in source files with the suffixes `.C`, `.U`, and `.H` (as appear in the SPLASH and SPLASH-2 applications). Invocation of the `unelf` command is not included in the generic makefile, as this command is only used on some target systems; thus, this command must be invoked separately if the user intends to run RSIM on non-ELF host platforms.

The generic makefile is in `apps/utils/lib/makefile_generic` in `rsim-1.0`. This makefile uses rules specified according to the rules of the Sun `make` utility for Solaris. At installation time, the fully-specified pathname of the directory in which this file is located should be inserted in the `LIBDIR` variable of the makefile (This field is set by default to `/home/rsim/rsim-1.0/apps/utils/lib`).

Makefiles for a specific application can include the 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`, and seeks to produce a predecoded executable named `execs/app_rc.out.dec`, the application makefile should simply read:

```
SRC = source1 source2
HEADERS = header1 header2
TARGET = app

include ../../utils/lib/makefile.generic
```

Then, to actually construct the desired predecoded executable, the user should type the following from the application directory:

prompt% `make execs/app_rc.out.dec`

Once the generic makefile is invoked, it will expand the PARMACS macros (if needed) in the source files, and compile, link, and predecode the target.

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

2.4.2 Using ordinary UNIX command sequences

Some users may want to invoke the commands for building applications directly rather than using the generic makefile. In particular, users intending to run RSIM on platforms without the ELF library must directly run `unelf` on the Solaris executables of the applications to be simulated.

The first step is to generate relocatable object files (`.o` files) from the source-code files. This can be done using an ordinary SPARC C compiler for Solaris. The recommended options for invoking the Sun C compiler version 4.0 to generate object code from the source file `src/source1.c` are:

prompt% `cc -xO4 -xtarget=ultra1/170 -xarch=v8plus -dalign -o obj/source1.o -c src/source1.c`

These options generate code with all optimizations recommended by the compiler, with code scheduled for Sun Ultra-1 workstations with a 170 MHz UltraSPARC processor. The code uses the SPARC V8plus subset of the V9 architecture, and assumes that double-precision accesses are properly aligned, allowing use of double-precision floating-point loads and stores.

Next, a SPARC application executable must be generated. Recommended options for invoking the Sun linker to generate the file `execs/appname.out` from the files `obj/source1.o` and `obj/source2.o` are (substituting the fully-specified pathname of the RSIM distribution for `/path_to_rsim/rsim-1.0` below):

prompt% `/usr/ccs/bin/ld -dn -z muldefs -L /path_to_rsim/rsim-1.0/apps/utils/lib -emystart -o execs/appname.out /path_to_rsim/rsim-1.0/apps/utils/lib/crt0.obj obj/source1.o obj/source2.o -l rsim -l c -l m -l rsim`

This generates a statically-linked executable that starts with the function `mystart`, linking the RSIM application startup object file (`crt0.obj`) with the application object files, and resolving unknown references with the RSIM library, the C library, and the system math library. The RSIM library is included twice so that unresolved references from the C and math libraries are resolved to RSIM library functions, when applicable.

If a different set of linker options is chosen, the user must guarantee that the linker output produced is a statically-linked application executable and that the entry point for the executable is the same as the base of the text segment, which in turn must correspond to the `mystart` function. Additional constraints on the application executable are given in Chapter 5.

After generating a SPARC application executable, the file to be run through RSIM must be predecoded, as described in Section 1. The syntax of `predecode` is:

prompt% `predecode execs/appname`

where `execs/appname.out` is the name of the SPARC application executable file to be predecoded. This command produces a file called `appname.out.dec` and also produces output on the screen related to the file being predecoded. As this output is generally not needed, the user will usually want to redirect this to `/dev/null`.

Users intending to run RSIM on target platforms that do not support ELF will need to first process the application executables to be simulated with the `unelf` utility as follows:

prompt% `unelf execs/appname.out`

where `appname.out` is the name of the file to be expanded. This command produces a file called `app.out_unelf`. `unelf` itself must be run on an ELF platform; however, we do not expect this to add any difficulty, as the applications themselves are currently built using an ELF platform.

2.5 Statistics processing utilities

The `bin` directory in `rsim-1.0` includes shell-scripts and `awk`-scripts for processing statistics files output by RSIM. This directory should be added to the `PATH` environment variable of each RSIM user. For `csh` and `tcsh` users, the following can be typed at the shell-prompt or added to the user's `.cshrc` or `.tcshrc` files (substitute the actual fully-specified path of the RSIM distribution for `/path_to_rsim/rsim-1.0/` below).

prompt% `setenv PATH /path_to_rsim/rsim-1.0/bin:${PATH}`

For `sh`, `ksh`, and `bash` users, the following command sequence is appropriate:

prompt$ `export PATH=/path_to_rsim/rsim-1.0/bin:$PATH`

Three of the shell scripts included in this directory must be modified according to the directory in which the RSIM distribution is download. Namely, the scripts `analyze_misses`, `rsim_analyze`, and `p_rsim_analyze` currently include a reference to the directory `/home/rsim/rsim-1.0/bin`. This reference should be changed to point to the actual fully-specified path of the RSIM distribution. For example, the file `p_rsim_analyze` contains the line:

```
nawk -f /home/rsim/rsim-1.0/bin/p_rsim_analyze.awk
```

This line must be changed to:

```
nawk -f /path_to_rsim/rsim-1.0/bin/p_rsim_analyze.awk
```

where `/path_to_rsim/rsim-1.0/` is replaced by the actual fully-specified path of the RSIM distribution.

The usage of the statistics-processing utilities is explained in Chapter 6. These utilities do not require compilation.

Chapter 3

Architectural Model

The following sections describe the instruction set, the processor microarchitecture, the cache and memory subsystem, and the multiprocessor interconnection network supported by RSIM.

3.1 RSIM instruction set architecture

RSIM simulates applications compiled and linked for SPARC V9/Solaris using ordinary SPARC compilers and linkers, with the following exceptions.

First, although RSIM supports most important user-mode SPARC V9 instructions, there are a few unsupported instructions. More specifically, all instructions generated by current C compilers for the UltraSPARC-I or UltraSPARC-II with Solaris 2.5 or 2.6 are supported. Unsupported instructions that may be most important on other SPARC systems include 64-bit integer register operations and quadruple-precision floating-point instructions. The other unsupported instructions are `tcc`, `flush`, `flushw`, and tagged add and subtract (described in the SPARC V9 ISA definition [23]).

Second, the system trap convention supported by RSIM differs from that of Solaris or any other operating system. Therefore, standard libraries and functions that rely on such traps cannot be directly used. We provide an RSIM applications library to support such commonly used libraries and functions; all applications must be linked with this library. Nevertheless, there are some unsupported traps and related functions (e.g., `strftime`), and our library has only been tested for application programs written in C. More details are given in Chapter 5.

The main simulator actually interprets input files generated by running an offline predecoder on the application executables generated. The predecoder generates a more loosely-encoded target format, which is used for all internal processing in RSIM. This removes the overhead of runtime instruction decoding and will facilitate modifications of RSIM to simulate other RISC ISAs. RSIM can use a predecoder because this simulator does not support self-modifying or dynamically generated code.

3.2 Processor microarchitecture

RSIM models a processor microarchitecture that aggressively exploits ILP. It incorporates features from a variety of current commercial processors. The default processor features include:

- Superscalar execution – multiple instructions issued per cycle.
- Out-of-order (dynamic) scheduling[1]
- Register renaming
- Static and dynamic branch prediction

[1] An option for in-order scheduling is provided as a straightforward modification to the out-of-order scheduling pipeline, but is not well tested. Details of the implementation of this feature are provided in Section 10.2.

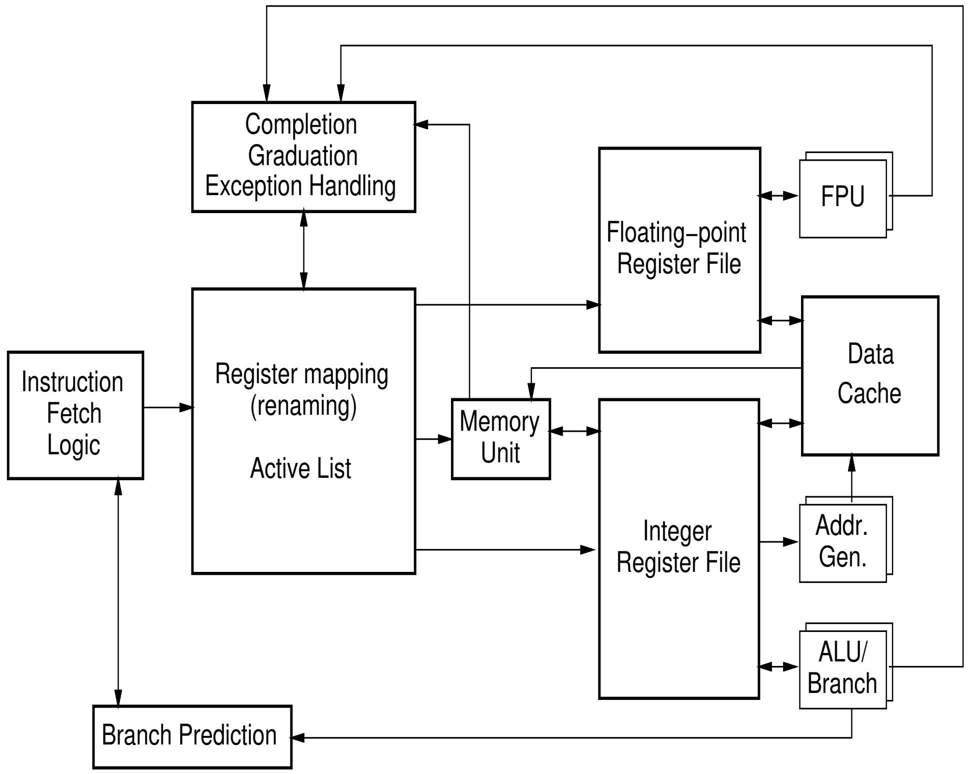


Figure 3.1: RSIM Processor Microarchitecture.

- Non-blocking memory load and store operations
- Speculative load execution before address disambiguation of previous stores
- Software-controlled non-binding prefetching
- Support for multiple memory consistency models and various implementations of these models [15]
- Precise exceptions
- Register windows

The processor microarchitecture modeled by RSIM is closest to the MIPS R10000[13] and is illustrated in Figure 3.1. In particular, RSIM 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 to physical registers), and *shadow mappers* (which store register map table information on branch prediction to allow single-cycle state recovery on mispredictions). The 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 RSIM to implement precise interrupts.

Most processor parameters are configurable at runtime. These parameters are listed in Chapter 4.

3.2.1 Pipeline stage details

The **instruction fetch stage** reads instructions from the predecoded input file. RSIM currently does not model an instruction cache. The maximum number of instructions brought into the processor per cycle is a configurable parameter.

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 mappers. These data strucutres closely follow the corresponding microarchitectural features of the MIPS R10000. The RSIM 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 [28]. 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. The previous value of the logical register remains in the physical register to which it was formerly mapped. This physical register is not returned to the free list until the instruction with the new mapping has graduated. Integer and floating point registers are mapped independently. Currently, RSIM assumes that the processor will always have sufficient renaming registers for its specified active list size[2].

An instruction is entered into the active list when it is decoded, and it remains in the active list until it graduates. This stage also dispatches memory instructions to the memory unit, which is used to insure that memory operations occur in the appropriate order, as detailed in Section 3.2.3[3]. The size of the active list and memory unit are configurable.

For branch instructions, the decode stage allocates a shadow mapper to allow a fast recovery on a misprediction, as discussed in Section 3.2.2. The prediction of a branch as taken stops the RSIM processor from decoding any further instructions in this cycle, as many current processors do not allow the instruction fetch or decode stage to access two different regions of the instruction address space in the same cycle. The number of shadow mappers is configurable.

The **instruction issue stage** issues ready instructions. For an instruction to issue, it must have no outstanding data dependences or structural hazards. With one exception, the only register data dependences that affect the issue of an instruction in RSIM are RAW (true) dependences; other register dependences are eliminated by register renaming[4]. RAW dependences are detected by observing the "busy bit" of a 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 RSIM: 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 dependence 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 if the address of the instruction has already been generated. Additional constraints for memory issue are described in Section 3.2.3.

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.

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

The completion 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.

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

[2]Code for fewer renaming registers and consequent register stalls is included but has not been tested and is not exposed to the user. Chapter 10 gives a more detailed explanation.

[3]A command-line option is also provided to specify that non-memory instructions should also be dispatched to an issue window; by default, these instructions are issued directly from the active list. More details about this option are provided in Section 4.1.1.

[4]Single-precision floating-point operations experience WAW (output) dependences because floating-point registers are mapped and renamed on double-precision boundaries. This is further discussed in Chapter 10

(before 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 3.2.3. After graduation, the instruction leaves the active list. The number of instructions that can graduate in a single cycle is configurable.

The RSIM processor also detects exceptions at the point of graduation. Section 3.2.4 describes how the processor simulator handles exceptions.

3.2.2 Branch prediction

The RSIM 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. Shadow mappers are 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.

RSIM currently supports three branch prediction schemes: dynamic branch predictors using either a 2-bit history scheme [22] or a 2-bit agree predictor [24], and a static branch predictor using only compiler-generated predictions. Return addresses are predicted using a return address stack [9]. 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.

3.2.3 Processor memory unit

The processor memory unit is the interface between the processor and the caches. Currently, instruction caches are not simulated and are assumed to be perfect. RSIM also does not currently support virtual memory[5]. A processor's accesses to its private data space (described in Section 5.1) are currently considered to be cache hits in all multiprocessor simulations and in uniprocessor simulations configured for this purpose. However, contention at all processor and cache resources and all memory ordering constraints are modeled for private accesses in all cases.

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

Constraints for precise exceptions

The RSIM 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 the next cycle and all previous instructions must have completed successfully. A store can be allowed to graduate, as 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, unless the multiprocessor memory constraints require the store to remain in the unit.

Loads always wait until completion before leaving the memory unit or graduating, as loads must write a destination register value. Prefetches can leave the memory unit as soon as they are issued to the cache,

[5]RSIM also does not include the ability to mark certain regions of memory uncached, a feature commonly associated with virtual memory

as these instructions have no destination register value. Furthermore, there are no additional constraints on the graduation of prefetches.

Constraints for uniprocessor data depedences

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, the processor may need to maintains this data dependence by enforcing additional constraints on the execution of the load. If 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 multiprocessor ordering constraints allow this. 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 11.

In the third case, however, the load may or may not have a data dependence on the previous store. The behavior of the RSIM memory unit in this situation depends on the configuration options. In the default RSIM configuration, the load is allowed to issue to the cache, if allowed by the multiprocessor ordering constraints. 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 3.2.4.

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 memory unit variation stalls the issue of the load altogether whenever a prior store has an unknown address.

Constraints for multiprocessor memory consistency model.

RSIM supports memory systems three types of multiprocessor memory consistency protocols:

- Relaxed memory ordering (RMO) [23] and release consistency (RC) [6]
- Sequential consistency (SC) [11]
- Processor consistency (PC) [6] and total store ordering (TSO) [26]

Each of these memory models is supported with a straightforward implementation and optimized implementations. We first describe the straightforward implementation and then the more optimized implementations for each of these models.

The *relaxed memory ordering* (RMO) model is based on the memory barrier (or fence) instructions, called `MEMBAR`s, in the SPARC V9 ISA [23]. Multiprocessor ordering constraints are imposed only with respect to these fence instructions. A SPARC V9 `MEMBAR` can specify one or more of several ordering options. An example of a commonly used class of `MEMBAR` is a `LoadStore MEMBAR`, which orders all loads (by program order) before the `MEMBAR` with respect to all stores following the `MEMBAR` (by program order). Other common forms of `MEMBAR` instructions include `StoreStore`, `LoadLoad`, and combinations of the above formed by bitwise or (e.g. `LoadLoad|LoadStore`). Instructions that are ordered by the above `MEMBAR` instructions must appear to execute in program order. Additionally, RSIM supports the `MemIssue` class of `MEMBAR`, which forces all previous memory accesses to have been globally performed before any later instructions can be initiated; this precludes the use of the optimized consistency implementations described below[6].

Release consistency is implemented using RMO with `LoadLoad|LoadStore` fences after acquire operations and `LoadStore|StoreStore` fences before release operations.

In the *sequential consistency* (SC) memory model, all operations must appear to execute in strictly serial order. The straightforward implementation of SC enforces this constraint by actually serializing all memory instructions; i.e. a load or store is issued to the cache only after the previous memory instruction by program order is globally performed[7] [19]. Further, stores in SC maintain their entries in the memory unit until they have globally performed to facilitate maintaining multiprocessor memory ordering dependences from stores to later instructions. Unless RSIM is invoked with the *store buffering* command line option (discussed in Chapter 4), stores in SC do not graduate until they have globally performed. Forwarding of values from stores to loads inside the memory unit is not allowed in the straightforward implementation of sequential consistency. `MEMBAR`s are ignored in the sequential consistency model.

The *processor consistency* (PC) and *total-store ordering* (TSO) implementations are identical in RSIM. With these models, stores are handled just as in sequential consistency with store buffering. Loads are ordered with respect to other loads, but are not prevented from issuing, leaving the memory unit, or graduating if only stores are ahead of them in the memory unit. Processor consistency and total store ordering also do not impose any multiprocessor constraints on forwarding values from stores to loads inside the memory unit, or on loads issuing past stores that have not yet disambiguated. `MEMBAR`s are ignored under the processor consistency and total store ordering models.

Beyond the above straightforward implementations, the processor memory unit in RSIM also supports optimized implementations of memory consistency constraints. These implementations use two techniques to improve the performance of consistency implementations: hardware-controlled non-binding prefetching from the active list and speculative load execution [5].

In the straightforward implementations of memory consistency models, a load or store is prevented from issuing into the memory system whenever it has an outstanding consistency constraint from a prior instruction that has not yet been completed at the memory system. Hardware-controlled non-binding prefetching from the active list allows loads or stores in the active list that are blocked for consistency constraints to be prefetched into the processor's cache. As a result, the access is likely to expose less latency when it is issued to the caches after its consistency constraints have been met. This technique also allows exclusive prefetching of stores that have not yet reached the head of the active list (and which are thus prevented from issuing by the precise exception constraints).

Speculative load execution allows the processor not only to prefetch the cache lines for loads blocked for consistency constraints into the cache, but also to use the values in these prefetched lines. Values used in this fashion are correct as long as they are not overwritten by another processor before the load instruction completes its consistency constraints. The processor detects potential violations by monitoring coherence actions due to sharing or replacement at the cache. As in the MIPS R10000, a soft exception is marked on any speculative load for which such a coherence action occurs [28]; this soft exception will force the load to reissue and will flush subsequent instructions. The soft exception mechanism used on violations is the same as the mechanism used in the case of aggressive speculation of loads beyond stores with unknown addresses. Speculative load execution can be used in conjunction with hardware-controlled non-binding prefetching.

---

[6] We do not expect applications to use this type of `MEMBAR`. It is currently used in RSIM only in the RSIM system trap library.

[7] A store is globally performed when its value is visible to all processors; i.e. all other caches with a copy of the line have been invalidated. In RSIM, this is indicated when an acknowledgment for the store is received by the processor. A load is globally performed when its return value is bound and when the store whose value it returns is globally performed. In RSIM, this is detected when the load returns its value to the processor.

3.2.4 Exception handling

RSIM supports precise exceptions[8] 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.

RSIM supports the following categories of exceptions: division by zero, floating point errors, segmentation faults, bus errors, system traps, window traps, soft exceptions, serializing instructions, privileged instructions, illegal or unsupported instructions, and illegal program counter value. RSIM simply emulates the effects of most of the trap handlers; a few of the trap handlers actually have their instructions simulated, and are indicated below. (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 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.

Segmentation faults are currently split into two types. The first type is a fault that occurs whenever a processor wishes to grow its stack beyond its current limits. For this trap, pages are added to the stack to accommodate stack growth, and execution recovers from the point of the excepting instruction. In the second type of fault, 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 nonrecoverable.

A *bus error* occurs whenever a memory access is not aligned according to its length. Generally, these exceptions are nonrecoverable. However, the SPARC architecture allows double-precision floating-point loads and stores to be aligned only to a word boundary, rather than to a double-word boundary[9]. RSIM currently traps these accesses and emulates their behavior. However, the cache accesses for these instructions are not simulated.

System traps are used to emulate the behavior of operating system calls in RSIM. The system traps supported are listed in Section 5.2 and serve several important functions, such as I/O, memory allocation, and process management. Additionally, some system traps are specific to RSIM and serve roles such as statistics collection or explicitly associating a home node to a region of shared memory. Some operating system calls are currently not supported; consequently, functions using these system calls (such as `strftime` and `signal`) cannot currently be used in applications to be simulated with RSIM. The RSIM simulator trap convention does not currently match the system trap convention of Solaris or any other operating system; however, a library is provided with RSIM to insure that the correct system traps are invoked for each supported function, as described in Section 5.2. The system trap handler restarts program execution at the instruction after the excepting instruction.

A *window trap* occurs when the call-depth of a window-save chain exceeds the maximum allowed by RSIM (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) [23]. The instructions used in the window trap handler are actually simulated by the RSIM processor, rather than merely having their effects emulated. The window trap handler returns control to the excepting instruction. The number of register windows is configurable, and can range from 4 to 32 (in all cases, 1 window is reserved for the system). The effect of window traps is not likely to significantly affect the performance of scientific codes written in C; however, code written in a functional language such as Scheme or an object-oriented language such as C++ may experience some performance degradation from window traps.

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 3.2.3).

RSIM uses traps (referred to as *serialization* traps in the code) to implement certain instructions that either modify system-wide status registers (e.g. `LDFSR`, `STFSR`) or are outdated instructions with data-paths

[8] We use the terms *exception* and *trap* interchangably

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

that are too complex for a processor with the aggressive features simulated in RSIM (e.g. `MULScc`). This can lead to significant performance degradation in code that uses old libraries, many of which internally use `MULScc`. The trap handler for the `STFSR` instruction is actually simulated, rather than merely emulated. The `LDFSR` instruction is slightly different in that it does not have a trap handler, but functions more like a soft exception. Specifically, the function of the trap on this instruction is to prevent later instructions from committing their values computed with an old floating-point status. Thus, the trap for `LDFSR` can be thought of as a soft-exception that does not retry the excepting instruction.

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 some instructions unsupported by RSIM (i.e. `tcc`, `flush`, `flushw`, and tagged addition and subtraction)[10] An *illegal program counter* value exception occurs whenever a control transfer instruction makes the program counter invalid for the instruction address region (e.g. out of range or unaligned program counters). These three exception types are all non-recoverable.

3.3 RSIM memory and network systems

Figure 3.2 shows the memory and network system organization in RSIM. RSIM simulates a hardware cache-coherent distributed shared memory system (a CC-NUMA), with variations of a full-mapped invalidation-based directory coherence protocol. Each processing node consists of a processor, a two level cache hierarchy (with a coalescing write buffer if the first-level cache is write-through), a portion of the system's distributed physical memory and its associated directory, and a network interface. A pipelined split-transaction bus connects the secondary cache, the memory and directory modules, and the network interface. Local communication within the node takes place on the bus. The network interface connects the node to a multiprocessor interconnection network for remote communication.

Both cache levels are lockup-free and store the state of outstanding requests using miss status holding registers (MSHRs)[10].

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. RSIM allows for a multiported and pipelined first level cache. Lines are replaced only on incoming replies. The size, line size, set associativity, cache latency, number of ports, and number of MSHRs can be varied.

The coalescing write buffer is implemented as a buffer with cache-line-sized entries. All writes 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. RSIM allows for a pipelined secondary cache. Lines are replaced only on incoming replies; more details of the protocol implementation are given in Chapter 13. The secondary cache maintains inclusion with respect to the first-level cache. The size, line size, set associativity, cache latency, and number of MSHRs can be varied.

The memory is interleaved, with multiple modules available on each node. The memory is accessed in parallel with an interleaved directory, which implements a full-mapped cache coherence protocol. The memory access time, the memory interleaving factor, the minimum directory access time, and the time to create coherence packets at the directory are all configurable parameters.

The directory can support either a MESI protocol with Modified, Exclusive, Shared, and Invalid states, or a three-state MSI protocol. The RSIM directory protocol and cache controllers support cache to cache transfers (shown in Figure 3.3 as "$ to $"). Figure 3.3 gives simplified state diagrams for both protocols, showing the key states and transitions at the caches due to processor requests and external coherence actions. Internally, the protocols also include transient states at the directory and caches; these conditions are handled according to mechanisms specified in Chapters 13 and 14.

For local communication within a node, RSIM models a pipelined split-transaction bus connecting the L2 cache, the local memory, and the local network interface. The bus speed, bus width, and bus arbitration delay are all configurable.

[10] RSIM does not yet raise any exception on some unsupported instructions, such as 64-bit integer operations or quadruple-precision floating-point accesses. It is the user's responsibility to insure that such instructions are not used. The compiler options we provide in Section 2.4 inform the compiler not to generate these instructions.

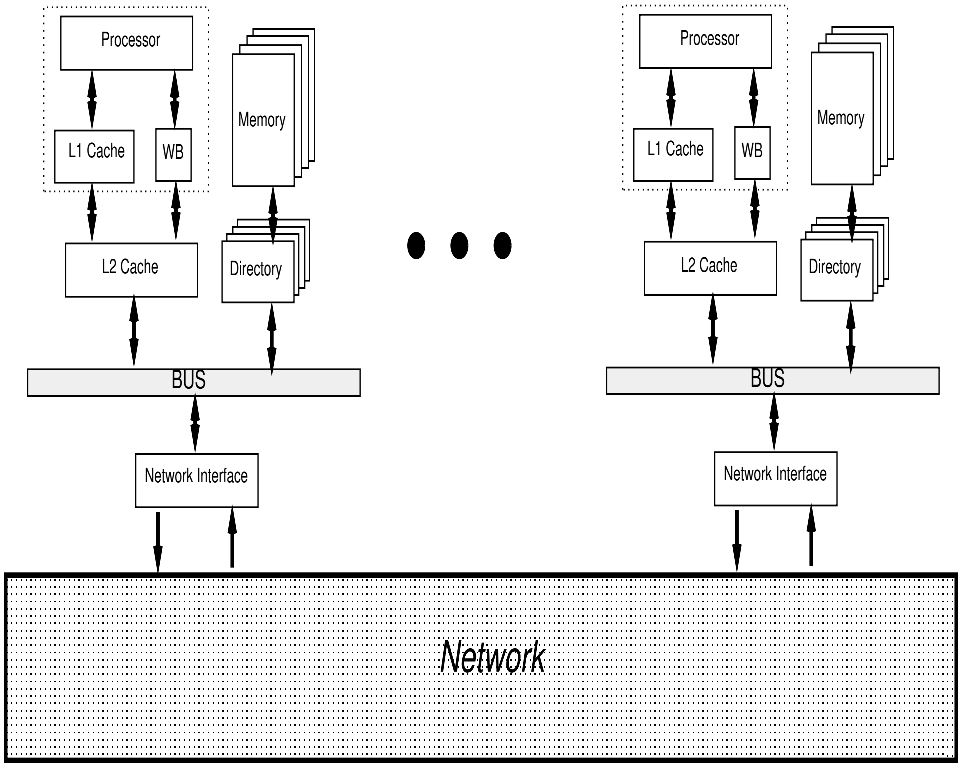
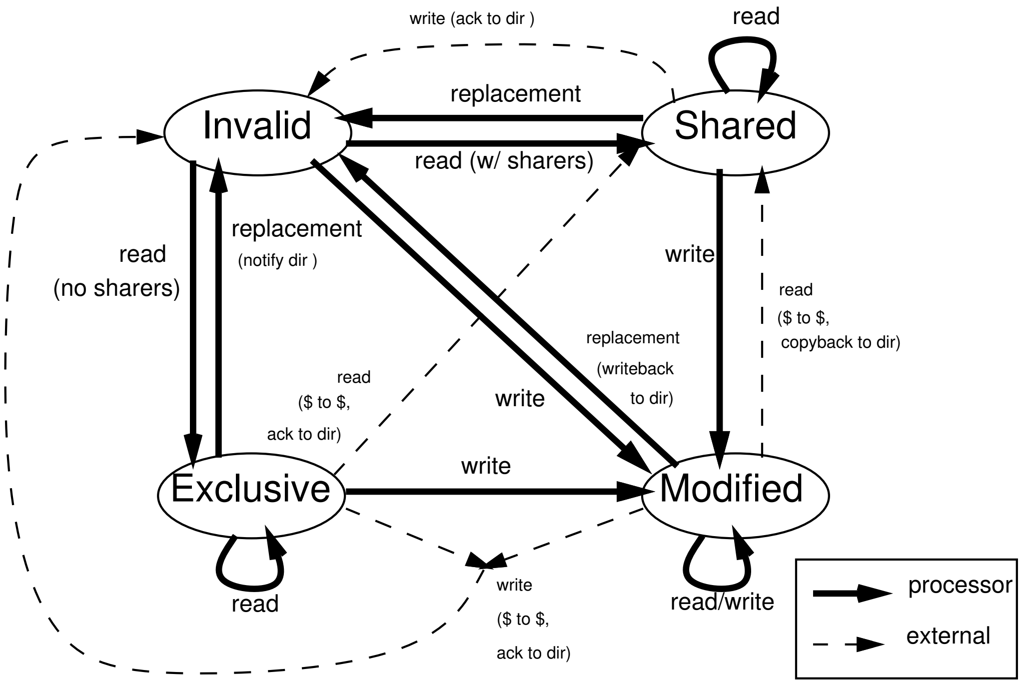


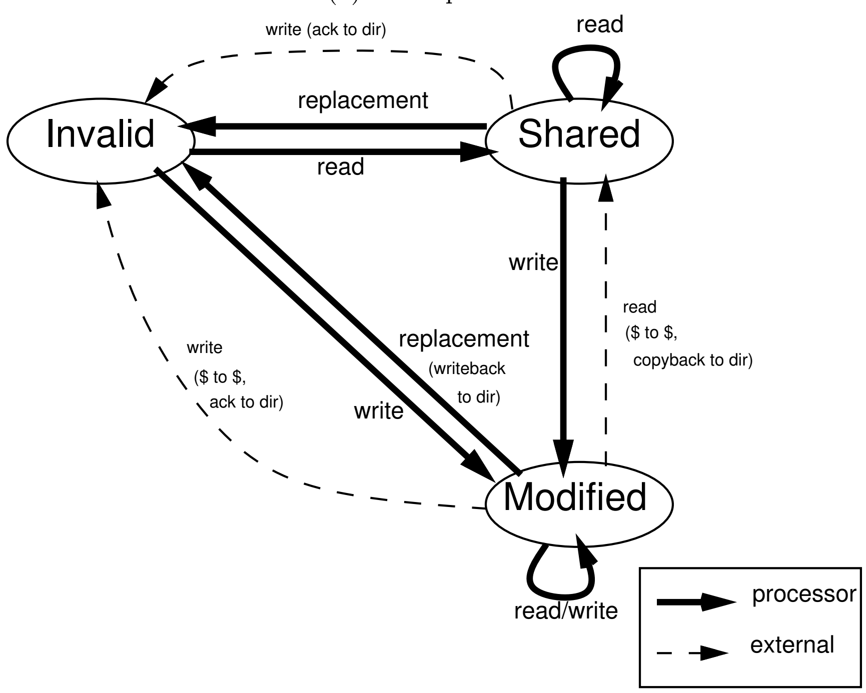
Figure 3.2: The RSIM memory system

For remote communication, RSIM currently supports a two-dimensional mesh network[11]. RSIM models a pipelined wormhole-routed network with contention at the various switches. For deadlock avoidance, the system includes separate request and reply networks. The flit delay per network hop, the width of the network, the buffer size at each switch, and the length of each packet's control header are user-configurable parameters.

[11]The potential for supporting other network models is discussed in Section 15.3.



(a) MESI protocol



(b) MSI protocol

Figure 3.3: Coherence protocols in RSIM

Chapter 4

Configuring RSIM

This chapter discusses the various run-time and compile-time options available to configure RSIM, and specifies the default values for the parameters. The parameters most frequently modified in our experience are available to the user on the RSIM command line; most other parameters are presented to RSIM via a configuration file. Many configuration files can be used for different simulation runs, as the name of the configuration file is passed to RSIM on the command line. Many of the parameters ask for time in processor cycles; this is the main metric used in RSIM. By default, we assume a processor with 300 MHz and choose some of the default latencies with this in mind.

4.1 Command line options

Many of the parameters controlling the simulation and the simulated configuration are passed to RSIM on the command line. Command line arguments to the application being simulated are given after a double-dash. For example, to simulate the application program `sort` with an active list of size 64 and with the application parameters `-n 1024 -p8`, one would use the command line:

```
rsim -a64 -f sort -- -n1024 -p8
```

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

4.1.1 Processor parameters

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

-a num Active list size. Defaults to 64.

-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).

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

-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.

-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. Supported only with the straightforward implementation of release consistency. The static scheduling supported in RSIM 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. This option has not yet been extensively tested. Unused by default.

4.1.2 Memory unit parameters

-m num Maximum number of operations in the processor memory unit, described in Section 3.2.3. 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 3.2.3). The following table specifies the policies supported:

Policy number	Description
0	Stall load until all previous store addresses known (supported only with release consistency)
1	Issue load, but do not let other instructions use load value until all previous store addresses known (supported only with release consistency)
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.

-p Turn on hardware-controlled prefetching for optimized consistency implementations (discussed in Section 3.2.3). Bring all hardware prefetches to L1 cache.

-P Same as "-p", but brings write prefetches only to L2 cache.

-J All prefetches (software and hardware) go only to L2 cache.

-K Enable speculative load execution for optimized consistency implementations (discussed in Section 3.2.3).

-N Store buffering in SC: allows stores to graduate before completion [4, 17] (useful in SC only; stores graduate before completion in all other models by default; discussed in Section 3.2.3)

-6 Processor consistency, if RSIM compiled with `-DSTORE_ORDERING`. RSIM compiled with `-DSTORE_ORDERING` provides SC by default.

4.1.3 Cache parameters

Most cache parameters are specified in the RSIM configuration file. The command line parameters available are:

-H num,num Number of MSHRs [10] supported at the L1 and L2 cache, respectively. Defaults to "8,8".

-T Discriminate prefetching. If a hardware or software prefetch is stalled for resource constraints at the L1 cache (e.g. all MSHRs full), 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.

4.1.4 Approximate simulation models

These parameters allow RSIM to simulate simple processors with single instruction issue, static scheduling, and blocking reads, possibly with increased processor and/or cache clock rates. These parameters were used in our previous work to investigate the effectiveness of models based on simple processors in approximating the behavior of ILP processors [14]. It is important to note that the change in processor speed brought about by these parameters do not affect the latencies in absolute times for the other modules. For example, if the DRAM memory latency is specified to be 18 processor cycles with the default processor speed of 300 MHz, this translates to a 60 ns access time. With the "`-F4`" option, which speeds up the processor by a factor of 4, RSIM automatically increases the DRAM speed in terms of processor cycles by a factor of 4 (i.e. 72 processor cycles and 60 ns in absolute time).

These approximate simulation models are not intended to speed up the performance of RSIM, but are provided only for purposes of comparison.

-k Turn off ILP simulation; i.e. simulate a processor with single-issue, static scheduling, and blocking reads. Supported only for RC.

-F num Increase processor clock speed by the factor specified in **num**. Defaults to 1.

-y num Increase L1 cache access speed by the factor specified in **num**. Defaults to 1.

4.1.5 Other architectural configuration parameters

-n Turn on simulation of private accesses. Currently supported only in single processor mode. It is highly recommended that this option be used if RSIM is used in uniprocessor studies.

-W Addresses in the shared region of the RSIM application address space are normally associated with a specific home node, which provides the directory services for those addresses. If an address being accessed has not been associated with a specific home node using `AssociateAddrNode` (described in Chapter 5), the default action for RSIM is to print a warning message and associate the cache line using a first-touch policy. If the "-W" option is used, no warning message is printed.

4.1.6 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.

-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 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, RSIM 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.

-z file Redirects simulator input (configuration file) to **file**. This option can be used (either alone or in conjunction with "-0") to redirect the input required by the simulator (such as the configuration file) separately from the input required by the application. (If "-0" is used without "-z", both the simulator input and application input come from the same file.) **Note:** to use the default values of all configuration file parameters, **file** should be set to `/dev/null`. Defaults to stdin.

-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.

4.1.7 Simulator control and debugging

-f file Name of application file to interpret with RSIM, without the `.out` or `.dec` suffix. Defaults to "a".

-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.

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

-t num Number of cycles to simulate before dumping detailed tracing information when RSIM is compiled with the debugging makefiles described in Section 2.2. Defaults to 0.

4.2 Configuration file

Several configuration inputs can be passed to RSIM through the configuration file on the simulation input (which can be redirected using "-0" or "-z" above). Sample configuration files are provided in the `inputs` directory of the distribution. Additional files can be generated with the aid of the Javascript utility `rsimconfig.html` in the `inputs` directory of the distribution. This utility can be used with any WWW-browser that supports Javascript (e.g. Netscape 2.0 or higher): the utility produces a list of configuration parameters for RSIM in the browser window. The configuration parameters produced should be copied from the browser window (using cut, copy, and paste) into a file that can be used as the RSIM configuration file.

The parameters that can be specified in the configuration file are given below. Each parameter in the input file will be followed by either a non-negative integer or a string, as specified below. The parameters are not case sensitive. If any one of these is not included, RSIM will use the default value specified in the list below (thus, if only default configuration is desired, the simulation input should be redirected to `/dev/null`). If any parameter is listed multiple times in the configuration file, the last one specifies the actual value used.

If the same input file will be used for both the simulator input and the application input, the configuration parameters should be given first, and the pseudo-parameter **STOPCONFIG** should be used to separate the simulator input from the application input. As with all other parameters, **STOPCONFIG** must be followed by an argument; however, this argument is ignored in this case.

4.2.1 Overall system parameters

numnodes The number given with this parameter specifies the number of nodes in the system. Defaults to 16.

reqsz This parameter gives the length of the control header for each packet, in bytes. The control header includes the requested address, the source node, the destination node, and the command type for the packet. Defaults to 16.

4.2.2 Processor parameters

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

2bit	2-bit history predictor [22]. This is the default.
2bitagree	2-bit agree predictor [24]
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.

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

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

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

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

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

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

maxstack This number specifies the maximum size of each simulated process stack, in KB. Defaults to 1024.

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 (e.g. add, subtract, move). Default latency and repeat delay of 1 cycle.

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

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

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

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

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

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

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

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

Thus, to specify that common FPU operations have a latency of 5 cycles and a repeat delay of 2 cycles, the configuration file should include:

latflt 5
repflt 2

4.2.3 Cache hierarchy parameters

l1type This parameter specifies the L1 cache type. If "WT" is chosen, a write-through cache with no-write-allocate is used. If "WB" 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 "WT".

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

l1size This number specifies the size of the L1 cache in kilobytes. Defaults to 16.

l1assoc This number specifies the set associativity of the L1 cache. The cache uses an LRU-like replacement policy within each set (the policy uses LRU ages, but prefers to evict lines held in shared state rather than lines held in exclusive state, and unmodified lines rather than modified lines). Defaults to 1.

l1ports Specifies the number of cache request ports at the L1 cache. Defaults to 2.

l1taglatency Specifies the cache access latency at the L1 cache (for both tag and data access). Defaults to 1. (With the assumption of a 300 MHz processor, this represents a 3 ns on-chip SRAM.)

l2size This number specifies the size of the L2 cache in kilobytes. Defaults to 64.

l2assoc This number specifies the set associativty of the L2 cache. The cache uses an LRU-like replacement policy within each set. Defaults to 4.

l2taglatency Specifies the access latency of the L2 cache tag array. Defaults to 3.

l2datalatency Specifies the access latency of the L2 cache data array. Defaults to 5.

wrbbufextra The L2 cache includes a buffer for sending subset enforcement messages to the L1 cache and write-backs to memory. Write-backs remain in the buffer only until issuing to the bus, and subset enforcement messages remain only until issuing to the level 1 cache. This buffer must contain at least one entry for each L2 MSHR, since each outbound request may result in a replacement upon reply. The number specified with **wrbbufextra** indicates the number of additional entries to provide for the write-back buffer in order to allow more outgoing requests while other write-backs still have not issued. Defaults to 0. (More details about the write-back buffer are given in Chapter 13.)

ccprot The string given here specifies the cache-coherence protocol of the system; `mesi` and `msi` are acceptable values. Defaults to `mesi`.

wbufsize 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.

mshrcoal 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).

4.2.4 Bus parameters

buswidth This number specifies the width of the bus, in bytes. Defaults to 32.

buscycle The number listed here gives the bus cycle time in processor cycles. Defaults to 3.

busarbdelay This parameter gives the bus arbitration delay in processor cycles. Defaults to 1.

4.2.5 Directory and memory parameters

memorylatency This number specifies the DRAM access latency in cycles. (Note that this is only the actual time at the DRAM, and does not include time on the bus, in the caches, or at any other resource.) Defaults to 18 cycles (60 ns with a 300 MHz processor).

dircycle This number gives the minimum directory access latency, which is the latency incurred by non-data responses to the directory. (Requests to the directory for data or data-carrying responses to the directory must access the DRAM and thus incur the latency specified by **memorylatency**.) This parameter is specified in processor cycles, and defaults to 3.

meminterleaving This number specifies the degree of memory and directory interleaving at each node. Defaults to 4.

dirbufsize This number sets the maximum number of pending requests at the directory at any time. If additional requests are received, they are bounced back to the sender with a retry request.

dirpacketcreate Some transactions require the directory to send coherence requests to the other caches (such as invalidations or cache-to-cache transfer requests). This number specifies the time to produce the first such coherence request for a given transaction, in processor cycles. Defaults to 12.

dirpacketcreateaddtl This number specifies the additional delay in processor cycles for each subsequent coherence request for a given transaction (after the first one). Defaults to 6.

4.2.6 Interconnection network parameters

flitsize This parameter specifies the number of bytes in each network flit, which is equivalent to the width of the network in bytes. Defaults to 8.

flitdelay This parameter gives the latency for each flit to pass through a network switch, in processor cycles. Defaults to 4 (with a 300 MHz processor, this delay indicates a 13 ns flit latency).

arbdelay This parameter gives the delay invoked by arbitration at each network multiplexer for the head flit of any packet. Defaults to 4.

pipelinedsw This parameter allows the use of pipelined network switches, in which the flit delay of multiple flits can be partially overlapped. With pipelined switches, the flit delay for each successive flit in a packet will begin `pipelinedsw` number of cycles after the start of the previous flit delay. If the number 0 or a number greater than the flit delay is specified here, the switches are not pipelined. Defaults to 2 (with an assumption of 300 MHz processors, this implies a 150 MHz network cycle for a fully pipelined switch.)

netbufsize This parameter specifies the number of flits that can be buffered in each network switch. Defaults to 64.

netportsize This parameter specifies the network interface buffer sizes, in packets. Each node has 2 network interfaces, one for sending messages to the network, and one for receiving messages from the network. Each network interface has 2 port queues (described in Section 12.1) connecting the network interface to the bus (one for requests and one for replies). This parameter specifies the number of entries available in each port queue. This parameter is also used to specify the number of internal buffer entries in each network interface for actually sending messages to the network or receiving messages from the network. Each network interface has two internal buffers: one for requests and one for replies. The total number of packets in these two internal buffers is set by the parameter **netportsize**, split evenly between requests and replies. The network interface and interconnection network is described more thoroughly in Chapter 15.3.

4.2.7 Queue sizes connecting memory and network modules

The modules in the memory system are connected through ports, which are FIFO queues (see Section 12.1 for more information). Each port is a bidirectional connection, but the queues in each direction can have different lengths. An entry in the port queue is removed as soon as the appropriate module can start processing it. Thus, the port queue size also limits the number of requests that can be processed by the module each time the module is activated. For example, if a cache is intended to initiate processing for four requests each cycle, the request port queue should contain at least four entries.

The port queue can also contain more than the minimum number of entries; in these cases, the queue acts a buffer to decouple a faster module from a slower module. For this reason, the default port sizes from the L2 cache to the bus are larger than most of the other cache ports; these port sizes are chosen so that the potentially slow processing rate of the bus will not cause the L2 cache itself to stall.

The configurable port sizes are listed below, along with their default values. When applicable, the transaction type carried by each specified port is included in parentheses. These transaction types are explained in more detail in Section 12.2.

Name	Description	Default
portszl1wbreq	L1 → Write buffer (data request)	2
portszwbl1rep	Write buffer → L1 (data reply)	1
portszwbl2req	Write buffer → L2 (data request)	1
portszl2wbrep	L2 → Write buffer (data reply)	1
portszl1l2req	L1 → L2 (data request)	1
portszl2l1rep	L2 → L1 (data reply)	1
portszl2l1cohe	L2 → L1 (coherence request [including subset enforcement])	1
portszl1l2cr	L1 → L2 (coherence reply)	1
portszl2busreq	L2 → Bus (data request)	8
portszl2buscr	L2 → Bus (coherence reply, cache-to-cache data reply)	8
portszbusl2rep	Bus → L2 (data reply)	2
portszbusl2cohe	Bus → L2 (coherence request)	2
portszbusother	Bus → other modules	16 (per port)
portszdir	Directory → bus	64 (per port)

4.3 Compile-time parameters

Some parameters are specified at compile time only. This table lists the file where each of the following parameters is specified, as well as the variable or macro controlling the value of the parameter.

Filename	Description	Variables	Default
Makefile	Simulate sequential consistency or processor consistency system	`STORE_ORDERING`	RC (not defined)
architecture.c	Width of module ports	`INTERCONNECTION_WIDTH_BYTES`	16
exec.cc	Address generation unit latency	`latencies[uADDR]`	1 cycle
exec.cc	Address generation unit repeat rate	`repeat[uADDR]`	1 cycle
state.h	Page size (used for loading executable or increasing stack size)	`ALLOC_SIZE`	4 KB

Chapter 5

Porting Applications to RSIM

When porting applications to run under RSIM, the user must consider the following issues:

1. RSIM process creation and shared memory model
2. RSIM applications library
3. Synchronization support for multiprocessor applications
4. Statistics collection
5. Performance tuning of the application
6. Options to improve simulation speed

This chapter describes each of the above issues and the effect they have on porting applications to RSIM. Assembly programmers will also need to account for the unsupported instructions discussed in Section 3.1. The reader is encouraged to see the example applications included in the `apps` directory of the RSIM distribution as illustrations of the concepts discussed in this section.

5.1 Process creation and shared memory model

The RSIM multiprocessor memory model is depicted in Figure 5.1. The regions below the dividing line are all private memory, while the regions above the dividing line are shared memory allocated with the `shmalloc` function. The stack for each process grows automatically, while the heap and shared region grow only through explicit memory allocation calls.

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

The only way to allocate shared memory is through the `shmalloc` function (which has syntax similar to `malloc`). Memory allocated using `shmalloc` cannot currently be freed; code that needs a substantial amount of runtime dynamic shared-memory allocation and freeing should implement its own shared-memory allocator, using the `shmalloc` call only to allocate large regions of shared memory.

In multiprocessor mode, RSIM assumes a perfect hit rate for private accesses. Resource contention and instruction scheduling for private accesses is simulated, but the actual cache behavior is not. Private memory includes the process stack, statically allocated data structures, and heap space allocated by `malloc` and its

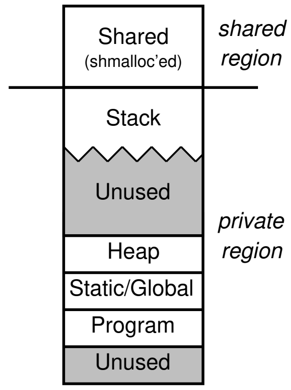


Figure 5.1: RSIM multiprocessor memory model

standard variants. All memory to be fully simulated must be explicitly allocated using `shmalloc`. Note that `shmalloc` can be used even in the case of uniprocessor simulations. However, in uniprocessor mode, all accesses (private and shared) can be simulated at the caches through the use of the "-n" option, discussed in Chapter 4.

Each region of shared memory has a "home node," which provides the directory services for the cache lines in that region. Shared-memory regions can be associated to specific home nodes using the `AssociateAddrNode` function. Regions can be associated at the granularity of a cache line. (The semantics for `AssociateAddrNode` are specified in more detail in Section 5.2.) If the user does not explicitly associate a shared region with a home node, the home node is chosen using a first-touch policy with a cache-line granularity.

5.2 RSIM applications library

As discussed in section 3.2.4, RSIM's trap convention differs from that of Solaris, so libraries that include Solaris system traps cannot be directly used with an application. Thus, RSIM is distributed with the RSIM Applications Library, which provides some of the library functions needed for linking the applications to be simulated. The RSIM Applications Library includes code derived from the GNU C Library and is distributed under the terms of the GNU Library General Public License, described at `http://www.gnu.org/copyleft/lgpl.html`.

The RSIM applications library includes the UNIX I/O functions (`close`, `dup`, `dup2`, `lseek`, `open`, `read`, and `write`), the `stdio` library (e.g. `printf`, `fscanf`, `fopen`), the `sbrk` memory allocation system trap, system traps associated with timing (`time` and `times`), the `exit` function, the `atexit` function, and the `getopt` function. The semantics of these functions are almost always identical to those on an ordinary UNIX system; however, in the case of an error, these functions will not set the `errno` variable (set by many of the standard UNIX functions).

Additionally, the RSIM applications library provides important multiprocessor primitives for process management, memory allocation, memory mapping, and synchronization. (The synchronization primitives are described in Section 5.3.) Other functions, such as the math library, the string library, the standard memory-allocation functions (`malloc`, `free`, `calloc`, etc.), and random-number generators can be linked from the ordinary system libraries.

Besides the above system traps used for supporting UNIX functions, a few additional system traps are provided with RSIM to enable ease of programming. The call `getpid()` returns the processor number of the calling process. The trap `sys_bzero(void *addr, int sz)` performs a fast, non-simulated clearing of a region of memory. This trap can be useful for speeding up initialization in some applications.

As described in Section 5.1, new processes are generated in RSIM using the `fork()` call, which spawns off a new process on a separate processor. RSIM also provides a new library function called `atfork()` which provides a set of optional cleanup functions to be called by the parent process before executing a `fork()`, just as the standard UNIX function `atexit()` does for the `exit()` call.

Shared memory can only be allocated through the use of the function `void *shmalloc(int size)`. This function returns a pointer to a region of shared memory with a size of `size`, just as `malloc` does for the private heap. Currently, this function returns regions starting at a cache-line boundary.

`AssociateAddrNode(void *start, void *end, int proc, char *name)` assigns `proc` as the home node for the memory range from `start` (inclusive) to `end` (exclusive). This assignment can be done at the granularity of a cache line. This function is an optional performance-tuning technique for applications; if a cache line being accessed is not associated, it will be associated using a first-touch policy on a cache-line granularity. The `name` argument is required and can be used internally for debugging, but currently has no externally visible role.

`abort(int code)` forces all processors to stop immediately; this differs from `exit(int code)` in that `exit` 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.

Some system traps support timing functions. `time` and `times` report the simulated time of the application execution and have nearly the same semantics as in Unix, although `time` starts the clock at the beginning of simulation rather than the beginning of 1970. `sysclocks()` gives a more detailed measurement, actually returning the number of simulated processor clock cycles since the start of the simulation.

Currently, some UNIX system traps are not supported with RSIM at all; consequently, functions based on these traps (such as `strftime` and `signal`) are not supported. Additionally, self-modifying code is not supported.

5.3 Synchronization support for multiprocessor applications

The RSIM applications library supports three types of multiprocessor synchronization primitives – locks, flags (or pauses), and barriers. To use the lock and flag synchronization functions, the application must include the header file `<locks.h>`; for barriers, the application must include `<treebar.h>`.

Locks are supported through a test-and-test-and-set mechanism. The lock functions act on a single integer in a shared region (one allocated with `shmalloc`). These locks should be declared volatile in order to prevent the compiler from optimizing accesses to them. The macro `GETLOCK(int *lock)` takes a pointer to the lock as an argument and spins with a test-and-test-and-set loop until the lock is available. The macro `FREELOCK(int *lock)` takes a pointer to the lock as an argument and frees the lock. Both of these macros invoke functions which perform the necessary operations. These macros also include the `MEMBAR` instructions required for release consistency. All instructions within a lock acquire form an aggregate class of type `ACQ` for statistics purposes, while instructions within a lock release are aggregated as `REL`.

Flags are supported through an ordinary spinning mechanism. The macro `WAITFLAG(int *flag, int val)` spins on the flag pointed to by `flag` until the integer value held in that address reaches the value of `val`. Similarly, the macro `WRITEFLAG(int *flag, int val)` sets the value addressed by `flag` to `val`. The functions associated with both of these macros include the necessary `MEMBAR` instructions for RC. Flag instructions are not aggregated by default, as these are often used in code to form novel synchronization types (such as prefix-summing trees). If the user desires to have flags counted as an aggregate class, he or she must explicitly place a `START_<agg>` and `END_<agg>` around flag macros, where `<agg>` represents one of the aggregate classes discussed in Section 5.4. The aggregate class `SPIN` is provided specifically for this purpose; alternatively, the `USR<num>` aggregates can be used to isolate several independent flags.

The only type of barrier supported is a simple binary tree barrier. A tree barrier is a structure of type `TreeBar` and can be declared as an ordinary global variable. The function `TreeBarInit(TreeBar *bar, int numprocs)` is used to initialize the tree barrier to a barrier that requires the specified number of processors. This function should be called before the `fork()` system call is invoked. When synchronizing, each processor should call the macro `TREEBAR(TreeBar *bar, int pid)`, where `pid` is the processor's unique ID number (initially determined through `getpid()`). The function called by this macro includes the necessary `MEMBAR` instructions. All instructions within a barrier are aggregated as type `BAR`.

The set of PARMACS macros provided with the RSIM distribution (described in Section 2.4) defines all the synchronization macros used in SPLASH and SPLASH-2 applications.

The user will need to provide any other synchronization desired (such as more advanced mutual exclusion mechanisms or barriers). The user should be careful to place the appropriate `MEMBAR`s in any such synchronizing libraries. The macros `ACQ_MEMBAR` and `REL_MEMBAR` are provided in the header file `<traps.h>` for the user's convenience: `ACQ_MEMBAR` should be inserted after an acquire, while `REL_MEMBAR` should be placed before a release operation to ensure that the appropriate fence operations are used with the RC model.

5.4 Statistics collection

RSIM automatically generates statistics for many important characteristics of the simulated system. RSIM 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` 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, described in Chapter 6.

There are additional macros that can be used within a processor phase to aggregate a set of instructions into a single statistics class. These macros are `START_USR<num>` and `END_USR<num>`, where `num` is an integer between 1 and 9. When RSIM prints the processor statistics for a phase, all the time spent between a set of these aggregation macros is lumped together, rather than being associated with the individual instructions included therein. In addition to the `USR<num>` aggregation classes, there are also aggregation classes for various synchronization operations called `ACQ`, `SPIN`, `REL`, and `BAR`. These aggregation classes can be used with `START_ACQ`, `END_ACQ`, and so forth.

Note that aggregate classes cannot be nested. Additionally, an aggregate class is not counted as graduating until all instructions in the class have graduated. Consequently, the partial statistics printed according to the "`-A`" option in Section 4.1.7 (i.e. the the number of cycles since each processor graduated an instruction) do not count instructions graduated within an aggregate class.

The functions `StatReportAll` and `StatClearAll` handle the statistics associated with the caches, memory system, and network. Each one applies to the entire system, and thus should usually be called only by processor 0 just after a barrier. `StatReportAll` prints out a detailed set of statistics associated with the caches, memory system, and network of the system, while `StatClearAll` clears all the statistics gathered.

5.5 Performance tuning

For best performance, the user should invoke the compiler with full optimization flags as described in Section 2.4. With such options, the compiler will not generate code using outdated instructions such as `MULScc` [23], which lead to poor performance with RSIM (see Section 3.2.4).

Whenever possible, the user should use double-precision floating-point operations rather than single-precision, as single-precision floating-point operations currently require additional overhead. Specifically, floating-point registers in the RSIM processor are mapped and renamed according to double-precision boundaries. As a result, operations with single-precision destination registers are must read the previous value of the destination register before writing a portion of the register. This causes single-precision floating point instructions to suffer from output dependences.

The user should also inform the compiler to assume that accesses are aligned, as this will avoid unnecessary single-precision floating point loads. The makefile provided with the sample applications has this option set for the Sun C compiler.

5.6 Options to improve simulation speed

RSIM provides a few special functions and macros to increase the speed of simulation. These functions allow faster simulation in two ways: by moving application initialization offline and by simulating less of the memory system.

5.6.1 Moving data initialization offline

The RSIM applications library provides two special-purpose functions, `SpecialInitInput` and `SpecialInitOutput`, that can be used in some cases to bypass the initialization phase of an application. In applications that spend a large amount of time in initialization, the initialization phase can be run through RSIM or natively on the host machine once. After the initialization phase, a "snapshot" of the state of the memory can be taken, such that later simulation runs can load this snapshot into their memory space in lieu of actually simulating the initialization phase.

The `SpecialInitOutput` function allows the memory values of a program variable to be stored to a file specified in the arguments. For example,

```
SpecialInitOutput(char *array, int size, char *filename);
```

allows the contents of the program variable specified by `bigarray` of size (in bytes) specified by `size` to be stored in the file specified by `filename`[1].

The `SpecialInitInput` function can then be used in the actual simulated run of the application to load the contents of this file into the data array and quickly initialize the data structures. `SpecialInitInput` can be invoked as follows,

```
SpecialInitInput(char *array, int size, char *filename);
```

The `SpecialInitInput` and `SpecialInitOutput` functions thus provide a simple way to facilitate offline (faster) initialization to be interfaced with a more detailed simulation methodology.

Note that the values held in these data arrays should not include pointers, as the pointers used in the native or simulated run which invoked `SpecialInitOutput` may not correspond to the pointers needed by the simulated runs which will call `SpecialInitInput`.

5.6.2 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 multiprocessor and 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. Note that each processor must independently invoke `MEMSYS_OFF` and `MEMSYS_ON`; the decision of whether or not to simulate the full memory system is made on a processor-by-processor basis.

[1] `SpecialInitOutput` is simply an `fopen` followed by an `fwrite`.

Chapter 6

Statistics Collection and Debugging

6.1 Statistics collection

RSIM provides a wide variety of statistics related to the processors, the caches, the network, and the memory system. RSIM prints a concise summary of the most important 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 functions and statistics-reporting functions described in Chapter 5 to print statistics for relevant portions of the application separately, rather than for the entire application at once.

Additionally, RSIM provides partial statistics periodically (with a period set by the "-A" option), in which each processor specifies the amount of simulated time it has been executing, and the number of cycles since the last graduation (for this statistic, an aggregate class is not considered to provide any graduations until all instructions in the set have graduated). This information can be useful for detecting application deadlocks (if each synchronization is made into an aggregate class), or for detecting deadlocks caused by simulator changes. RSIM can be forced to print partial statistics immediately if the user sends an alarm signal to the RSIM process by invoking a `kill -ALRM <pid>` at the shell prompt, where `<pid>` is the UNIX process ID of the simulator.

6.1.1 Overall performance statistics

RSIM displays the total execution time and the IPC (instructions per cycle) achieved by the program on the system simulated. In order to better characterize the bottlenecks in application performance, the total execution time is further categorized into busy time and stalls due to various classes of instructions. These classes of instructions include ALU, FPU, data reads, data writes, exceptions, branches, synchronization, and up to 9 user-defined aggregate classes discussed in Section 5.4. Data read and write stalls are further split according to the level of the memory hierarchy at which the memory operations were resolved: L1 cache, L2 cache, local memory, or remote memory.

With ILP processors, the various components of execution time described above are not easily separable, as multiple instructions can execute in parallel and out of order on such systems. We use the following policy, also used in other studies (e.g. [14, 15, 18]) to attribute execution time to the various components. If, in any given cycle, the processor retires the maximum allowable number of instructions, we count that cycle as part of busy time. Otherwise, we charge that cycle to the stall time component corresponding to the first instruction that could not be retired. Thus, the stall time for a class of instructions represents the number of cycles that instructions of that class spend at the head of the active list before retiring.

6.1.2 Processor statistics

RSIM provides statistics on the branch prediction behavior, the occupancy of the active list, and the utilization of various functional units. RSIM also provides statistics related to the performance of the instruction 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.

6.1.3 Cache, network, and memory statistics

RSIM classifies memory operations at various levels of the memory hierarchy into hits and misses. Misses are further classified into cold, conflict, capacity, and coherence misses. The method for distinguishing between conflict and capacity misses is discussed in Section 13.6. Statistics on the average latency of various classes of memory operations, MSHR occupancy, and prefetching effectiveness are also provided by default. RSIM also provides statistics on bus utilization, write-buffer utilization, network contention, traffic, the usage of the network switch buffers, and the types of packets sent in the network (based on packet length and distance traveled).

6.2 Utilities to process statistics

The RSIM distribution includes several shell-scripts and `awk`-scripts to process the statistics files produced by the simulator. These are located in the `bin` directory. These are useful for collecting information about the behavior and performance of applications being simulated.

Each of the example applications provided with the RSIM distribution includes a `testoutputs` directory that has sample concise and detailed statistics generated by RSIM, as well as the output of several of the utilities described in this section.

6.2.1 The `stats` and `pstats` programs

The `stats` program is used to further condense the concise statistics generated by RSIM. This program displays the most important variables from the concise statistics file produced by RSIM, averaged across all processors. The `stats` program takes as input the phase of interest for the application and the concise statistics file of RSIM (the file redirected using the "-2" option, or the `_err` file produced as a result of the "-S" option). Multiple files can be processed with a single invocation of `stats`; each file is processed separately. For example, to condense the statistics of phase 2 from the files `app1_err` and `app1_opt_err`, one would type:

```
stats app1_err app1_opt_err 2
```

The statistics displayed are execution time and its various components, as well as the average latencies of memory operations, as seen from the point of entering the processor active list, from the point of address generation, and from the point of issue to the caches.

`pstats` is invoked with the same arguments as `stats`: the only difference is that `pstats` gives a more detailed categorization of the memory component of execution time according to the level of the memory hierarchy at which each access was resolved.

6.2.2 The `plotall` program

The `plotall` program converts the application statistics generated by `pstats` into a form that can be processed using the `splot` program. (At the time of this release, `splot` is available from the URL `ftp://cag.lcs.mit.edu/pub/splot/`.) Multiple files can be specified with a single invocation of `plotall`. The `plotall` script generates two different types of statistics. The first set of statistics gives the execution time for each simulation run specified, split into the components discussed in Section 6.1.1. The execution times for the various runs in this graph are normalized to the execution time of the first specified simulation run. The second set presents the relative weights of each of the components of execution time; thus the execution-time statistics for each application concise statistics file is normalized to its own execution time. This set of statistics is useful for determining the relative contribution of each specific component of execution time. The `plotall` program then invokes `splot` to produce PostScript plots from these statistics.

This final stage requires the `splot` program to be available and in a directory included in the user's `PATH` environment variable.

To create plots called `app1eff_exec.ps` (execution times) and `app1eff_wt.ps` (relative component weights) of the phase 2 statistics from the files `app1_err` and `app1_opt_err`, one would type:

```
plotall app1 eff app1_err app1_opt_err 2
```

The read and write components of execution time are shown in shades of red and green, respectively. Four shades of these colors are used to represent L1 hits (the lightest shade), L2 hits, local memory accesses, and remote memory accesses (the darkest shade).

6.2.3 The `stats_miss` program

The `stats_miss` program is used to generate some commonly used performance metrics from the concise and detailed statistics output files produced by RSIM, averaging across all processors or caches. The statistics displayed include the total number of read misses, the average absolute read miss latency from the point of address generation, the average read miss latency overlapped by the ILP processor between the point of address generation and graduation, and statistics about the number of references to various levels of the cache hierarchy. The output from `stats_miss` also includes information about the MSHR (miss status holding register) utilization at the L1 and L2 caches, as well as the average absolute latency beyond the L2 cache seen by read misses in the system.

Multiple simulation runs can be specified with a single invocation of `stats_miss`; each run is processed separately. The `stats_miss` program takes as input the the simulation output file names (without the `_stat` or `_err` suffixes), as well as the phase of interest for the program. The *processor phase* is the same as the phase used in the `stats` utility. However, the `stats_miss` utility also introduces the concept of the *cache phase*. The *cache phase* is the number of sets of statistics generated by `StatReportAll` before the specific call to `StatReportAll` that is of interest. Thus, the first call to `StatReportAll` leads to cache phase 0, and so on.

The syntax for invoking `stats_miss` to process the files associated with two different runs of an application (here `app1_stat` and `app1_err`, along with `app1opt_stat` and `app1opt_err`) is:

```
stats_miss -cache 1 app1 app1opt 2
```

where the cache phase of interest is assumed to be `1` and the processor phase of interest is assumed to be `2`. If the cache phase and processor phase are the same for the files specified, the cache phase does not need to be specified.

6.2.4 The `MSHR` program

The `MSHR` utility interfaces with `splot` to generate plots of the request MSHR utilization at the caches. These plots can be used to indicate overlap and contention in the system. Each plot gives the percentage of the time spent in the phase of interest (on the Y-axis) for which misses occupy at least N MSHRs, where N is the number on the X-axis. The phases understood by `MSHR` correspond to the cache phases of `stats_miss`. The maximum X-axis value is currently set to 8 in this script. Multiple simulation runs can be plotted on the same graph.

The `MSHR` program uses the following command line. `MSHR plotname graph1 app1_stat phase1 [graph2 app2_stat phase2] ...`

`plotname` specifies the title of the plot. `graph1` and `graph2` specify the names of the individual graphs in the plot, and are used in the legend of the plot. `app1_stat` and `app2_stat` are the filenames of the detailed statistics files generated by the simulation runs being plotted. `phase1` and `phase2` specify the cache phase to be used for each application detailed statistics file.

The program generates two command files for the L1 and L2 MSHR utilization in the working directory called `MSHR_plotnameL1.cmd` and `MSHR_plotnameL2.cmd` respectively, where `plotname` is the title of the graph specified in the command line. `MSHR` then runs `splot` on these command files to generate Postscript output files, which are displayed using `ghostview`.

6.3 Debugging

6.3.1 Support for debugging RSIM

RSIM provides compile-time options to enable copious debugging and diagnostic output to be printed to the standard output and to separate files. Such tracing information is likely to be very important to anyone seeking to modify RSIM. There are various compile time flags which the user can selectively turn on to get debugging information for one section of the system (e.g., the network interfaces, the first level cache), but it is often the case that the user will want all debugging flags available turned on together, as in the debugging makefiles provided with the RSIM distribution.

The processor debugging output is written into files called *corefiles*. Each processor has its own corefile, and the suffix of each file represents the processor number (with a `corefile.`*i* file for each processor with processor number *i*.) These files contain detailed information on each stage of simulation for every instruction.

The debugging output related to the memory hierarchy and network is printed on the simulation output file. This debugging information is also very detailed, providing information about nearly all relevant activity in the cache hierarchy, the directory, the busses, and the network interfaces.

Because of the amount of information provided by these debugging options, these files can quickly grow into several megabytes of data in just thousands of processor cycles. The "-t" and "-c" options described in Chapter 4 can be used to limit this tracing to the exact spot where the problem is suspected. In the case of deadlocks, this moment of simulation time can be determined to some extent through the use of the periodic partial statistics displayed by RSIM, as processors will usually stop graduating instructions soon after the deadlock ensues.

6.3.2 Debugging applications

RSIM does not currently include support for debugging application programs with a debugger like `gdb` or `dbx`, as RSIM does not expose information about the application being simulated to such a debugger. If RSIM encounters a nonrecoverable exception (such as a segmentation fault or bus error), all processors halt immediately and a termination message is printed on the standard error file. Application errors can be debugged either by running the applications natively on a machine using an ordinary debugger, or by running the application through RSIM instrumented with many `printf` calls. 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 RSIM.

A common mistake in porting applications to RSIM is to forget the "volatile" declaration for shared synchronization variables. This is important for avoiding compiler optimizations that may cause infinite loops.

Part II

RSIM DEVELOPER'S GUIDE

Chapter 7

Overview of RSIM Implementation

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

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

RSIM is implemented in a modular fashion for ease of development and maintenance. The primary subsystems in RSIM are the event-driven simulation library, the processor out-of-order execution engine, the processor memory unit, the cache hierarchy, the directory module, and the interconnection system. These modules perform the following roles:

Event-driven simulation library Steers the course of the simulation, activating the processors, memory hierarchy, and network subsystem according to the demands of the simulated application and system. This subsystem is based on the YACSIM event-driven simulation library [3, 8].

Processor out-of-order execution engine Maintains the processor pipelines described in Section 3.2.

Processor memory unit Interfaces the processor pipelines with the caches, maintaining the various ordering constraints described in Section 3.2.3.

Cache hierarchy Processes requests to the caches, including both demands from the processor and demands from external sources. A significant part of this subsystem is based on code from the RPPT (Rice Parallel Processing Testbed) direct-execution simulator.

Directory and memory module Processes requests from various sources, maintaining the cache-coherence protocol of the system. This module is also based on code from RPPT.

Interconnection system Connects the various modules within each node, and the nodes within the multiprocessor system. The multiprocessor interconnection network is based on the NETSIM simulation environment [7].

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 RSIM to be quite focussed, and affecting only the desired functionality. Our experience with RSIM in a classroom setting has borne these expectations out to some extent; however, each type of simulator change does require detailed knowledge of the subsytem being modified.

The remaining chapters in this part of the manual describe the above subsystems in detail and provide other additional information needed to understand the implementation of RSIM. Chapter 8 gives a brief explanation of the event-driven simulation library underlying RSIM and the manner in which RSIM uses it. Chapter 9 describes the initialization routines in RSIM. Chapter 10 gives an overview of `RSIM_EVENT` and describes the details of each stage in the processor out-of-order execution engine. Chapter 11 explains the implementation of the processor memory unit. Chapter 12 describes fundamental data structures used for the cache, memory, and network systems. Chapter 13 explains the key functions within the simulation of the cache hierarchy. Chapter 14 describes the implementation of the directory module. Chapter 15 explains the principles of the interconnection system simulated in RSIM. Chapter 16 explains the additional support provided by RSIM for statistics and debugging. Finally, Chapter 17 describes the RSIM predecoder and the `unelf` utility.

Chapter 8

Event-driven Simulation Library

The event-driven simulation library underlies the entire RSIM simulation system, guiding the course of the various subsystems in RSIM. The event-driven simulation library used in RSIM is a subset of the YACSIM library distributed with the Rice Parallel Processing Testbed [3, 8].

Section 8.1 describes the YACSIM event-manipulation functions used by RSIM. Section 8.2 describes the use of YACSIM semaphores, which can be used to control the interaction between YACSIM events. Section 8.3 describes the fast memory-allocation pools used in RSIM.

8.1 Event-manipulation functions

Source files: `src/MemSys/driver.c`, `src/MemSys/act.c`, `src/MemSys/pool.c`, `src/MemSys/userq.c`, `src/MemSys/util.c`

Header files: `incl/MemSys/simsys.h`

All actions that take place during the course of an RSIM simulation occur as part of YACSIM *events*. 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 state value and argument for use on a future invocation.

An event is a specific type of YACSIM *Activity*; however, it is the only type used in RSIM. The following functions are used for manipulating events in RSIM.

- `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. `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 `ActivitySetArg` described below and is read with `ActivityGetArg`. `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. The `type` argument is available for any use by the user of the event-driven simulation library. RSIM events always have this field set to 0.

- `int EventSetState(int stval)`

  This function can only be called within the body 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 the event to determine the current state of the event.

- `void ActivitySetArg(ACTIVITY *aptr, char *argptr, int argsize)`

  This function sets the argument of the event pointed to by `aptr` to the value of `argptr`, with `argsize` 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 strucutre at the time when the argument is set, if intervening operations reset the value of the structure.

- `char *ActivityGetArg(ACTIVITY *aptr)`

  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.

- `int ActivityArgSize(ACTIVITY *aptr)`

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

- `ActivitySchedTime(ACTIVITY *aptr, double timeinc, int blkflg)`

  This operation schedules the event pointed to be `aptr` for `timeinc` cycles in the simulated future. The only valid value of `blkflg` for events is `INDEPENDENT`.

- `EventReschedTime(double timeinc, int stval)`

  This operation schedules the next invocation of the event for `timeinc` cycles in the simulated future. The state of the event upon rescheduling will be `stval`. (This function must be called within the event to be rescheduled, whereas `ActivitySchedTime` can be called from another event or from outside an event.)

- `ActivitySchedSema(ACTIVITY *aptr, SEMAPHORE *semptr, int blkflg)`

  This operation schedules the event pointed to be `aptr` for the time when the semaphore pointed to by `semptr` becomes available (described in Section 8.2). The only valid value of `blkflg` for events is `INDEPENDENT`.

- `EventReschedSema(SEMAPHORE *semptr, int stval)`

  This operation schedules the next invocation of the event according to the time when the semaphore pointed to by `semptr` becomes available. The state of the event upon rescheduling will be `stval`. (This function must be called within the event to be rescheduled, whereas `ActivitySchedTime` can be called from another event or from outside an event.)

The YACSIM event-list is implemented as a calendar queue [2]. Event-list processing in YACSIM is controlled by the function `DriverRun(double timeinc)`, which processes the event list for `timeinc` cycles or until the event list has no more events scheduled (if the value of `timeinc` given is less than or equal to 0).

The function `void YS__errmsg(char *s)` can be used at any point in the simulation to print out the error message `s` and terminate the simulation. This function is commonly used for unexpected simulation occurrences. The function `void YS__warnmsg(char *s)` prints out the warning message `s` on the simulation output file, but does not terminate the simulation. This function can be used to warn of unexpected happenings in the simulated system.

8.2 Semaphore functions

Source files: `src/MemSys/userq.c`

Header files: `incl/MemSys/simsys.h`

YACSIM *Semaphores* are used for controlling the interaction of events in the simulator. The following semaphore functions are supported in RSIM:

- `SEMAPHORE *NewSemaphore(char *sname, int i)`

  This function creates a new semaphore with the name specified in `sname` and the initial semaphore value `i`.

- `void SemaphoreSignal(SEMAPHORE *sptr)`

  This function activates the event at the head of the semaphore queue (added through a previous call to `ActivitySchedSema` or `EventReschedSema`). If there is no event, the semaphore value is incremented.

8.3 Memory allocation functions

Source files: `src/MemSys/pool.c`

Header files: `incl/MemSys/simsys.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 RSIM are:

- `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.

- `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 cool to `YS__PoolInit`.

  This function also performs minimal initialization when allocating from the pool of memory system transactions.

- `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.

- `void 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 [8].

Chapter 9

Initialization and Configuration Routines in RSIM

Source files: `src/Processor/mainsim.cc`, `src/Processor/simio.cc`, `src/Processor/config.cc`, `src/Processor/exec.cc`, `src/Processor/units.cc`, `src/Processor/funcs.cc`, `src/Processor/traptable.cc`, `src/MemSys/architecture.c`, `src/MemSys/net.c`, `src/Processor/state.cc`, `src/Processor/startup.cc`

Header files: `incl/Processor/mainsim.h`, `incl/Processor/simio.h`, `incl/Processor/units.h`, `incl/MemSys/arch.h`

RSIM 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.

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 4.

The various input and output files used by the simulator and application are redirected according to the command-line options. The `FILE` data structures called `simin`, `simout`, and `simerr` are set up through the `SimIOInit` function; if the simulator input and output are redirected separately from the application input and output, the function `RedirectSimIO` is called to point these to the appropriate files.

Next, the function `ParseConfigFile` is invoked to read in the options from the configuration file (described in Chapter 4) and set global simulation variables based on these options. 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 `ConfigureCacheProt` function converts the protocol names "`mesi`" and "`msi`" to values of type `enum CacheCoherenceProtocolType`, while the `ConfigureInt` function merely calls `atoi` to read a string into an integer variable. The parameter names and values are currently case-insensitive. If the user adds any new entries to this table, the macro `NUM_CONFIG_ENTRIES` must also be changed accordingly.

The application to be simulated is chosen based on the command line options. 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. The hash table for the `SharedPageTable` structure is initialized, and initially contains no elements.

`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. After this, the function `TrapTableInit` sets up the instruction lists for the simulator traps that are actually simulated (as opposed to only having their effects emulated). These traps include window traps and stores of the floating-point status register.

Next, the `SystemInit` function is called to set up the RSIM memory system and multiprocessor interconnection network according to the parameters earlier read from the configuration file with `ParseConfigFile`. After setting some basic parameters according to whether or not certain features are present (such as the write-buffer or pipelined network switches), this function calls `dir_net_init`. More information about `dir_net_init` is provided in Section 12.3.

After this point, the first processor data structure (or `state`) 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 10.9).

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 stack and register set. The `startup` function performs each of these actions. For systems with ELF, `startup` interfaces with the ELF library to extract the relevant sections from the application file; for systems without ELF, `startup` relies on a version of the application file preprocessed with the `unelf` utility discussed in Section 2. The 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 (processor program counter) is set to the entry point of the application executable, while the NPC (next program counter) points to the subsequent instruction.

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

Chapter 10

RSIM_EVENT and the Out-of-order Execution Engine

Section 10.1 gives an overview of RSIM_EVENT, the event corresponding to the processors and cache hierarchies. The rest of this chapter focuses on the 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 11 and 13.

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 from the active list.

10.1 Overview of RSIM_EVENT

Source files: `src/Processor/state.cc`, `src/Processor/pipestages.cc`

RSIM_EVENT simulates the processors and cache hierarchies of the simulated system. It is scheduled every cycle as described in Chapter 7. On every invocation, RSIM_EVENT loops through all the processors and caches in the system, calling the functions described in this section.

Note that RSIM_EVENT should seek to process messages from other portions of the system without any unexpected or unnecessary delays. Suppose, for example, that the bus unit provided a reply to the L2 cache at time X. However, if the RSIM_EVENT planned for time X had already occurred, the RSIM_EVENT function would not be able to pick up the reply until time $X + 1$. On the other hand, if the RSIM_EVENT function had not already been processed for the cycle, RSIM_EVENT would pick up the reply at time X. Thus, the timing behavior of the would be non-deterministic and could include unexpected delays. To avoid this sort of problem, RSIM_EVENT is actually scheduled to occur at an offset of 0.5 cycles from the processor cycle. RSIM_EVENT starts by completing operations that finished during the previous cycle, and then initiates new operations based on the current cycle.

For each processor, RSIM_EVENT first calls `L1CacheOutSim` and `L2CacheOutSim` (described in Section 13), which are used to process cache accesses. Then, `CompleteMemQueue` is called to inform the memory unit of any operations that have completed at the caches. `CompleteQueues` is used to process other instructions that have completed at their functional units.

Then, RSIM_EVENT calls `maindecode`. This function starts out by using `update_cycle` to update the register file and handle other issues involved with the completion stage of the 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.

RSIM_EVENT then calls `IssueQueues`, which sends ready instructions to their functional units. After this, the functions `L1CacheInSim` and `L2CacheInSim` are called for the caches to bring in new operations that have been sent to them. After this, RSIM_EVENT loops on to the next processor.

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.

10.2 Instruction fetch and decode

Source files: `src/Processor/pipestages.cc`, `src/Processor/tagcvt.cc`, `src/Processor/active.cc`, `src/Processor/stallq.cc`

Headers: `incl/Processor/state.h`, `incl/Processor/instance.h`, `incl/Processor/instruction.h`, `incl/Processor/mainsim.h`, `incl/Processor/decode.h`, `incl/Processor/tagcvt.h`, `incl/Processor/active.h`, `incl/Processor/stallq.h`

Since RSIM currently does not model an instruction cache, the instruction fetch and decode peline stages are merged. This stage starts with the function `decode_cycle`, called from `maindecode`.

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.

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 strucutre 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 RSIM. If the program counter is not valid for the application, the processor checks to see if the processor is in privileged mode. If so, and if the program counter points to a valid instruction in the trap-table, the processor reads an instruction from the trap-table instead. If the processor is not in privileged mode, or the PC is not valid in the trap-table, 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).

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

Then, 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 `win_num` field of the instance is set. This represents the processor's register window pointer (`cwp` or current window pointer) at the time of decoding this instruction.

`decode_instruction` then sets the functional unit type and 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.

At this point, 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 [23]. 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.

In a release consistent system, the processor will now detect `MEMBAR` operations and note the imposed ordering constraints. These constraints will be used by the memory unit.

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; 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 instructions 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 an `instance` data structure pointers. This structure is used internally for communication among the modules of the simulator.

Now the `check_dependencies` function is called for the dynamic instruction. If RSIM 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 RSIM 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 architecutre [23]. 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. Although the simulator assumes that there are enough renaming registers for the specified active-list size by default, `check_dependences` also includes code to stall if the instruction could not obtain its desired renaming registers.

After the `instance` has received its renaming registers and active list space, `check_dependences` 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 bits of `rs2` or `rscc` are 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 RSIM), or the processor is stalled until space is available (if the processor has static scheduling, or has dynamic scheduling with the "-q" option to RSIM).

If the instruction has no true dependences, the SendToFU function is called to allow this function to issue in the next stage.

decode_cycle continues looping until it decodes all the instructions it can (and is allowed to by the architectural specifications) in a given cycle.

10.3 Branch prediction

Source files: src/Processor/branchpred.cc, src/Processor/branchqelt.cc, src/Processor/branchresolve.cc

Headers: incl/Processor/bpb.h, src/Processor/branchq.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 "branch always"), then 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.

10.4 Instruction issue

Source files: src/Processor/pipestages.cc, src/Processor/exec.cc

Header files: incl/Processor/units.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 queue for the specified functional unit.

The issue function places the specified instance pm 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 structure of the processor, which is used to revive the instruction for completion after its functional unit latency has passed.

The IssueQueues function also calls IssueMem in the memory unit, which checks to see if any new memory accesses can be issued.

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

10.5 Instruction execution

Source files: src/Processor/funcs.cc, src/Processor/branchresolve.cc

The actual execution of instructions at their functional units is simulated through the functions in the file src/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 between the RSIM address of the reference to the corresponding address in the simulator's UNIX address space.

Some instructions are not currently supported in RSIM. These are tcc, flush, flushw, and tagged addition and subtraction instructions. These are considered illegal instructions and see the corresponding exception.

10.6 Completion

Source files: src/Processor/exec.cc, src/Processor/pipestages.cc, src/Processor/branchresolve.cc, src/Processor/branchq.cc, src/Processor/stallq.cc, src/Processor/active.cc,

Header files: incl/Processor/units.h, incl/Processor/state.h

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 src/Processor/funcs.cc and then inserts the instance onto the processor's DoneHeap. For memory instructions, this function marks the completion of address generation, and thus calls the Disambiguate function (described in Section 11). This function is also responsible for freeing 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.

The function update_cycle processes instructions from the DoneHeap data structure. For each instruction removed from the DoneHeap in a 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 approriate) 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 Section 11) is used to send the instruction to the address generation unit.

10.7 Graduation

Source files: `src/Processor/graduate.cc`

The `graduate_cycle` function controls the handling associated with 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, RSIM 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, an exception is detected, or the processor's maximum graduation rate is reached. At that point, control is returned to `graduate_cycle`.

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 this function, 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 nonblocking stores (PC, RC, or SC 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.

10.8 Exception handling

Source files: `src/Processor/except.cc`, `src/Processor/traps.cc`, `src/Processor/traptable.cc`

Header files: `incl/Processor/traptable.h`, `incl/Processor/instance.h`, `incl/Processor/hash.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. This function makes sure that all stores in the memory unit prior to the excepting instruction have issued to the caches before allowing any exception to trap to 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 3.2.4) 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. 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 RSIM 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 segmentation faults, the processor must determine whether this is a fault that simply indicates the need for growing the stack or an actual error. If the address is in a region that would be considered appropriate for the processor stack, the function `StackTrapHandle` is called. This function allocates space for the stack increase and adds those pages to the processor `PageTable`. For other types of segmentation faults, the function `FatalException` is called to print information about the type of violation and force all processors in the simulation to a halt.

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, RSIM currently does not simulate cache behavior for these accesses, instead calling the corresponding functions in `src/Processor/funcs.cc` immediately. In cases of genuine alignment failures, the exception is considered nonrecoverable, and `FatalException` is called.

The function `SysTrapHandle` handles the emulation of all supported system traps, which include some functions with UNIX-like semantics (`close`, `dup`, `dup2`, `exit`, `lseek`, `open`, `read`, `sbrk`, `time`, `times`, and `write`), and some additional traps provided by RSIM (corresponding to the following functions and macros: `abort`, `AssociateAddrNode`, `endphase`, `fork`, `GET_L2CACHELINE_SIZE`, `getpid`, `MEMSYS_OFF`, `MEMSYS_ON`, `newphase`, `shmalloc`, `StatClearAll`, `StatReportAll`, `sys_bzero`, and `sysclocks`). The UNIX I/O 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 `sbrk` function is processed by adding pages to the address space for the simulated processor. The `time` and `times` functions use the simulated cycle time to set the appropriate return values and structure fields. Although all of these functions have the same behavior as UNIX functions on success (except as noted in Section 5.2) and return the same values on failure, these functions do not set the `errno` variable on failure. The additional functions provided by RSIM are handled through calls to simulator internal functions, setting the `%o0` register value to the result of the trap.

RSIM also uses exceptions to implement certain instructions that either modify system-wide status registers (e.g. `LDFSR`, `STFSR`), are outdated instructions with data-paths too complex for a processor with the aggressive features simulated in RSIM (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 [23]). All of these instructions types are marked with `SERIALIZE` traps, and are handled in `ProcessSerializedInstructions`. In the case of `STFSR` and `STXFSR`, control will be transfered to instructions in the trap-table, while `DONE` and `RETRY` transfer control back to the `trappc` and `trapnpc` fields saved aside before entering the trap-table. Other serialized instructions continue with normal execution starting from the instruction after the serialized one.

Window traps dispatch control to the trap table through the `TrapTableHandle` function. This function puts the processor into privileged state and saves aside the PC and NPC of the faulting instruction as `trappc` and `trapnpc`. The processor PC and NPC are then set to the appropriate instruction sequences for the window traps, and the processor restarts execution from within those trap-handlers.

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

10.9 Principal data structures

Source files: `src/Processor/active.cc`, `src/Processor/freelist.cc`, `src/Processor/tagcvt.cc`, `src/Processor/stallq.cc`, `src/Processor/branchqelt.cc`, `src/Processor/branchresolve.cc`, `src/Processor/instheap.c`

Header files: `incl/Processor/active.h`, `incl/Processor/freelist.h`, `incl/Processor/tagcvt.h`, `incl/Processor/stallq.h`, `incl/Processor/circq.h`, `incl/Processor/branchq.h`, `incl/Processor/heap.h`, `incl/Processor/instheap.h`, `incl/Processor/hash.h`, `incl/Processor/memq.h`, `incl/Processor/FastNews.h`

The majority of the data structures used in the out-of-order execution engine are associated with the processor's `state` structure.

The first type of data structures are concerned with instruction fetching, decoding, and graduation. These structures include the register `freelist` class, the `activelist` class, the tag converted (`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 deals 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 `instances` 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 `incl/Processor/FastNews.h`.

Chapter 11

Processor Memory Unit

The processor memory unit includes nearly as much complexity as the rest of the processor, which was discussed in Chapter 10. The functions provided include adding new memory instructions to the memory unit, generating 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 3.2.3: constraints for precise exceptions, constraints for uniprocessor data dependences, and constraints for multprocessor 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.

11.1 Adding new instructions to the memory unit

Source files: `src/Processor/memunit.cc`

Header files: `incl/Processor/memory.h`

The function `AddToMemorySystem` is called to add new instructions to the memory unit. This function first initializes some fields of the `instance`. If this instruction is a store, it is added to the list of stores which have not yet had their addresses generated (called `ambig_st_tags`). For SC and PC, this instruction is added to the unified memory unit queue, `MemQueue`. For RC, this instruction is inserted into either the `LoadQueue` or `StoreQueue` as appropriate. The `LoadQueue` and `StoreQueue` are currently used only as a simulator abstraction – the memory unit is thought of as a unified whole, rather than two split parts.

If this `instance` does not have any outstanding address dependences (i.e. `addrdep` is clear, as discussed in Section 10), it is sent on to the address generation unit by calling `CalculateAddress`.

11.2 Address generation

Source files: `src/Processor/memunit.cc`, `src/Processor/pipestages.cc`, `src/Processor/exec.cc`

Header files: `incl/Processor/memory.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 `addr` and `finish_addr` fields of the `instance` are filled in using the `GetAddr` function. Additionally, the `instance` will be marked with a bus error (misalignment

[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 processor pipeline.

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, 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 Section 10.

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 4). 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 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.

11.3 Issuing instructions to the memory hierarchy

Source files: src/Processor/memunit.cc, src/Processor/memprocess.cc, src/MemSys/cpu.c

Header files: incl/Processor/memory.h, incl/Processor/hash.h, incl/Processor/memprocess.h, incl/MemSys/cpu.h

Every cycle, the simulator calls the IssueMem function. In the case of RC, this function first checks if any outstanding memory fences (MEMBAR instructions) can be broken down – this occurs when every instruction in the class of operations that the fence has been waiting upon has completed. If the processor has support for a consistency implementation with speculative load execution (chosen with "-K"), all completed speculative loads beyond the voided fence that are no longer blocked for consistency or disambiguation constraints are allowed to leave the memory unit through the PerformMemOp function.

The IssueMem function then seeks to allow the issue of actual loads and stores in the memory system. If the system implements SC or PC, the IssueMems function is called. With RC, IssueStores is called first, followed by IssueLoads. We issue instructions in this order with RC not to favor stores, but rather to favor older instructions (As discussed in Section 10, no store can be marked ready to issue until it is one of the oldest instructions in the active list and all previous instructions have completed).

The functions IssueStores and IssueLoads, or IssueMems for SC and PC systems, scan the appropriate part of the memory unit for instructions that can be issued this cycle. At a bare minimum, the instruction must have passed through address generation and there must be a cache port available for the instruction. The following description focuses on the additional requirements for issuing each type of instruction under each memory consistency model. Steps 1a-1e 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: Stores in sequential consistency or processor consistency

[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.

If the instruction under consideration is a store in SC or PC, it must be the oldest instruction in the memory unit and must have been marked ready in the graduate stage (as described in Section 10.7) before it can issue to the 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 1b: Stores in release consistency

Stores in RC issue after being marked ready, if there are no current ordering constraints imposed by memory fences. If any such constraints are present and if the system has hardware prefetching, the system can mark the store for a possible hardware prefetch. A store can be removed from the memory unit as soon as it issues to the cache, rather than waiting for its completion in the memory hierarchy (as in sequential consistency and processor consistency). When a store is issued to the caches, the processor's `StoresToMem` field is incremented. However, as we do not currently simulate data in the caches, stores remain in what we call a *virtual store buffer*. The virtual store buffer is part of the `StoreQueue` data structure and has a size equivalent to the processor's `StoresToMem` field. These elements are not counted in the memory unit size, but may be used for obtaining values for later loads.

Step 1c: Loads in sequential consistency

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.

Step 1d: Loads in processor consistency

Loads issue in PC under circumstances similar to those of SC. 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 1e: Loads in release consistency

In RC, loads can issue non-speculatively whenever they are not prevented by previous memory barriers[3]. As in SC and PC, a load that is preceded by store operations must check previous stores for possible forwards or stalls. However, in RC, such checks must also take place against the virtual store buffer. As the virtual store buffer is primarily a simulator abstraction, forwards from this buffer are used only to learn the final value of the load; the load itself must issue to the cache as before. However, loads must currently stall in cases of partial overlaps with instructions in the virtual store buffer. This constraint is not expected to hinder performance in applications where most data is either reused (thus keeping data in cache and giving partial overlaps short latencies) or where most pointers are strongly typed (making partial overlaps unlikely). However, if applications do not meet these constraints, it may be more desirable to simulate the actual data in the caches. As in the other models, loads hindered by memory consistency model constraints can be marked for prefetching or speculatively issued if the consistency implementation supports such accesses. Although speculative loads and prefetching are allowed around ordinary `MEMBAR` instructions, such optimizations are not allowed in the case of fences with the `MemIssue` field set.

Step 2: Issuing an instruction 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

[3]Because of the single-precision floating point problems discussed in Chapter 10, single-precision loads do not issue until their output dependences are resolved. With static scheduling, subsequent loads will also be prevented from issuing.

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 access is private or 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 either the processor `PageTable` or the shared-memory `SharedPageTable` 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. either a private access or 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`.

`StartUpMemRef` and the other functions in `src/Processor/memprocess.cc` are responsible for interfacing between the processor memory unit and the memory hierarchy itself. `StartUpMemRef` translates the format specified in the `instance` data structure to a format understood by the cache and memory simulator. This function then calls the function `addrinsert` to begin the simulation of an access.

`addrinsert` starts by initializing a memory system request data structure for this memory access. (This data structure type is described in Section 12.2.) 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.

11.4 Completing memory instructions in the memory hierarchy

Source files: `src/Processor/memprocess.cc`, `src/Processor/memunit.cc`, `src/Processor/funcs.cc`

Header files: `incl/Processor/memory.h`, `incl/Processor/memprocess.h`

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 `src/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.

The function `CompleteMemQueue` processes instructions from the `MemDoneHeap` of the processor by calling `CompleteMemOp` for each instruction to complete in a given cycle. The corresponding instruction emulation function is called for accesses that were not simulated at the caches. 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 re-issue, 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` has two functions: removing instructions from the memory unit and passing values down from `limbo` loads. In the case of RC, `PerformMemOp` always removes the operation from either the memory unit or virtual store buffer (as appropriate) except in the case of loads that are either marked with a `limbo` field or past a `MEMBAR` that blocks loads. In 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 or, in the case of RC, when prior memory fences have been cleared.

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 the lowest level of local cache (L2 in our case) must be sent to the 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 oustanding, 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 [5, 28].

Chapter 12

Memory Hierarchy and Interconnection System Fundamentals

This chapter describes fundamental aspects of the memory hierarchy, bus, and multiprocessor interconnection network implementation. We will refer to the implementation of these subsystems collectively as the *memory system simulator*. Subsequent chapters provide detailed investigations of each part of the memory system simulator.

Section 12.1 describes the various memory system simulator modules and their interconnection. Section 12.2 describes the essentials of the message data structure used to convey information within the memory system simulator (much as the `instance` data structure is used in the processor simulator). Section 12.3 describes the steps to construct the multiprocessor portion of the simulated architecture. Section 12.4 explains the steps taken throughout the memory system simulator to avoid deadlock.

12.1 Fundamentals of memory system modules

Source files: `src/MemSys/architecture.c`, `src/MemSys/module.c`, `src/MemSys/route.c`

Header files: `incl/MemSys/module.h`, `incl/MemSys/net.h`

The `SMMODULE` data structure contains a basic module framework that is used by many of the modules in the memory system simulator. This framework includes fields common to all the module types, and is initialized using the `ModuleInit` function. This function sets the node number for the module, along with fields related to the module's input and output ports. These ports, which are of the type `SMPORT`, are used to connect between the various modules in the memory system simulator. `SMPORT` data structures act as queues between the various modules and have memory system simulator messages as their entries. (These messages are described in Section 12.2.) Each queue has a fixed maximum size, initialized by the call to `ModuleInit` (but possibly modified using `QueueSizeCorrect` later). Note that only output ports are actually created; the input ports of one module are set to the same data structures as the output ports of another module using the `ModuleConnect` function described in Section 12.3.

Each module also has a *routing* function, which generates the output port number for each type of message that leaves the module. The various modules and default port connections used in RSIM are shown in Figure 12.1. Input ports are shown on this figure as `i`*X*, where *X* is the port number; similarly, output ports are shown as `o`*X*. The terms *Request*, *Reply*, *Cohe*, and *Cohe reply* correspond to the types of memory system messages, as explained in Section 12.2. If no write-buffer is included, port `o1` of the L1 cache connects directly to port `i0` of the L2 cache, while port `o0` of the L2 cache connects to port `i1` of the L1 cache.

RSIM uses a single module for each bank of the directory and memory. The input and output ports connecting the directory and memory to the bus are listed in the form `o6,8,...` to indicate that each interleaved bank has a separate port in place of the single port shown on the diagram.

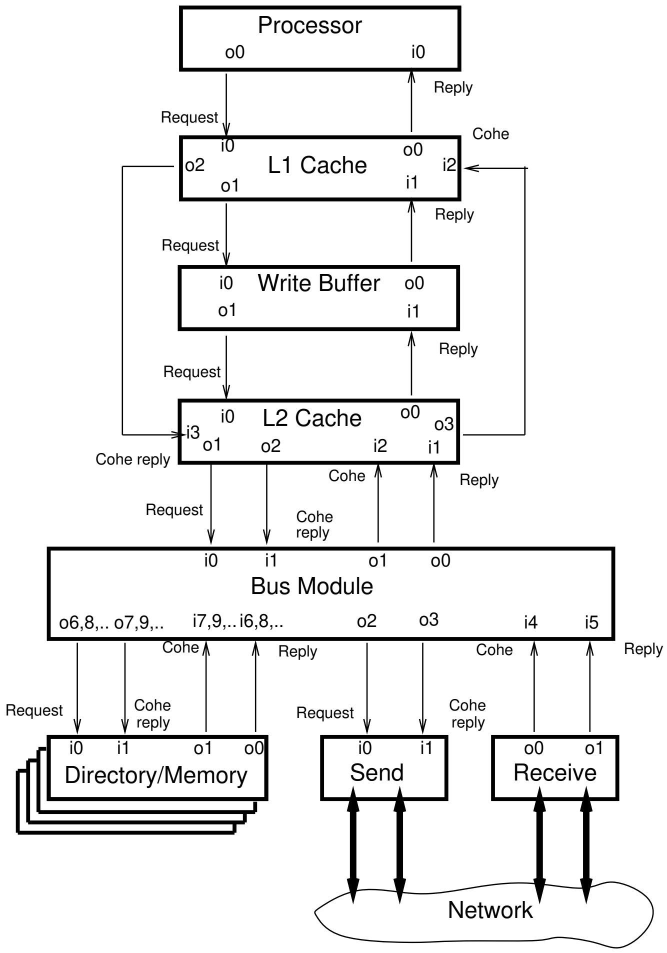


Figure 12.1: Modules and port connections in RSIM

The network interface is shown on the diagram as being split into two parts, *Send* and *Receive*. *Send* moves new messages from the node to the multiprocessor interconnection network, while *Receive* brings messages from the interconnection network into the node. Note that the network system and its connections to the network interface do not use the standard `SMMODULE` and `SMPORT` data structures used by the rest of the memory system simulator. The data structures for the network and its connections to the network interface are explained in Section 15.3.

The queue sizes for the various ports are configurable at run-time, as discussed in Chapter 4. The size of the queue specifies the number of transactions of any type (request or response) that can be held in the connection between the modules at any given time. If the size of the queue configured at runtime differs from that originally initialized with `ModuleInit`, the `QueueSizeCorrect` function is called to set the sizes of these ports to the desired values. Note that the port queue size also limits the number of requests that can be processed by the module each time the module is activated. For example, if a cache is intended to start processing four requests each cycle, the request port queue should contain at least four entries.

The port queue can also contain more than the minimum number of entries; in these cases, the queue acts as a buffer to decouple a faster module from a slower module. For this reason, the default port sizes from the L2 cache to the bus are larger than most of the other cache ports; these port sizes are chosen so that the potentially slow processing rate of the bus will not cause the L2 cache itself to stall.

The implementation of port queues is such that each port queue holds one of its entries as an “overflow entry.” Thus, most of the modules subtract 1 from the port queue sizes specified before calling `ModuleInit` or `QueueSizeCorrect`, as this accounts for the separate overflow entry provided. The network interface module does not use the overflow request; these units stop adding to their output ports before allowing the queues to fill up the overflow request.

12.2 Memory system message data structure

Header files: `incl/MemSys/req.h`

The fundamental unit of information exchange among the RSIM memory system simulator modules is referred to in this manual as a memory system message[1]. This data structure conveys essential information about the access being simulated, just as the `instance` structure acts as the basic unit of information exchange among the processor pipeline stages.

The memory system message specifies an action that must be performed on a certain portion of the memory hierarchy, as described below. The five most important fields of the message data structure are the `tag` field, the `s.type` field, the `req_type` field, the `s.reply` field, and the `s.nack_st` field. The `tag` field specifies the cache line to which the action in question applies. This field is used with the same meaning in all varieties of messages. The following sections describe how each of the other fields is used to distinguish the types of messages in the memory system simulator.

12.2.1 The `s.type` field

Memory system messages come in four basic varieties, as conveyed by the `s.type` field:

REQUEST 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 directory in response to the demands of a `REQUEST`; may include a data transfer.

COHE Sent by a directory 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.

Each of these four basic categories is further divided according to the `req_type` field and, in some cases, the `s.reply` and `s.nack_st` fields. These fields are described in the following sections.

12.2.2 The `req_type` field

The `req_type` field can take on several values, some of which are currently reserved for future expansion. The types that are actually supported in RSIM are split into the following categories:

1. Those seen only at the processors and caches
2. System transaction requests
3. Replies
4. Coherence actions
5. Replacement messages

[1] In the code, the data structure allocated for such a message is called the `REQ` data structure. This data structure is used for requests and reply messages, for both data and coherence transactions. We avoid using the term `REQ` in this manual to avoid any possible confusion with the `REQUEST` message type.

The names and functions of some of these field values resemble those used in the UltraSPARC-II [25].

`req_type` *values used only at the processors and caches:*

The names of the `req_type` values used only at the processors and caches are largely self-explanatory: `READ`, `WRITE`, `RMW`, `L1WRITE_PREFETCH`, `L1READ_PREFETCH`, `L2WRITE_PREFETCH`, and `L2READ_PREFETCH`. (The prefetch requests represent the type of prefetch to issue and the level of cache to which to bring the line).

`req_type` *values that specify system transaction requests:*

System transaction requests are `REQUEST`s seen beyond the caches (i.e. at the busses and directory). These request have the following `req_type` values and demand the stated actions:

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

`req_type` *values that indicate replies:*

Each of the request types discussed above receives a `REPLY` from the module at which it is serviced. The following values of `req_type` indicate such 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 `READ_SH`[2], `READ_OWN`, or `UPGRADE`[3]

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

REPLY_EXCLDY Brings a line to cache in modified (exclusive dirty) state. Valid response to `READ_OWN`, or `UPGRADE`[4]

`req_type` *values that specify coherence actions:*

In response to one of the system transaction requests described above and based on the current sharing status of the line, the directory may send coherence messages with any of the following `req_type` values to bring the line to an acceptable state: (These messages may have an `s.type` of either `COHE` or `COHE_REPLY`.)

COPYBACK Transitions a line from exclusive or modified state to shared state. Invoked for `READ_SH` if held elsewhere with ownership. Involves a cache-to-cache transfer to the requester and either an acknowledgment (if exclusive state) or a copyback (if modified) to the directory.

COPYBACK_INVL Transitions a line from exclusive or modified state to invalid state. Invoked for `READ_OWN` if held elsewhere with ownership. Involves a cache-to-cache transfer to the requester and an acknowledgment to the directory.

INVL Transitions a line from shared to invalid state. Invoked for `READ_OWN` or `UPGRADE` if held elsewhere in shared state. Involves only acknowledgment to directory.

[2]If no other processors sharing

[3]In certain race conditions, `UPGRADE` is converted to `READ_OWN`, as described in Section 14.

[4]In certain race conditions, `UPGRADE` is converted to `READ_OWN`, as described in Section 14.

`req_type` *values for replacement messages:*

Replacement messages are sent when a line in exclusive or modified state is evicted from the cache due to an incoming reply. These messages are sent with an `s.type` of `COHE_REPLY` and with one of the following `req_type` values:

WRB Indicates a replacement from the modified state, and sends the updated line to the directory and memory module

REPL Indicates a replacement from the exclusive state, and informs the directory of the replacement

The RSIM caches do not inform the directory of replacements from shared state.

12.2.3 The `s.reply` field

`REPLY`s and `COHE_REPLY`s also use a field called `s.reply` to indicate the response type. This field can have the following values:

REPLY Successful completion of the desired action.

RAR Request a retry: action could not be completed immediately due to transient condition, and demand should be retried. These are sent by a directory when its pending request buffer fills up or when certain negative-acknowledgment responses cannot be immediately resolved (as explained in Chapter 14).

NACK Action could not be completed as specified (`COHE_REPLY`s only). These are sent on invalidation messages when the line is not present in the cache, or on write-back subset-enforcement messages when the L1 cache does not have the line in modified state.

NACK_PEND Action could not be completed as specified due to a transient condition (`COHE_REPLY`s only). The cache sends these responses during certain races described in Chapter 13. The directory handles these replies by either retrying the corresponding `COHE` or reprocessing the original `REQUEST`.

12.2.4 The `s.nack_st` field

`COHE`s additionally have a field called `s.nack_st`, which indicates whether or not a `NACK` is an acceptable response to the coherence action. Specifically, a `NACK` is not acceptable if the coherence message demands a data transfer, as this indicates that the directory currently considers the cache to be the exclusive owner of the line. Thus, a value of `NACK_NOK` is specified for `COPYBACK` and `COPYBACK_INVL` messages, while `NACK_OK` distinguishes `INVL` messages.

12.3 Memory system simulator initialization

Source files: `src/MemSys/architecture.c`, `src/MemSys/mesh.c`, `src/MemSys/net.c`, `src/MemSys/module.c`, `src/MemSys/setup_cohe.c`

Header files: `incl/MemSys/arch.h`, `incl/MemSys/net.h`

As mentioned in Chapter 9, RSIM uses the `SystemInit` function to set the desired characteristics of the multiprocessor architecture being simulated. After setting some global fields according to the modules present in the system, the `SystemInit` function calls `dir_net_init`.

First, `dir_net_init` constructs the multiprocessor interconnection network. This is a 2-dimensional mesh network with a square number of nodes (if a non-square system is desired, the user should either round up to the nearest square or some other appropriate square configuration size). `dir_net_init` calls `CreateMESH` to construct the request and reply networks. This function initializes all the buffers and ports of the network (described in Section 15.3), calling the functions `MeshCreate`, `Create1DMesh`, `ConnectMeshes`, `ConnectComponents`, and `NetworkConnect` to set up the system. After creating these meshes, `dir_net_init` next sets the mesh parameters of flit delay and arbitration delay: if pipelined switches are being modeled, these parameters are modified from the input parameters in order to simulate the desired degree of pipelining. The network used in RSIM is taken from the NETSIM interconnection network simulator [7]. More details on the network interconnection and parameters are available in Section 15.3 and in the NETSIM reference manual (available from `http://www-ece.rice.edu/~rsim/rppt.html`).

Next, `dir_net_init` creates the `Delays` data structures for many of the individual modules. Each module has parameters for access time, initial transfer time, and flit transfer time. Additionally, the directory module has parameters for the first packet creation time associated with a request and each subsequent packet creation time, which are used to effect delays when sending coherence messages from a directory. The access time and packet creation times of the `DirDelays` structures are configurable at run-time. These `Delays` are not currently used for the CPU, L1 cache, L2 cache, and write-buffer, but are provided for possible expansion. The actual latencies for the L1 and L2 caches are stored in the variables `L1TAG_DELAY`, `L2TAG_DELAY`, and `L2DATA_DELAY`.

Next, `dir_net_init` allocates all the needed modules for the individual nodes (e.g. caches, write buffers, network interfaces, busses, and directory banks). The system then initializes each of these modules, using `NewProcessor`, `NewCache`, `NewWBuffer`, `NewSmnetSend` (creates new network send interface), `NewSmnetRcv` (creates new network receive interface), `NewBus`, and `NewDir` to set the parameters for each module type. As caches are created, the `setup_tables` function is called to configure the coherence behavior of the system. Within the memory system simulator, the MESI states are referred to as `PR_DY`, `PR_CL`, `SH_CL`, and `INVALID`, respectively.

The `ModuleConnect` functions are then used to connect the various modules through their appropriate `SMPORT` data structures. The invocation `ModuleConnect(src,dest,a,b,width)` creates a bidirectional connection between the modules: output port `a` of module `src` is connected to input port `b` of module `dest`, and input port `a` of module `src` is connected to output port `b` of module `dest`. The output ports must have already been constructed by the `ModuleInit` function described in Section 12.1; `ModuleConnect` simply sets the input ports of one module to point to the same data structures as the output ports of the other module. The interconnection between these modules has a width of `width` bytes. (Currently, the width parameter for all module connections is a compile-time parameter, `INTERCONNECTION_WIDTH_BYTES`.)

As described in Section 12.1, the `ModuleInit` function initializes all output ports of a given module with queues of the same size. However, the user may override these through the runtime parameters specified in Section 4.2.7. These changes in queue lengths are incorporated using the `QueueSizeCorrect` function.

12.4 Deadlock avoidance

RSIM uses limited buffers in order to accurately simulate contention for resources at the various modules, raising the possibility of deadlock in the system. To avoid deadlock, requests and replies are kept physically separate. Conceptually, requests are messages that allocate resources (such as MSHRs, pending coherence bits, or directory buffer entries). Replies conceptually release resources, and must be guaranteed to be accepted in a finite amount of time, even if they lead to new messages. This is achieved by allocating a resource for the reply at the time the request is sent out. Thus, the reply is guaranteed to have an associated resource waiting for it when it arrives. The incoming reply can be accepted by the module immediately and held in that resource until the completion of any additional processing it may require.

In our system, for deadlock avoidance purposes, requests include `REQUEST` and `COHE` messages, while replies include `REPLY` and `COHE_REPLY` (including write-back and replacement) messages. Although write-backs and replacement messages are unsolicited and do not already have resources reserved for them in the directory and memory module, they can be considered replies because they do not require additional resources or send out additional messages. (In our system, write-backs do not receive acknowledgments from the memory controller. If such acknowledgments are required, write-backs should be sent as requests with a resource held at the cache until the acknowledgment arrives.)

Each subsystem in the RSIM memory system simulator takes specific precautions to prevent the creation of new deadlocks. These steps are described separately with each subsystem.

Chapter 13

Cache Hierarchy

RSIM simulates two levels of data cache. 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 multiported. 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 `src/MemSys/pipeline.c`

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`, `L2CacheInSim`, `L1CacheOutSim`, and `L2CacheOutSim` for each cache, as mentioned in Section 10.1. Each of these functions, as well as the functions called by those functions, are described in this chapter.

13.1 Bringing in messages

Source files: `src/MemSys/l1cache.c`, `src/MemSys/l2cache.c`, `src/MemSys/pipeline.c`

Header files: `incl/MemSys/cache.h`, `incl/MemSys/pipeline.h`

Two functions are used to bring new messages into each level of cache. These functions are `L1CacheInSim` and `L2CacheInSim`. Each of these functions checks incoming messages from the ports of the module. The function then attempts to insert each incoming message into the appropriate tag-array pipeline, according to its `s.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.

At the L2 cache, if the incoming message is a `REPLY`, then some bookkeeping is done for managing the write-back buffer. Specifically, before a `REQUEST` can be sent out from the L2 cache, it must have allocated a write-back buffer entry for a potential replacement caused by its `REPLY`. When the `REPLY` returns to the cache, the `wrb_buf_used` field of the cache is incremented to indicate that the new `REPLY` may need to send out a replacement.

13.2 Processing the cache pipelines

Source files: `src/MemSys/l1cache.c`, `src/MemSys/l2cache.c`, `src/MemSys/pipeline.c`, `src/MemSys/cachehelp.c`

Header files: `incl/MemSys/cache.h`, `incl/MemSys/pipeline.h`

For each cycle in which there are accesses in the cache pipelines, the functions `L1CacheOutSim` and `L2CacheOutSim` are called.

These functions start out by checking what the system calls its *smart MSHR list*. The smart MSHR list is an abstraction used for simulator efficiency. In a real system, this list would correspond to state held at the cache resources (MSHRs or write-back buffer entries). Entries in the smart MSHR list correspond to messages being held in one of the above resources, waiting to be sent on one of the cache output ports. Messages can be held in their previously-allocated cache resources in order to prevent deadlock, as the cache must always accept replies in a finite amount of time. If there are any such messages held in their resources, the cache attempts to send one to its output port. If the cache successfully sends the message, the corresponding resource may be freed in some cases.

After attempting to process the smart MSHR list, the cache considers the current state of its 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: namely, `L1ProcessTagReq`, `L2ProcessTagReq`, or `L2ProcessDataReq`. If the corresponding function returns successfully, the element is removed from the pipe. After elements have been processed from the head of their pipelines, the cache advances the elements by calling `CyclePipe`. The following sections describe functions `L1ProcessTagReq`, `L2ProcessTagReq`, and `L2ProcessDataReq` in detail.

13.3 Processing L1 cache actions

Source files: `src/MemSys/l1cache.c`, `src/MemSys/mshr.c`, `src/MemSys/cachehelp.c`, `src/MemSys/setup_cohe.c`

Header files: `incl/MemSys/cache.h`, `incl/MemSys/mshr.h`

The function `L1ProcessTagReq` processes messages that have reached the head of the L1 cache pipeline. Its behavior depends on the type of message (i.e. `s.type`). The following sections describe the manner in which each type of message is processed.

13.3.1 Handling `REQUEST` type

For `REQUEST`s, `L1ProcessTagReq` first checks the tag in the tag array and in the outstanding MSHRs by calling `notpres_mshr`.

Step 1: Calling `notpres_mshr`

For a `REQUEST`, `notpres_mshr` first checks to see if the request has a tag that matches any of the outstanding MSHRs. Additionally, the `notpres` function is called to determine if the desired line is available in the cache. If the line is not present in any MSHR, the coherence routine for the cache (`cohe_pr` for L1, `cohe_sl` for L2) is called to determine the appropriate actions for this line, based on whether or not the line is present, the current MESI state of the line in cache, and the type of cache.

Step 1a: Behavior of `notpres_mshr` when `REQUEST` does not match MSHR

If the `REQUEST` being processed does not match an outstanding MSHR, the operation of `notpres_mshr` depends on whether or not the line hits in the cache and the state of the line.

If the line being accessed hits in the cache in an acceptable state, this request will not require a request to a lower module. As a result, the cache will return `NOMSHR`, indicating that no MSHR was involved or needed in this request[1].

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 was required, but that the request must be sent forward.

[1]In the L2 cache, this will return `NOMSHR_STALL_COHE` if there is currently a `COHE` request pending on the line. This indicates that no MSHR has been consumed, but this `REQUEST` must wait for the pending `COHE` first.

If the request needs a new MSHR, but none are available, the value `NOMSHR_STALL` is returned. In the case of the L2 cache, a request that is not able to reserve a space in the write-back buffer leads to a `NOMSHR_STALL_WRBBUF_FULL` return value.

Otherwise, the cache books an MSHR and returns a response based on whether this access was a complete miss (`MSHR_NEW`) or an upgrade request (`MSHR_FWD`). In the case of upgrades, the line is locked into cache by setting `mshr_out`; this guarantees that the line is not victimized on a later `REPLY` before the upgrade reply returns. In all cases where the line is present in cache, the `hit_update` function is called to update the ages of the lines in the set (for LRU replacement).

Step 1b: Behavior of `notpres_mshr` when `REQUEST` matches MSHR

On the other hand, if the `REQUEST` matches a current MSHR, the operation of `notpres_mshr` depends on the type of the `REQUEST` and the previous accesses to the matching MSHR.

If the access is a shared prefetch or an exclusive prefetch to 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_IN_PROGRESS` to indicate that the `REQUEST` should be dropped.

If the access is an L2 prefetch or an exclusive 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 was an exclusive prefetch, it is converted to an L2 exclusive prefetch before being forwarded around the cache.

In certain cases, the `REQUEST` may need to be stalled. Possible scenarios that can result in stalls and the values they return are as follows. If the MSHR is being temporarily held for a write-back, `notpres_mshr` returns `MSHR_STALL_WRB`. If the MSHR is marked with an unacceptable pending coherence message, the function returns `MSHR_STALL_COHE`. If the MSHR already has the maximum number of coalesced `REQUEST`s 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*[2]. 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 [15, 16].

If the access was 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 `notpres_mshr`

`L1ProcessTagReq` continues processing the `REQUEST` based on the return value of `notpres_mshr`.

For a hit (`NOMSHR`), the request is marked as a hit and returned to the processor through the `GlobalPerformAndHeapInsertAllCoalesced` function.

For new misses (`MSHR_NEW`), upgrades (`MSHR_FWD`), or write-throughs (`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_IN_PROGRESS`, the request is dropped.

For each of the stall 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.

13.3.2 Handling `REPLY` type

Step 1: Checking status of MSHR entry

[2] WAR stalls are not usually seen with straightforward implementations of consistency models, as stores following loads to the same line generally depend on the values of those loads. Consequently, such stores cannot issue in straightforward implementations until the loads complete.

For a REPLY, the function L1ProcessTagReq uses the FindInMshrEntries function to find the MSHR number corresponding to the REPLY.

If there is no MSHR entry for the REPLY, then it must be a response to a non-allocated write or to an L2 prefetch REQUEST. For such a REPLY, GlobalPerformAndHeapInsertAllCoalesced should be immediately called. This function calls the GlobalPerform function for the corresponding REQUEST and any other REQUESTs that may have coalesced with this REQUEST at the write-buffer or L2 cache, while also inserting the REPLYs into the MemDoneHeap described in Section 11.

Step 2: Processing REPLYs that match an MSHR

If the REPLY matches an MSHR, the function GetCoheReq is used to determine if any coherence message coalesced into the MSHR while it was outstanding. If so, any possible effects of this coherence message will be processed along with the REPLY.

Step 2a: Processing upgrade REPLYs

For upgrades, the REPLY handler calls the cache's coherence routine to determine the final state of the cache line. If any writes are present in the MSHR for the cache line and the state of the line is exclusive, the state changes to modified. If a COHE was merged into the MSHR, its effect on the cache line state is now carried out. (Note: no coherence message that requires a copyback or cache-to-cache transfer will ever merge into an MSHR, as discussed in Section 13.3.3.)

Step 2b: Processing cache miss REPLYs

For cache misses (REPLYs other than upgrades), the REPLY handler calls either premiss_ageupdate or miss_ageupdate based on whether or not the line is a "present miss" (a line whose tag remains in cache after a COHE, but is in an INVALID state).

If the line is not a "present miss", the miss_ageupdate function tries to find a possible 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[3].

If no victim is available because all lines in the set have upgrades pending, the REPLY is not taken and a message needs to be sent back to the sender. This is done in the NackUpgradeConflictReply function. This function uses the MSHR allocated reserved by the REQUEST corresponding to this rejected REPLY as a resource from which to issue the new message. This resource is added to the smart MSHR list simulator abstraction (discussed in Section 13.2).

For REPLYs that do find a space in which to insert the new line, the cache's coherence routine is called to determine the state for the new line being brought in. If any writes are present in the MSHR for the cache line and the state of the line is exclusive, the state changes to modified. If a COHE was merged into the MSHR, its effect on the cache line state will also be carried out.

If this line replaces a current line, the cache's coherence routine is called to determine any possible requests that must be sent as a result. If the variable blw_req_type is set, a writeback must be sent as a result of this replacement (occurs if the victim had been in modified state). In this case, the GetReplReq 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 Step 3.

Step 3: Returning replies to processor

Regardless of REPLY type processed, the system now prepares to remove the corresponding entry from the MSHRs.

If some non-MSHR accesses (L2 prefetches, writes with a write-through cache) are also coalesced with the reply, the function GlobalPerformAndHeapInsertAllCoalescedWritesOnly or GlobalPerformAndHeapInsertAllCoalescedL2PrefsOnly is called to provide replies all of them to the processor.

[3]Recall that the simulator code refers to the MESI states as PR_DY, PR_CL, SH_CL, and INVALID, respectively.

Then, `MSHRIterateUncoalesce` is called, passing through all the responses coalesced into the `REPLY` and the MSHR itself and informing the processor of these.

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. As with other resources held to issue messages to the ports, the MSHR is added to the smart MSHR list.

`L1ProcessTagReq` does not currently account for cache fill time.

13.3.3 Handling `COHE` type

The types of incoming `COHE` transactions understood by the L1 cache are `COPYBACK`, `COPYBACK_INVL`, `INVL`, and `WRB` (as indicated by the `req_type` field of the message). The first three are described as coherence transactions in Section 12.2. The latter is used to enforce inclusion on the L2 replacement of an exclusive line. (Note that the L1 cache cannot receive a `COHE_REPLY` from any module.)

Step 1: Determining if `COHE` hits in cache or matches an MSHR

If the message being handled is an incoming `COHE` transaction, `L1ProcessTagReq` starts by calling `notpres` and `notpres_mshr` to determine the status of the line in the cache. For `COHE` messages, `notpres_mshr` starts out by checking to see if the line is being held in any of the outstanding MSHRs.

Step 1a: Behavior of `notpres_mshr` when `COHE` does not match MSHR

If the incoming coherence message does not match any MSHR, the function immediately returns `NOMSHR` to indicate that no MSHR was involved in the transaction.

Step 1b: Behavior of `notpres_mshr` when `COHE` matches MSHR

If a `COHE` matches an MSHR, the response of `notpres_mshr` depends upon the type of MSHR and the type of coherence transaction.

Step 1b/i: Matching MSHR is for a read miss

If the MSHR is for a read miss, the reply may come back in either shared or exclusive state. Thus, the type of `COHE` determines whether or not this message can be processed.

If the coherence message demands a copyback (`COPYBACK` or `COPYBACK_INVL`), the directory still considered this cache to be the owner of the line at the time of the message. Thus, the cache either had previously been the owner or is going to be the owner because of an exclusive `REPLY`. Because of the latter possibility, this response must be a `NACK_PEND` so that the sender can either retry the coherence message, reprocess the original `REQUEST` that caused the coherence message, or simply drop the `COHE` (in the case of `WRB` subset enforcement messages where the L2 cache request is already stalling at an MSHR). `notpres_mshr` returns a response of `MSHR_COAL`.

If the coherence message does not require a copyback (`INVL` or `WRB`), the invalidation message can be merged with the MSHR for later processing. In the case of an `INVL`, the message can receive a positive acknowledgment of `REPLY` (indicating that the `COHE` will be acted upon). If the incoming `COHE` is a `WRB`, the response is sent as a negative acknowledgment to the L2 cache (indicating to the L2 cache that no data is being provided by the L1). In the `INVL` case, `notpres_mshr` returns `MSHR_COAL`, while the `WRB` case sees a return value of `NOMSHR`.

Step 1b/ii: Matching MSHR is for a write miss or upgrade

If the MSHR is for a write miss or upgrade, however, the message must be handled differently.

If the request demands a data copyback from a write-back cache, the request must be sent back with a `NACK_PEND`. If the request is a type that seeks a data copyback, but the cache is write-through, the request is handled by acknowledging the message and handling it later. In either case, this function returns `MSHR_COAL`.

On the other hand, if the request does not demand a data copyback, it must have originated before the private request was serviced. Thus, the coherence action can be done immediately and returned with a `NACK`.

If the line was being upgraded, it can still be unlocked and invalidated here: the directory must convert the processor's UPGRADE request to a READ_OWN when it realizes that the processor doesn't actually have the line it wants. notpres_mshr returns MSHR_FWD.

Step 2: Processing based on the results of notpres_mshr

If notpres_mshr returns NOMSHR, the response depends on whether or not the line is in cache. If the line is a cache miss, the COHE is NACKed. If the line is a cache hit and the COHE message is of type WRB, the message is NACKed if the line is held in a state other than PR_DY and positively acknowledged if the line is in PR_DY. Either way, the line is invalidated. If the access is a cache hit and not a WRB, the cache line state is changed according to the results of the coherence function and the message is positively acknowledged. In each of the NOMSHR cases, SpecLoadBufCohe is called if the coherence type indicates an invalidation, even if the line is not present in the L1 cache (as such a message may indicate an L2 invalidation or replacement).

If notpres_mshr returns MSHR_COAL, L1ProcessTagReq first sends the coherence message to SpecLoadBufCohe if the system implements speculative load execution and the message conveys an invalidation. If the cache is write-through or the COHE does not demand a copyback, it is positively acknowledged. Otherwise, the COHE will receive a NACK_PEND response.

If notpres_mshr returns MSHR_FWD, the COHE receives a response of either NACK_PEND (for WRB requests) or NACK (for other message types). If the request indicates an invalidation and the system implements speculative load execution, L1ProcessTagReq sends the coherence message to SpecLoadBufCohe. Additionally, if the line is present in the cache, this case removes the mshr_out field from the line and invalidates it, thus removing the line from the cache even in a case formerly considered an upgrade.

In all of the above cases for COHE messages, L1ProcessTagReq returns success if the COHE_REPLY can be sent down immediately. If not, the cache will try to resend the COHE_REPLY until it is accepted by its port.

13.4 Processing L2 tag array accesses

Source files: src/MemSys/l2cache.c, src/MemSys/mshr.c, src/MemSys/cachehelp.c, src/MemSys/setup_cohe.c

Header files: incl/MemSys/cache.h, incl/MemSys/mshr.h

The function L2ProcessTagReq is called for accesses that have reached the head of an L2 tag array pipeline. L2ProcessTagReq is largely similar to L1ProcessTagReq but has some key differences, described below.

Difference 1: Presence of a data array

The first difference between L1ProcessTagReq and L2ProcessTagReq deals with the data array. REQUESTs that hit, the data response for REPLYs, as well as the copyback portions of COHE messages and write-backs (whether replacements at the L2 or unsolicited fills from the L1) all require data array accesses.

Difference 2: Servicing COHE messages

For COHE messages, the L2 cache marks the line in question with a "pending-coherence" bit and then forwards possible actions to the L1 cache first.

The actual actions for the message are processed at the time of the COHE_REPLY; however, NACK_PEND responses from the L1 cache are forwarded to the directory immediately for non-WRB messages. The pending-coherence 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 COHE_REPLY acknowledgment or copyback to the directory, but also sends a REPLY to the requesting processor with the desired data. The cache-to-cache transfer policy follows that depicted in Figure 3.3.

Difference 3: Additional conditions for stalling REQUEST messages

The notpres_mshr function adds two further conditions for stalling REQUEST messages in the case of the L2 cache.

REQUESTs that hit a line with its pending-coherence bit set receive a return value of NOMSHR_STALL_COHE from notpres_mshr. This indicates that no MSHR has been consumed, but that this REQUEST must wait for the pending COHE first. This case does not appear in the L1 cache because that cache does not have pending-coherence bits for lines.

If the REQUEST is not able to reserve a space in the write-back buffer for its expected REPLY, the notpres_mshr function returns NOMSHR_STALL_WRBBUF_FULL. This indicates to the function L2ProcessTagReq that no MSHR will be allocated for this REQUEST until space in the write-back buffer becomes available.

Difference 4: Handling retries

The L2 cache accepts REPLYs from the directory with the s.reply field set to RAR (request a retry). This indicates that the directory could not process the REQUEST and is returning it to avoid deadlock. In this case, the cache must reissue the original REQUEST. The cache uses the MSHR originally allocated by the REQUEST as a resource from which to reissue the REQUEST, thus allowing the retry message to be accepted even if the outbound REQUEST port is blocked.

Difference 5: Handling replacements caused by replies

The most significant differences between L1ProcessTagReq and L2ProcessTagReq deal with the handling of replacements. The L2 cache has a write-back buffer for victimization messages to the directory. Before a REQUEST can be sent out, the notpres_mshr function must ensure that there will be a write-back buffer space available for the reply.

When the REPLY returns to the cache, a write-back buffer is tentatively booked.

Difference 5a: Replies that replace no line or a shared line

If the REPLY causes no replacement, the write-back buffer space is freed. If the REPLY replaces a shared line, the cache sends a subset enforcement invalidation to the L1 cache, possibly using the write-back buffer entry as a resource from which to send the invalidation. This resource is added to the smart MSHR list simulator abstraction.

Difference 5b: Replies that replace an exclusive line

However, if the REPLY causes a replacement of an exclusive line, the write-back buffer space may actually be used for data as well.

If the L1 cache above is write-through, an invalidation message is sent up to the L1 cache. If it cannot be sent immediately, the cache uses the write-back buffer entry as a resource from which to send the message. After the invalidation message is sent up, the write-back or replacement message tries to issue from the write-back buffer entry to the port below it. Again, if this message cannot be sent immediately, the write-back buffer entry will be used as a resource to hold the message until it can be sent. In both cases, the write-back buffer entry is added to the smart MSHR list simulator abstraction.

If the L1 cache is write-back, the WRB first passes through the L2 data array if the line in L2 is held in dirty state, then to the L1 cache as a subset-enforcement COHE message. The next WRB coherence reply from the L1 is used to either replace the data currently held for the line in the write-back buffer (on a positive acknowledgment) or to inform the cache that the L1 cache did not have the desired data (on a NACK). A variety of races are handled in these cases, as the L1 may send an unsolicited write-back at nearly the same time as the L2 sends a request for a write-back. The details of these races are explained in the inline documentation with the code.

If neither the L2 nor the L1 had the line in dirty state, a REPL message, rather than a WRB is sent to the directory. The directory-bound write-back or replacement message uses its write-back buffer entry as a resource from which to send the message to insure that an inability to send out a WRB does not stall the REPLY that caused the replacement or the COHE_REPLY from the subset-enforcement WRB. Once the write-back or replacement message issues to the port below it, its space in the write-back buffer is cleared.

13.5 Processing L2 data array accesses

Source files: `src/MemSys/l2cache.c`, `src/MemSys/cachehelp.c`

Header files: `incl/MemSys/cache.h`

`L2ProcessDataReq` is a short function that handles the data-array access of each request type. The additional processing for each request type here is minimal. Some message types use write-back buffer entries as resources from which to issue messages from this stage. This use of the write-back buffer entries prevents write-backs from blocking any resources, thus preventing one possible deadlock condition. As with other resources held to send out messages, the cache uses the smart MSHR list as a simulator abstraction in these cases.

13.6 Cache initialization and statistics

Source files: `src/MemSys/cache.c`, `src/MemSys/cache2.c`, `src/Processor/capconf.cc`, `src/MemSys/mshr.c`

Header files: `incl/MemSys/cache.h`, `incl/MemSys/stats.h`, `incl/Processor/capconf.h`

The functions `NewCache` and `init_cache` initialize the data structures used by the cache, including the the cache-line state structures, the MSHR array, the write-back buffer, the cache pipelines, and the statistics structures.

Each data access to the cache calls `StatSet` to specify whether the access hit or missed, and the type of miss in the case of misses. Each cache module classifies misses into cold, 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, new insertions 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 its taken to refill the entire cache; 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 are 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.

The caches also keep track of the MSHR occupancy, determining the percentage of time any given number of MSHRs is in use. This statistic is calculated through an *interval statistics record*, described in Chapter 16.1.

13.7 Discussion of cache coherence protocol implementation

Source files: `src/MemSys/setup_cohe.c`

RSIM supports two coherence protocols – MSI and MESI. Both of these protocol implementations are depicted in Figure 3.3. In the MSI system, an explicit upgrade message is required for a read followed by a write, even if there are no other sharers. The MESI system overcomes this disadvantage. However, our MESI implementation requires a message to be sent to the directory on elimination of an exclusive line from the L2 cache. Some available MESI implementations avoid this replacement message; however, in such systems, the write-back of a modified line requires an acknowledgment from the memory controller and holds an entry in the write-back buffer until the reply arrives [12]. In our system, on the other hand, a write-back does not cause a reply, and the write-back buffer entry is freed as soon as the write-back issues to the ports below the cache. The bandwidth tradeoff between these two choices is application-dependent.

13.8 Coalescing write buffer

Source files: `src/MemSys/wb.c`, `src/MemSys/wbuffer.c`

Header files: `incl/MemSys/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 (Figure 3.2), the simulated module sits between the two caches. To provide the semblance of parallel access, the write buffer has zero delay.

The `WBSim` function implements the write buffer and is called from `L2CacheOutSim` (for `REPLY`s) and from `L1CacheOutSim` (for `REQUEST`s). This simulation module first checks for a message on any of 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 [21][4].

If the incoming `REQUEST` is a write, the write-buffer attempts to add it to the queue of outstanding write-buffer entries by calling the function `notpres_wb`. 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. This processing takes place in `case 9` of the function `WBSim`. As soon as a write is added to an L2 port, its entry is freed from the write-buffer.

`REPLY`s are immediately forwarded to the L1 cache with no additional processing. The write buffer does not receive `COHE` or `COHE_REPLY` messages.

13.9 Deadlock avoidance

The caches guarantee that all incoming replies (`REPLY`s and `COHE_REPLY`s) are accepted even if they require further messages to be sent out. If a `REPLY` leads to a retry (either because of an `RAR` or a `NackUpgradeConflictReply`), the MSHR originally allocated by the `REQUEST` will be used as a resource from which to send the message to the cache ports. If a `REPLY` leads to a writeback or replacement message, its write-back buffer entry or MSHR can be used as a resource from which to send the message. In all cases, the resources needed to send any possible messages on `REPLY` are reserved at the time of the `REQUEST`.

Additionally, the tag arrays of the caches have a special port reserved for `COHE` messages to insure that these messages do not interact with `REQUEST`s or `REPLY`s to cause a deadlock. Such a port would likely be considered excessive in a real system; an alternative is to have the caches break deadlocks caused by `COHE`s by sending these messages back to the directory marked with `s.reply` fields set to `RAR` (request a retry).

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

Chapter 14

Directory and Memory Simulation

Source files: `src/MemSys/directory.c`

Header files: `incl/MemSys/directory.h`

The processor and cache modules are simulated on a cycle-by-cycle basis, as these units are likely to have activity nearly every cycle. However, the remaining building blocks of RSIM are activated only according to whether or not they are ready to process a transaction.

The directory controller modeled in RSIM implements a 4-state MESI coherence protocol or a 3-state MSI protocol. The directory is merged with the system DRAM, as accesses generally need to process information from both the DRAM and the directory at the same time (an exception is noted below). The directory and memory banks are interleaved on a cache-line basis, using as many modules as specified with the "-I" option. The directory is responsible for maintaining the current state of a cache line, serializing accesses to each line, generating and collecting coherence messages, sending replies, and handling race conditions. In addition, the directory coherence protocol used in RSIM relies on cache-to-cache transfers and uses replacement messages, as depicted in Figure 3.3. Coherence replies are collected at the directory, and in the case of transfers that require coherence actions (other than cache-to-cache transfers), the data is sent to the requestor only after all coherence replies have been collected.

The `DirSim` function simulates the actions of the directory and includes many stages based on the current type of access being processed and the progress of that access. The following sections describe each of the stages in the directory.

14.1 Obtaining a new or incomplete transaction to process

The first stage is `DIRSTART`. In this stage, the directory first sees if it can send out a message that was previously added to its list of outbound transactions (`OutboundReqs`). If a message can be sent, the directory jumps to state `DIROUTBOUND`; otherwise, the message at the head of the `OutboundReqs` list is pushed to the end.

Next, the directory starts to check its input ports (`REQUEST` and `COHE_REPLY`) in a round-robin fashion. However, the directory cannot bring in a new `REQUEST` if it still has a partially completed `REQUEST` transaction (described in Sections 14.2 and 14.3). This transaction must be completed before the any new `REQUEST`s are brought in. In this case, the directory transitions to state `DIRSENDCOHE` to attempt to complete the previous `REQUEST` transaction rather than bringing in a new one. Further, the directory cannot bring in a new `REQUEST` if a formerly pending request (described in Section 14.2) has become ready for processing. In this case, the formerly pending `REQUEST` is chosen rather than a new `REQUEST`. Before looking at the input port for new `REQUEST`s, the directory will check to see if it has a partially completed transaction in flight (noted in the `req_partial` field). This transaction must be completed before any new `REQUEST`s are brought in. In this case, the directory transitions to state `DIRSENDCOHE`. If there are no partially completed transactions, the directory then looks at its list of previously pending `REQUEST`s to process an old `REQUEST` before bringing in a new one. Only if there is no partially completed transactions or formerly pending REQUEST available will the directory bring in a new REQUEST. However, there is no restriction on the directory bringing in a new COHE_REPLY: such an access will be brought in and processed regardless of previous pending REQUESTs or partially-completed REQUEST transactions.

For new REQUESTs, formerly pending REQUESTs, or COHE_REPLYs, the directory must now stall for its access latency. This is calculated based on the width of the directory port and the flit transfer time, discussed in Section 12.3. Additionally, for all accesses except COHE_REPLYs that do not access memory (simple acknowledgments without copy-back), the directory must delay by the memory latency specified with "-M". For COHE_REPLYs that do not access memory, the directory only stalls for the amount of time specified with the configuration parameter `dircycle`, which represents the minimum directory delay.

After these stalling, the directory jumps to state DIRSERV, where REQUESTs are dispatched to DIRREQ, write-backs or other replacement messages are sent to case DIRWRB, and other COHE_REPLYs are sent to state DIRCOHEREP. Each of these states is discussed in the following sections.

14.2 Processing incoming REQUESTs

The first part of DIRREQ handles "preprocessed" REQUESTs, which have already come to the directory before but were bounced back at the cache by `NackUpgradeConflictReply` (Section 13). These are simply bounced back to the cache. For other REQUESTs, the directory coherence routine (`Dir_Cohe`) must be called in order to determine the course of handling this access.

`Dir_Cohe` determines the state transition for the line at the directory, as well as the possible COHE messages that this access will need to send out. As `Dir_Cohe` may consume a directory buffer, the directory must first check to make sure that a buffer is available; if none is available, the directory sends the REQUEST back as an RAR.

If `Dir_Cohe` responds with a DIR_REPLY or VISIT_MEM, no COHE actions must be sent as a result of this transaction. Consequently, the directory attempts to send a REPLY back to the caches. If a REPLY cannot be sent, the REQUEST is held in its input port or pending buffer until it can be processed later; the directory returns to case DIRSTARTOVER in such cases. If a REPLY can be sent, the directory moves to DIRSENDREQ.

If the REQUEST comes from a cache that the directory believes to have the line in exclusive state, then the REQUEST must have bypassed an in-flight write-back or replacement message from the same node. In this case, `Dir_Cohe` returns WAITFORWRB to force this REQUEST to wait for a write-back or replacement message before being processed. Such a REQUEST is moved to the pending queue to allow other REQUESTs to be processed in the meanwhile. Similarly, if the REQUEST is to a line that is currently in a transient directory state, caused by outstanding COHE messages or an outstanding WAITFORWRB, `Dir_Cohe` returns WAIT_PEND and the REQUEST is also added to the pending buffer.

If the REQUEST requires new COHE messages to be sent out and the line is currently held in shared state by other nodes, the `Dir_Cohe` function returns WAIT_CNT. In this case, the directory must create new invalidation COHE packets. These messages are sent as INVL coherence messages, with NACK_OK set to indicate that a negative-acknowledgment is acceptable.

If the line is currently held in exclusive state by another node, `Dir_Cohe` returns FORWARD_REQUEST. This reponse indicates that the outbound coherence message should request the owner of the cache line to send a cache-to-cache transfer to the requester. If the REQUEST being sent is a shared-mode access (read), the cache-to-cache transfer COHE request is sent as a COPYBACK, indicating that the memory should also be sent a copy of any dirty data. This is needed because the cache-coherence protocol does not support a shared-dirty state; any line shared by multiple caches must be held with the same value at the memory. If the REQUEST is an exclusive-mode access, the cache-to-cache transfer COHE request is sent as COPYBACK_INVL, indicating that it is sufficient to send only an acknowledgment to the directory after sending the cache-to-cache transfer. In either case, negative acknowledgments are not acceptable, so the cache-to-cache transfer request is sent with NACK_NOK.

In either the WAIT_CNT or FORWARD_REQUEST cases, the directory has to delay the processing of outbound coherence messages according to the packet creation time described in Section 4. Namely, the first COHE message will wait for a delay of `pkt_create_time`, while each subsequent message will wait for `addtl_pkt_crt_time`. Each time the directory delays for a packet creation, the REQUEST is put into the directory's partially-completed transaction structure (req_partial), and the directory transitions to DIRSENDCOHE.

14.3 Sending out COHE messages

State DIRSENDCOHE attempts to send out COHE messages needed by a new or partially-completed REQUEST being processed. If there is no space available on the output port for COHEs, the REQUEST is placed in the req_partial field to indicate that it is a partially-completed transaction. The directory is then sent back to state DIRSTARTOVER so that it may be able to process incoming COHE_REPLYs in the meanwhile.

If there is space available on the output COHE port, a COHE for this transaction is sent out. If the transaction requires more coherence messages to be sent out, the request is placed in the req_partial field and the directory must delay for the additional packet creation time (a configurable parameter). After delaying, the directory will remain in state DIRSENDCOHE, which will then attempt to send the next coherence message out. These steps repeat until all COHEs for this transaction have been sent out.

14.4 Processing incoming write-back and replacement messages

On write-back or replacement messages, the directory transitions to state DIRWRB. These accesses call Dir_Cohe in order to make changes to the state of the line. If any pending REQUESTs are waiting on this write-back or replacement message, those are marked as being ready for processing.

14.5 Processing other incoming COHE_REPLYs

For other coherence replies, the directory uses state DIRCOHEREP. The operation of the directory in these cases depends on whether the response is a positive acknowledgment or a negative acknowledgment (as indicated by the s.reply field of the message).

14.5.1 Handling positive acknowledgments

In the case of positive acknowledgments (or negative acknowledgments to coherence messages sent with NACK_OK), the directory sees if this response enables it to send out a REPLY (if the access is not a cache-to-cache transfer and all coherence replies have now been collected). If so, the directory attempts to send out a reply, adding the reply to its OutboundReqs structure if no space is currently available in its port.

If the positive acknowledgment is an acknowledgment from a cache-to-cache transfer (whether it includes data or not), the buffer entry for the access is freed, as these accesses do not require any further messages to be sent.

14.5.2 Handling negative acknowledgments

If s.reply is set to NACK, the COHE_REPLY indicates that the line missed in the remote cache. This response type is acceptable and handled like a positive acknowledgment unless the COHE message from the directory was sent with NACK_NOK. NACK_NOK indicates that the coherence message specifically expected that data would be transfered. In this case, the cache must have issued a write-back or replacement message just prior to receiving the coherence message in question. If this write-back or replacement has not yet been received, the original REQUEST that started the coherence transaction sequence is sent back to its cache with an RAR to be retried. (If the RAR cannot be sent immediately, the directory puts it in OutboundReqs to be tried later). If the write-back has been received at the directory, though, the original REQUEST can be reprocessed through the standard directory request-handling case.

On NACK_PEND coherence replies, the directory is expected to reevaluate its status and resend the COHE request if needed. A response of NACK_PEND is sent from the cache in certain cases of a COHE received for a line with an outstanding MSHR. Such a race can occur either if the cache sent a write-back or replacement message before the directory sent out the COHE or if the COHE was received by the cache before the REPLY, which had been sent to the cache earlier. If the directory has received a write-back for the line in question from the node that responded with a NACK_PEND, the original REQUEST is reprocessed by the normal request handler. Otherwise, the coherence message is retried again, as it may have been NACK_PENDed only because it bypassed an earlier REPLY.

14.6 Deadlock avoidance

Certain sequences of transactions at the directory require the directory to reprocess or reissue previous messages. Just as the caches reissue retries from previously allocated MSHRs or write-back buffer entries, the directories reprocess REQUESTs or reissue COHE messages from the buffers already allocated at the time of the original REQUEST.

The directory is also reponsible for breaking request-request cycles in the system. If such a condition arises, the directory buffers of some directory must have filled up, since all REQUESTs have a directory as an ultimate destination and all COHEs issue from the directory. If the request buffers of a directory fill up, the directory sends back later requests as RAR replies. These will not lead to deadlock, since the caches can always accept retries in a finite amount of time.

In certain pathological cases, the RSIM directory may starve a specific processor by always choosing its requests for retry. This can be changed by adding additional constraints on the requests that a directory can process once it resorts to retries.

Chapter 15

System Interconnects

RSIM simulates a bus connecting modules within a node and a multiprocessor interconnection network connecting the nodes within a system. Section 15.1 explains the internals of the bus. Section 15.2 describes the function of the network interface on each node. Section 15.3 explains the operation of the multiprocessor interconnection network.

15.1 Node bus

Source files: `src/MemSys/bus.c`

Header files: `incl/MemSys/bus.h`

The RSIM bus module simulates an aggressive split-transaction bus that imposes no limits on outstanding requests. This bus connects the L2 cache, the network interfaces, and the directory/memory modules within a node. Arbitration is round-robin among the bus agents. As in a real system, a bus agent is not allowed to acquire the bus unless its destination is ready. The bus speed, bus width, and arbitration delays can be configured as described in Chapter 4.

The function `node_bus` represents the main operation of the bus simulator. This function is split up into several stages according to the progress of a request on the bus. In the `BUSSTART` stage, the bus has not started processing a transaction. In this case, the bus peeks at the ports round-robin (starting with the port after the one last accessed) for a new message. If a transaction is available, the bus moves to the `SERVICE` stage.

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

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

15.2 Network interface modules

Source files: `src/MemSys/smnet.c`

Header files: `incl/MemSys/net.h`

The `SMNET` (Shared Memory-NETwork) interfaces in RSIM are the modules that connect each node's local bus to the interconnection network. The primary functions of the `SMNET` are as follows:

- Receive messages destined for the network from the bus. (These may originate from the cache or directory controller.)
- Create the message packets.
- Inject the messages into the appropriate network ports and initiate communication.
- Handle incoming messages from the network by removing them from the network port and delivering them to the bus.

The main procedure for sending packets to the network is `SmnetSend`. This event handles communication between the bus and the network interface. Upon receiving a new message, `SmnetSend` schedules an appropriate event to insert the new message into the request or reply network as appropriate. The events that provide this interface have the body functions `ReqSendSemaWait` and `ReplySendSemaWait` for the request network and reply network, respectively. These events ensure that there is sufficient space in the network interface buffers before creating the packets and initiating communication.

In addition to sending messages, the Smnet module handles receiving messages through the `ReqRcvSemaWait` and `ReplyRcvSemaWait` events. These events wait on semaphores associated with the network output ports to receive messages. (Semaphores are discussed in Section 8.2.) As soon as a message is received, it is forwarded to the appropriate bus port, according to whether it is a request or a reply. The bus will actually deliver the message to the caches or the directory.

15.3 Multiprocessor interconnection network

Source files: `src/MemSys/net.c`, `src/MemSys/mesh.c`

Header files: `incl/MemSys/simsys.h`, `incl/MemSys/net.h`

The base network provided in the RSIM distribution is a 2-dimensional bi-directional mesh (without wraparound connections), and is taken from the NETSIM simulation system [7]. The interconnection network includes separate request and reply networks for deadlock-avoidance. Unlike the other subsystems discussed in this chapter, the network is not built using the standard module framework.

The network flit delay, arbitration delay, width, and buffer sizes can be configured as described in Chapter 4. Additionally, the system can be directed to simulate pipelined switches, by which the flit delay of multiple flits can be incurred in a pipelined manner. To model this behavior, a system with pipelined switches uses a flit delay equal to the granularity of pipelining and adds the remainder of the originally specified flit delay to the arbitration delay of the multiplexers. With these adjustments, the latency of the head flit of a packet remains the same, but subsequent flit delays are based on the degree of pipelining.

The processor connection to the network of each node is depicted in Figure 15.1. Similar components connect the node to the interconnection network and neighboring processors along the X and Y axes. Messages are injected into the network using the `SmnetSend` event and are received from the interconnection network using the `ReqRcvSemaWait` and `ReplyRcvSemaWait` events, corresponding to the request and reply networks, respectively.

The network routes packets using dimension-ordered routing, and each switch provides wormhole routing. At each buffer, port, multiplexor or demultiplexor, the packet's head flit determines its next destination in the network. When moving from one buffer to the next, the head flit encounters a delay of `flitdelay` cycles, which corresponds to the flit latency, possibly adjusted for pipelining as described above. In addition, the head flit consumes arbitration delays (possibly adjusted for pipelining) at each multiplexor. (NETSIM allows the head flit to experience routing delays at demultiplexors, but RSIM does not currently use this feature; these delays are set to 0.) A packet's remaining flits are moved when the tail flit is allowed to move. A tail flit is allowed to move in the network as long as it does not share a buffer with the head flit since the tail is never allowed to overtake the head of a packet. Once the tail is allowed to move, the simulator moves all intermediate flits in a pipelined fashion every `flitdelay` cycles until the tail flit itself moves. The various flits in the packet may thus span several network buffers at any time. This tail movement process continues until the packet has reached the destination output port or until the tail has caught up with its head flit.

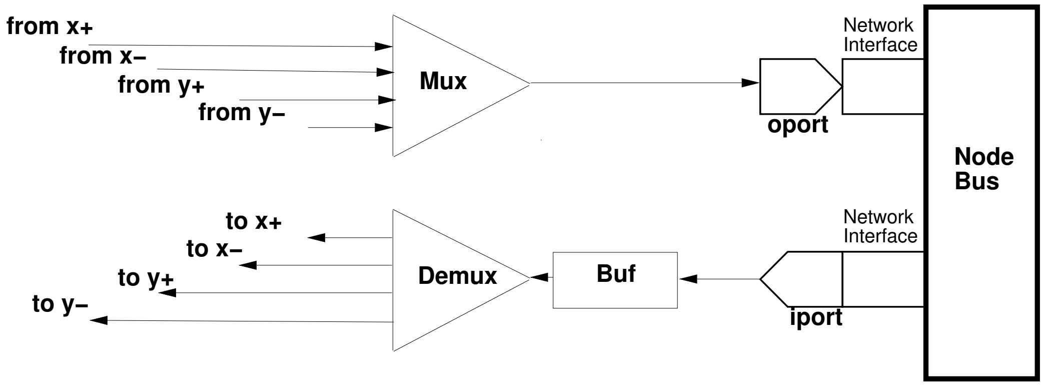


Figure 15.1: Processor side 2D-mesh switch connection.

The NETSIM reference manual is recommended reading for anyone intending to add other interconnection network types or policies [7]. RSIM supports all of the primary functions in NETSIM.

15.4 Deadlock avoidance

The node bus avoids deadlock by accessing subsequent bus agents in a round-robin order. If the desired target of a transaction is not available, the bus does not allow a new transaction to progress beyond arbitration. Thus, no access is allowed to stall the bus.

The interconnection network avoids deadlock through three design decisions. First, replies and requests are sent on separate networks. Since replies are guaranteed to be processed by the other modules in a finite amount of time, there is no possibility of the reply network deadlocking. Requests can depend on other requests or replies. If a request depends only on other replies, there is no chance of deadlock since no circular dependence exists (as replies cannot in turn depend on requests). However, if a request depends on other requests, deadlock may arise. This type of deadlock is resolved at the directory, as described in Section 14.6.

Chapter 16

Statistics and Debugging Support

This chapter discusses the statistics provided for the various subsystems as well as the debugging information available.

16.1 Statistics

Source files: `src/MemSys/stat.c`

Header files: `incl/MemSys/simsys.h`

The detailed set of statistics is printed on simulation output at the end of each phase and at the end of the simulation; a concise summary of the most important statistics is provided on the simulation standard error. The types of statistics and the utilities for processing these statistics are described in Chapter 6.

Some of the statistics are kept track of by simple counters, but most of these statistics are computed using the `STATREC` functions of YACSIM [8]. The key functions used are:

- `STATREC *NewStatrec(char *name, int type, int means, int hist, int numbins, double lowbin, double hibin)`

  Returns a new statistics record with the specified name and type (`POINT` or `INTERVAL`). A `POINT` statistics record uses the weight passed in through `StatrecUpdate` for the weight of each sample, whereas an `INTERVAL` statistics record uses the difference between the weight parameter for the current sample and the weight parameter for the previous sample as the actual weight of the current sample. The most common way to use an `INTERVAL` statistics record is to pass in the current simulation time as the weight parameter. In this way, the weight of the sample being passed in is the length of time since the last call to `StatrecUpdate` for this record.

  `means` (can be set to `MEANS` or `NOMEANS`) indicates whether or not this statistics record should calculate mean and standard deviation. `hist` (can be set to `HIST` or `NOHIST`) indicates whether or not a histogram should be generated and reported for this record. (RSIM has also added the value `HISTSPECIAL`, which indicates that a histogram should be generated, but only those bins with a non-zero number of entries should be displayed when reporting statistics.) If a histogram is used, it will have `numbins` primary bins, equally distributed with values from `lowbin` to `highbin`. There will also be overflow bins provided.

- `void StatrecReset(STATREC *s)`

  Clears out the statistics recorded in `s`

- `void StatrecUpdate(STATREC *s, double val, double wt)`

  Adds the value `val` into the statistics record with a weight parameter of `wt`

- `void StatrecReport(STATREC *s)`

  Prints a report of the mean, standard deviation, high, low, sample count, and histogram of the `STATREC` on the simulation output (mean, standard deviation, and histogram provided only if thus configured when calling `NewStatrec`).

- `int StatrecSamples(STATREC *s)`

  Returns number of samples recorded

- `double StatrecMean(STATREC *s)`

  Returns the mean of the samples

- `double StatrecSum(STATREC *s)`

  Returns the sum of the samples

- `double StatrecSdv(STATREC *s)`

  Returns the standard deviation of the samples

16.2 Debugging Support

RSIM provides a variety of debugging information for the developer looking to make changes to the system. Currently, the caches, bus, network interfaces, and directory have corresponding `#define`s which, when set at compile time, cause these modules to trace each of their important actions on the simulation output, along with information related to the specific access being considered. Any or all of these debugging options can be used at any time to trace the behavior of these modules. Additionally, the processor provides a variety of tracing information for each pipeline stage and each instruction execution if the `COREFILE` option is defined at compile-time. Each processor produces a trace file called `corefile.`*i*, where i is the unique identifier for the processor. The `Makefile` provided in the `obj/dbg` directory has all the common debugging options set so as to produce the maximum possible tracing output. Generally speaking, this is the desired level of tracing for most significant simulator changes.

In addition to this tracing provided by RSIM, RSIM can be run through any standard debugger. However, applications being simulated under RSIM cannot be debugged using a standard debugger from within the RSIM environment, as RSIM does not expose information about the application being simulated to the debugger.

Chapter 17

Implementation of `predecode` and `unelf`

This chapter provides implementation details for the `predecode` and `unelf` utilities used to process applications to be simulated with RSIM.

17.1 The `predecode` utility

Source files: `src/predecode/predecode.cc`, `src/predecode/predecode_instr.cc`, `src/predecode/predecode_table.cc`,

Header files: `incl/Processor/instruction.h`, `incl/Processor/table.h`, `incl/Processor/decoding.h`, `incl/Processor/archregnums.h`

This utility converts application executable files from the SPARC format to a format understood by RSIM. The `main()` function starts by looping through the ELF sections of the executable looking for instruction sections. Once a text section is found, the `start_decode` function is called on every SPARC instruction in the region.

The `start_decode` function begins by calling either `branch_instr`, `call_instr`, `arith_instr`, or `mem_instr`, based on the first two-bits of the instruction type. `call_instr` directly interprets the `CALL` instruction specified and returns. The other functions mentioned begin by dispatching the instruction to another function, based on up to 6 opcode bits. This dispatch is done through a table set in the function `TableSetup`, which corresponds to the instruction mapping specified in the SPARC V9 architecture [23]. In some cases, multiple dispatch functions may need to be invoked. Finally, however, the functions given in `predecode_instr.cc` convert from the SPARC instruction with tightly-encoded fields to the more loosely-encoded RSIM instruction format (specified in the `instr` data structure). The opcode and the way in which it defines its fields determines the final encoding used.

More information about the opcodes supported and their fields can be found in the SPARC V9 Architecture Reference Manual [23].

17.2 The `unelf` utility

Source files: `src/predecode/unelf.cc`

`unelf` uses the functions in the ELF library to expand an `ELF` executable into a format that can be processed by a machine without support for the ELF library. `unelf` loops through the ELF sections for the file. If any section contains data (which can either be instructions, initialized static data, or uninitialized data), space for that section should be allocated into the overall data array being constructed. If the ELF section actually provides data values (instructions or initialized static data), those should be copied from the section into the data array being constructed. If the section is for uninitialized static data, the corresponding region of the data array being constructed should be cleared.

After constructing a data array, the utility writes an `UnElfedHeader` data structure to the output file. The `UnElfedHEader` specifies the size of the data array, the entry point, and the virtual address for the start of the data array. Next, `unelf` writes the entire data array to the output file.

More information about ELF can be found from the Unix manual pages.

Bibliography

[1] James E. Bennett and Michael J. Flynn. Performance Factors for Superscalar Processors. Technical Report CSL-TR-95-661, Stanford University, February 1995.

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

[3] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *Intl. Journal in Computer Simulation*, 1:31–58, January 1991.

[4] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[5] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

[7] J. Robert Jump. *NETSIM Reference Manual*. Rice University Electrical and Computer Engineering Department, March 1993. Available at http://rsim.cs.uiuc.edu/rsim/rppt.html.

[8] J. Robert Jump. *YACSIM Reference Manual*. Rice University Electrical and Computer Engineering Department, March 1993. Available at http://rsim.cs.uiuc.edu/rsim/rppt.html.

[9] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Targer Branches Due to Subroutine Returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 1991.

[10] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Eighth Symp. on Computer Architecture*, pages 81–87, May 1981.

[11] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[12] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

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

[14] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. Intl. Symp. on High Performance Computer Architecture*, pages 72–83, 1997.

[15] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[16] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 144–156, 1997.

[17] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, 1997.

[18] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmet Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[19] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proc. 14th Intl. Symp. on Computer Architecture*, pages 234–243, Pittsburgh, PA, June 1987.

[20] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[21] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 144–155, February 1997.

[22] J. E. Smith. A study of branch prediction strategies. In *Proc. Eighth Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

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

[24] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[25] Sun Microelectronics. *UltraSPARC-II: Second Generation SPARC v9 64-Bit Microprocessor with VIS*, July 1997.

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

[27] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.

[28] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

Appendix A

RSIM Version 1.0 License Terms and Conditions

Developed by: Professor Sarita Adve's RSIM research group
University of Illinois at Urbana-Champaign and William Marsh Rice University
http://www.cs.uiuc.edu/rsim and http://www.ece.rice.edu/~rsim/dist.html

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.
- Neither the names of Professor Sarita Adve's RSIM research group, the University of Illinois at Urbana-Champaign, William Marsh Rice University, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.