The Multi2Sim Simulation Framework

A CPU-GPU Model for Heterogeneous Computing
(For Multi2Sim v. 4.0.1)

List of authors contributing to the development of the simulation framework and/or writing of this document.

Zhongliang Chen, Northeastern University, Boston, MA, USA
Tahir Diop, University of Toronto, ON, Canada
Xiang Gong, Northeastern University, Boston, MA, USA
Steven Gurfinkel, University of Toronto, ON, Canada
Byunghyun Jang, University of Mississippi, MS, USA
David Kaeli, Northeastern University, Boston, MA, USA
Pedro López, Universidad Politécnica de Valencia, Spain
Nicholas Materise, Northeastern University, Boston, MA, USA
Rustam Miftakhutdinov, University of Texas, Austin, TX, USA
Perhaad Mistry, Northeastern University, Boston, MA, USA
Salvador Petit, Universidad Politécnica de Valencia, Spain
Julio Sahuquillo, Universidad Politécnica de Valencia, Spain
Dana Schaa, Northeastern University, Boston, MA, USA
Sudhanshu Shukla, Indian Institute of Technology Kanpur, India
Rafael Ubal, Northeastern University, Boston, MA, USA
Yash Ukidave, Northeastern University, Boston, MA, USA
Mark Wilkening, Northeastern University, Boston, MA, USA
Norm Rubin, AMD, Boxborough, MA, USA
Amir Kavyan Ziabari, Northeastern University, Boston, MA, USA

Contents

1 Introduction **8**
- 1.1 Organization of Multi2Sim's Source Code . . . 8
- 1.2 The Four-Stage Architectural Model . . . 8
  - 1.2.1 Disassembler . . . 10
  - 1.2.2 Functional Simulator . . . 10
  - 1.2.3 Detailed Simulator . . . 11
  - 1.2.4 Visual Tool . . . 12
- 1.3 Full-System vs. Application-Only Emulation . . . 12
  - 1.3.1 Full-System Emulation . . . 13
  - 1.3.2 Application-Only Emulation . . . 13
- 1.4 Getting Started . . . 14
  - 1.4.1 Installation . . . 14
  - 1.4.2 First Execution . . . 15
  - 1.4.3 Statistics Summary . . . 16
  - 1.4.4 Launching Multiple Guest Programs . . . 16

2 The x86 CPU Model **18**
- 2.1 The x86 Simulation Paradigm . . . 18
  - 2.1.1 The x86 Functional Simulator . . . 18
  - 2.1.2 The Detailed Simulation . . . 20
- 2.2 The x86 CPU Statistics Summary . . . 20
- 2.3 Compiling and Simulating Your Own Source Code . . . 21
  - 2.3.1 Static and Dynamic Linking . . . 22
  - 2.3.2 Observing the Differences . . . 22
  - 2.3.3 Execution Variability . . . 24
  - 2.3.4 Error Messages when Simulating your Program Binaries . . . 25
- 2.4 The Processor Pipeline . . . 25
- 2.5 Branch Predition . . . 26
  - 2.5.1 Perfect branch predictor . . . 26
  - 2.5.2 Taken branch predictor . . . 27
  - 2.5.3 Bimodal branch predictor . . . 27
  - 2.5.4 Two-level adaptive predictor . . . 27
  - 2.5.5 Combined predictor . . . 28
- 2.6 Multiple Branch Prediction . . . 28
- 2.7 CISC Instructions Decoding . . . 29
- 2.8 Trace Cache . . . 30
  - 2.8.1 Creation of traces . . . 32

2.8.2 Trace cache lookups . . . 32
2.8.3 Trace cache statistics . . . 33
2.9 The Fetch Stage . . . 33
2.10 The Decode Stage . . . 35
2.11 Integer Register Renaming . . . 35
2.11.1 Logical Registers . . . 35
2.11.2 Physical Register File . . . 36
2.11.3 Renaming Process . . . 36
2.12 Floating-Point Register Renaming . . . 37
2.12.1 The x86 Floating-Point Stack . . . 37
2.12.2 Two-Stage Renaming Process . . . 37
2.13 The Dispatch Stage . . . 38
2.14 The Issue Stage . . . 38
2.15 The Writeback Stage . . . 39
2.16 The Commit Stage . . . 39
2.17 Support for Parallel Architectures . . . 39
2.18 Multithreading . . . 40
2.18.1 Configuration of storage resources . . . 40
2.18.2 Configuration of bandwidth resources . . . 41
2.19 Multicore Architectures . . . 41
2.20 The Context Scheduler . . . 42
2.20.1 The Static Scheduler . . . 42
2.20.2 The Dynamic Scheduler . . . 42
2.21 Statistics Report . . . 43
2.21.1 Global statistics . . . 43
2.21.2 Statistics related to all pipeline stages . . . 44
2.21.3 Statistics related to the dispatch stage . . . 44
2.21.4 Statistics related to the execution stage . . . 45
2.21.5 Statistics related to the commit stage . . . 45
2.21.6 Statistics related to hardware structures . . . 45
2.22 Periodic Performance Report . . . 45

3 The OpenCL Programming Model 47
3.1 Basic Concepts of the OpenCL Programming Model . . . 47
3.2 The Execution Model for OpenCL Programs . . . 47
3.2.1 Native vs. Emulated Execution of OpenCL . . . 47
3.2.2 Execution of an OpenCL Program on an AMD-based Native Environment . . . 48
3.2.3 Execution of an OpenCL Program on Multi2Sim . . . 49
3.2.4 Implementation of the Multi2Sim OpenCL Library . . . 50
3.3 Building and Simulating Your Own OpenCL Program . . . 50
3.3.1 Building the Multi2Sim OpenCL Library . . . 51
3.3.2 Compiling Source Files . . . 51
3.3.3 Linking Object Files for Native Execution . . . 52
3.3.4 Linking Object Files for Execution on Multi2Sim . . . 52
3.3.5 Simulating an OpenCL Program Linked for Native Execution . . . 53
3.3.6 The Multi2Sim OpenCL Kernel Compiler . . . 53

4 The AMD Evergreen GPU Model **55**
4.1 Mapping the OpenCL Model into an AMD Evergreen GPU . . . 55
4.2 The Evergreen Instruction Set Architecture (ISA) . . . 57
4.2.1 Evergreen Assembly . . . 57
4.2.2 Control Flow and Thread Divergence . . . 58
4.2.3 Evergreen Emulation at the ISA Level . . . 59
4.3 The Evergreen GPU Device Architecture . . . 60
4.3.1 Work-Group Scheduling and Configuration . . . 60
4.3.2 Mapping Work-Groups to Compute Units . . . 61
4.4 The Compute Unit Architecture . . . 62
4.4.1 The Control-Flow (CF) Engine . . . 63
4.4.2 The Arithmetic-Logic (ALU) Engine . . . 65
4.4.3 The Texture (TEX) Engine . . . 67
4.4.4 Periodic Report . . . 68
4.5 The Evergreen GPU Memory Architecture . . . 69
4.5.1 Private Memory . . . 70
4.5.2 Local Memory . . . 70
4.5.3 Global Memory . . . 71
4.6 The GPU Occupancy Calculator . . . 71
4.6.1 Number of Registers per Work-Item . . . 71
4.6.2 Local Memory Used per Work-Group . . . 72
4.6.3 Work-Group Size . . . 72
4.7 Trying it out . . . 73
4.7.1 First Executions . . . 73
4.7.2 The Evergreen GPU Statistics Summary . . . 74
4.7.3 The OpenCL Trace . . . 75
4.7.4 The Evergreen ISA Trace . . . 76

5 The AMD Southern Islands GPU Model **78**
5.1 Mapping the OpenCL Model into an AMD Southern Islands GPU . . . 78
5.2 The Southern Islands Instruction Set Architecture (ISA) . . . 80
5.2.1 Southern Islands Assembly . . . 80
5.2.2 Control Flow and Thread Divergence . . . 80
5.3 The Southern Islands GPU Device Architecture . . . 81
5.3.1 Work-Group Scheduling and Configuration . . . 81
5.3.2 Mapping Work-Groups to Compute Units . . . 82
5.4 The Compute Unit Architecture . . . 83
5.4.1 The SIMD Unit . . . 86
5.4.2 The Scalar Unit . . . 87
5.4.3 The Branch Unit . . . 88
5.4.4 The Local Data Share (LDS) Unit . . . 89
5.4.5 The Vector Memory Unit . . . 90
5.5 The Southern Islands Memory Architecture . . . 91
5.5.1 Private Memory . . . 91
5.5.2 Local Memory . . . 92
5.5.3 Global Memory . . . 93
5.6 Trying It Out . . . 93
5.6.1 First Executions . . . 93

5.6.2 The Southern Islands GPU Statistics Summary . . . 94
5.6.3 The OpenCL Trace . . . 95
5.6.4 The Southern Islands ISA Trace . . . 96

6 The Memory Hierarchy 98
6.1 Memory Hierarchy Configuration . . . 98
6.1.1 Sections and Variables . . . 98
6.1.2 Memory Hierarchy Commands . . . 101
6.2 Examples of Memory Hierarchy Configurations . . . 102
6.2.1 Cache Geometries . . . 102
6.2.2 Example: Multicore Processor using Internal Networks . . . 102
6.2.3 Example: Multicore with External Network . . . 104
6.2.4 Example: Multicore with Ring Network . . . 106
6.2.5 Heterogeneous System with CPU and GPU cores . . . 109
6.3 Default Configuration . . . 111
6.4 Cache Coherence . . . 111
6.4.1 Cache Directories . . . 112
6.4.2 Main Memory Directories . . . 112
6.4.3 Deadlocks . . . 113
6.5 Statistics Report . . . 114

7 Interconnection Networks 116
7.1 Model Description . . . 116
7.2 Communication Model . . . 117
7.3 Routing . . . 117
7.4 Network Configuration . . . 118
7.5 Example of Network Configuration . . . 120
7.6 Example of Manual Routing . . . 120
7.7 Example Using Virtual Channels . . . 124
7.8 Statistics Report . . . 126
7.9 Stand-Alone Network Simulation . . . 127

8 M2S-Visual: The Multi2Sim Visualization Tool 128
8.1 Introduction . . . 128
8.1.1 Compilation of M2S-Visual . . . 128
8.1.2 Running M2S-Visual . . . 128
8.2 Main Window . . . 130
8.2.1 The Cycle Bar . . . 130
8.2.2 Panels . . . 130
8.3 The x86 CPU Visualization . . . 131
8.4 The Evergreen GPU Visualization . . . 132
8.5 Memory Hierarchy Visualization . . . 133

9 M2S-Cluster: Launching Massive Simulations 135
9.1 Introduction . . . 135
9.2 Requirements . . . 135
9.3 Installation . . . 136
9.3.1 Installation Steps . . . 136

9.3.2 Keeping Repositories up to Date . . . 137
9.4 The Client Tool `m2s-cluster.sh` . . . 137
9.4.1 Command `create` . . . 137
9.4.2 Command `add` . . . 138
9.4.3 Command `submit` . . . 139
9.4.4 Command `state` . . . 139
9.4.5 Command `wait` . . . 140
9.4.6 Command `kill` . . . 140
9.4.7 Command `import` . . . 140
9.4.8 Command `remove` . . . 141
9.4.9 Command `list` . . . 141
9.4.10 Command `list-bench` . . . 141
9.4.11 Command `server` . . . 141
9.5 Automatic Creation of Clusters: Verification Scripts . . . 141
9.5.1 Command `submit` . . . 142
9.5.2 Command `kill` . . . 142
9.5.3 Command `state` . . . 142
9.5.4 Command `wait` . . . 143
9.5.5 Command `process` . . . 143
9.5.6 Command `remove` . . . 143
9.6 Benchmark Kits . . . 143
9.7 Usage Examples . . . 145
9.7.1 Tying It Out . . . 145
9.7.2 Using a Modified Copy of Multi2Sim . . . 146
9.7.3 Transferring Configuration Files and Reports . . . 147

10 Tools **149**
10.1 The INI file format . . . 149
10.1.1 The inifile.py tool . . . 149
10.1.2 Reading INI files . . . 150
10.1.3 Writing on an INI file . . . 150
10.1.4 Using scripts to edit INI files . . . 150
10.2 McPAT: Power, Area, and Timing Model . . . 151
10.2.1 McPAT input file . . . 151
10.2.2 Interaction with Multi2Sim . . . 151
10.2.3 McPAT output . . . 152

Chapter 1

Introduction

Multi2Sim is a simulation framework for CPU-GPU heterogeneous computing written in C. It includes models for superscalar, multithreaded, and multicore CPUs, as well as GPU architectures. In this chapter, an introduction to Multi2Sim is presented, and it is shown how to perform basic simulations and extract performance results.

Throughout this document, the term *guest* will be used to refer to any property of the simulated program, as opposed to the term *host*, used to refer to the simulator properties. For example, the *guest code* is formed by instructions of the program whose execution is being simulated, whereas the *host code* is the set of instructions executed by Multi2Sim natively in the user's machine.

1.1 Organization of Multi2Sim's Source Code

The Multi2Sim software package is organized in a directory structure as represented in Figure 1.1. Directory `images` contains program icons. Directory `samples` contains sample programs and configuration files to test the simulator. Directory `tools` includes mostly runtime libraries linked with some guest programs using GPU simulation. Finally, the `src` directory contains the C source code that compiles into a single executable file called `m2s`.

The organization of subdirectories in `src` is similar to the structure of this document. Subdirectory `arch` contains the implementation for each microprocessor architecture supported in Multi2Sim (`x86`, `evergreen`, etc., described in Chapter 2 and following). Subdirectories `mem-system` (Chapter 6) and `network` (Chapter 7) contain the implementation of the memory system and the interconnection network models. And subdirectory `visual` (Chapter 8) includes the visualization tool, with one lower-level subdirectory per microprocessor architecture.

1.2 The Four-Stage Architectural Model

The development of the model for a new microprocessor architecture on Multi2Sim consists of four phases, represented in Figure 1.2 in order from left to right. The development phases involve the design and implementation of four independent software modules: a *disassembler*, a *functional simulator*, a *detailed simulator*, and a *visual tool*.

These four software modules communicate with each other with clearly defined interfaces, but can also work independently. Each software component though requires previous (left) design modules to work as a stand-alone tool. In this manner, the detailed simulator can operate by interacting with the functional simulator and disassembler, and the functional simulator can likewise

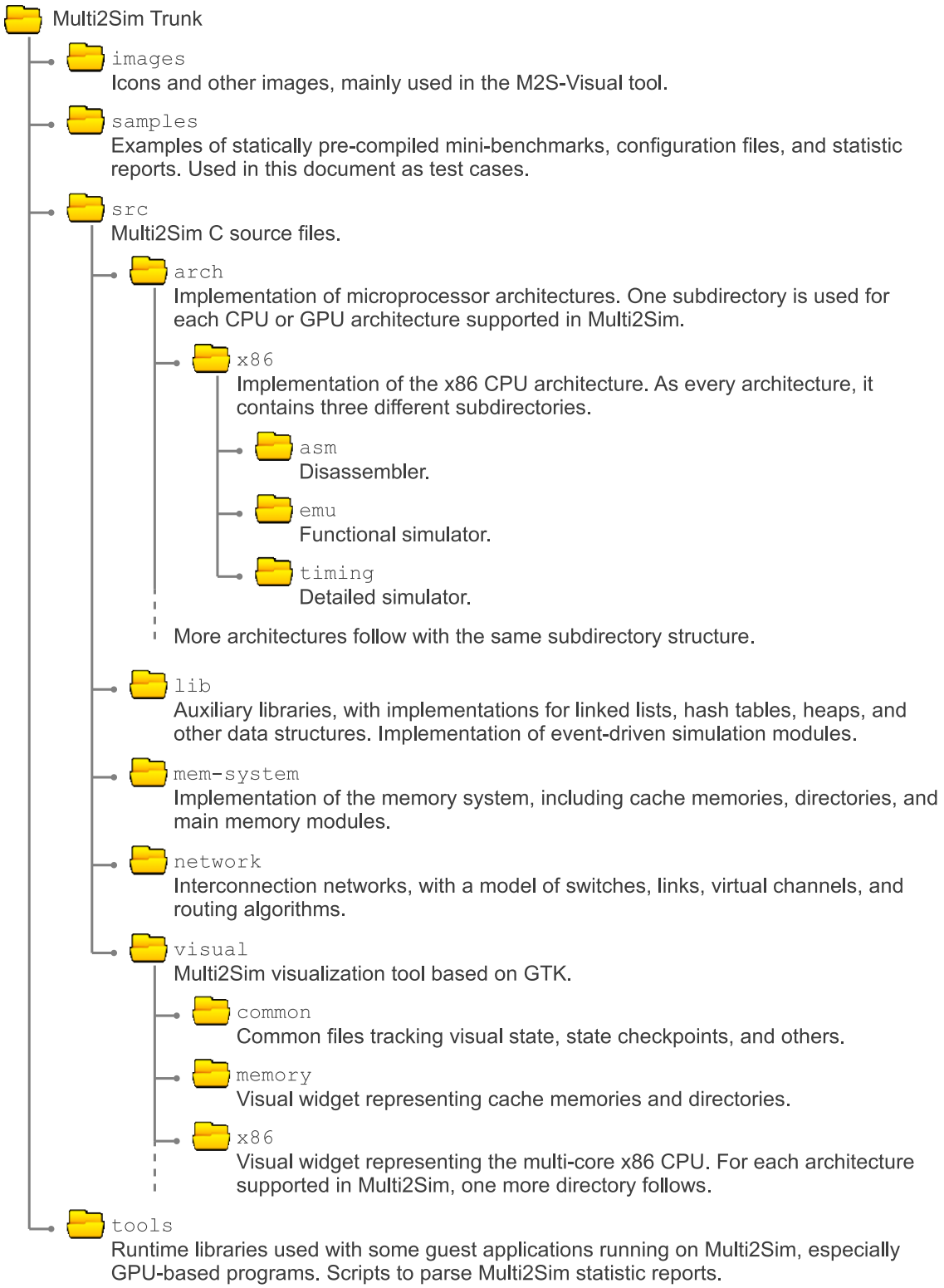


Figure 1.1: Structure of Multi2Sim's source code.

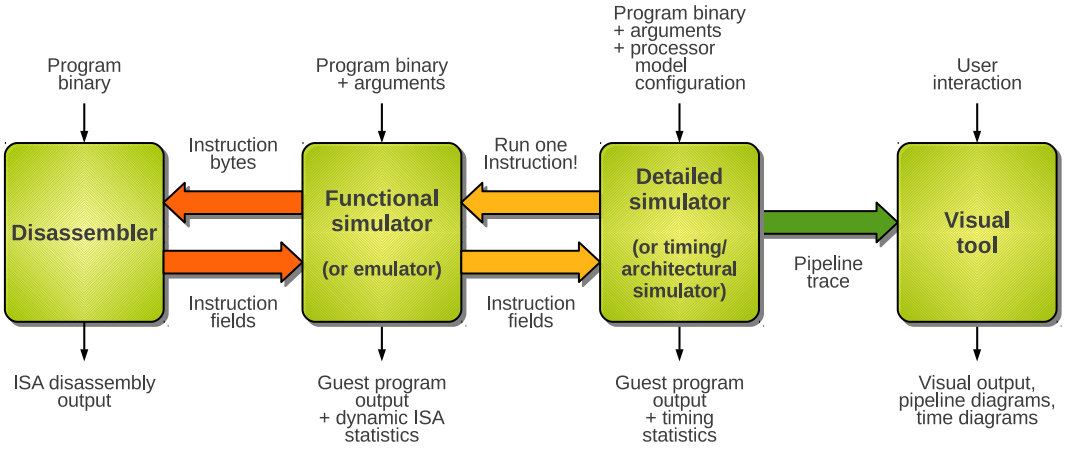


Figure 1.2: Multi2Sim's simulation paradigm for a microprocessor architecture.

be used in isolation together with the disassembler. However, it is not possible to use the visual tool without the remaining components.

1.2.1 Disassembler

Given a bit stream representing machine instructions for a specific instruction set architecture (ISA), the goal of a disassembler is to decode these instructions into an alternative representation that allows for a straightforward interpretation of the instruction fields, such as operation code, input/output operands, or immediate constants.

Multi2Sim's disassembler for a given microprocessor architecture can operate autonomously or serve later simulation stages. In the first case, the disassembler reads directly from a program binary generated by a compiler, such as an x86 application binary, and dumps a text-based output of all fragments of ISA code found in the file. In the second case, the disassembler reads from a binary buffer in memory, and outputs machine instructions one by one in the form of organized data structures that split each instruction into its comprising fields.

1.2.2 Functional Simulator

The purpose of the functional simulator, also called the *emulator*, is to reproduce the original behavior of a guest program, providing the illusion that it is running natively on a given microarchitecture. For example, an ARM program binary can run on top of Multi2Sim's ARM emulator. Even though Multi2Sim runs on an x86 architecture, the ARM guest program provides the same output as if it ran on a real ARM processor.

To accomplish this effect, an ISA emulator needs to keep track of the guest program state, and dynamically update it instruction by instruction until the program finishes. The state of a program can be most generally expressed as its virtual memory image and the architected register file. The virtual memory image consists of the set of values stored at each possible memory location addressable by the program. The state of the architected register file is formed of the values for each register defined in a specific architecture (e.g., `eax`, `ebx`, `ecx`, ... in x86).

Given a program state associated with a specific point in its execution, the emulator is capable of updating it to the next state after consuming one single ISA instruction. This process is done in 4 steps: *i*) the new instruction is read from the memory image containing the program's code at that location pointed to by the *instruction pointer* architected register, *ii*) the instruction is decoded,

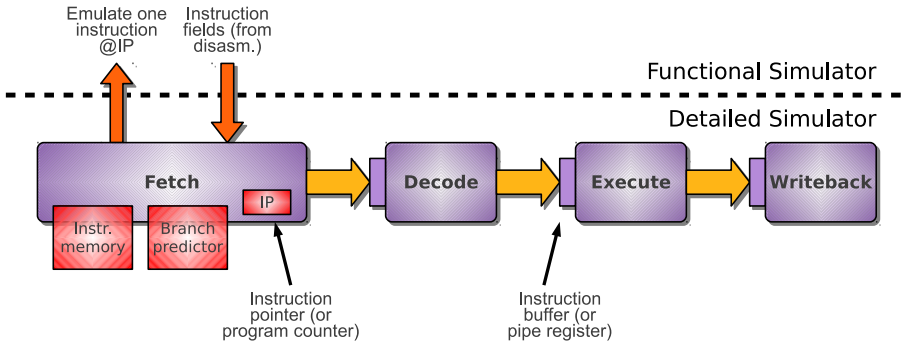


Figure 1.3: Example of a 4-stage processor pipeline, illustrating the communication between the detailed and functional simulators.

taking advantage of the interface provided by the disassembler software module, *iii*) the instruction is emulated, updated the memory image and architected registers according to the instruction's opcode and input/output operands, and *iv*) the instruction pointer is moved to the next instruction to be emulated.

When used independently, the functional simulator runs a program to completion. Initially, the simulated instruction pointer is at the *program entry*, i.e., the address of the first instruction to be executed. Then, a simulation loop keeps emulating instructions repeatedly until the program runs its termination routine. But the functional simulator also provides an interface for next simulation stages. In this case, an independent software entity can request that it emulate its next available instruction. After internally updating the guest program's state, the functional simulator returns all information related with the emulated instruction, as per the information returned by its internal call to the disassembler.

1.2.3 Detailed Simulator

The detailed simulator, interchangeably referred to as *timing* or *architectural* simulator, is the software component that models hardware structures and keeps track of their access time. The modeled hardware includes pipeline stages, pipe registers, instruction queues, functional units, cache memories, and others.

A processor microarchitecture is organized, in general, as a pipeline where each stage is devoted to a specific purpose. Symmetrically, the detailed simulator body is structured as a main loop, calling all pipeline stages in each iteration. One iteration of the loop models one clock cycle on the real hardware. While hardware structures are wholly modeled in the detailed simulator, the flow of instructions that utilize them is obtained from invocations to the functional simulator.

To illustrate the interaction between the detailed and functional simulators, let us use a 4-stage pipeline as an example, as shown in Figure 1.3. Instruction execution is assumed in order, and branch prediction is used to obtain the address of the next instruction to fetch. When the timing simulator detects free ports in instruction memory and free space in the *fetch*/*decode* pipe register, it decides to fetch a new instruction at the address dictated by the branch predictor. The timing simulator requests emulation of a new instruction to the functional simulator, after which the latter returns all information about the emulated instruction, as propagated by its internal call to the disassembler.

While the instruction travels across pipeline stages, it accesses different models of hardware resources —functional units, effective address calculators, data caches, etc.— with potentially diverse

latencies. The timing simulator also keeps track of instruction dependences: An instruction at the decode stage, for instance, could consume an operand produced by an instruction at the execute stage not committed to the register file yet. Structural and data hazards potentially cause pipeline stalls that are propagated all the way back to the fetch stage.

After the emulator runs an instruction, it internally knows exactly what is going to be the effect of that instruction in the state of the program. But the timing simulator should pretend that the instruction output is not known until later stages of the pipeline, which is critical for branch instructions. Under normal circumstances, the branch predictor provides a value for the next IP (instruction pointer) matching the internal value recorded in the emulator. But when a branch misprediction occurs, instruction emulation should begin through a wrong execution path.

An emulator with support for wrong-path execution allows the detailed simulator to force a new value for the IP. When this occurs, the emulator state is automatically checkpointed. The timing simulator then continues fetching invalid instructions, until the original mispeculated branch is resolved. At this point the contents of the pipeline are squashed, and the emulator receives the command to restore the previous checkpoint. This mechanism is used, for example, in the simulation of the x86 superscalar pipeline, described in Chapter 2.

The detailed simulator is the ultimate tool for architectural exploration studies. When used in isolation, it provides detailed hardware state and performance statistics, headed by the universal performance metric *execution time*. But the detailed simulator also allows for the generation of an exhaustive simulation trace, that comes in form of a plain-text file, and can be parsed manually or automatically in later simulation steps.

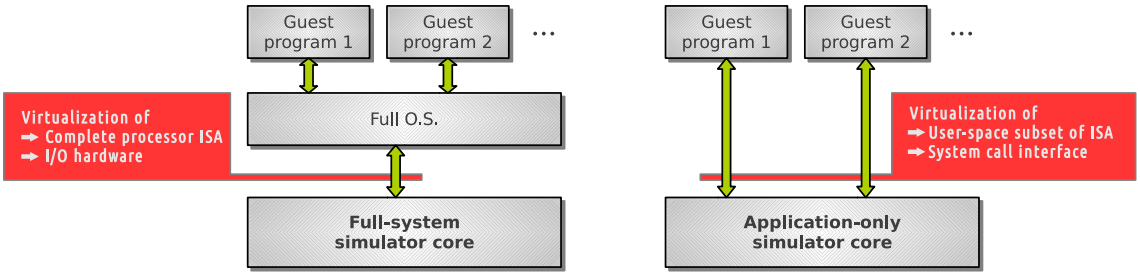
1.2.4 Visual Tool

The last software component involved in a microarchitecture model is the graphic visualization tool. As opposed to the runtime interaction scheme observed previously, the visual tool does not communicate with the detailed simulator during its execution. Instead, the detailed simulator generates a compressed text-based trace in an output file, which is consumed by the visual tool in a second execution of Multi2Sim.

The visual tool provides the user with a cycle-based interactive navigation. For each simulation cycle, one can observe the state of the processor pipelines, instructions in flight, memory accesses traversing the cache hierarchy, etc. This level of detail complements the global statistics provided by the timing simulator: Not only can one observe final performance results, but also the cause for access contention on a specific hardware resource, as well as other performance bottlenecks. More details about visualization of simulation traces can be found in Chapter 8.

1.3 Full-System vs. Application-Only Emulation

Two families of ISA emulators can be distinguished, according to the guest software they are devoted to run. A *full-system emulator* runs the entire software stack that would normally run on a real machine, including an operating system (OS), a set of device drivers, and user-level applications. On the other hand, an *application-only* emulator —Multi2Sim can be classified as such— concentrates in the execution of a user-level application, removing OS and device drivers from the software stack. A comparison between these two emulation approaches is presented next.



a) Full-system simulator b) Application-only simulator

Figure 1.4: Infrastructure of different simulation approaches. The full-system simulator boots a full-fledged operating system, on top of which guest programs run. The application-only simulator removes the intermediate layer and runs only the guest programs.

1.3.1 Full-System Emulation

A full-system emulator, represented in Figure 1.4a, begins execution by running the master-boot record of a disk image containing an unmodified operating system (guest OS). Its state is represented as the physical memory image of the modeled machine, together with the values of the architected register file. The management of guest applications is done by the guest OS, transparently to the full-system emulator.

The following software components are present in a full-system emulator: a complete implementation of the ISA specification, including user-level and privileged instructions; a virtualization service for all I/O devices, intercepting instructions from the guest OS performing I/O operations; and in most cases, a checkpointing mechanism that allows for a guest OS to start its normal operation, skipping the costly booting process.

A full-system emulator behaves similarly to a virtual machine in the way it runs a guest OS and abstracts I/O. However, an emulator intercepts every single ISA instruction to update a dedicated copy of the emulated OS state, while a virtual machine runs ISA instructions natively, taking advantage of the support in the host processor for hardware virtualization. Thus, a virtual machine can be much faster, in fact having an efficiency comparable to the non-virtualized OS execution, but it cannot capture a trace of ISA instructions to feed other software components devoted to timing simulation.

1.3.2 Application-Only Emulation

The execution scheme of an application-only emulator is represented in Figure 1.4b. As opposed to the full-system emulator, instruction emulation begins straight at the guest program entry point (i.e., initial virtual address), as encoded in the program ELF binary. In an OS, there are two main services to allow an application to run on top of it: preparation of an initial memory image for the application (process known as *program loading*), and communication between the application and OS at runtime via system calls. Since the OS is removed from the guest software stack in an application-only simulation, these two services are abstracted by the emulator itself.

The program loading process consists, in turn, of three steps. First, the application ELF binary is analyzed and those sections containing ISA instructions and initialized static data are extracted. An initial memory image is created for the guest program, copying these ELF sections into their corresponding base virtual addresses. Second, the program stack is initialized mainly by copying

the program arguments and environment variables to specific locations of the memory image. And third, the architected register file is initialized by assigning a value to the stack- and instruction-pointer registers. After program loading, emulation is ready to start with the first ISA instruction at the program entry point.

A guest program executes software interrupts to request an OS service through a system call, abstracted by the application-only emulator. When a system call is intercepted, the emulator gathers information about all its input arguments. Then, it updates its internal state as specified by the requested service, as well as the guest program's state, giving it the illusion of having executed the system call natively. For example, the invocation of system call `open` involves collecting a file name from the guest program, updating the emulator's internal file table, and returning the new file descriptor to the guest program. While the execution of a software interrupt in a native environment is equivalent to a jump to OS code, the application-only model wholly runs the system service as a consequence of one single ISA instruction emulation — the software interrupt.

An application-only emulator requires the following software components: a partial implementation of an ISA specification, not including privileged ISA instructions designed for exclusive OS use; implementation of all system calls, as specified in the Linux application binary interface (ABI); and management of process tables, file tables, signal masks, and other structures representing the internal state of an OS.

1.4 Getting Started

1.4.1 Installation

To install Multi2Sim, download the simulator source package from the home page at *www.multi2sim.org*. The package is a compressed *tar* file, that can be unpacked and compiled using the following commands, replacing `<version>` with the appropriate value:

```
$ tar -xzvf multi2sim-<version>.tar.gz
$ cd multi2sim-<version>
$ ./configure
$ make
```

If all library dependences are satisfied and compilation succeeds, the main Multi2Sim command-line tool is found at `multi2sim-<version>/src/m2s`. If you have root privileges on the machine, you can optionally install the package to make it available to all users of the system. This is not recommended though if you plan to make modifications in the source code and rebuild the simulator often. The command to install the simulator is

```
$ sudo make install
```

Multi2Sim has a rich set of command-line options that allow for simulations of complex processor designs and guest program configurations. But it also provides a simple command-line interface to launch initial test simulations. The generic command-line syntax is

```
$ m2s [<options>] [<program> [<args>]]
```

The string `<options>` represents a (possibly empty) set of command-line options that define the type of simulation to be performed, as well as the processor model to be used. All options start with a double dash (`--`), and optionally receive one or more arguments. A list of command-line options can be obtained by running command

```
$ m2s --help
```

After the last command-line option is parsed, `m2s` can receive further strings. The first string following the last option is interpreted as the executable guest program to run on Multi2Sim. All remaining strings are interpreted as arguments for the guest program. Some simple execution examples are shown next.

1.4.2 First Execution

As an initial example, let us choose a simple x86 program as our guest application running on Multi2Sim. The `samples/x86` directory in the downloaded package contains a set of mini-benchmarks that can be used as test cases. Each mini-benchmark is provided with its source code, as well as its statically compiled executable file. Let us start executing benchmark `test-args` natively, passing three argument strings to it:

```
$ ./test-args how are you
number of arguments: 4
    argv[0] = ./test-args
    argv[1] = how
    argv[2] = are
    argv[3] = you
```

As observed, `test-args` is a simple program that outputs the number of arguments passed in the command line, followed by a list of the arguments themselves. The `test-args` mini-benchmark can be run on top of Multi2Sim by just prepending `m2s` (or the full-path location of the executable, if not installed) to the command line:

```
$ m2s test-args how are you

; Multi2Sim 4.0.1 - A Simulation Framework for CPU-GPU Heterogeneous Computing
; Please use command 'm2s --help' for a list of command-line options.
; Last compilation: Nov. 30, 2012 16:44:36

number of arguments: 4
    argv[0] = test-args
    argv[1] = how
    argv[2] = are
    argv[3] = you

;
; Simulation Statistics Summary
;

[ General ]
Time = 0.01
SimEnd = ContextsFinished

[ x86 ]
SimType = Functional
Time = 0.04
Contexts = 1
Memory = 9093120
EmulatedInstructions = 9655
EmulatedInstructionsPerSecond = 254219
```

The output of the code above is split into the output coming from the simulator, provided in the standard error output `stderr` and shown red in the listing. There is also the output coming

from the guest application, provided in the standard output `stdout`, and shown in black text. Even if the guest program tries to write into `stderr`, Multi2Sim redirects the output text into the host standard output `stdout`. In Unix, standard and standard error outputs can be dumped into real or virtual files using the redirection syntax (e.g., "`> my-file`" or "`2> /dev/null`").

A format used throughout Multi2Sim's modules, both for input configuration files and output configuration reports, is the *INI file* format. An INI file is a piece of plain-text composed of section headers, variable-value pairs, and comments. A section header is a string surrounded with square brackets (e.g., "`[My-section]`"). A variable-value pair is formed of two strings separated with an equal sign, always appearing after a section header (e.g., "`Var-name = 123`"). And a comment is a line starting with a semicolon (";").

Multi2Sim's standard error output `stderr` follows the INI file format. It is used to output a summarized report of the simulation execution. The report contains a welcome string, dumped before simulation starts, and a set of statistics classified in sections, shown at the tend of the simulation. There is one section named `[General]` presenting global simulation statistics, and one individual section for each activated processor model — in this case only `[x86]`.

1.4.3 Statistics Summary

Section `[General]` of the statistics summary, presented in the standard error output at the end of a simulation, contains the following variables:

- `Time`. Total simulation time. This value is **not** a performance metric for a guest program, but just the time that Multi2Sim took to finish simulation.
- `SimEnd`. Reason for simulation end. Multi2Sim can finish execution for multiple reasons. Possible general values for this variable are:
  - `ContextsFinished`. All guest programs and their spawned threads finished execution.
  - `MaxTime`. The maximum simulation time specified with command-line option `--max-time <time>` has been reached.
  - `Signal`. Multi2Sim received a signal to stop simulation (the user pressed Ctrl+C).
  - `Stall`. Detailed simulation has stalled. All architectural simulation modules include sanity checks asserting that some constant progress must be made in the simulated pipelines. If no instruction is processed in a large amount of cycles, a stall is assumed and simulation automatically stops.

  There are additional possible values for variable `SimEnd` specific to each microprocessor architecture. These values are described in the corresponding chapter of this document.
- `Cycles`. Total number of simulation cycles. This variable is present only if at least one architecture is using detailed simulation. The concept of simulation cycle is not applicable for functional simulation.

The rest of the sections in the statistics summary correspond to each microprocessor architecture. The meaning of each variable is presented in the corresponding chapter of this document.

1.4.4 Launching Multiple Guest Programs

Using the standard command-line syntax, only one guest program can be launched on Multi2Sim at a time. In some cases, though, it makes sense to execute several applications in parallel on the simulator. For example, a model of an x86 multi-core processor can be fully stressed either using

one single multi-threaded application —a program based on *pthreads*, OpenMP, MPI, etc.—, or using several single-threaded programs, each running on a different virtual core.

To run more than one initial guest program (or context), a context configuration file should be used, passed to the simulator with command-line option `--ctx-config <file>`. The context configuration file follows the INI file format, with as many sections as initial contexts are to be launched. The section for the first context should be named `[Context 0]`, followed by section `[Context 1]`, and so on. Multi2Sim will stop processing the file once the consecutive order of context sections is broken, or obviously, if the file ends.

A section in the context configuration file accepts the following variables:

- `Exe = <path>`. Executable file containing the guest program. The presence of this variable is mandatory, while all following variables can be omitted.
- `Args = <arg_list>`. Command-line arguments for the simulated program.
- `Env = <env_list>`. Additional environment variables for the simulated program. The list of given environment variables will be accessible to the guest program, together with the set of host environment variables. Each environment variable should be given as a string in double quotes.

  Example: `Env = "var1=value1" "var2=value2"`
- `Cwd = <path>`. Current working directory for the simulated program. Whenever the simulated program uses relative paths, this will be the directory used to build absolute paths. If omitted, the host current directory will be used by default.
- `StdIn = <file>`. Standard input for the program. If omitted, the host standard input (standard input for Multi2Sim) is used by default.
- `StdOut = <file>`. Standard output and standard error output of the guest application. If omitted, the host standard output `stdout` will be used by default for both the guest standard output `stdout` and standard error output `stderr`.

The guest programs launched in the context configuration file will be part of the list of initial contexts together with the additional program specified in the command line, if any. As an example, the following listing corresponds to a context configuration file creating two contexts. The first context uses program `test-args` with three arguments, sending its output to file `context-0.out`. The second context launches program `test-sort`, dumping its output to file `context-1.out`.

```
[ Context 0 ]
Exe = test-args
Args = how are you
Stdout = context-0.out

[ Context 1 ]
Exe = test-sort
Stdout = context-1.out
```

Chapter 2

The x86 CPU Model

2.1 The x86 Simulation Paradigm

The simulation of an x86 guest program can be divided into two main modules: the *functional simulation* (or *emulation*) and the *timing* (*detailed* or *architectural* simulation. Given an executable ELF (*Executable and Linkable Format*) file, the functional simulator provides the same behavior as if the program was executed natively on an x86 machine. The detailed simulator provides a model of the hardware structures of an x86-based machine; it provides timing and usage statistics for each hardware component, depending on the instruction flow supplied by the functional simulator.

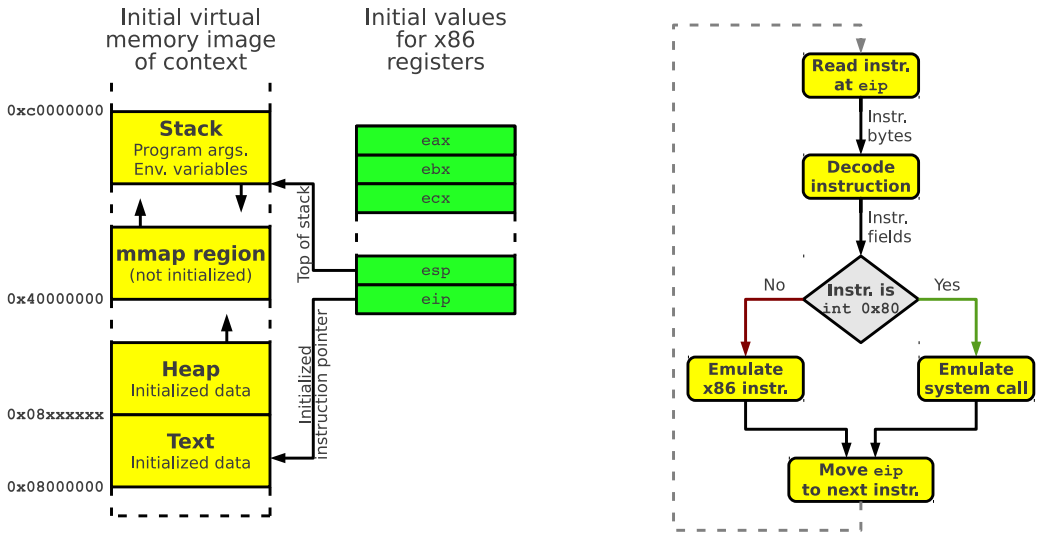
2.1.1 The x86 Functional Simulator

The executable generated when building Multi2Sim is `m2s`, which is a unified tool for both functional and detailed simulation. Optional command-line option `--x86-sim functional` enables x86 functional simulation (default configuration). This configuration provides an implementation for the functional simulation. It takes as an input one or more ELF files, and emulates their execution, providing a basic set of statistics based on the guest code run. The main actions performed by the functional simulator can be classified into *program loading*, *x86 instructions emulation*, and *system calls emulation*, described next:

- **Program Loading.** The state of a guest program's execution, referred to as *context*, is basically represented by a *virtual memory image* and a set of *logical register values* (Figure 2.1a). The former refers to the values stored in each memory location in the context's virtual memory, while the latter refers to the contents of the x86 registers, such as `eax`, `ebx`, etc.

  The Linux Application Binary Interface (ABI) specifies an initial value for both the virtual memory image and register values, before control is transferred to the new context. The initial state of the context is inferred mainly from the program ELF binary, and the command-line run by the user to launch it, during the process called *program loading*. In a real system, program loading is performed by the operating system after an `execv` system call or any of its variants. In the simulation environment, Multi2Sim is in charge of performing program loading for each guest context run on top of it. Program loading consists of the following steps:

  – First, the x86 binary is analyzed with an ELF parser. An ELF file contains sections of code (x86 instructions) and initialized data, jointly with the virtual address where they should be initially loaded. For each ELF section, the program loader obtains its virtual offset and copies it into the corresponding location in the virtual memory image.



a) Program loading.

b) Functional simulation loop

Figure 2.1: Initialization and central loop of the functional simulation of an x86 program.

  - The context stack is initialized. The stack is a region of the guest virtual memory image pointed to by register `esp`. Initially, it contains a set of program headers copied from the ELF file, followed by an array of environment variables, and the sequence of command-line arguments provided by the user.
  - The x86 registers are initialized. The `esp` register is set to point to the top of the stack, and the `eip` register is set to point to that memory location containing the code to run when control is first transfered to the new context.

- **Emulation of x86 instructions.** Once the initial image of the new context is ready, its emulation can start. Iteratively, the functional simulator reads a sequence of bytes at the guest memory address pointed to by guest register `eip`. Then, the represented x86 instruction is obtained by calling the Multi2Sim x86 decoder and disassembler. The instruction is emulated by updating accordingly the guest virtual memory image and registers[1]. Finally, guest register `eip` is updated to point to the next x86 instruction to be executed.
- **Emulation of system calls.** A special case of machine instruction is the software interrupt x86 instruction `int`. Specifically, instruction "`int 0x80`" is used to perform a system call. When Multi2Sim encounters a system call in the emulated application, it updates the context status accordingly depending on the system call code and its arguments, providing the guest program with the view of actually having performed the system call natively.

  In most cases, Multi2Sim will need to perform the same host system call as the guest program is requesting, by making a pre- and post-processing of the arguments and result, respectively. For example, when a guest program runs an `open` system call, it provides in specific x86 registers a pointer to the string containing the path to be opened, and it expects a file descriptor as a return value, also in a specific x86 register. Roughly, Multi2Sim deals with this by locating the path string in guest memory, performing its own host `open` system call, and

[1]For example, an `add` instruction would read the source operands from guest memory or registers, perform an addition, and store its result back into guest memory or registers, depending on the location of the destination operand.

placing the resulting file descriptor into guest register *eax*, where the guest context expects it to be once execution resumes at the instruction following the system call.

The high-level actions performed by the functional simulator loop are represented in Figure 2.1b, including the emulation of both x86 instructions and system calls.

2.1.2 The Detailed Simulation

Command-line option `--x86-sim detailed` enables x86 detailed simulation. Detailed simulation activates the model of an x86 CPU pipeline with support for branch prediction and speculative execution. The first stage of this pipeline (*fetch* stage) interacts with the functional simulator module or Multi2Sim kernel library, by using a simple interface. Iteratively, `m2s` asks the functional simulator to emulate the next guest x86 instruction and return some information about it. Based on this information, `m2s` can figure out which hardware structures are activated and performs a timing simulation.

The functional simulator knows at each time which is exactly the next instruction to execute. In contrast, real hardware obtains the address of the next instruction to fetch from the output of a branch predictor. This address can be correct or might be the start of a sequence of mispredicted instructions, followed by a pipeline squash and recovery. This process is modeled in `m2s` as follows.

As long as the predicted address for the next instruction matches the functional simulator statement, both functional and detailed simulation are synchronized. However, a branch misprediction will make `m2s` start fetching instructions through the wrong execution path. At this time, `m2s` forces a new value for the `eip` register in the functional simulator, which automatically checkpoints the context state. `m2s` keeps fetching instructions and forcing the functional simulator to execute them, until the mispredicted branch is resolved in an advanced stage of the pipeline. The branch resolution leads `m2s` to squash the modeled pipeline contents, and makes the functional simulator return to the last valid checkpointed state, from which correct execution is normally resumed.

2.2 The x86 CPU Statistics Summary

At the end of an simulation, Multi2Sim presents a summary of statistics in the standard error output (see Section 1.4.3) that follows the INI file format. If an x86 functional or detailed simulation took place, a section named `[x86]` is included in this report, including the following variables:

- `SimType`. Simualtion type, as specified in the command line. Possible values are `Functional` and `Detailed`.
- `Time`. Total x86 simulation time in seconds. This value includes the time in which the x86 functional/detailed simulator was active either in exclusive execution, or in parallel with other architecture simulations. It does not include the time that only other simulation modules were running.
- `Contexts`. Maximum number of active contexts during the simulation.
- `Memory`. Maximum amount of memory in bytes used in total by all contexts during the simulation.
- `EmulatedInstructions`. Total number of emulated instructions. In a detailed simulation, this value can be higher than the effective number of instructions committed in the guest program, since it also includes instructions emulated through speculative execution paths.

- `EmulatedInstructionsPerSecond`. Number of instructions emulated per second, calculated as the quotient of `EmulatedInstructions` and `Time`. This value is not a performance metric for the guest program. It is used to measure simulation speed.

The following variables are present in the statistic summary only when detailed simulation is selected in the command line:

- `Cycles`. Number of simulation cycles during which an x86 detailed simulation was active, either exclusively or in parallel with simulation of other architectures.
- `CyclesPerSecond`. Number of simulation cycles per second, calculated as the quotient of `Cycles` and `Time`. This metric does not measure performance of the guest program. It measures simulation speed.
- `FastForwardInstructions`. Total number of instructions running with fast-forward execution before the timing simulation begins, as specified in the x86 configuration file (option `--x86-config <file>`).
- `CommittedInstructions`. Number of x86 instructions committed in all x86 CPU pipelines. This value is always equal to or lower than the number of emulated instructions captured in statistic `EmulatedInstructions`.
- `CommittedInstructionsPerCycle`. Number of x86 instructions committed per cycle, calculated as the quotient of `CommittedInstructions` and `Cycles`.
- `CommittedMicroInstructions`. Number of micro-instructions committed in all x86 CPU pipelines. Since each x86 instruction generates at least one micro-instruction, this value is always equal or greater than `CommittedInstructions`.
- `CommittedMicroInstructionsPerCycle`. Number of micro-instructions committed per cycle, calculated as the quotient of `CommittedMicroInstructions` and `Cycles`. This is the guest program's performance metric formerly reported as `IPC` in Multi2Sim versions 4.0 and earlier.
- `BranchPredictionAccuracy`. Branch predictor accuracy, calculated as the number of correctly predicted divided by the total number of branch micro-instructions.

Additionally, the x86 model and its associated command-line options can cause the simulation to end. This cause is recorded in variable `SimEnd` in section `[General]` of the statistics summary. Besides those values presented in Section 1.4.3, the following additional values are possible for `SimEnd`:

- `x86LastInst`. The emulation of an x86 program has executed the last instruction, as specified in command-line option `--x86-last-inst <inst>`.
- `x86MaxInst`. The maximum number of x86 instructions has been reached, as specified in command-line option `--x86-max-inst <num>`. In functional simulation, this limit is given as the maximum number of emulated instructions. In detailed simulation, the limit is given in number of committed (non-speculative) x86 instructions.
- `x86MaxCycles`. The maximum number of x86 simulation cycles has been reached, as specified in command-line option `--x86-max-cycles <cycles>`. This cause for simulation end is only possible for detailed simulation.

2.3 Compiling and Simulating Your Own Source Code

Day after day, Multi2Sim provides a more and more robust and complete support for the x86 instruction set and Unix system calls. This means that every new release of the simulator makes it

more likely for your own compiled source files to be supported, regardless of your `gcc`, `glibc`, or Linux kernel versions. The main test scenarios for Multi2Sim have been executions of the pre-compiled benchmark suites provided in the website, but also support has been added for missing features, based on reports sent by users in the past years. The next sections show some considerations when simulating your own program sources.

2.3.1 Static and Dynamic Linking

When compiling a program, there are two main approaches to link the object files into the final executable, called *dynamic* and *static* linking. It is important to understand the characteristics of each approach and their impact on the program execution, either native or simulated.

- **Static linking**. The `gcc` linker can be configured to generate a statically linked executable by adding the `-static` option into the command line. In this case, the code of any shared library used by the program (such as the mathematic library, the POSIX thread library, `glibc` library, etc.) is linked together with the program. This code includes, for example, the implementation of the `printf` function, along with many other program initialization procedures. Even for the simplest *hello world* program, a huge executable file is generated. The advantage thereof is that this file can be used on any Linux machine with a compatible version of the kernel, regardless of the versions of the remaining installed development packages and libraries.
- **Dynamic linking**. This is the default behavior for `gcc`. When a program is linked dynamically, the library code is not attached for the final executable. Instead, every reference to an external symbol, such as the `printf` function, is left unresolved initially. The compiler adds into the executable some code to load the *dynamic loader*, which is also a dynamic library present in your system, usually under the `/etc` directory. When the program is executed, the guest code itself copies the dynamic loader code into its own context image, and then jumps into it to transfer control. Then, the dynamic loader tries to find all shared libraries required by your program (usually `*.so` files under the `/lib` or `/usr/lib` directories), and loads their code into the process image as well. Finally, control is transferred back to the program code, which continues with other initialization actions.

Based on the previous description, the following difference can be noted between the static and dynamic linking approaches, regarding the creation of the process executable code image. In the case of static linking, this initialization is exclusively performed by the program loader, implemented by the OS in a real machine, and by a simulator library in the Multi2Sim environment. In contrast, dynamically linked programs follow two steps in the code initialization: first, the OS (or Multi2Sim) creates an initial program image; second, once the program starts running, it continues to update its image by loading the shared libraries.

Thus, it is important to note that the OS (or the simulator) is not involved in the dynamic linking process. Since the update of the program image relies on the dynamic loader and dynamic libraries provided by your distribution, it is much more likely for a given pre-compiled, dynamically linked executable program to generate incompatibility issues. Thus, all benchmark packages available for download include statically linked programs.

2.3.2 Observing the Differences

This section shows a practical example to observe the implications of static versus dynamic linking in the program execution, using the Multi2Sim functional simulator. Let us base this example on the execution of a *hello world* program, stored in a file called `hello.c`, and containing the following code:

```
#include <stdio.h>

int main()
{
    printf("hello world\n");
    return 0;
}
```

First, let us generate a statically linked version of the program, and run it on top of Multi2Sim. A good clue of the program behavior is going to be given by the system calls. Similarly to the output provided by the `strace` tool, a trace of the performed system calls, their arguments, and return values can be obtained with Multi2Sim by using command-line option `--x86-debug-syscall <file>`, where `<file>` is the file name of the file where the trace is dumped. If `stdout` is specified, it dumps the trace into the standard output:

```
$ gcc hello.c -o hello -static
$ m2s --x86-debug-syscall stdout hello

syscall 'newuname' (code 122, inst 418, pid 1000)
syscall 'brk' (code 45, inst 738, pid 1000)
syscall 'set_thread_area' (code 243, inst 850, pid 1000)
     [...]
syscall 'open' (code 5, inst 911, pid 1000)
  filename='/dev/urandom' flags=0x0, mode=0x0
  return=0x3
syscall 'read' (code 3, inst 932, pid 1000)
  guest_fd=3, pbuf=0xfffdffbd, count=0x3
  return=0x3
syscall 'close' (code 6, inst 948, pid 1000)
  guest_fd=3
  return=0x0
     [...]
syscall 'fstat64' (code 197, inst 7973, pid 1000)
  fd=1, pstatbuf=0xfffdfe58
  return=0x0
syscall 'mmap2' (code 192, inst 8028, pid 1000)
  addr=0x0, len=4096, prot=0x3, flags=0x22, guest_fd=-1, offset=0x0
  prot={PROT_READ|PROT_WRITE}, flags={MAP_PRIVATE|MAP_ANONYMOUS}
  return=0xb7fb0000
syscall 'write' (code 4, inst 8881, pid 1000)
  guest_fd=1, pbuf=0xb7fb0000, count=0xc
  buf="hello world\n"
  return=0xc
syscall 'exit_group' (code 252, inst 9475, pid 1000)
```

The system call trace should look similar to the one shown above (it may vary across systems or even executions). We can observe that the program is retrieving some kernel information (`newuname`), updating the heap size and allocating memory (`brk`, `mmap2`), getting some random numbers for initialization purposes (`open`, `read`, `close`), getting information about the standard output (`fstat64`), displaying the *hello world* string (`write`), and exiting the program (`exit_group`). Now let us try the same with the dynamically linked version of the program:

```
$ gcc hello.c -o hello
$ m2s --x86-debug-sysx86-debug-sysx86-debug-sysx86-debug-sysx86-debug-sysx86-debug-sysx86-debug-syscal

syscall 'brk' (code 45, inst 1122, pid 1000)
syscall 'newuname' (code 122, inst 2499, pid 1000)
syscall 'open' (code 5, inst 6906, pid 1000)
  filename='/etc/ld.so.cache' flags=0x0, mode=0x0
  return=0x3
syscall 'fstat64' (code 197, inst 6931, pid 1000)
  fd=3, pstatbuf=0xfffdf924
  return=0x0
syscall 'mmap2' (code 192, inst 6967, pid 1000)
  addr=0x0, len=61684, prot=0x1, flags=0x2, guest_fd=3, offset=0x0
  prot={PROT_READ}, flags={MAP_PRIVATE}
  host mapping created for '/etc/ld.so.cache'
    host_ptr=0xb77e9000
    host_fd=5
  return=0xb7fa0000
syscall 'close' (code 6, inst 6983, pid 1000)
  guest_fd=3
  return=0x0
syscall 'open' (code 5, inst 8171, pid 1000)
  filename='/lib/libc.so.6' flags=0x0, mode=0x0
  return=0x3
syscall 'read' (code 3, inst 8192, pid 1000)
  guest_fd=3, pbuf=0xfffdfa58, count=0x200
  return=0x200

     [...]

syscall 'mprotect' (code 125, inst 80556, pid 1000)
  start=0xb7f9a000, len=0x2000, prot=0x1
  return=0x0
syscall 'write' (code 4, inst 91187, pid 1000)
  guest_fd=1, pbuf=0xb7faf000, count=0xc
  buf="hello world\n"
  return=0xc
syscall 'exit_group' (code 252, inst 92139, pid 1000)
  return=0x0
```

The trace above is an excerpt of the system calls obtained with the dynamically linked version of the program. It can be observed that the program issues a couple of `open` system calls for files `/etc/ld.so.cache` and `/lib/libc.so.6`, followed by `read`, `mmap`, and `mprotect` calls, aimed at updating the guest memory image for the context code, and assigning execution permissions to it.

The overhead of dynamic linking can be observed in the value of variable `Instructions` in section `[x86]` of the statistics summary at the end of the simulation in both cases. While the statically linked program runs around 10K instructions, the dynamically linked version needs about 100K instructions when adding the overhead for the initialization.

2.3.3 Execution Variability

A frequent question that Multi2Sim users come up with is the cause for unexpected variability among program executions. This variability might occur even if the program is run in the same machine, by the same user, etc., and just a few of seconds after the previous run.

A reason for this can be found in the first system call trace shown above, corresponding to the statically linked program. As observed, one of the multiple initializations performed by the program libraries, and specifically by `glibc`, involves getting a sequence of random bytes by reading file `/dev/urandom`. Even though we do not really care what `glibc` needs this random numbers for,

we should assume that the following instructions will use these numbers to perform some actions, which will vary depending on the actual values read. This would be a possible cause for slight execution variability.

Another frequent source of variability can occur when the same program is executed in different paths. There is an environment variable called "_" (underscore) that represents the current working directory. When an `execv` system call (real system) or the program loader (Multi2Sim environment) initializes a process image, the environment variables are copied into the new process's stack. It could perfectly happen that a specific version of `glibc` analyzes these variables by, for example, performing a `strlen` operation on each of them for any purpose. In this case, the number of iterations for each string would vary depending on the string lengths. Thus, the total number of instructions executed in the program might depend on the current directory length, among others.

In conclusion, a small variability among different execution instances of the same program should be considered as a normal behavior, even if all "environment conditions" seem to be exactly the same. Notice that this is not a behavior exclusively present in a simulated program, but also incurred by a program's native execution on the host machine.

2.3.4 Error Messages when Simulating your Program Binaries

A very extensive implementation is provided in Multi2Sim for both the x86 instruction set and some more than the most common Unix system calls. The implementation of these features is based on the usage of both instructions and system calls by the supported benchmarks and tested executables. At the current point, many different versions of `gcc` have been used to generate executables, including different versions of shared libraries and Linux kernels. During this process, new instructions and system calls have been added, providing a pretty stable and complete version for the functional simulation of a program.

However, it is possible that your specific combination of shared libraries and `gcc` compiler generate an executable file that includes a specific instruction or system call that is not supported by Multi2Sim. When you try to perform a functional simulation of the unsupported program, you would obtain an error message like this:

```
fatal: context 1000 at 0x0804f800: instruction not implemented: 0f a1 b0 ...
```

— or —

```
fatal: not implemented system call 'tgkill' (code 270) at 0x0802010
```

In any of these cases, don't give up! We are very interested in increasing the support for any program executable that a user might need to run. So please send an email to `development@multi2sim.org`, and the problem will be solved as soon as possible. Since the origin of the simulator, the fact of providing a complete implementation of the x86-based Linux ABI has been a priority for its development.

2.4 The Processor Pipeline

Figure 2.2 shows a block diagram of the processor pipeline modeled in Multi2Sim. The gray-painted boxes represent hardware structures, whereas the round shapes represent pipeline stages. Six stages are modeled in Multi2Sim, named *fetch*, *decode*, *dispatch*, *issue*, *writeback*, and *commit*.

In the fetch stage, instructions are read from the instruction or the trace cache. Depending on their origin, they are placed either in the *fetch queue* or the *trace queue*. The former contains raw

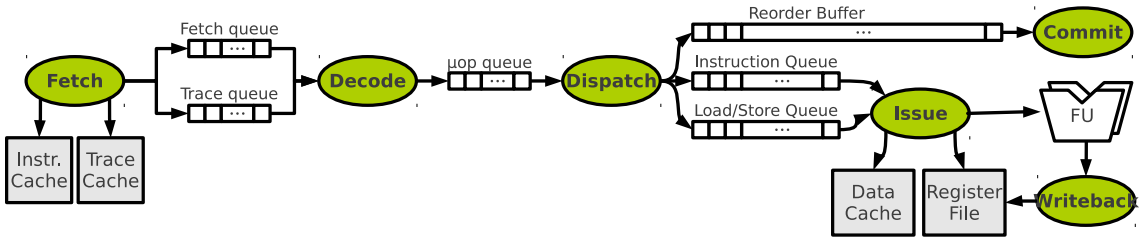


Figure 2.2: Processor pipeline.

macroinstruction bytes, while the latter stores pre-decoded microinstructions (uops). In the decode stage, instructions are read from these queues, and decoded if necessary. Then, uops are placed in program order into the *uop queue*. The fetch and decode stages form the front-end of the pipeline.

The dispatch stage takes uops from the uop queue, renames their source and destination registers, and places them into the *reorder buffer* (ROB) and the *instruction queue* (IQ) or *load-store queue* (LSQ). The issue stage is in charge of searching both the IQ and LSQ for instructions with ready source operands, which are schedule to the corresponding functional unit or data cache. When and uop completes, the writeback stage stores its destination operand back into the register file. Finally, completed uops at the head of the ROB are taken by the commit stage and their changes are confirmed.

A detailed report of the simulation statistics related with the processor pipeline can be obtained with option `--x86-report`. Please refer to Section 2.21 for a detailed description of its format and meaning.

2.5 Branch Predition

There are two different components involved in branch prediction: the Branch Target Buffer (BTB) and the branch predictor itself. The BTB is a set-associative cache indexed by a macroinstruction address. If an address is present, i.e., the corresponding entry contains a tag equals to the address, the contents of the entry specify the target of the branch. Moreover, it conveys additional information, such as the type of branch: conditional, unconditional (or jump), call, or return. The variables to specify the BTB characteristics are `BTB.Sets` and `BTB.Assoc` in section `[BranchPredictor]`. The argument `BTB.Sets` is a power of 2 indicating the number of sets of the BTB, while `BTB.Assoc` refers to the number of ways or associativity of the BTB, also a power of 2.

On the other hand, the branch predictor provides the direction of a branch located at a given address, i.e., whether it is taken or not. The branch predictor kinds modeled in Multi2Sim are *Perfect*, *Taken*, *NotTaken*, *Bimodal*, *TwoLevel*, and *Combined*. In the processor front-end, branch instructions are identified by accessing the BTB. Afterwards, the branch predictor states if the branch is actually taken or not. The branch predictor type is specified by means of the variables `Kind = {Perfect|Taken|NotTaken|Bimodal|TwoLevel|Combined}`. Each type of predictor is described next.

2.5.1 Perfect branch predictor

The *perfect* predictor (variable `Kind = Perfect`) provides a totally accurate prediction with a 100% hit ratio. Calls to the BTB-related functions always return the correct target address even if the branch has not been committed before, and calls to the branch predictor functions always return the right direction.

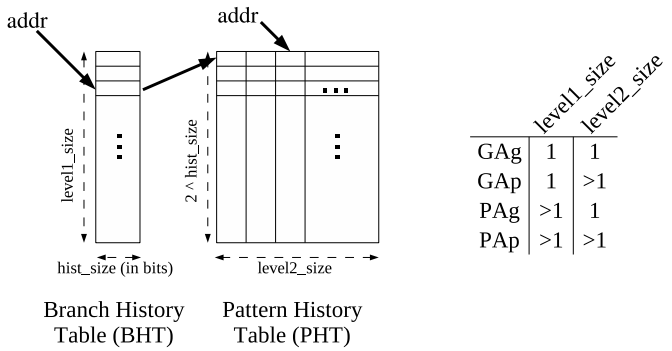


Figure 2.3: Two-level adaptive branch predictor.

2.5.2 Taken branch predictor

The *taken* predictor (variable `Kind = Taken`) assumes that branches are always taken. However, those branches whose target address is not contained in the BTB will not access the branch predictor, and the flow of fetched instructions will continue as if the branch had not been taken. Finally, the *not-taken* predictor assumes that conditional branches are never taken. However, this predictor is smart enough to consider as taken those branches that are certainly known as such, that is, unconditional branches, calls, and returns.

2.5.3 Bimodal branch predictor

A *bimodal* predictor (variable `Kind = Bimodal`) is a table indexed by the least significant bits of an instruction address. The entries of the table are 2-bit up-down saturating counters. A counter represents the current prediction for a given branch. Values of 0 and 1 represent a not-taken prediction, while values 0 and 2 mean that the branch is taken. The number of entries in the table is a power of 2 given by the variable `Bimod.Size`.

2.5.4 Two-level adaptive predictor

A *two-level adaptive* predictor (variable `Kind = TwoLevel`) two tables, each corresponding to one prediction level. There are three variables involved with this predictor, namely `TwoLevel.L1Size`, `TwoLevel.L2Size`, and `TwoLevel.HistorySize`, which defines specific parameter of the predictor components.

As shown in Figure 2.3, the first accessed table is the Branch History Table (BHT). This table is indexed by the least significant bits of the branch instruction address, and contains `TwoLevel.L1Size` entries (power of 2). Each entry contains a branch history register of `TwoLevel.HistorySize` bits that indicates the behavior of the last `TwoLevel.HistorySize` occurrences of the branch. Every time a branch commits, this register is shifted left, and the least significant bit is set or cleared according to whether the branch was actually taken or not.

The contents of the history register obtained from the BHT is used to index the row of a second two-dimensional table called Pattern History Table (PHT). Because the history register has `TwoLevel.HistorySize` bits, the PHT is forced to have 2^`TwoLevel.HistorySize` entries. The column of the PHT is also indexed by the least significant bits of the branch instruction address. The number of columns in the PHT is given by the `TwoLevel.L2Size` parameter. Each entry in the PHT contains a 2-bit up-down saturating counter that gives the final prediction for the inquired branch.

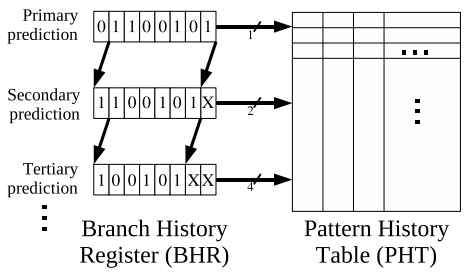


Figure 2.4: Multiple branch prediction.

By properly tunning the variables described above, one can form the four two-level adaptive configurations commonly known as GAg, GAp, PAg, and PAp. See [1] for a more detailed description about these predictors. The table shown on the right of Figure 2.3 lists the restrictions that need to fulfill the predictor parameters in order to be classified as each of the cited configurations.

2.5.5 Combined predictor

The *combined* predictor (option `Kind = Combined`) combines the bimodal and the two-level adaptive predictors. On an inquiry, both components are looked up, and their corresponding predictions are temporarily stored. Then, an additional table, called choice predictor, is accessed to decide whether to obey to the bimodal predictor statement or to the two-level predictor statement. The variable `Choice.Size` specifies the number of entries in the choice predictor (power of 2).

Each entry contains a 2-bit saturating counter. If its value is 0 or 1, the statement of the bimodal predictor is considered. If its value is 2 or 3, the two-level predictor is used to give the final prediction. The choice predictor counters are updated at the commit stage only in the case that the bimodal and the two-level predictors gave a contradicting prediction when looked up.

2.6 Multiple Branch Prediction

Multi2Sim supports the prediction of multiple non-consecutive branches in the same cycle by using the so-called *multiple branch prediction* algorithms, as proposed in [2]. This advanced prediction scheme is required by some other microarchitectural improvements, such as the trace cache. If multiple non-contiguous basic blocks are fetched in the same cycle, it is necessary to predict the behavior of the branches located at the end of each of these blocks.

The branch at the end of the first basic block is referred to as *primary branch*, while the branches located at the end of the second and third basic blocks following the execution path are called *secondary* and *tertiary branch*, respectively. To make multiple prediction, a branch predictor is needed which uses a branch history register. In Multi2Sim, multiple branch prediction is allowed only when a two-level adaptive predictor is used.

Figure 2.4 shows a working scheme of multiple branch prediction. First, a branch history register (BHR) is used to make the primary prediction. In the case of the two-level predictor, this register is obtained either straightforwardly from the first level when it uses global history, or by indexing the branch history table (BHT) using the branch address when it uses per-address history. With the obtained BHR, the PHT is indexed in the second level to obtain a counter that gives the inquired prediction. This is the output for the primary branch.

To predict the second branch, the BHR is left shifted, and two new BHRs are obtained by filling the least significant position with 0 and 1. These BHRs are now used to index the PHT and obtain

the predictions for the two possible secondary branches. The choice between these predictions is performed according to the output of the primary branch prediction.

The tertiary branch is again predicted by left shifting the BHR two positions, and obtaining four predictions from the PHT, corresponding to the four possible paths that can reach four different tertiary basic blocks and branches. The choice between these predictions is made by means of the output of the primary and secondary predictions. This mechanism can be generalized to any number of additional predictions.

There is a chain connection between the outputs of the primary, secondary, tertiary, etc., predictions. However, notice that the accessed to the PHT are performed simultaneously. Moreover, those predictions requiring multiple PHT accessed (e.g., four of a tertiary prediction) always read consecutive positions in the PHT, since only least significant bits in the BHR differ.

2.7 CISC Instructions Decoding

The x86 architecture defines a CISC instruction set [3]. Each single instruction defined in the x86 ISA has a size between 1 and 15 bytes, and can perform a wide set of different actions. Some of these instructions involve several complex actions, such as a memory read followed by an arithmetic computation and a memory write. Such complex instructions are internally decoded as separate micro-instructions (*uops*). The set of possible uops may vary among x86 implementations, and Multi2Sim defines its own uop set, listed in Appendix IV.

Each uop has a maximum of four output dependences, formed of logical registers and status flags. There can be at the most one logical register in this set. The maximum number of input dependences for an uop is three, considering logical registers and status flags, among which a maximum of two elements can be logical registers.

Next, some examples are shown to illustrate the modeled x86 instruction decoding mechanism. In each case, an x86 instruction is given, followed by the microcode (uop) sequence that it generates.

- `mov edx, DWORD PTR [ebx-0x4]`

  This instruction reads the 32-bit value at the address pointed by register `ebx` minus 4, and stores it in register `edx`. The generated microcode has to *i*) calculate the effective memory address based on the `ebx` register, *ii*) load the value at this address, and *iii*) store the result in `edx`. Steps *ii* and *iii* can actually be done with a single uop, by just placing the result loaded from memory directly into the corresponding register. This is the generated sequence of uops:

  ```
  effaddr ea/ebx
  load edx/ea [0x8004300,4]
  ```

  The arguments for uops are logical registers (dependences). They are presented as `odep1,odep2,.../idep1,idep2,...`, where `idepXX` is an input dependence, and `odepYY` is an output dependence. Register `ea` is an internal logical register used to store results of effective address computations. Memory uops (`load` and `store`) are followed by a tuple `[addr,size]`, indicating the accessed memory location and the access size.

- `add DWORD PTR [ebx+0x8], eax`

  This is an example of a complex instruction requiring to load a value from memory location pointed to by `ebx` plus 8, add this value with the contents of register `eax`, and store the result back into memory. In the microcode sequence below, notice how the effective memory address

is reused for the memory read and subsequent write. The `data` register is in this case another temporary register used to store data for memory uops.

```
effaddr ea/ebx
load data/ea [0x8004300,4]
add data/data,eax
store -/data,ea [0x800430,4]
```

- `rep movsb`

  Finally, let us consider a complex x86 string operation. The `movsb` instruction copies one byte from the memory location pointed to by register `esi` into the memory location pointed to by `edi`. The additional prefix `rep` causes these actions to be repeated as many times as the value in register `ecx` specifies, while in each iteration the value of `esi` and `edi` is incremented (or decremented, depending on the value of flag `DF`) by 1. One iteration of the loop generates the following microcode:

  ```
  load aux/edi [0x80d789f,1]
  store -/esi,aux [0x80d789c,1]
  add edi/edi,df
  add esi/esi,df
  sub ecx/ecx
  ibranch -/ecx
  ```

  The `load` instruction places in temporary register `aux` the value read from address `edi`, which is stored next at address `esi`. Then, registers `esi` and `edi` are incremented using two `add` uops, the value in `ecx` is decremented, and a branch occurs to the first instruction of the microcode depending on the value of `ecx`.

  When a generic string operation is decoded, the number of iterations might not be known in advance. Thus, the decode stage will keep decoding iterations, assuming that uop `ibranch` always jumps to the beginning of the loop. This will cause a continuous flow of uops into the pipeline, and a final burst of mispredicted uops after the last iteration is decoded. When the last `ibranch` instruction (first non-taken branch) is resolved, all subsequent uops are squashed, and execution resumes at the x86 instruction following the string operation.

  Figure 2.5 shows an example of a timing diagram generated from the execution of a `rep movsb` instruction. Time diagrams can be generated automatically using the M2S-Visual tool (see Chapter 8).

2.8 Trace Cache

Multi2Sim models a trace cache roughly with the design proposed originally in [4]. The aim of the trace cache is to provide a sequence of predecoded x86 microinstructions with intermingled branched. This increases the fetch width by enabling instructions from different basic blocks to be fetched in the same cycle. The trace cache model is implemented in the `tcache.c` file. It is activated with the variable `Present` in section `[TraceCache]` in the x86 CPU configuration file (option `--x86-config`), and the number of sets and associativity of the trace cache are tuned with the variables `Sets` and `Assoc`, respectively. Additionally, the variables `TraceSize` and `BranchMax` specify the maximum number of microinstructions and branches in a single trace cache line, respectively. Additionally, option `QueueSize` controls the size of the queue where predecoded instructions are placed after fetching (more details in Section 2.9).

Besides the microinstruction data, each trace cache line contains the following attached fields:

Figure 2.5: Timing diagram for the execution of the `rep movsb` x86 string operation.

- *valid* bit: bit indicating whether the line contains a valid trace.
- *tag*: address of the first microinstruction in the line. This is always the first one from the set of microinstructions belonging to the same macroinstruction. If a macroinstruction is decoded into more than *trace_size* microinstructions, they cannot be stored in the trace cache.
- *uop_count*: number of microinstructions in the trace.
- *branch_count*: number of branches in the trace, not including the last microinstruction if it is a branch.
- *branch_flags*: bit mask with predictions for branches.
- *branch_mask*: bit mask indicating which bits in *branch_flags* are valid.
- *fall_trough*: address of the next trace to fetch.
- *target*: address of the next trace in case the last microinstruction is a branch and it is predicted taken.

2.8.1 Creation of traces

Traces are created non-speculatively at the so-called *fill unit*, after the commit stage. In this unit, a temporary trace is created and dumped into the trace cache when the number of stored microinstructions exceeds *trace_size* or the number of branches reaches *branch_max*. The following algorithm is used every time an instruction I commits.

If the trace is empty, the address of I is stored as the trace *tag*. If I is a branch, the *fall_through* and *target* fields are stored as the address of the contiguous next instruction, and the branch target if it is taken, respectively. This information is needed only in case this branch is the last microinstruction stored in the trace line. Then, the predicted direction is added to *branch_flags* in the position corresponding to bit *branch_count*. The mask *branch_mask* indicates which bits in *branch_flags* are valid. Thus, the bit position *branch_count* in *branch_mask* is set in this case. Finally, the counters *uop_count* and *branch_count* are incremented.

If I is not a branch, the *fall_through* field is updated, and the *target* field is cleared. Likewise, the counter *uop_count* is incremented.

Before adding I the temporary trace, it is checked whether I actually fits. If the maximum number of branches of microinstructions is exceeded, the temporary trace is first dumped into a new allocated trace cache line. When this occurs, the *target* field is checked. If it contains a value other than 0, it means that the last instruction in the trace is a branch. The prediction of this branch must not be included in the branch flags, since the next basic block is not present in the trace. In contrast, it will be used to fetch either the contiguous or the target basic block. Thus, the last stored bit in both *branch_flags* and *branch_mask* is cleared, and *branch_count* is decremented.

2.8.2 Trace cache lookups

The trace cache is complementary to the instruction cache, i.e., it does not replace the traditional instruction fetch mechanism. To the contrary, the trace cache simply has priority over the instruction cache in case of a hit, since it is able to supply a higher number of instructions along multiple basic blocks.

The trace cache modeled in Multi2Sim is indexed by the *eip* register (i.e., the instruction pointer). If the trace cache associativity is greater than 1, different ways can hold traces for the same address and different branch prediction chains. The condition to extract a microinstruction sequence into the pipeline, i.e. to consider a trace cache hit, is evaluated as follows.

First, the two-level adaptive branch predictor is used to obtain a multiple prediction of the following *branch_max* branches, using the algorithm described in Section 2.6. This provides a bit mask of *branch_max* bits, called *pred*, where the least significant bit corresponds to the primary branch direction. To check if a trace is valid to be fetched, *pred* is combined with the trace *branch_mask* field. If the result is equal to the trace *branch_flags* field, a hit is detected.

On a trace cache hit, the next address to be fetched is updated as follows. If the trace *target* field is 0, the *fall_through* field is used as next address. If *target* is non-0, the last predecoded microinstruction in the trace is a branch. In this case, the bit *branch_count*+1 in the *pred* bit mask is used to choose between *fall_through* and *target* for the next fetch address.

2.8.3 Trace cache statistics

When the trace cache is activated, a set of statistics are attached to the pipeline report (option `--x86-report`). Since each hardware thread has its private trace cache, there is a different set of statistics prefixed with the `TraceCache` identifier in each thread section (`[c0t0]`, `[c0t1]`, etc). The following list gives the meaning of each reported statistic related with the trace cache:

- `TraceCache.Accesses`. Number of cycles when the trace cache is looked up for a valid trace in the fetch stage. This is not necessarily the same as the number of execution cycles, since trace cache accesses are avoided, for example, when the trace queue is full.
- `TraceCache.Hits`. Number of accesses to the trace cache that provided a hit for the given fetch address and branch sequence prediction. Notice that this does not imply that the fetched micro-instructions are in the correct path, since the branch prediction used to access the trace cache might have been incorrect.
- `TraceCache.Fetched`. Number of micro-instructions fetched from traces stored in the trace cache.
- `TraceCache.Dispatched`. Number of micro-instructions fetched from the trace cache and dispatched to the reorder buffer.
- `TraceCache.Issued`. Number of micro-instructions fetched from the trace cache and issued to the functional units for execution.
- `TraceCache.Committed`. Number of micro-instructions in the correct path that came from a trace cache access and have committed. These are a subset of the micro-instructions included in the `Commit.Total` statistic for the same hardware thread section.
- `TraceCache.Squashed`. Number of micro-instructions fetched from the trace cache that where dispatched and later squashed upon a branch misprediction detection. This number is equal to `TraceCache.Dispatched` − `TraceCache.Commited`.
- `TraceCache.TraceLength`. Average length of the traces stored in the trace cache. This is a value equal or (probably) lower than the trace cache line size. Since micro-instructions from the same macro-instruction cannot be split among traces, a trace may be dumped into the trace cache before being full. The fact that there is a maximum number of allowed branches in a trace is an additional reason for having traces shorter than the maximum trace length.

2.9 The Fetch Stage

The fetch stage is the first pipeline stage modeled in Multi2Sim. It is in charge of fetching instructions either from the instruction cache or from the trace cache at the addresses provided by the branch predictor. The fetched instructions are used to fill the fetch queue and the trace queue,

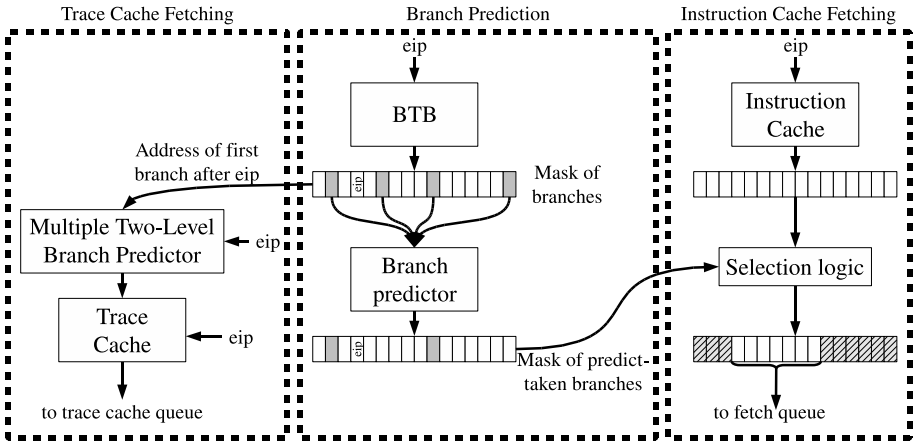


Figure 2.6: The fetch stage.

depending on the structure they were fetched from. Figure 2.6 shows a block diagram of the fetch stage.

The fetch stage is divided into three main parts, as shown in its block diagram, called *branch prediction*, *instruction cache fetching*, and *trace cache fetching*. The branch prediction part provides information about the branches located within the fetched block, and this information is sent to the instruction cache and trace cache fetching parts. The two latter work in parallel, and priority is given to the trace cache part in case it is able to fetch the requested block. The modeled fetching mechanism works as follows:

- *i*) First of all, the BTB is accessed with the current instruction pointer, i.e., the *eip* register. Though the number of associative ways of the BTB can be specified in a command-line option, it is assumed to have as many interleaved ways as the size of the instruction block size in bytes. In Figure 2.6, this value is set to 16. This means that 16 concurrent accesses can be performed in parallel to the BTB, as long as no pair of accesses matches the same interleaved way, which is true as only contiguous addresses belonging to the same block are looked up.
- *ii*) The concurrent accesses to the BTB provide a mask of those instructions known to be branches, jointly with their corresponding target addresses. The branch predictor is next looked up to obtain the predicted direction for the branches, i.e., whether they are taken or not. Since the BTB also provides the type of branch, the branch predictor will consider this information for its output. This means that an unconditional branch will always provide a predict-taken output, and function calls and returns will access the Return Address Stack (RAS) to obtain the actual target addresses. After the access to the branch predictor, the input mask is converted to an output mask that only tracks those taken branches.
- *iii*) In parallel with *i*), the instruction cache is accessed in the instruction cache fetching part (right block in Figure 2.6). After a variable latency, depending on whether there was a cache hit or miss, the cache block is available, and the mask provided by the branch prediction part is used to select the useful bytes. Specifically, a selection logic takes those bytes ranging from the address contained in register *eip* until the address of the first predict-taken branch, or until the end of the block if there was none.

  The filtered bytes are then placed into the fetch queue, which communicates the fetch stage with the next pipeline stage. After fetching, the *eip* register is set either to the starting address of the next block if no predict-taken branch was found, or to the target address of

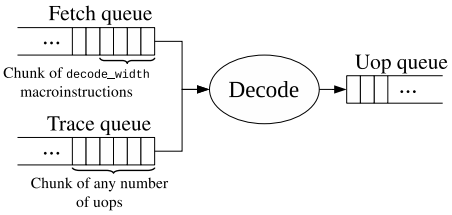


Figure 2.7: The decode stage.

the first taken branch, as provided by the BTB. Notice that data fetched directly from the instruction cache is formed of x86 complex instructions (or macroinstructions) of variable size, which are not straightforwardly interpretable by the processor. Instead, they must be first split into fixed-size microinstructions (or *uop*s) in the decode stage.

- *iv*) An alternative to *iii*) and more efficient fetch mechanism is provided by the trace cache fetching part. It works in parallel with the instruction cache access, and can start as soon as the BTB access completes and the mask of branches is available. With this mask, the address of the first branch after *eip*, be it taken or not, is grabbed to feed a multiple two-level branch predictor (see Section 2.6). This component gives a prediction for the next group of branches regardless of whether they are placed in the same or separate blocks.

  The multiple output prediction, jointly with the *eip* register, is used to index the trace cache. On a hit, the trace cache provides a sequence of predecoded microinstructions, which may span across multiple blocks and have various intermingled branches. This trace is placed into the trace cache queue, which also communicates with the next pipeline stage. However, notice that this queue stores decoded microinstructions, and thus, they require a different and more lightweight handling in the decode stage.

2.10 The Decode Stage

In the decode stage (Figure 2.7), instructions are taken either from the fetch queue or from the trace queue. Instructions coming from the fetch queue are decoded and placed into the uop queue. Instructions coming from the trace queue were fetched from the trace cache. These are predecoded instructions stored as uops, and can be copied straightforwardly into the uop queue.

A single decode cycle can perform the following actions: *i*) decode as many instructions from the fetch queue as the decode bandwidth allows (specified by the variable `DecodeWidth` in section `[Pipeline]` in the x86 CPU configuration file, option `--x86-config`) and place them into the uop queue, and *ii*) copy any number of subsequent predecoded uops from the trace queue into the uop queue.

2.11 Integer Register Renaming

2.11.1 Logical Registers

The register renaming mechanism implemented in Multi2Sim uses a simplification of the x86 logical registers. There are 32 possible logical dependences between microinstructions, which are listed in Figure 2.8a. Logical registers `eax...edz` are general purpose registers used for computations and intermediate results. Registers `esp...edi` are specific purpose registers implicitly or explicitly modified by some microinstructions, such as the stack pointer or base pointer for array accesses.

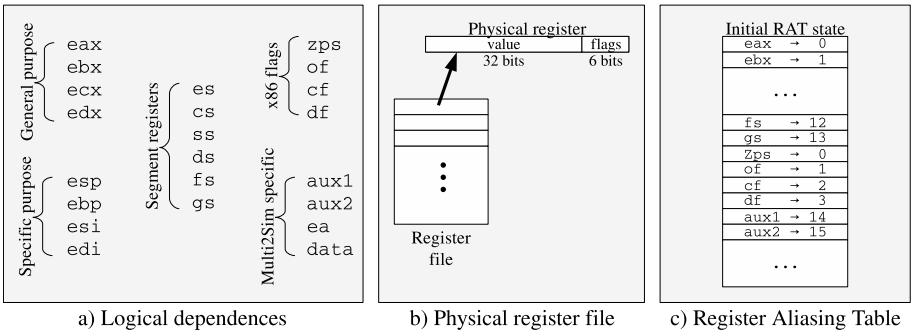


Figure 2.8: Register renaming.

Registers `es...gs` are segment registers, while `aux1...data` are internally used by the macroinstruction decoder to communicate corresponding microinstructions with one another.

The x86 architecture uses a set of flags that are written by some arithmetic instructions, and later consumed mainly by conditional branches to decide whether to jump or not. Flags `of`, `cf`, and `df` are the overflow, carry, and direction flags, respectively, and are tracked as separate dependences among instruction. On the other hand, flags `zf`, `pf`, and `sf` are the zero, parity, and sign flags, respectively, and any x86 instruction modifying any of these three flags is modifying all of them. Thus, they are tracked as a single dependence, called `zps`.

2.11.2 Physical Register File

The value associated with each logical register, i.e., each potential input dependence for an instruction, is stored in the physical register file. As represented in Figure 2.8b, the register file consists of a set of physical registers that store operation results. Each physical register is formed of a 32-bit data, jointly with a 6-bit field storing the x86 flags. The number of integer physical registers can be established with the `RfIntSize` variable in section `[Queues]` in the x86 configuration file (option `--x86-config`).

2.11.3 Renaming Process

At any moment of a guest program simulation, each logical register is mapped to a given physical register in the register file, containing the associated value. In the Multi2Sim renaming model, logical register and flags renaming works independently. This means, for example, that register `eax` and flag `cf` can be mapped to the same register file entry. In this case, the *value* field stores the contents of *eax*, while a specific bit in the *flags* field contains the value for *cf*. Each logical register is mapped to a different physical register, but x86 flags can be mapped all to the same physical register, even if the latter already has an associated logical register.

A Register Aliasing Table (RAT) holds the current mappings for each logical register. Its initial state is shown in Figure 2.8a. Additionally, a Free Register Queue (FRQ) contains the identifiers corresponding to free (not allocated) physical registers. When a new instruction writing into logical register l is renamed, a new physical register is taken from the FRQ and the new mapping for l is stored in the RAT. The previous mapping p' of logical register l will be needed later, and is stored in the ROB entry associated with the renamed instruction. When subsequent instructions consuming l are renamed, the RAT will make them read its contents in p, when they will find the associated value.

When the instruction writing on l is committed, it releases the previous mapping of l, i.e., physical register p', returning it to the FRQ if necessary. Notice that, unlike a classical renaming implementation ignoring flags, a physical register can have several entries in the RAT pointing to it (the maximum is the number of flags plus one logical register). Thus, a counter is associated with each physical register, which will only be freed and sent back to the FRQ in case this counter is 0.

2.12 Floating-Point Register Renaming

2.12.1 The x86 Floating-Point Stack

The x86 floating-point (FP) unit is based on a stack of 8 extended precision 80-bit FP registers. Each of these registers is structured as a 64-bit fraction, 15-bit exponent, and 1-bit sign, representing a single FP number. The operations performed by the x86 FP instruction set can be classified in three groups: *i*) instructions pushing/popping values to/from the stack, *ii*) exchanging the values at different stack positions, and *iii*) performing arithmetic operations involving the value at the top of the stack.

For example, a piece of code performing an FP addition would push into the stack the two operands fetched from memory (instructions of type *i*), add the two registers at the top of the stack by replacing the top-most register with the result of the operation (type *iii*), and finally pop the result from the stack and store it to memory (type *i* again). When several simple operations are involved in a complex computation, stack-based arithmetic requires to frequently exchange register positions (type *ii* instructions) within the stack, since the requested source operands might not be located at the top.

Push and pop operations on the stack modify the FP stack pointer, while any operation consuming an FP register first reads the stack pointer to actually locate its source operand. This causes an implicit dependence between any FP operation and the previous stack push/pop. Moreover, some arithmetic instructions also modify the stack pointer, which would cause another implicit dependence for all subsequent FP instructions, even though the FP result generated by the old instruction is not consumed by the young one.

A naive implementation of the FP arithmetic would conservatively enforce a sequential execution of all arithmetic operations. This would under-utilize the out-of-order execution potential of a superscalar processor, and prevent FP independent operations from overlapping each other to hide latencies. Since Version 2.4, Multi2Sim solves this problem by implementing some of the techniques published in [5], which rely on a 2-stage register renaming of FP instructions.

2.12.2 Two-Stage Renaming Process

The FP register renaming scheme implemented in Multi2Sim works as follows. In a first stage, source and destination registers are translated to a flat space independent of the stack pointer. In a second stage, an additional renaming is performed to remove WAW and WAR hazards while still enforcing RAW dependences, exactly the same way as it is done for integer registers. More specifically:

- **1st Stage**. When FP macroinstructions are decoded, the hardware determines by how many positions (up or down) the top of the stack (ToS) should be modified. This information is passed to the renaming hardware, jointly with the sequence of corresponding FP microinstructions. FP instructions (as well as the generated microinstructions) use *relative stack registers*, referred to as `ST(0)` to `ST(7)`, and being `ST(0)` the FP register located at the head of the stack. Once in the renaming stage, the ToS pointers is first updated in case it is affected

by the next FP microinstruction. Then, the relative stack register identifiers included in the handled microinstruction (both for the source and destination operands), are translated into *absolute stack registers* by adding the ToS value to their identifier.

- **2nd Stage**. Traditional register renaming is then performed for each absolute stack register, by translating it into a final *floating-point physical register*. To this aim, a Floating-Pointer Register Alias Table (FP-RAT) is used to store register associations. This table has as many entries as possible absolute stack registers, while each entry contains a value between 0 and the number of FP physical registers minus 1. For each FP operation, the renaming hardware allocates a new FP physical register, and associates it to the destination absolute stack register of the operation, if any. This mapping is then stored in the FP-RAT. If there was no free physical register available for allocation, the renaming stage stalls until any occupied physical register is released. Each source absolute stack register is also translated into an FP physical register by looking up the corresponding FP-RAT entries and finding out their current associations.

Since version 2.4, Multi2Sim distinguishes between the integer and the floating-point register files. The number of floating-point registers can be specified with variable `RfFpSize` in section `[Queues]`.

2.13 The Dispatch Stage

In the dispatch stage, a sequence of uops is taken from the uop queue. For each dispatched uop, register renaming is carried out, by looking up the RAT for the current source and previous destination mappings, and allocating a new physical register for the current destination operand. Then, the uop is inserted in the ROB, and either in the LSQ or the IQ, depending on whether the uop is a memory instruction or an arithmetic operation, respectively.

The number of instructions dispatched per cycle is specified with the `DispatchWidth` variable in section `[Pipeline]`. Instruction dispatching can stall for several reasons, such as the unavailability of physical registers, a lack of space in the ROB/IQ/LSQ, or an empty uop queue. Since the dispatch stage acts as a bridge between the processor front- and back-end, a stall in this stage is a symptom of some processor bottleneck constraining performance.

2.14 The Issue Stage

The issue stage operates on the IQ and the LSQ. The uops placed in these queues are instructions waiting for their source operands to be ready, or for their associated processor resource to be available. The issue stage implemented the so-called *wakeup logic*, which is in charge of selecting at the most `IssueWidth` uops from each queue that can be scheduled for execution. After selecting the proper candidates, instructions from the IQ are sent to the corresponding functional unit, whereas *load* instructions placed in the LSQ are sent to the data cache.

Since *store* instructions irreversibly modify the machine state, they are handled in an exceptional manner both in the issue and the commit stage. On one hand, *stores* are allowed to access the data cache only after they are known to be non-speculative, which can be ensured after they have safely reached the ROB head. On the other hand, *stores* have no destination operand, so they need not perform any renaming action at the commit stage. Thus, they are allowed to leave the ROB as soon as they have been issued to the cache, without waiting for the cache access to complete.

2.15 The Writeback Stage

The writeback stage is in charge of taking the results produced by the functional units or by a read access to the data cache, and store them to the corresponding physical register mapped to the logical destination of the executed instruction. If the executed instruction is a mispeculated branch, this is when mispeculation is detected, since both the branch condition and the target address are known at this time.

Processor recovery on mispeculation can be performed either at the writeback or at the commit stage, as specified in variable `RecoverKind` in section `[General]`. If recovery is performed at the writeback stage, instructions following the mispeculated branch are drained from the ROB, IQ, and LSQ, the RAT is returned to a previous valid state, and instruction fetching is delayed as many cycles as specified by variable `RecoverPenalty` in section `[General]`.

2.16 The Commit Stage

The commit stage is the last stage of a superscalar processor pipeline, in which instructions commit their results into the architected machine state in program order. The oldest instruction in the pipeline is located at the head of the ROB. The condition for a *store* instruction to be extracted from the ROB is that it be issued to the cache, while the rest of instructions must be completed before committing.

If the instruction at the head of the ROB is a mispeculated branch and the recovery process is specified to be carried out at the commit stage, the contents of the ROB, IQ, and LSQ are completely drained (only mispeculated instructions following the branch remain in the pipeline at this time), the RAT is recovered to a valid state, and `RecoverPenalty` cycles go by before instruction fetch resumes.

When a completed, non-speculative uop commits, the following actions are carried out. First, the register renaming mechanism frees the physical registers corresponding to the previous mappings of the uop's destination logical registers (see Section 2.11). Then, the branch prediction mechanism updates the appropriate tables with the behavior of the committed uop if it is a branch (see Section 2.5). And finally, the uop is added to a temporary buffer in the trace cache, which is used to construct traces of committed instructions (see Section 2.8).

2.17 Support for Parallel Architectures

To describe how a parallel architecture is modeled in Multi2Sim, the following definitions are first given.

- A *context* is a software task (sometimes referred to as *software thread*) whose state is defined by a virtual memory image and a logical register file. Logical register values are exclusive for a context, while a memory map can be either exclusive or shared with other contexts. A running application is represented with one single context when it executes sequential code. However, programs can run parallel code by spawning contexts at runtime (using the OpenMP, or POSIX threads libraries, for example).
- A *(hardware) thread* is a hardware entity capable of storing the status of a single context and executing it. In order to store the logical register values, a thread has its own register aliasing table (RAT), which maps the logical registers into a physical register file. To store the state of a private memory image, a thread has its own memory map cached in a translation look-aside buffer (TLB), which maps virtual memory locations into physical memory pages.

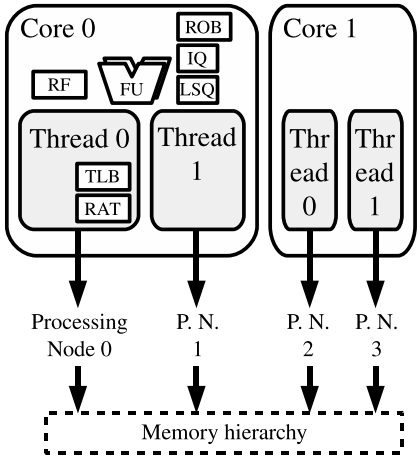


Figure 2.9: Parallel architecture scheme.

Functional units (adders, multipliers, FP execution unit...) are shared among threads, while other pipeline structures, stages, and queues (such as ROB, IQ, LSQ) can be private or shared.

- A *(processor) core* is formed of one or more threads. It does not share any pipeline structure, execution resource, or queue with other cores, and the only communication and contention point among cores is the memory hierarchy.
- A *processing node* is the minimum hardware entity required to store and run one context. In a multithreaded processor, each thread is one processing node. Likewise, each core in a multicore (and not multithreaded) processor is considered one processing node. Finally, an c-core, t-threaded processor (meaning that each core has t threads) has $c \times t$ processing nodes, since it can store and run $c \times t$ contexts simultaneously. Each processing node can have its own entry point to the memory hierarchy to fetch instructions or read/write data. The number of processing nodes limits the maximum number of contexts that can be executed at a time in Multi2Sim. If this limit is exceeded, dynamic context scheduling is required to perform a dynamic mapping of contexts to hardware threads (see Section 2.20).

Based on this definitions, Figure 2.9 represents the structure of the parallel architecture modeled in Multi2Sim. Specifically, the figure plots a processor with 2 cores and 2 threads, forming 4 processing nodes with independent entry points to the memory hierarchy.

2.18 Multithreading

A multithreaded processor is modeled in Multi2Sim using variable `Threads` in section `[General]` in the x86 CPU configuration file (option `--x86-config`), and assigning a value greater than 1. In a multithreaded design, most execution resources can be either private or shared among threads. These resources can be classified as *storage resources* and *bandwidth resources*. The former refer to pipeline structures (such as the ROB, IQ, LSQ, or register file), while the latter refer to the uops that a pipeline stage can handle in a single cycle (such as dispatch slots, issue slots, etc.).

2.18.1 Configuration of storage resources

Multi2Sim uses variables `RobKind`, `IqKind`, `LsqKind`, and `RfKind` in section `[Queues]` in the x86 CPU configuration file to specify the sharing strategy of the ROB, IQ, LSQ, and register file, respectively.

Variable	Coarse-Grain MT	Fine-Grain MT	Simultaneous MT
FetchKind	SwitchOnEvent	TimeSlice	TimeSlice/Shared
DispatchKind	TimeSlice	TimeSlice	TimeSlice/Shared
IssueKind	TimeSlice	TimeSlice	Shared
CommitKind	TimeSlice	TimeSlice	TimeSlice/Shared

Table 2.1: Classification of multithreading paradigms depending on Multi2Sim variables in the x86 CPU configuration file.

The possible values for these options are `Private` and `Shared`. The parameter specifying the size of each structure always refers to the number of entries per thread. For example, when the ROB is shared in an n-threaded, the total number of ROB entries that can be occupied by a single thread is $n \times$`rob_size`.

The fact of sharing a storage resource among threads has several implications in performance and hardware cost. On one hand, private storage resources constrain the number of structure entries devoted to each thread, but it is a natural manner of guarantying a fair distribution of available entries. On the other hand, a shared resource allows an active thread to occupy resource entries not used by other threads, but a greedy thread stalled in a long-latency operation may penalize other active threads by hundreds of cycles if it is holding resource entries for too long.

2.18.2 Configuration of bandwidth resources

The variables to specify how pipeline stages divide their slots among threads are `FetchKind`, `DispatchKind`, `IssueKind`, and `CommitKind` in section `[Pipeline]`. The values that these options can take are `TimeSlice` and `Shared`. The former means that a stage is devoted to a single thread in each cycle, alternating them in a round-robin fashion, while the latter means that multiple threads can be handled in a single cycle. The stage bandwidth always refers to the total number of slots devoted to threads. For example, a value of 4 for `IssueWidth` means that at the most 4 uops will be issued per cycle, regardless of whether the issue stage is shared or not.

The fetch stage can be additionally configured with *long term* thread switches, by assigning the value `SwitchOnEvent` for the `FetchKind` variable. In this case, instructions are fetched from one single thread either until a quantum expires or until the current thread issues a long-latency operation, such as a *load* instruction incurring a cache miss.

Depending on the combination of sharing strategies for pipeline stages, a multithreaded design can be classified as coarse-grain (CGMT), fine-grain (CGMT), and simultaneous multithreading (SMT). The combination of parameters for each stage and its classification are listed in Table 2.1. The main enhancement of FGMT with respect to CGMT is a round-robin fetch stage, which constantly feeds the rest of the pipeline with uops from different threads, thus increasing thread-level parallelism. The key improvement of SMT with respect to FGMT is the shared issue stage, which feeds functional units with a higher rate of ready instructions, regardless of the thread they belong to.

2.19 Multicore Architectures

In Multi2Sim, a multicore architecture is modeled by assigning a value greater than 1 to variable `Cores` in section `[General]` in the x86 CPU configuration file. Since processor cores do not share any pipeline structure, there is no other option related with the multicore processor configuration.

When the number of cores is greater than 1, all processor pipelines and their associated structures are simply replicated, and they work simultaneously in every execution cycle. As mentioned above, the only common entity for cores is the memory hierarchy.

2.20 The Context Scheduler

Multi2Sim introduces the concept of context scheduling after version 2.3.3, similar to the idea of process scheduling in an operating system. The scheduler is aimed at mapping software contexts to processing nodes (hardware threads) to run them. There are two types of context scheduling, selected by the `ContextSwitch` variable. A value of false (`f`) for this option corresponds to the *static scheduler*, while a value true (`t`) actives the *dynamic scheduler*. Both of them are implemented in the `sched.c` file, and their behavior is explained in the following sections.

2.20.1 The Static Scheduler

The static scheduler is implemented in function `p_static_schedule()`. This type of scheduling maps contexts to hardware threads in a definitive manner, using the following criterion:

- At startup, the context configuration file specified in the `--ctx-config` option is analyzed and each software context is mapped to a different hardware thread. The followed allocation order maps first threads within a single core and then goes to the next core after the first one fills up.
- An application using parallel code might spawn new contexts at runtime. New spawned contexts are allocated in the same order as initial contexts (first threads, then cores). This allocation is definitive, meaning that the allocated processing node will not be assigned to any other context and vice versa, even if the context is suspended or finishes. Thus, a suspended context cannot be evicted to allow the hardware thread to be occupied by other context.
- Context switches are not allowed. A running context holds the allocated hardware thread until the simulation ends.
- The total number of created contexts (initial plus spawned contexts) is limited by the total number of processing nodes, that is, the number of cores multiplied by the number of threads. For example, a 2-core, 2-threaded system with one initial context is not allowed to spawn more than 3 additional contexts during execution, even after any of them finishes.

2.20.2 The Dynamic Scheduler

The dynamic scheduler, implemented in function `p_dynamic_schedule()`, offers a more flexible handling for software contexts, with the following criterion:

- The mapping of initial contexts at startup does not differ from the static scheduler. However, these allocations are not definitive, and they can vary at runtime.
- When a context is selected for eviction from a processing node, the scheduler labels it with a *deallocation flag*. Hereafter, it fetches no more instructions until the associated thread's pipeline is empty (including fetch queue, uop queue, and reorder buffer). Then, the context is effectively evicted, and the processing node becomes available for allocation by some other waiting context.

- Contexts have a time quantum, specified as a number of cycles by the `ContextQuantum` variable. If an allocated context exceeds this quantum, and there is any unallocated context waiting for execution, it is selected for eviction by the dynamic scheduler.
- Contexts have an affinity to hardware threads. This means that a context stores the processing node identifier of its last allocation, and likewise, a hardware thread stores the *pid* of the last allocated context. When a context is suspended (or evicted after its quantum expires), it tries to return to the same processing node where it was run for the last time, if it is available.
- New spawned contexts try to find a processing node that has not been used before by any other context, rather than choosing a processing node that was already allocated by any suspended or evicted context.
- When an allocated context is suspended, it is immediately selected by the dynamic scheduler for eviction, so that any unallocated context waiting for execution can allocate the released processing node again.

The aim of the simple affinity scheme implemented by the Multi2Sim dynamic scheduler is preventing the overhead of data migration among caches when possible. This can still occur if a context is executed in different processing nodes with private caches after a context switch is performed. Finally, note that the dynamic scheduler behaves identically as the static scheduler if there are less or equal contexts than processing nodes.

2.21 Statistics Report

A detailed report of the simulation statistics related with the processor pipeline can be obtained by assigning a file name to option `--x86-report`. The output file is a plain-text INI file, with one section for each thread, one for each core, and one for the complete processor.

Using the context and x86 CPU configuration files `ctx-config-args-sort` and `x86-config-args-sort` provided in the `samples/x86` directory, let us run a simulation with these two benchmarks on a processor with 1 core and 2 threads, dumping the pipeline statistics into file `x86-report`. This command should be used:

```
m2s --ctx-config ctx-config-args-sort --x86-sim detailed --x86-config x86-config-args-sort \
        --x86-report x86-report
```

The generated report `x86-report` has four different sections. The first section `[Global]` summarizes statistics for the whole processor. Section `[c0]` contains simulation results for core 0, whereas sections `[c0t0]` and `[c0t1]` show the statistics corresponding to threads 0 and 1, respectively.

2.21.1 Global statistics

The following statistics in the `[Global]` section provide generic simulation results:

- `Cycles`. Number of simulation cycles.
- `Time`. Simulation time in seconds.
- `CyclesPerSecond`. Simulation speed, equal to `Cycles` divided by `Time`.
- `MemoryUsed`. Physical memory used by contexts (as allocated physical pages) at the end of the simulation.
- `MemoryUsedMax`. Maximum physical memory allocated by contexts during the simulation.

2.21.2 Statistics related to all pipeline stages

The statistics prefixed by a pipeline stage name summarize the uops processed by each stage. They have the following meaning:

- `<stage>.Uop.<uop_name>`. Number of uops of a given type (`move`, `add`, `sub`, etc.) that have been processed in a pipeline stage throughout the simulation. The `<stage>` field can be `Dispatch`, `Issue`, or `Commit`. This information is given globally for the processor, per core, and per hardware thread.
- `<stage>.SimpleInteger`, `<stage>.ComplexInteger`. Integer operations are classified in these statistics as complex integer computations (multiplications and divisions) and simple integer computations (rest).
- `<stage>.Integer`, `<stage>.Logical`, `<stage>.FloatingPoint`, `<stage>.Memory`, `<stage>.Ctrl`. Number of processed uops in a given pipeline stage classified as per the computation type in integer, logical, floating-point, memory, and control (branches, jumps, and function calls and returns) operations.
- `<stage>.WndSwitch`. Total number of context switch in a given pipeline stage.
- `<stage>.Total`. Total number uops processed in a given pipeline stage.
- `<stage>.IPC`. Instructions per cycle processed in a given pipeline stage. This value is equal to the number of uops shown in `<stage>.Total` divided by the total number of simulation cycles. The specific value `Commit.IPC` gives the throughput of the computational node, as it refers to the number of uops committed per cycle for a thread/core/processor. The value taken by this statistic within the `[global]` section is equal to the `sim.ipc` statistic reported in the simulation summary.
- `<stage>.DutyCycle`. Value between 0 and 1 indicating the ratio between the obtained IPC and the peak IPC for a given stage. For example, a 4-way processor with a commit duty cycle value of 0.5 retires 2 uops per cycle on average.

2.21.3 Statistics related to the dispatch stage

Multi2Sim measures the usage of dispatch slots for each core and classifies them in different categories. A dispatch slot is the opportunity for an uop to be dispatched; so if there is a dispatch bandwidth of 4 uops/cycle, there are 4 dispatch slots that can be used by 4 different uops to be dispatched. These are the possible classifications:

- `Dispatch.Stall.used`. The slot is used to dispatch a non-speculative uop in the correct path.
- `Dispatch.Stall.spec`. The slot is used to dispatch a mispeculated uop in the wrong execution path.
- `Dispatch.Stall.uopq`. The slot is wasted because there is no uop to consume from the uop queue.
- `Dispatch.Stall.rob`. The uop cannot be dispatched due to a lack of space in the ROB.
- `Dispatch.Stall.iq`. The uop cannot be dispatched due to a lack of space in the IQ.
- `Dispatch.Stall.lsq`. Lack of space in the LSQ.
- `Dispatch.Stall.rename`. Lack of space in the physical register file.
- `Dispatch.Stall.ctx`. The slot is wasted because all contexts allocated to this core are suspended or finished. Thus, there is no uop to grab from the uop queue.

The sum of all `Dispatch.Stall.<how>` statistics is equal to the number of simulation cycles multiplied by the dispatch bandwidth.

2.21.4 Statistics related to the execution stage

The functional units utilization is presented for each core with the statistics prefixed with `fu`. They have the following meaning:

- `fu.<type>.Accesses`. Number of uops issued to a given functional unit. The field `<type>` can be `IntAdd`, `IntSub`, etc.
- `fu.<type>.Denied`. Number of uops that failed to allocate the functional unit. This occurs when an uop is ready to be issued, but the corresponding functional unit is busy.
- `fu.<type>.WaitingTime`. Average time since an uop is ready to be issued until it is able to allocate the corresponding functional unit.

2.21.5 Statistics related to the commit stage

In the commit stage, some statistics are recorded regarding speculative execution and misprediction recovery.

- `Commit.Branches`. Number of committed control uops, including jumps, branches, and function calls and returns.
- `Commit.Squashed`. Number of squashed uops after branch misprediction detections.
- `Commit.Mispred`. Number of control uops mispredicted in the correct path, or committed control uops that caused processor recovery.
- `Commit.PredAcc`. Prediction accuracy. This is equal to $1-$(`Commit.Mispred`/`Commit.Branches`).

2.21.6 Statistics related to hardware structures

Multi2Sim tracks occupancy and access counters for the modeled hardware structures, including the reorder buffer (ROB), instruction queue (IQ), load-store queue (LSQ), register file (RF), branch target buffer (BTB), and register aliasing table (RAT). These statistics are shown in the `[cXtY]` sections for private-per-thread structures, and in the `[cX]` sections for structures shared among threads (`X` and `Y` are core and thread identifiers, respectively).

- `<struct>.Size`. Number of entries, as specified in the corresponding configuration parameter.
- `<struct>.Occupancy`. Average number of occupied entries. This number lies between 0 and `<struct>.Size`.
- `<struct>.Full`. Number of cycles in which the structure was full, i.e., with a number of occupied entries equals to `<struct>.Size`.
- `<struct>.Reads`, `<struct>.Writes`. Number of accesses to the structure.
- `IQ.WakeupAccesses`. Number of associative searches in the instruction queue performed by the wakeup logic.

2.22 Periodic Performance Report

Periodic IPC (instructions per cycle) values can be obtained for each x86 context using additional fields in the context configuration file, passed to the simulator with command-line option

`--ctx-config <file>`. Under each `[Context <id>]` section of this file corresponding to one x86 context, the following additional variables can be used:

- `IPCReport = <file>`. Output file to dump a report with intermediate IPC statistics for the context. If this variable is specified, the simulator periodically dumps plain-text lines including a cycle number, IPC, and number of committed instructions. IPCs are given for the last interval, and also globally since the beginning of the simulation. This option must be specified together with command-line option `--x86-sim detailed` (architectural x86 CPU simulation enabled).
- `IPCReportInterval = <num>`. If variable `IPCReport` is specified, this variable determines the number of cycles of each execution interval. A new record is dumped in the IPC report file every `<num>` cycles. If this value is omitted, a default interval of 10k cycles is considered.

The following listing is an example of an IPC periodic report for a simulation of 600k cycles, with a dump interval of 100k cycles.

```
     cycle     inst   ipc-glob    ipc-int
-----------------------------------------
    100000     7903     0.0790     0.0790
    200000     8434     0.0817     0.0843
    300000    14226     0.1019     0.1423
    400000    21687     0.1306     0.2169
    500000    21333     0.1472     0.2133
    600000    11877     0.1424     0.1188
```

Each column has the following meaning:

- `Cycle`. Current simulation cycle. The increment between this value and the value shown in the next record is the interval specified in the context configuration file with variable `IPCReportInterval`.
- `Inst`. Number of non-speculative instructions executed in the current interval.
- `IPC-glob`. Global IPC observed so far. This value is equal to the number of executed non-speculative instructions divided by the current cycle.
- `IPC-int`. IPC observed in the current interval. This value is equal to the number of instructions executed in the current interval divided by the number of cycles of the interval.

Chapter 3

The OpenCL Programming Model

3.1 Basic Concepts of the OpenCL Programming Model

OpenCL is an industry-standard programming framework [6] designed specifically for developing programs targeting heterogeneous computing platforms, consisting of CPUs, GPUs, and other classes of processing devices. OpenCL's programming model emphasizes parallel processing by using the *Single Program Multiple Data* (SPMD) paradigm, in which a single piece of code, called a *kernel*, maps to multiple subsets of input data, creating a massive number of parallel threads.

Figure 3.1a provides a graphical representation of the basic execution elements defined in OpenCL. An instance of the OpenCL kernel is called a *work-item*, which can access its own pool of *private memory*. Work-items are arranged into *work-groups* with two basic properties: i) those work-items contained in the same work-group can perform efficient synchronization operations, and ii) work-items within the same work-group can share data through a low-latency *local memory* pool.

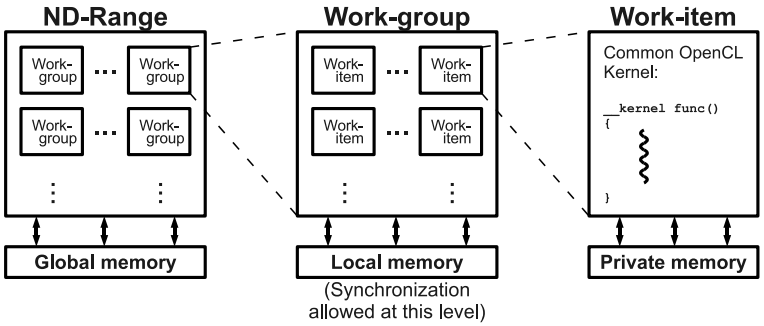
The totality of work-groups form the *ND-Range* (grid of work-item groups), which shares a common *global memory* space. To comply with the OpenCL model, work-groups should be able to execute in any order. Thus, neither synchronizations nor communications through global memory accesses are allowed among work-groups. This restriction allows for compatibility and scalability with any generation of compute devices, regardless of the amount of available parallel processing resources.

3.2 The Execution Model for OpenCL Programs

3.2.1 Native vs. Emulated Execution of OpenCL

An OpenCL application is composed of a *host program* and a *device kernel*. The host program is initially run by the CPU, which invokes a sequence of OpenCL functions to set up the execution of the device kernel on the GPU (or other target device). To provide some insight into the heterogeneous infrastructure, this section describes the software elements involved in the execution of the OpenCL program on a native AMD-based environment (Figure 3.2a), compared with the Multi2Sim simulated environment (Figure 3.2b).

When an OpenCL function call is performed in a native environment, control is transferred to the AMD proprietary implementation of the OpenCL library (`libOpenCL.so`). This library manages OpenCL objects and translates OpenCL functions into sequences of lower-level, AMD-specific, function calls, defined in a forward-compatible interface called the *Compute Abstraction Layer* (CAL) [7]. The CAL interface is in turn divided into user-space and system software, which



Elements defined in the OpenCL programming model. Work-items running the same code form work-groups, which in turn, compose the whole ND-Range.

Figure 3.1: OpenCL Programming Model.

communicate by means of system calls (e.g., `ioctl`). CAL system software includes the GPU device driver specific to the hardware device installed on the host machine.

On the other hand, the call stack of an OpenCL program running on the Multi2Sim simulator differs from the native one starting at the top level. When an OpenCL function call is performed, an alternative implementation the OpenCL runtime (`libm2s-opencl`) handles the call. In this implementation, each OpenCL function generates a function call with a special code currently unused in Linux. When run on Multi2Sim, this system call is intercepted by the x86 emulator, which transfers control to the GPU emulator as soon as the guest application launches the device kernel execution. This infrastructure allows unmodified x86 binaries (pre-compiled OpenCL host programs) to run on Multi2Sim with total binary compatibility with the native environment.

3.2.2 Execution of an OpenCL Program on an AMD-based Native Environment

To further understand the execution model, let us describe first the native execution of an OpenCL program based on the APP software kit provided by AMD [8]. Figure 3.2a presents the generated call stack when a program running on the host CPU performs an OpenCL function call, such as `clEnqueueNDRangeKernel`.

In a first step, control is transfered to the AMD implementation of the OpenCL library, provided in a file called `libOpenCL.so` in their current distributions. This code manages OpenCL objects and translates OpenCL functions into sequences of lower-level, AMD-specific function calls, defined in a forward-compatible AMD standard called *Compute Abstraction Layer* (CAL) [7]. In this case, one of the involved CAL functions would be `calCtxRunProgram`.

The software provided by AMD to implement the CAL interface can be divided into user-space and system-software. The user-space software will handle CAL function calls, until some actual communication with the GPU device is required. At this point, privileged system-software is invoked by means of system calls, such as `open`, `mmap`, or `ioctl`. This lowest-level software includes the GPU device driver, which is specific to the hardware installed on the host machine.

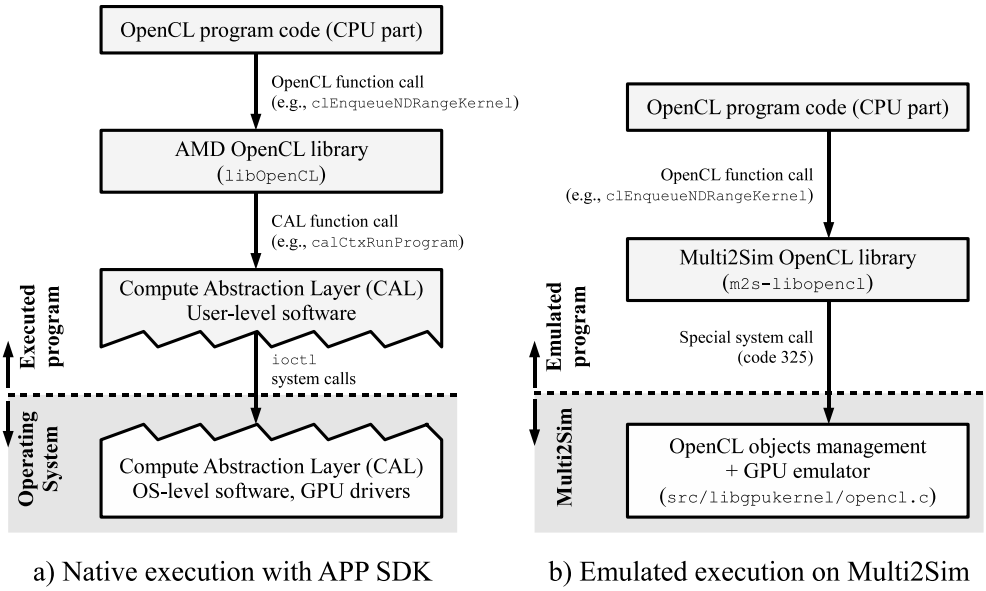


Figure 3.2: Execution of an OpenCL program in a real vs. a simulated execution environment.

3.2.3 Execution of an OpenCL Program on Multi2Sim

The call stack for OpenCL functions changes immediately after the OpenCL function call in the case of the Multi2Sim execution environment, as shown in Figure 3.2b. When the CPU program performs an OpenCL function call, the Multi2Sim OpenCL library provides the implementation for it.

Specifically, the function call is just straightforwardly converted into a system call with code 325, currently unused in Linux. Two arguments are passed with this system call. The first argument is a specific code for the OpenCL function requested. The second argument is a pointer to an array containing the arguments for the OpenCL function. The number of elements in this array depends on the OpenCL function code, and complies to the OpenCL standard.

The Multi2Sim functional simulator intercepts and emulates the system calls performed by the application (see Section 2.1.1). Thus, the special OpenCL system call will be handled by Multi2Sim by analyzing the function arguments, managing OpenCL objects, and eventually launching the GPU simulation module to run the OpenCL kernel provided by the application.

In the described design, it can be observed that the Multi2Sim OpenCL library just acts as a messenger between the program and Multi2Sim, without providing any user-space actions, such as management of OpenCL objects. There are two reasons for choosing a centralized design of the OpenCL implementation in the simulator kernel. First, the source code of the Multi2Sim OpenCL library has been generated automatically, by using a parser of the OpenCL header files provided by the Khronos Group [6] that generates the appropriate system calls. This makes it very unlikely for a bug to be present in this code.

Second, a complete user-level support is provided for the OpenCL interface since the first version, while the actual OpenCL implementation is done in the Multi2Sim kernel, whose support will be increased progressively in subsequent releases. This prevents the user from having to update libraries every time an improved OpenCL support is released.

3.2.4 Implementation of the Multi2Sim OpenCL Library

The Multi2Sim OpenCL library can be found in directory `tools/libm2s-opencl`, and all its source code is centralized in on single file called `m2s-opencl.c`. The following code is an excerpt of this file, corresponding to the implementation of the `clEnqueueNDRangeKernel` function:

```
cl_int clEnqueueNDRangeKernel(
        cl_command_queue command_queue,
        cl_kernel kernel,
        cl_uint work_dim,
        const size_t *global_work_offset,
        const size_t *global_work_size,
        const size_t *local_work_size,
        cl_uint num_events_in_wait_list,
        const cl_event *event_wait_list,
        cl_event *event)
{
        unsigned int sys_args[9];
        sys_args[0] = (unsigned int) command_queue;
        sys_args[1] = (unsigned int) kernel;
        sys_args[2] = (unsigned int) work_dim;
        sys_args[3] = (unsigned int) global_work_offset;
        sys_args[4] = (unsigned int) global_work_size;
        sys_args[5] = (unsigned int) local_work_size;
        sys_args[6] = (unsigned int) num_events_in_wait_list;
        sys_args[7] = (unsigned int) event_wait_list;
        sys_args[8] = (unsigned int) event;
        return (cl_int) syscall(SYS_CODE_OPENCL, OPENCL_FUNC_clEnqueueNDRangeKernel, sys_args);
}
```

The function header matches the function prototype provided in the OpenCL header files distributed by the Khronos Group [6]. In the body of the function, an array `sys_args` is declared, with as many elements as function parameters. For each of them, the actual value passed by the application is copied into the corresponding array position. Finally, a system call of type `SYS_CODE_OPENCL` is performed.

The `SYS_CODE_OPENCL` code is set to 325, as an agreement between the `libm2s-opencl` library and the Multi2Sim kernel. The additional parameters defined for the new OpenCL system call are: *i*) a code corresponding to the OpenCL function to be performed (here `OPENCL_FUNC_clEnqueueNDRangeKernel = 1067`) and *ii*) a pointer to the first element of an array containing the arguments for that specific function.

When the simulated program makes a system call, it is processed by the Multi2Sim emulator. In the case of an OpenCL system call, the OpenCL function code is first inspected, and the actions specific to the OpenCL function are carried out by the Multi2Sim Evergreen functional simulator. Since the actual OpenCL actions are performed in the simulator code, they don't interfere with the program execution. The library user code just serves as a bridge between the program code and the Multi2Sim OpenCL implementation, with the unique burden of packing arguments and launching a system call.

3.3 Building and Simulating Your Own OpenCL Program

The process of building a program executable from C source files can be divided in two steps. First, object files (`*.o`) are generated by compiling the source files (`*.c`), and then the final executable is generated by linking these object files together with shared libraries. If the program aims a native execution, the APP software kit by AMD needs to be installed. If it aims simulated execution, the

Multi2Sim OpenCL library must be built. Additionally, the generation of Evergreen binaries from OpenCL kernel sources is supported by a kernel compilation tool provided within the Multi2Sim package. All these steps are detailed in the next sections.

3.3.1 Building the Multi2Sim OpenCL Library

Multi2Sim's implementation for the OpenCL library is provided in directory `tools/libm2s-opencl`. Notice that this code is outside of the `src` directory, and it is not linked or included with the rest of the simulator code. The reason is that this is not part of the simulator, but a piece of simulated code. Thus, the Multi2Sim OpenCL library should be compiled for a 32-bit x86 target (using the `-m32` compilation flag), regardless of whether Multi2Sim is compiled on a 32- or 64-bit machine. If you want to build your own OpenCL program with simulation purposes, you need to build first the OpenCL library, using the following commands:

```
$ cd $M2S_ROOT/tools/libm2s-opencl
$ make
```

In the commands above, `$M2S_ROOT` refers to the root path where the simulator has been unpacked. If you are running on a 64-bit machine, the command `gcc -m32` will fail, unless your distribution includes the libraries and packages supporting compilation for 32-bit target code. After executing these commands successfully, the following files are generated:

- `libm2s-opencl.so`: dynamic version of the Multi2Sim OpenCL library implementation. When an OpenCL program is compiled dynamically (this is the default linking mechanism for *gcc*), you will need to provide this file jointly with the executable program. It must be locatable at runtime either in the current working directory, or in any of the default library paths, such as `/lib` or `/usr/lib`.
- `libm2s-opencl.a`: static version of the Multi2Sim OpenCL library implementation. An OpenCL program can be linked statically with the libraries it uses by adding the `-static` option to the *gcc* command-line. In this case, no action is performed by the dynamic linker at runtime, since every piece of code used by the program is included in the final executable. This increases portability of the generated binary at the cost of increasing its size.

Recall that the compilation of the Multi2Sim OpenCL library is only required to build your own OpenCL program for simulation purposes. This becomes an unnecessary step if you are just interested in running the OpenCL benchmarks provided in the package, since they are released as statically linked executables, which embed a pre-compiled version of the Multi2Sim OpenCL library ready for simulation. Please refer to Section 2.3.1 for further details about static versus dynamic linking of programs.

3.3.2 Compiling Source Files

The CPU part of an OpenCL program is characterized by performing OpenCL function calls, whose prototype is defined by the OpenCL standard. This standard is encoded in several header files, included by every C source file with an `#include <opencl.h>` directive. If we need to compile a source file called `test-opencl.c`, the only additional files we require so far are the OpenCL headers (`*.h`) provided by the Khronos Group [6]. There are three options to obtain them:

- If you have downloaded Multi2Sim, a distribution of the OpenCL headers is included in directory `$M2S_ROOT/tools/libm2s-opencl/CL`, where `$M2S_ROOT` refers to the path where Multi2Sim has been unpacked.

- If you have installed the APP software kit by AMD, the OpenCL headers can be found in `$AMDAPPSDKROOT/include/CL`, where `$AMDAPPSDKROOT` refers to the root path where the software kit was unpacked.
- If none of the packages above are installed, you can still compile your OpenCL sources by downloading the headers directly from the Khronos Group website [6].

In any of these cases, the following command can be used to compile `test-opencl.c` into `test-opencl.o`, where `$OPENCL_HEADERS` refers to the path where the `opencl.h` file is located:

```
$ gcc -I$OPENCL_HEADERS -c test-opencl.c -o test-opencl.o
```

Additionally, flag `-m32` should be used if you are running on a 64-bit machine, in order to generate 32-bit code suitable for Multi2Sim. This might require a prior manual installation of the necessary `gcc` libraries supporting 32-bit code generation.

3.3.3 Linking Object Files for Native Execution

The compilation of source files is a common process for both native and simulated execution, whereas two different options are given next for linking, depending on the target execution environment. First, let us discuss how OpenCL object files should be linked for a native execution based on the APP software kit.

For this purpose, you first need a correct installation of the APP software kit [8] provided by AMD. In this package, an implementation of the OpenCL library can be found in `$AMDAPPSDKROOT/lib/x86/libOpenCL.so`. Your object files need to be linked with this library, among others, by using a command-line like this:

```
$ gcc -L$AMDAPPSDKROOT/lib/x86 -lOpenCL test-opencl.o -o test-opencl
```

This causes the `gcc` linker to generate an executable file called `test-opencl`. By using the `-L` option, `gcc` is notified to search for shared libraries within the specified path. Notice that the APP software kit does not distribute an `*.a` version of the OpenCL library, meaning that it is not possible to generate a statically linked executable.

3.3.4 Linking Object Files for Execution on Multi2Sim

If your OpenCL program is aimed at running on Multi2Sim, it should be linked with the alternative OpenCL library implementation included in the simulator package. First, the Multi2Sim OpenCL library needs to be compiled, following the guidelines in Section 3.3.1. After a successful compilation, files `libm2s-opencl.so` and `libm2s-opencl.a` can be found in path `$M2S_ROOT/tools/libm2s-opencl`. A program executable can be generated as follows:

```
$ gcc -L$M2S_ROOT/tools/libm2s-opencl test-opencl.o -o test-opencl -lm2s-opencl
```

This command uses the dynamic library `libm2s-opencl.so` to generate a dynamically linked program. Alternatively, a statically linked program using `libm2s-opencl.a` can be generated by appending the `-static` flag to the `gcc` command line shown above. See Section 2.3.1 for the implications of choosing either option.

3.3.5 Simulating an OpenCL Program Linked for Native Execution

You might have an OpenCL executable that was built targeting native execution on AMD's APP software kit, and be lacking the source or object files that generated it. In this case, Multi2Sim still offers the possibility of running it, by using the following technique.

Since the executable is most likely linked dynamically, the initial execution of the program will cause the dynamic linker to be loaded into guest memory, which in turn locates and loads all shared libraries in form of `*.so` files required by the program (see Section 2.3.2). When the dynamic loader finds a shared library location, its code is included into the guest memory image by opening the library, reading the file contents, and closing it, using `open`, `read`, `mmap`, and `close` system calls.

The emulation of system calls allows Multi2Sim to detect the special case where the `open` system call targets a file path suffixed by the "`/libOpenCL.so`" string. At this point, the simulator will try to locate the dynamic version of the Multi2Sim OpenCL library (`libm2s-opencl.so`), and will "cheat" the program by returning a file descriptor to the alternative OpenCL implementation. The dynamic loader does the rest. The user is notified about this manipulated program behavior with a warning like this:

```
warning: path '.../libOpenCL.so' has been redirected to '.../libm2s-opencl.so'
    Your application is trying to access the default OpenCL library, which is being
    redirected by Multi2Sim to its own provided library. Though this should work,
    the safest way to simulate an OpenCL program is by linking it initially with
    'libm2s-opencl.so'. See the Multi2Sim Guide for further details (www.multi2sim.org).
```

All this is done in a completely automatic manner. The only requirement is an available version of the Multi2Sim OpenCL library in the current path, or in any of the paths accessible by the dynamic loader, such as `/lib` or `/usr/lib`. Notice that this mechanism allows for completely unmodified OpenCL program binaries to run correctly on the Multi2Sim simulation environment.

3.3.6 The Multi2Sim OpenCL Kernel Compiler

The source code for an OpenCL program includes the OpenCL kernel, that is, the piece of code aimed at being run on the GPU. For the next examples, let us assume that this code is typed in a separate file, called `test-opencl.cl`. The CPU part of the OpenCL program can load this source code by reading from the file, storing it in a temporary string buffer, and performing a function call to `clCreateProgramWithSource` to make the runtime libraries compile it. Alternatively, `test-opencl.cl` can be compiled off-line before running the OpenCL program, generating a binary file called `test-opencl.bin`. In this case, file `test-opencl.bin` should be loaded by the CPU program using function `clCreateProgramWithBinary`.

In the AMD native execution environment, the OpenCL compiler is implemented in the CAL libraries, an OpenCL sources can be compiled targeting several GPU instruction set architectures either by using the CAL interface, or the higher level OpenCL function. However, Multi2Sim abstracts all these layers (see Figure 3.2), and the OpenCL compiler is not implemented in the simulator. Thus, the only remaining option when targeting simulation is off-line compilation of the kernels.

For this purpose, a tool is provided within the Multi2Sim package, called Multi2Sim OpenCL Kernel Compiler, and located under directory `tools/m2s-opencl-kc`. This tool provides a user-friendly command-line interface for generation of GPU-specific kernel binaries, based on CAL functions. To build this tool, the APP software kit needs to be installed on your system, and its home directory needs to be referred to by environment variable `$AMDAPPSDKROOT` (this is one of the requirements of its installation). You can use the following commands to build the tool:

```
$ cd $M2S_ROOT/tools/m2s-opencl-kc
$ make
```

Again, `$M2S_ROOT` refers to the root directory for the Multi2Sim installation. Any wrong installation of the APP software kit will cause these commands to fail. If the Multi2Sim OpenCL Kernel Compiler is built properly, the executable file `m2s-opencl-kc` will be generated. It can be initially run with no arguments to obtain a list of the available command-line options. Then, option `-l` can be used to obtain a list of the GPU device families that the CAL libraries can generate code for. An output like this is obtained:

```
$ ./m2s-opencl-kc -l

 ID    Name, Vendor
----  -------------------------------------------------------
  0    Cypress Advanced Micro Devices, Inc.
  1    ATI RV770 Advanced Micro Devices, Inc.
  2    ATI RV710 Advanced Micro Devices, Inc.
  3    ATI RV730 Advanced Micro Devices, Inc.
  4    Juniper Advanced Micro Devices, Inc.
  5    Redwood Advanced Micro Devices, Inc.
  6    Cedar Advanced Micro Devices, Inc.
  7    WinterPark Advanced Micro Devices, Inc.
  8    BeaverCreek Advanced Micro Devices, Inc.
  9    Loveland Advanced Micro Devices, Inc.
 10    Cayman Advanced Micro Devices, Inc.
 11    Barts Advanced Micro Devices, Inc.
 12    Turks Advanced Micro Devices, Inc.
 13    Caicos Advanced Micro Devices, Inc.
 14    Intel(R) Core(TM)2 CPU          6300  @ 1.86GHz GenuineIntel
-----------------------------------------------------------
15 devices available
```

For example, the GPU family implementing the Evergreen architecture is referred to as `Cypress`, so this is the device that needs to be chosen to compile the OpenCL kernel sources for Evergreen. The target device is specified using the `-d` option, and the name of the source file containing the OpenCL kernel code must follow. To compile the `test-opencl.cl` kernel, the following command should be used:

```
$ ./m2s-opencl-kc -d Cypress test-opencl.cl

Device 3 selected: Cypress
Compiling 'opencl_basic.cl'...
    test-opencl.bin: kernel binary created
```

As the output reflects, the created binary kernel is stored in file `test-opencl.bin`, whose contents are ready to be loaded by a `clCreateProgramWithBinary` function in a CPU program. The Multi2Sim OpenCL Compiler tool was used to generate the kernel binaries, such as `MatrixMultiplication_Kernels.bin`, included in the OpenCL Benchmarks package distributed in the simulator website. Notice that the binaries are self-contained and do not have any dependency on any other installed software. Thus, they can be distributed unchanged.

Chapter 4

The AMD Evergreen GPU Model

Since Version 3.0, Multi2Sim introduces a model for AMD graphics processing units (GPUs). This work is possible thanks to a collaboration with AMD, providing some unpublished details about the interface between software and hardware modules, as well as undocumented features of the targeted instruction set. The AMD Evergreen ISA [9] has been chosen as the baseline architecture for this model.

4.1 Mapping the OpenCL Model into an AMD Evergreen GPU

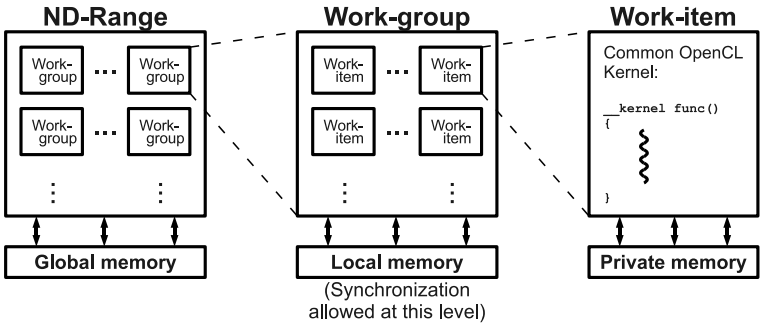
The Evergreen family of AMD GPUs (a.k.a., Radeon HD 5000 series) is a flagship in the AMD's APP lineup, designed to target not only graphics applications, but also general-purpose data-intensive applications. Figure 4.1b presents a block diagram of the Radeon HD 5870 GPU [10], a mainstream device in the Evergreen family. As discussed next, this architecture is designed to provide a conceptual match with the OpenCL programming model (3.1).

When an OpenCL kernel is launched on the Radeon HD 5870 compute device, the ND-Range is initially transferred to it. A global front-end (*ultra-threaded dispatcher*) processes the ND-Range, and assigns work-groups to any of the 20 available *compute units* in any order. Each compute unit has access to the *global memory*, implemented as a hierarchy of private 8KB L1 caches, 4 shared 512KB L2 caches, and the global memory controllers.

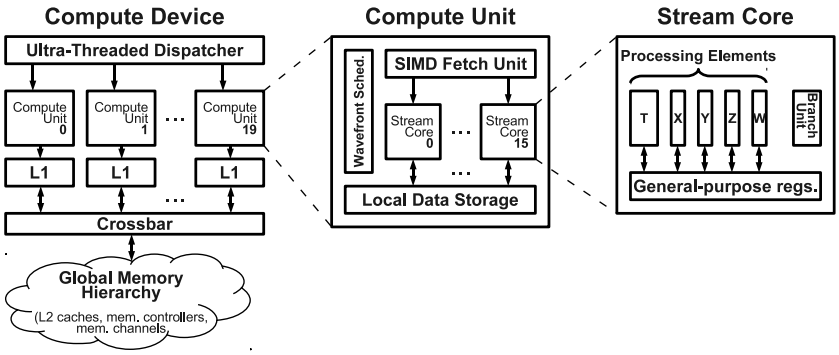
Each compute unit contains a set of 16 *stream cores*, each devoted to the execution of one work-item. All stream cores within the compute unit have access to a common 32KB *local data storage* (LDS), used by the work-items to share data at the work-group level. The LDS is the implementation of the *local memory* concept as defined in OpenCL. Finally, each stream core contains 5 processing elements to execute Evergreen machine instructions in a work-item, plus a file of *general-purpose registers*, which provides the support for the *private memory* concept as defined in OpenCL.

There are two important novelties in an Evergreen device that are not exposed to the programming model. First, considering the mapping between a work-group with a compute unit, the number of stream cores (16) in the compute unit is much lower than the maximum number of work-items (256) in a work-group. To resolve this, stream cores are on one hand time-multiplexed in 4 slots, providing the illusion that each stream core is capable of running 4 work-items concurrently. This defines the concept of a *wavefront* as the total number of work-items (64) virtually executing at the same time on a compute unit.

Still, a work-group contains up to 4 wavefronts that share execution resources. To manage these resources, a wavefront scheduler is in charge of dynamically selecting wavefronts for execution using



a) Elements defined in the OpenCL programming model. Work-items running the same code form work-groups, which in turn, compose the whole ND-Range. (3.1)



b) Simplified block diagram of the Radeon HD 5870 hardware architecture. This GPU belongs to the Evergreen family of AMD devices.

Figure 4.1: OpenCL Programming Model and Evergreen Hardware Architecture.

a round-robin policy. Thus, the wavefront is also commonly referred to as the *scheduling unit* in a GPU. Moreover, the Evergreen family runs all work-items from the same wavefront in a *single instruction multiple data* (SIMD) fashion. In other words, a shared instruction fetch unit provides the same machine instruction for all stream cores to execute.

The second distinctive feature of the Evergreen family is the support for 5-way *Very Long Instruction Word* (VLIW) bundles of arithmetic instructions, defined at compile time. The high performance associated with the Radeon HD 5870 comes from the ability to issue up to 5 floating-point scalar operations in a single cycle, one per VLIW slot. The hardware support for this is contained in each stream core as a set of five *processing elements*, labeled x, y, z, w, and t. The latter provides extended functionality for complex (or *transcendental*) operations, such as logarithm, exponential, square root, etc.

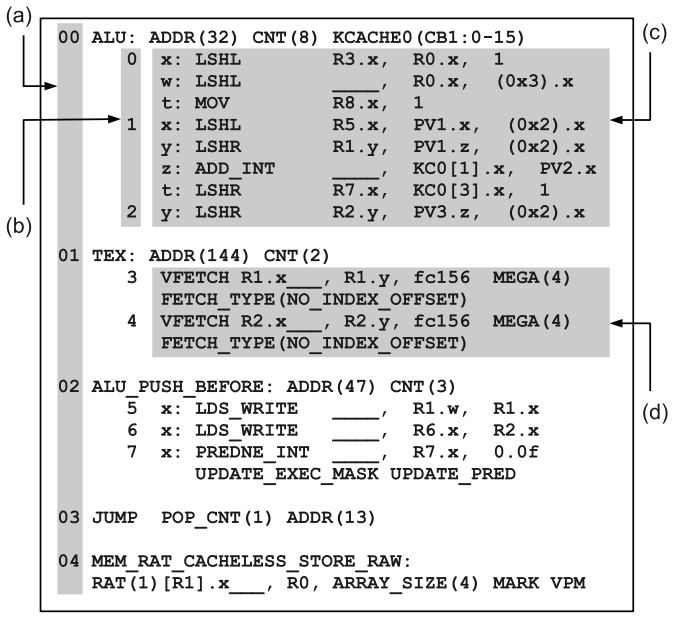


Figure 4.2: Example of AMD Evergreen assembly code: (a) main CF clause instruction counter; (b) internal clause instruction counter; (c) ALU clause; (d) TEX clause.

4.2 The Evergreen Instruction Set Architecture (ISA)

4.2.1 Evergreen Assembly

When the GPU emulator receives the OpenCL kernel to execute, an emulation loop starts in which Evergreen instructions are fetched, decoded, and executed. In this section, the basic format and characteristics of the AMD Evergreen instruction set are discussed, based on the sample assembly code shown in Figure 4.2.

Evergreen assembly uses a clause-based format. The kernel execution starts with an external CF (*control flow*) clause, whose instructions are labeled with 2-digit numbers in the code. CF instructions can affect the program control flow (such is the case of instruction `03`), write data to global memory (`04`), or transfer control to a secondary clause, such as an ALU (*arithmetic-logic unit*) clause (`00`, `02`), or a TEX (*fetch through a texture cache*) clause (`01`). In the code, indented instructions are those belonging to secondary clauses.

In an ALU clause, instructions are packed into *ALU groups*, also called *VLIW bundles*. In the sample code, ALU groups are those preceded by 1-digit labels. An ALU group is run at a time in a stream core, where each ALU instruction label reflects the processing element assigned to that instruction (`x`, `y`, `z`, `w`, or `t`). ALU instructions include data transfers (`MOV`), arithmetic-logic operations (`LSHR`, `ADD_INT`), accesses to local memory (`LDS_WRITE`), or condition evaluations (`PREDNE_INT`).

Possible types of operands for ALU instructions are immediate values (such as `0x3` or `0.0f`), or any of the 128 general purpose logical registers (`R0`, `R1`, etc.), where each register is a set of 4 32-bit values (`x`, `y`, `z`, `w`). Also, an ALU instruction operand can be any processing element's output for the last executed ALU group: the outputs of four regular processing elements can be accessed through the four components (`x`, `y`, `z`, `w`) of the *Previous Value* special register (`PV`), while the output of transcendental processing element is accessed through the *Previous Scalar* special register (`PS`). Finally, *constant memory* is defined as globally accessible storage initialized by the CPU, whose positions can be also used as ALU instruction operands (`KC`).

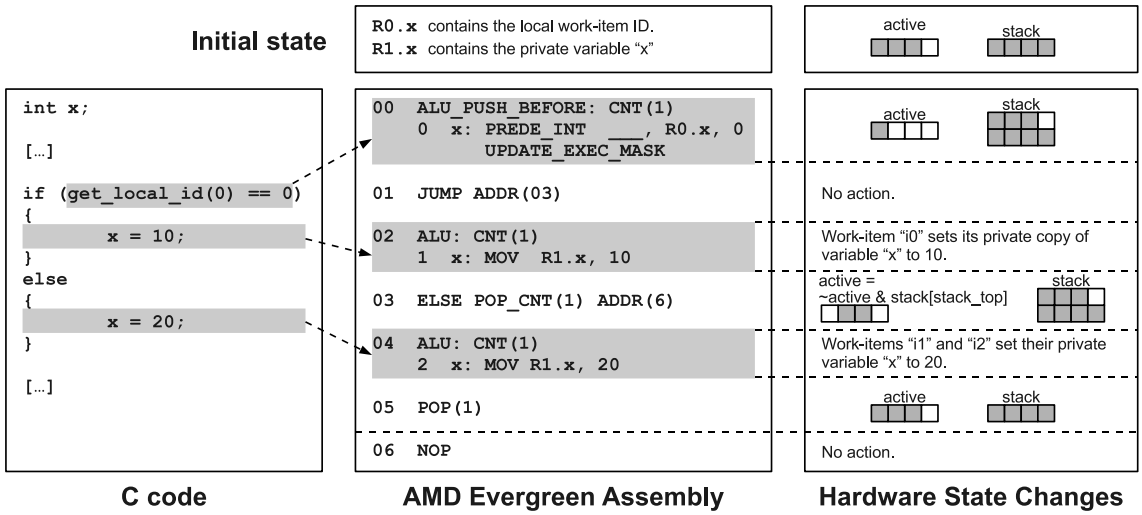


Figure 4.3: Example of 4 work-items (*i0..i3*) from the same wavefront executing Evergreen flow-control instructions. Conditional branches are evaluated differently for each work-item, causing thread divergence.

Regarding a TEX clause, associated instructions are in charge of performing global memory reads. By running in separate hardware blocks of a compute unit, execution of TEX clauses can be overlapped with the execution of other clauses, hiding potentially long latencies when accessing global memory.

This brief description is aimed at roughly interpreting the format of the Evergreen assembly. To obtain more information about the Evergreen ISA and instruction formats, please refer to [9].

4.2.2 Control Flow and Thread Divergence

The SIMD execution model used by the compute units present on an Evergreen GPU causes the same machine instruction to be executed concurrently by all work-items belonging to the same wavefront (see Section 4.1). This implementation simplifies hardware by allowing a common instruction fetch engine to be shared among stream cores, but becomes problematic when a conditional branch instruction is resolved differently in any pair of work items, causing *thread divergence*.

To address thread divergence, the Evergreen ISA provides each wavefront with an *active mask* and an *active mask stack*. The active mask is a 64-bit map, where each bit represents the *active* status of an individual work-item in the wavefront. If a work-item is labeled as inactive, the result of any arithmetic computation performed in its associated stream core is ignored, preventing it from changing the kernel state. The strategy to support thread divergence consists in bringing all work-items together through all possible execution paths, while keeping active only those work-items whose conditional execution matches the currently fetched instruction flow. To support nested branches and procedure calls, the active mask stack is used to push and pop temporary active masks [9].

Figure 4.3 shows an example in which 4 work-items (*i0..i3*) execute the C code shown on the left. The corresponding Evergreen assembly code (center) assumes registers `R0.x` and `R1.x` to contain the work-item ID and the private variable `x`, respectively. The active mask is initially set to 1110 (set bits represented by gray-filled squares on the right), and the stack contains one previously pushed mask set to 1111, as the result of an hypothetical previous flow-control operation. When

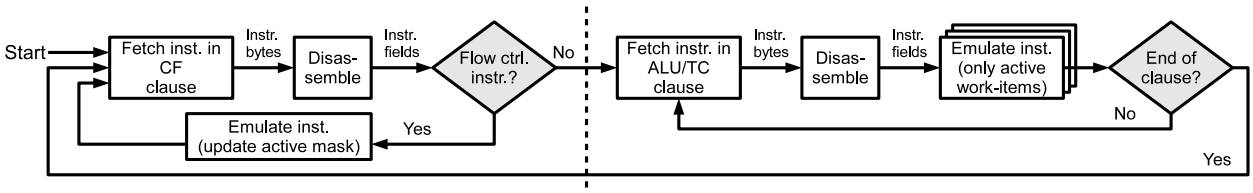


Figure 4.4: The emulation loop in the Evergreen functional simulation module.

running this code, those initially active work-items with an ID other than 0 (i.e., *i1* and *i2*) will set their private variable `x` to 20, whereas work-item *i0* will set its copy of `x` to 10.

Instruction `00 ALU_PUSH_BEFORE` pushes the old active mask into the stack, evaluates condition `get_local_id(0)==0`, and updates the active mask as per those work-items satisfying the condition. Thus, clause `02 ALU` is effective only for work-item *i0*. Instruction `03 ELSE` inverts the active mask and *and*'s it (logic product) with the mask at the top of the stack. In other words, it activates only those work-items that do not satisfy the condition and that were already active before starting the *if* block. This causes clause `04 ALU` to affect only work-items *i1* and *i2*. Finally, instruction `05 POP` restores the original active mask by popping it from the stack.

Instructions `01 JUMP` and `03 ELSE` provide an additional functionality in the case where all work-items evaluate the conditional expression equally (not reflected in the example). Specifically, instruction `01 JUMP ADDR(03)` jumps to instruction `03` if the active mask is completely set to 0, and performs no action otherwise. This improves performance, since there is no need to fetch instruction `04`, if no work-items are going to be active during its execution. Similarly, instruction `03 ELSE POP_CNT(1) ADDR(6)` pops the active mask from the stack and jumps to `06` if its updated active mask is completely clear. Otherwise, its operation is limited to the aforementioned active mask update.

4.2.3 Evergreen Emulation at the ISA Level

The portion of the OpenCL program running on the CPU (x86 binary) is in charge of setting up the OpenCL kernel to be executed on the GPU. For example, the CPU application can load first an OpenCL kernel binary (with a call to `clCreateProgramWithBinary`), then set up the kernel arguments (`clSetKernelArg`), and finally send the OpenCL kernel to the GPU for execution (`clEnqueueNDRangeKernel`). The latter call causes Multi2Sim to transfer control to the GPU emulator, which is fed with the information gathered from the former intercepted functions. Similarly to the x86 CPU functional simulation, the GPU emulation process is split into a *program loading* step and a *emulation loop*.

Program loading is the first stage in the GPU emulation, in which all storage elements representing the OpenCL kernel state are initialized. As previously shown in Figure 4.1a, the kernel state is represented by a common *global memory*, a per work-group *local memory*, and a per work-item *local memory*. The initialization of storage components can be local work-item identifiers, kernel parameters, or information extracted from the kernel binary image, such as the assembly code of the kernel, or initialized variables.

With a proper initial state of the OpenCL kernel, the GPU emulation loop is started, according to the flow diagram shown in Figure 4.4. In every iteration, an instruction is read and disassembled from the main CF clause, and emulated once for an entire group of work-items running in a SIMD fashion. CF instructions can either affect the actual flow of the program (`JUMP`, `ELSE`, etc.), or invoke the execution of a secondary clause (`ALU_PUSH_BEFORE`, `TEX`, etc.).

In the former case (left of dashed line in Figure 4.4), the emulation consists in updating the

active mask of the SIMD group so as to disable execution of those work-items for which the global execution path does not temporarily match their private flow control. In the latter case (right of the dashed line), a new internal loop is run for the emulation of ALU or TEX clauses, in which each instruction is emulated separately for all those active work-items. When the clause finishes, the emulation returns to the initial loop.

4.3 The Evergreen GPU Device Architecture

Since Multi2Sim 3.1, the processor model includes the architectural simulation of an Evergreen GPU. This option can be activated by using the command-line argument `--evg-sim detailed`. Similarly to the x86 architectural simulator, the Evergreen architectural model is based on calls to the Evergreen functional simulation, which provides traces of executed machine instructions.

Evergreen GPU devices are formed of groups of *compute units*, and each compute unit contains a set of *stream cores*. In turn, each stream core is composed of five processing elements, aimed at executing one Evergreen VLIW instruction bundle. Each hardware component is mapped to a different OpenCL software entity at runtime, as described in section 4.1. This section describes the GPU architectural model implemented in Multi2Sim, based on the real architecture of an AMD Evergreen GPU.

4.3.1 Work-Group Scheduling and Configuration

The GPU device can be seen as the hardware unit aimed at running an OpenCL ND-Range. Each GPU compute unit executes one or more OpenCL work-groups at a time. When the CPU launches an OpenCL kernel into the GPU, work-groups are initially mapped into compute units until all of them reach their maximum occupancy. When a work-group finishes execution, the associated compute unit allocates a new waiting work-group, and this process is repeated until the entire ND-Range is executed.

The main Evergreen GPU architectural parameters can be tuned in the Evergreen GPU configuration INI file used with option `--evg-config <file>`. This option should always be used together with option `--evg-sim detailed` for a detailed Evergreen GPU simulation. Section `[Device]` in this file can contain any of the following main configuration variables:

- `NumComputeUnits`. Number of compute units in the GPU. Each compute unit executes one work-group at a time.
- `NumStreamCores`. Number of stream cores in a compute unit. Each stream core contains five processing elements able to execute an Evergreen VLIW bundle.
- `NumRegisters`. Number of registers in a compute unit, also referred to as private memory. The register file is shared among all work-items and work-groups executing in the compute unit at a time.
- `WavefrontSize`. Number of work-items within a wavefront.

A statistics report of the Evergreen architectural simulation can be obtained with option `--evg-report <file>`. Like the configuration files, this report follows a plain text INI file format, and provides the following variables in the `[Device]` section:

- `NDRangeCount`. Number of OpenCL kernels scheduled into the GPU with calls to `clEnqueueNDRangeKernel` performed by the OpenCL host program.

- `Instructions`. Total number of Evergreen machine instructions executed in the GPU. This counter is incremented by one for each instruction executed by a whole wavefront, regardless of the number of work-items forming it.
- `Cycles`. Number of cycles the GPU has been active. The device is considered active as long as any of its compute units has a work-group mapped to it.
- `InstructionsPerCycle`. Quotient of `Instructions` and `Cycles`.

4.3.2 Mapping Work-Groups to Compute Units

A GPU compute unit can run several OpenCL work-groups at a time. However, the specific number of work-groups depends on several architectural and run-time parameters, given by the GPU configuration files, the launched OpenCL kernel binary, and the ND-Range global and local sizes. The architectural factors that limit the number of work-groups mapped to a compute unit are listed next, together with the associated configuration variables in the evergreen simulator (option `--evg-config <file>`, section `[Device]`):

- **Limit in number of work-groups**. In a real GPU, each work-group needs a hardware structure to store information related to it. Since the total architectural storage in a compute unit devoted to this is limited, there is a maximum predefined number of work-groups that can be mapped. Variable `MaxWorkGroupsPerComputeUnit` in the GPU configuration file controls this limit.
- **Limit in number of wavefronts**. There is also a limited amount of total wavefronts whose state can be held at a time by a compute unit, specified by variable `MaxWavefrontsPerComputeUnit` in the configuration file. The number of wavefronts forming a work-group is determined by the OpenCL host program, during the call to `clEnqueueNDRangeKernel` that specifies the local (work-group) size. Depending on this runtime parameter, the actual limit in work-groups per compute unit is limited by `MaxWorkGroupsPerComputeUnit` and `MaxWavefrontsPerComputeUnit`, whichever is reached first.
- **Limit in number of registers**. Each work-item needs a specific amount of registers to execute, which can be found out from encoded metadata in the OpenCL kernel binary. Since there is a limited amount of registers in the compute unit, specified with variable `NumRegisters`, the number of work-groups can be also constrained by this.

  Registers are allocated in chunks, whose size and granularity can be tuned with two additional configuration variables. Variable `RegisterAllocSize` defines the minimum amount of registers that can be allocated at a time, while variable `RegisterAllocGranularity` defines the granularity of these allocations. If the latter is equal to `Wavefront`, the number of registers allocated per wavefront is the first multiple of `RegisterAllocSize` equal or greater than the number of registers needed by all its work-items. In contrast, if `RegisterAllocGranularity` is set to `WorkGroup`, a multiple of the chunk size if allocated at the granularity of the whole work-group.
- **Limit in local memory size**. Finally, each work-group uses a specific amount of local memory, which is determined by the sum of the static local variables encoded in the OpenCL binary kernel, and the dynamic local memory specified by the OpenCL host program at runtime. The total amount of local memory used by all work-groups allocated to a compute unit cannot exceed the size of the physical local memory (section `[LocalMemory]`, variable `Size`). Thus, this imposes an additional limit in number of allocated work-groups per compute unit.

  Similarly to registers, local memory bytes are allocated in chunks (section `[LocalMemory]`, variable `AllocSize`). The amount of local memory allocated by a work-group is the first

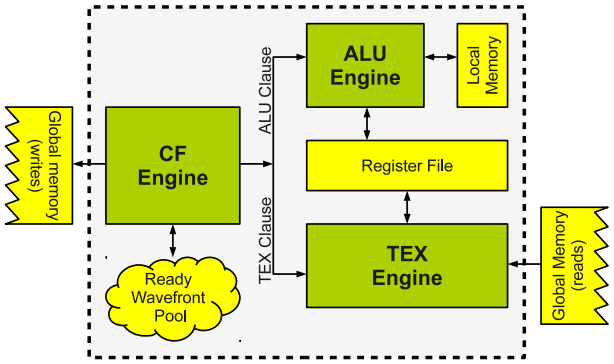


Figure 4.5: Block Diagram of a GPU Compute Unit pipeline.

multiple of `AllocSize` equal or greater than the actual local memory required by a work-group.

Notice that the runtime global and local sizes must allow at least one single work-group to be mapped to a compute unit. If this condition is not satisfied, for example because the number of registers allocated by a single work-group exceeds `NumRegisters`, the simulator will stop with an error message reporting this problem.

The final limit in number of work-groups per compute unit is determined by Multi2Sim right after the OpenCL function `clEnqueueNDRangeKernel` is executed by the simulated OpenCL host program. This value is computed as the minimum of the four limiting factors presented above.

4.4 The Compute Unit Architecture

The compute unit architecture of an Evergreen GPU is represented in Figure 4.5. There are three main components in a compute unit, called CF (Control-Flow) engine, ALU (Arithmetic-Logic) engine, and TEX (Texture) engine, devoted to execute CF, ALU, and TEX clauses of an OpenCL kernel binary, respectively.

When an OpenCL work-group is initially mapped to the compute unit, the main CF clause of the OpenCL kernel is started on the CF engine. The work-items forming the running work-group are combined into smaller groups called *wavefronts*. The main property of a wavefront is that all its work-items execute in a SIMD (single-instruction multiple-data) fashion, that is, only one instruction is fetched for all wavefront's work-items, but each of them runs it based on its own private data. All wavefronts forming the mapped work-group are contained initially in the *ready wavefront pool*, from which they are selected by the CF engine for execution.

CF instructions, or in other words, instructions forming a CF clause, can be classified in three major categories:

- **Secondary ALU clause trigger**. When this type of instruction executes, a secondary ALU clause starts. The current wavefront allocates the ALU engine until all instructions in the secondary clause complete. ALU instructions are VLIW bundles formed of at most five instruction slots, which can perform arithmetic-logic operations and accesses (both reads and writes) to local memory.
- **Secondary TEX clause trigger**. When this type of instructions executes, the current wavefronts launches a secondary TEX clause on the TEX engine, which remains allocated until the TEX clause completes. TEX instructions are read accesses to the global memory hierarchy.

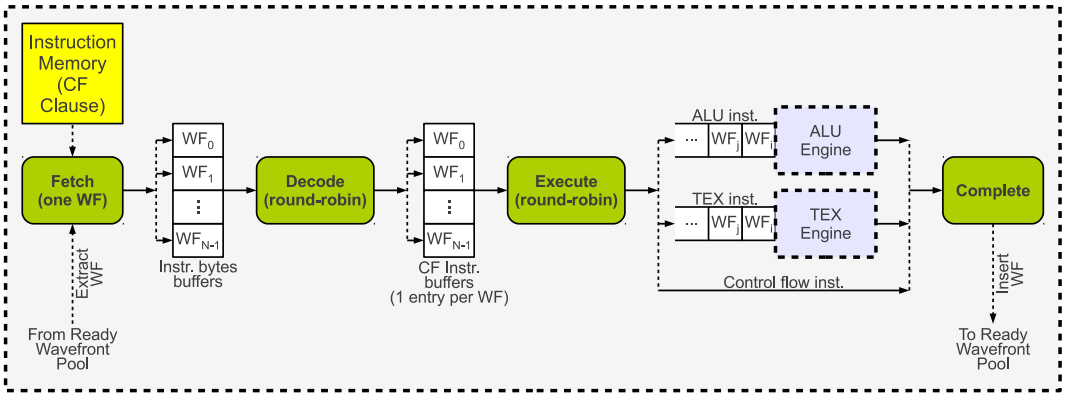


Figure 4.6: Block Diagram of the CF Engine of a Compute Unit.

- **Standard CF instructions**. The remaining instructions not triggering any secondary clause are executed right in the CF engine. These instructions are aimed at updating the work-item active mask, performing synchronizations, issuing global memory writes, or jumping to other positions of the CF clause.

The report dumped with simulator option `--evg-report <file>` includes detailed statistics for every compute unit in the device. Each of the N compute units has an associated section named `[ComputeUnit <id>]`, where `<id>` is a number between 0 and $N-1$. The included variables and their meaning are:

- `WorkGroupCount`. Number of work-groups mapped to the compute unit. At a given time, only one work-group is mapped, so multiple mappings are always done sequentially.
- `Instructions`. Total number of instructions run in the compute unit. For each wavefront executing an Evergreen instruction in a SIMD manner, this counter is incremented once.
- `Cycles`. Number of cycles that the compute unit had some work-group mapped to it.
- `InstructionsPerCycle`. Quotient of `Instructions` and `Cycles`.

The architecture of the execution engines is based on pipeline stages. All of them have a front-end fetch stage that reads instructions from an instruction memory private to the engine, and the rest of the stages vary depending of the engine purpose. The next sections detail the architecture of each execution engine.

4.4.1 The Control-Flow (CF) Engine

Figure 4.6 shows the CF engine architecture, based on a pipeline with 4 stages.

- **Fetch stage**. A running wavefront is selected from the wavefront pool. Since all work-items in the wavefront execute the same instruction at a time, each wavefront has a single program counter associated with it, which is used to address the instruction memory module containing the CF clause. When a wavefront is selected by the fetch hardware, it is extracted from the wavefront pool, and it will be only returned to it at the last stage of the pipeline. This ensures one single instruction to be in flight for each wavefront at a time.

  After selecting a wavefront, an entry in a *fetch buffer* is allocated, associated with the current wavefront. This buffer contains as many entries as number of wavefronts, and each entry has an 8-byte capacity, which is the size of a CF instruction. After the latency of the instruction

memory access, the instruction bytes stored in the allocated fetch buffer entry will be ready for the next stage to be processed.

Every cycle, the fetch stage selects a new wavefront to be fetched, switching among them in a round-robin fashion at the granularity of one single CF instruction. If eventually the wavefront pool runs out of wavefronts, instruction fetch is stalled until a wavefront is placed back into the pool.

- **Decode stage**. This stage selects one wavefront with an occupied entry from the fetch buffer, and decodes the instruction contained in the associated entry. The decoded instruction is placed in the corresponding entry of the CF instruction buffer, which similarly contains one entry per possible wavefront.

  The decode stage selects wavefronts in a round-robin fashion from the fetch buffer, skipping those entries that contain no instruction bytes in them. If there is no empty entry in the instruction buffer corresponding to a ready entry in the fetch buffer, the decode stage stalls.

- **Execute stage**. Based on one entry in the instruction buffer corresponding to a selected wavefront, the execute stage runs the contained CF instruction. If the CF instruction triggers a secondary clause, this instruction will be placed in an entry of the input queue of the ALU/TEX engine. When the secondary ALU or TEX execution engine becomes available, it will start executing the secondary clause triggered by the CF instruction. Finally, when the secondary clause completes, the CF instruction can move to the next pipeline stage. Instructions requiring different execution engines can run in parallel.

  Since CF instructions run by different execution engines might have different latencies, they reach the last stage of the CF engine pipeline in a different order than they were fetched. However, since these instructions belong to different wavefronts, the order of execution does not alter the correctness of the program.

  The execute stage selects wavefronts from the instruction buffer in a round-robin fashion, skipping empty entries. If the instruction buffer has no candidate to be launched, the execution stage stalls. No stall can occur due to unavailable execution resources, since input queues for ALU/TEX engines are assumed to have enough space for all wavefronts if needed.

- **Complete stage**. When CF instructions complete execution (including all instructions run in a secondary clause, if any), they are handled by the last stage of the pipeline. The only purpose of the complete stage is placing the associated wavefront back into the wavefront pool, making it again a candidate to be fetched.

  An exception of instructions that need not complete before reaching this stage are global memory writes. These are CF instructions run in the CF engine (not a secondary clause), and they just issue the write access to the global memory hierarchy without waiting for the actual write operation to complete.

The configuration parameters of the CF engine can be specified in a section named `[CFEngine]` in the GPU configuration file (option `--evg-config <file>`). The allowed configuration variables are:

- `InstructionMemoryLatency`. Latency of an access to the instruction memory in number of cycles.

The set of statistics related with the CF Engine can be found in variables `CFEngine.<xxx>` under section `[ComputeUnit <id>]` in the simulation report dumped with option `--evg-report <file>`. This is a list of the statistics and their meaning:

- `CFEngine.Instructions`. Number of CF instructions executed in the CF engine.

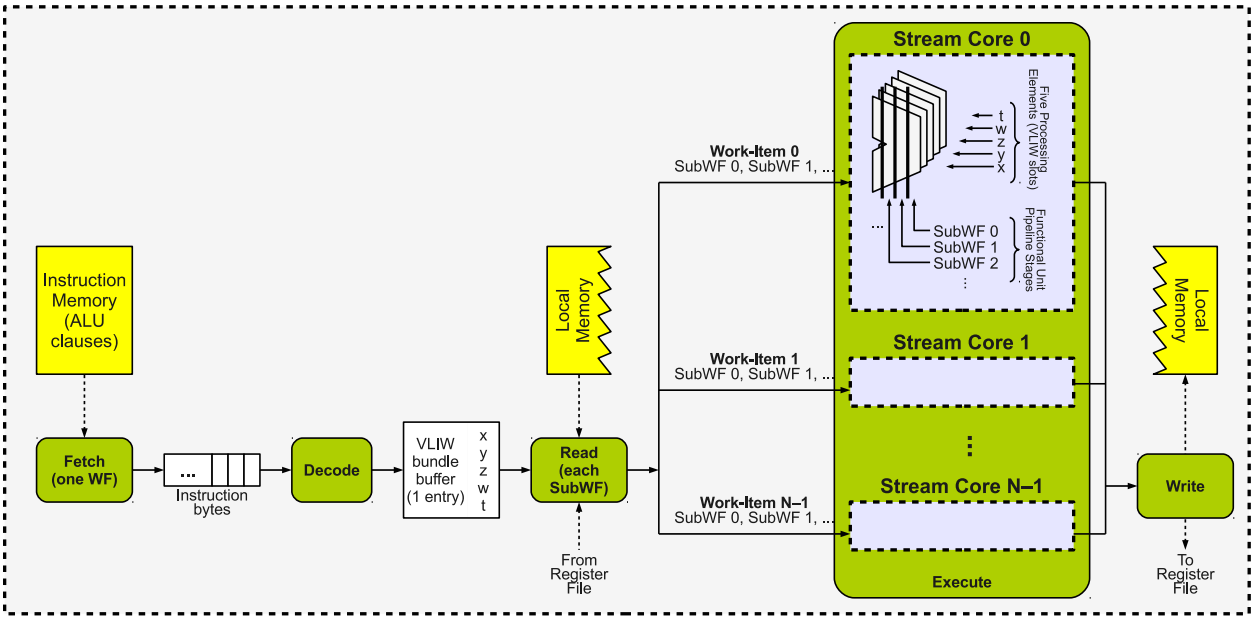


Figure 4.7: Block Diagram of the ALU Engine of a Compute Unit.

- `CFEngine.InstructionsPerCycle`. Quotient of `CFEngine.Instructions` and `Cycles`, as reported in the same `[ComputeUnit <id>]` section. Notice that the CF engine is active exactly as long as the compute unit has a work-group mapped to it.
- `CFEngine.ALUClauseTriggers`. Number of CF instructions triggering a secondary ALU clause.
- `CFEngine.TEXClauseTriggers`. Number of CF instructions triggering a secondary TEX clause.
- `CFEngine.GlobalMemWrites`. Number of CF instructions writing into global memory.

4.4.2 The Arithmetic-Logic (ALU) Engine

The architecture of the ALU engine is shown in Figure 4.7. The ALU engine is a 5-stage pipeline that is mapped to a wavefront after a CF instruction triggers a secondary ALU clause. CF instructions triggering ALU clauses will be placed in an input queue at the ALU engine. The ALU engine selects the CF instruction at the head of this queue, and run all instructions in its secondary clause. When the last instruction of the clause is fetched, the ALU engine extracts the CF instruction from the input queue, and starts fetching instructions from the next secondary clause. Instructions from different secondary clauses can coexist in the pipeline at a given time.

- **Fetch stage**. The memory instruction module is accessed to fetch instructions from the current position in the ALU clause. There is only one wavefront associated with the ALU engine, and a predetermined sequential range for an initial and an ending program counter (instruction address) that needs to be fetched, without any jump in between.

  ALU instructions are VLIW bundles that can contain up to 5 arithmetic-logic instructions (8 bytes each) and 4 literal constants (4 bytes each). The size of a VLIW bundle is variable, and can be as large as 56 bytes. To deal with this variability, the fetch stage just dumps continuous sequences of bytes in to the fetch buffer (circular queue with a minimum capacity of 56 bytes), that will be interpreted as actual instructions in the following stages.

- **Decode stage**. The sequence of bytes at the head of the fetch queue corresponding to a complete VLIW bundle are analyzed and extracted. An interpreted version of it is stored in an instruction buffer able to store one VLIW bundle. If the bytes at the head of the fetch queue do not correspond to a complete VLIW bundle yet, or if the instruction buffer after the decode stage is occupied by the previous VLIW bundle, the decode stage stalls.
- **Read stage**. The source operands for each instruction comprising the VLIW bundle are read. These operands can come from the register file (also referred to as per-work-item private memory), or from local memory. In the former case, one cycle is enough to complete the operands read, while the latter depends on the latency specified for the local memory access, and the amount of work-item accesses that can be coalesced.
- **Execute stage**. This is the core stage of a GPU, where arithmetic instructions are carried out in each stream core. When the source operands for all work-items in the wavefront are ready, the execution stage starts to issue the operations into the stream cores. Each stream core accepts one VLIW bundle every cycle. However, notice that the number of available stream cores does not necessarily match (i.e., might be smaller than) the number of work-items in the current wavefront.

  The solution for this is splitting the wavefront into subwavefronts at the execute stage, where each subwavefront contains as many work-items as available stream cores (say N). In the first execution cycle, work-items 0 to $N-1$ are issued to the pool of stream cores. In the second execution cycle, work-items N to $2N-1$ are issued, and so on. This mechanism is known as *time-multiplexed* execution of a wavefront, with as many time slots as number of subwavefronts.

  Time-multiplexing at the cycle granularity relies on the processing elements (functional units) on the stream cores to be fully pipelined. When a stream core receives a VLIW bundle, its execution latency will be several cycles (configurable), and the result of the operation will be available only after this latency. However, the stream core is ready to accept a new operation right in the next cycle.

  In the Radeon HD 5870 GPU, wavefronts are composed of 64 work-items, while there are 16 stream cores per compute unit. Thus, there are 4 subwavefronts in a wavefronts or, in other words, stream cores receive VLIW bundles from a wavefront in a 4-slot time-multiplexed manner. Notice that if the latency of an operation matches exactly (or is lower than) the number of subwavefronts, it will be completely hidden, and the processing elements will be fully utilized.

  The division of a wavefront into subwavefronts is an architectural decision that allows the wavefront size and the number of stream cores per compute unit to be chosen as independent parameters. Stream cores are expensive resources forming the bulk of the GPU area, and an increase of their number has a significant hardware cost impact. However, increasing the wavefront size reduces the need for fetch resources (more work-items execute one common instruction), although it might increase thread divergence. Thus, the hardware cost versus thread divergence trade-off can be handled as a separate problem, without involving the number of stream cores in the design decision.
- **Write stage**. The result of the computation is written back to the destination operands. Again, these operands can be located in private memory (register file), or in local memory. Writes to local memory are asynchronous, so the write stage need not stall until they complete. When the last VLIW bundle of an ALU clause exits the write stage, the CF instruction triggering this clause will continue in the CF engine.

The configuration parameters of the ALU engine can be specified in a section named `[ALUEngine]` in the GPU configuration file (option `--evg-config <file>`). The allowed configuration variables are:

- `InstructionMemoryLatency`. Latency of an access to the instruction memory in number of cycles.
- `FetchQueueSize`. Size in bytes of the fetch queue. The minimum size is equal to the maximum size of an ALU instruction (56 bytes).
- `ProcessingElementLatency`. Latency of each processing element (`x`, `y`, `z`, `w`, `t`) of a stream core in number of cycles. This is the time since an instruction is issued to a stream core until the result of the operation is available.

The set of statistics related with the ALU Engine can be found in variables `ALUEngine.<xxx>` under section `[ComputeUnit <id>]` in the simulation report dumped with option `--evg-report <file>`. This is a list of the statistics and their meaning:

- `ALUEngine.WavefrontCount`. Number of wavefronts mapped to the ALU engine. At a given time, only one wavefront can be executing an ALU clause, so the mappings occur sequentially.
- `ALUEngine.Instructions`. Number of VLIW bundles executed in this engine. The counter is incremented once for each SIMD instruction, regardless of the number of work-items affected by it.
- `ALUEngine.InstructionSlots`. Number of instructions executed in this engine. Each VLIW bundle can contain up to five instruction slots. The counter is incremented once for each SIMD instruction.
- `ALUEngine.LocalMemoryInstructionSlots`. Subset of `ALUEngine.InstructionSlots` accessing local memory.
- `ALUEngine.VLIWOccupancy`. List of five integer numbers. The first element represents the number of VLIW bundles with one occupied slot. The second number represents the number of VLIW bundles with two occupied slots, and so on.
- `ALUEngine.Cycles`. Number of cycles that a wavefront was mapped to the ALU engine.
- `ALUEngine.InstructionsPerCycle`. Quotient of `ALUEngine.Instructions` and `ALUEngine.Cycles`.

4.4.3 The Texture (TEX) Engine

The TEX engine consists of a 4-stage pipeline, devoted to the execution of global memory fetch instructions, embedded in TEX clauses. The TEX engine executes instructions from the secondary clause triggered by the CF instruction at the head of the input queue. When the last TEX instruction is fetched for the clause, the CF instruction is extracted from the input queue, and the next TEX clause begins to be fetched.

- **Fetch stage**. Instruction bytes are fetched from instruction memory starting from the TEX clause initial address sequentially until the clause end, without any possible execution branch. The fetched bytes are stored at the queue of a circular fetch buffer. Each TEX instruction is 16-byte wide, so this is the minimum size required for the fetch buffer.
- **Decode stage**. A TEX instruction is decoded from the fetch buffer, and its interpreted contents are moved into the following instruction buffer. If the contents of the fetch buffer do not correspond to a complete TEX instruction yet, or the instruction buffer is occupied by a previous TEX instruction, the decode stage stalls.
- **Read stage**. Memory addresses are read from the register file, independently for each work-item forming the current wavefront. For each work-item, a read request to the global memory

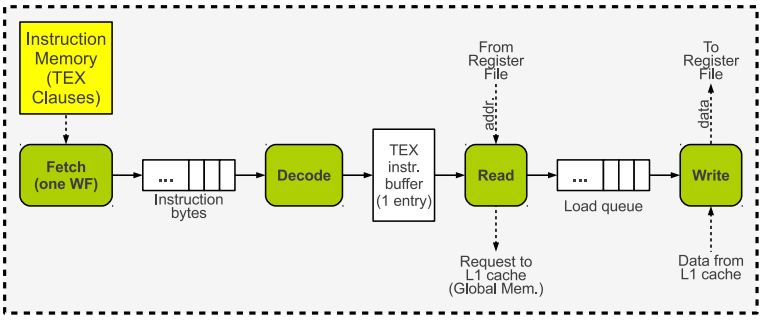


Figure 4.8: Block Diagram of the TEX Engine of a Compute Unit.

hierarchy is performed. Simultaneous read requests of different work-items might be coalesced into a single read operation (see Chapter 6).

The read instruction is inserted at the end of the load queue, which contains all in-flight global memory reads. If there is no free entry in the load queue, the read stage stalls.

- **Write stage**. A completed global memory read instruction is extracted from the head of the load queue. The data fetched from memory is written back into the register file. If the read instruction at the head of the load queue is not complete, or there is no instruction in the load queue, the write stage stalls. When the last TEX instruction of a secondary clause leaves the write stage, the CF instruction triggering this clause can continue traversing the CF engine pipeline.

The configuration parameters of the TEX engine can be specified in a section named `[TEXEngine]` in the GPU configuration file (option `--evg-config <file>`). The allowed configuration variables are:

- `InstructionMemoryLatency`. Latency of an access to the instruction memory in number of cycles.
- `FetchQueueSize`. Size in bytes of the fetch queue. The minimum size is 16 bytes, which is the exact size of a TEX instruction.
- `LoadQueueSize`. Size of the load queue in number of instructions. The load queue communicates the read and write stages of the TEX engine pipeline. The number of entries in this queue determines the maximum number of global memory reads in flight.

The set of statistics related with the TEX Engine can be found in variables `TEXEngine.<xxx>` under section `[ComputeUnit <id>]` in the simulation report dumped with option `--evg-report <file>`. This is a list of the statistics and their meaning:

- `TEXEngine.WavefrontCount`. Number of wavefronts mapped to the TEX engine. At a given time, only one wavefront can be executing a TEX clause, so the mappings occur sequentially.
- `TEXEngine.Instructions`. Number of TEX instructions executed in this engine. The counter is incremented once for each instruction, regardless of the number of work-items affected by it.
- `TEXEngine.Cycles`. Number of cycles that a wavefront was mapped to the TEX engine.
- `TEXEngine.InstructionsPerCycle`. Quotient of `TEXEngine.Instructions` and `TEXEngine.Cycles`.

4.4.4 Periodic Report

Multi2Sim allows to track the progress of a set of performance statistics over the dynamic execution of an OpenCL kernel. A periodic report can be dumped for either a subset or all of the wavefronts in execution at specific configurable intervals. Section `[PeriodicReport]` is used in the

GPU configuration file (option `--evg-config <file>`) for this purpose. The following variables can be used:

- `File`. Prefix for files where the periodic report is dumped. For example, if `File` is set to `my-report`, the report for wavefront 2 within work-group 1 will be stored in file `my-report-wg1-wf2`.
- `Interval`. Number of instructions executed by a wavefront after which a new entry in the report file will be dumped. Instructions are given here as VLIW bundles, where CF and TEX instructions count as one single bundle each.
- `Scope`. This variable specifies the subset of wavefronts in the ND-Range that should dump a report. The following values are accepted:
  - `FirstWavefrontFirstWorkgroup`. Only the first wavefront in the entire ND-Range creates a report file and dumps periodic statistics.
  - `FirstWavefrontAllWorkgroups`. First wavefront of all work-groups.
  - `AllWavefrontsFirstWorkgroup`. All wavefronts of only the first work-group.
  - `AllWavefrontsAllWorkgroups`. All wavefronts in the ND-Range.

The periodic report consists of a set of records, with as many entries as the total number of VLIW bundles executed by the wavefront divided by `Interval`. Each field of a record represents a given performance statistic, and its meaning is specified in a header as part of the dumped report files. The following performance statistics are presently provided:

- `local_mem_accesses`.Number of local memory accesses performed in the interval, adding up all accesses performed by all work-items in the wavefront.
- `global_mem_accesses`. Number of global memory accesses performed in the interval, adding up all accesses performed by all work-items in the wavefront. This statistic assumes that for any global memory read / write action, all the work items in the wavefront will access global memory.

4.5 The Evergreen GPU Memory Architecture

The GPU memory hierarchy is divided into three *memory scopes*, called *private memory*, *local memory*, and *global memory*. The access to each memory scope is defined by software, so there are different instructions or instruction fields specifying which memory scope is targeted in a given memory access. Private memory is accessible per work-item, local memory is shared by a work-group, and global memory is common for the whole ND-Range.

Global memory has a significantly higher latency than local and private memory. To improve its performance, Evergreen GPUs use multiple levels of caches in the global memory scope, forming the *global memory hierarchy*. As opposed to the GPU memory scopes, accesses to different components within the global memory hierarchy are decided by hardware, transparently to the programmer, in a similar way as the cache hierarchy works on a CPU.

This section describes the model and configuration used in Multi2Sim for the private and local memory scopes. Since private and local memory are on chip and accessed by the ALU engines, their configuration is discussed in this chapter. Global memory is accessed by the TEX engine and is discussed in Chapter 6 in detail.

4.5.1 Private Memory

GPU private memory is a different way to refer to the compute unit's register file. The register file provides a private copy of register values for each work-item of the work-group mapped to a compute unit at a given time. It is accessed by the ALU and TEX engines (see Sections 4.4.2 and 4.4.3) during the corresponding read and write stages in their respective pipelines.

Multi2Sim provides a model with no contention for register file accesses. In the worst case, a wavefront mapped to the ALU engine is accessing the register file at the same time as a wavefront mapped to the TEX engine is trying to access it. Even in this case, the involved wavefronts are different, and their work-items will access separate regions of the register file. A per-wavefront banked organization of the register file with enough ports to feed each wavefront's work-item is the equivalent hardware implementation for a model without register file access contention.

4.5.2 Local Memory

There is one local memory module in each compute unit of the GPU, accessible to all work-items of the current work-group running on it. Local memory is accessed by specific ALU instructions, i.e., instructions within an ALU clause mapped to the ALU engine. In a GPU, the local memory latency is higher than private memory because its capacity is higher, and each work-item has access to its entire contents. In the case of local memory, accesses happen in the read or write stages of the ALU engine pipeline. When an instruction accesses local memory, each work-item in the wavefront mapped to the ALU engine issues an access to a potentially different memory location.

Every memory instruction causes each work-item executing it to provide memory addresses based on their private copies of the registers containing them. This causes a chunk of memory address to be enqueued in an access buffer associated to the accessed memory. However, all enqueued addresses need not be translated into actual memory accesses in most of the cases. Since it is likely for adjacent work-items to access also adjacent memory locations, some contiguous accesses in the access queue might fall within the same memory block, and thus they can be coalesced into one single memory access. The coalescing degree depends on the memory block size and the generated memory addresses. The algorithm discussing the coalescing of addresses is discussed in Chapter 6.

Multi2Sim allows a flexible configuration of the local memory parameters in the `[LocalMemory]` section of the GPU configuration file (`--evg-config <file>` option). Also, detailed statistics about local memory accesses are obtained in variables `LocalMemory.<xxx>` of the GPU pipeline report (`--evg-report <file>` option). In the case of local memory, configuration parameters are specified in section `[LocalMemory]` in the GPU configuration file, using option `--evg-config <file>`. The following variables can be used:

- `Latency`. Number of cycles since the time a read/write port is allocated until the access to the memory component completes.
- `BlockSize`. Size of the block. This is the minimum access unit for a memory component. For cache memories, it determines the transfer unit between different cache levels. For any memory component, it determines the coalescing degree among concurrent accesses.
- `Banks`. Number of banks (N) in which the memory contents are organized. Given a continuous memory address space, bank 0 stores memory blocks starting at address 0, N, $2N$, ..., bank 1 stores memory blocks 1, $N+1$, $2N+1$, ..., and bank $N-1$ stores memory blocks $N-1$, $2N-1$, $3N-1$, etc.
- `ReadPorts`, `WritePorts`: Number of read and write ports per bank.

For each local memory element, a set of statistics is dumped in the simulation reports. The statistics can be found in the report associated with the `--evg-report <file>` option, using variables prefixed with `LocalMemory` under sections `[ComputeUnit <id>]`. The set of statistics related to local memory elements and their meaning is the following. Since local memory is explicitly managed by the OpenCL program, every read / write access to local memory is a hit in the local memory.

- `Accesses`. Total number of accesses requested from a compute unit.
- `Reads`, `Writes`. Number of read requests received from a compute unit.
- `CoalescedReads`, `CoalescedWrites`. Number of reads/writes that were coalesced with previous accesses. These are requested accesses that never translated into an effective memory access, since they matched a block targeted by a previous access in the same cycle. See Chapter 6 for more details on coalescing.
- `EffectiveReads`. Number of reads actually performed (= `Reads` − `CoalescedReads`).
- `EffectiveWrites`. Number of writes actually performed (= `Writes` − `CoalescedWrites`).

4.5.3 Global Memory

The GPU global memory, as modeled in Multi2Sim, is structured as a cache hierarchy completely configurable by the user. A dedicated configuration file is used for this purpose, passed to the simulator with the `--mem-config <file>` option. The configuration of the CPU and the GPU memory hierarchy is done in a similar manner and is discussed in Chapter 6 The statistics report of the accesses performed on each component of the global memory hierarchy can be obtained at the end of a simulation by using option `--mem-report <file>`. See Chapter 6 for details on the reported statistics.

4.6 The GPU Occupancy Calculator

In a compute unit of an AMD GPU, there are several limits on the number of OpenCL software elements that can be run on top of it at a time, referred here as compute unit occupancy. Multi2Sim optionally dumps occupancy plots based on static and run-time characteristics of the executed OpenCL kernels.

This option is used by using option `--evg-calc <prefix>` to the command line used to run the simulation, where `<prefix>` is part of the file names used to dump the generated figures in EPS format. The occupancy calculator requires the tool `gnuplot` to be installed in your system.

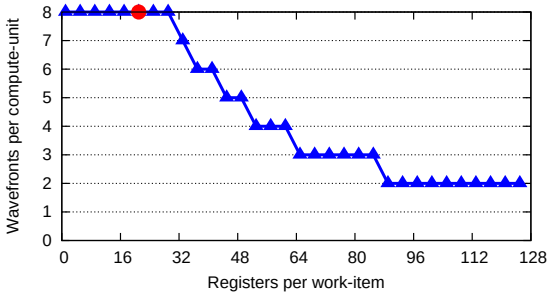
As an example, the following command can be used to run a 64×64 matrix multiplication kernel, and generate the GPU occupancy plots. This example is based on the `MatrixMultiplication` benchmarks included in the AMD OpenCL benchmark suite, available on Multi2Sim's website:

```
m2s --evg-sim detailed --evg-calc calc MatrixMultiplication \
    --load MatrixMultiplication_Kernels.bin -q
```

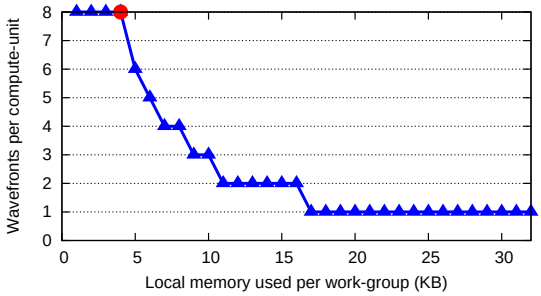
This command produces three EPS images as an output, named `calc.0.registers.eps`, `calc.0.local_mem.eps`, and `calc.0.work_items.eps`, where `0` is the index of the launched OpenCL ND-Range. These three plots are shown in Figure 4.9.

4.6.1 Number of Registers per Work-Item

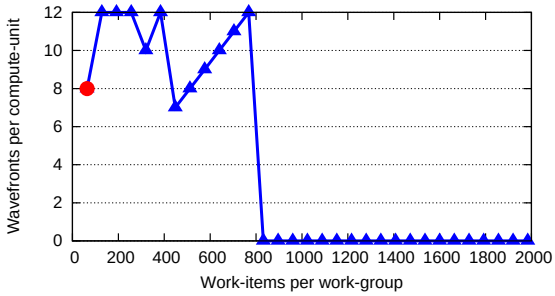
The total number of registers in a compute unit is limited. If a work-item uses too many registers, it will eventually prevent other wavefronts from being executed concurrently (Figure 4.9a). The



a) Number of registers used per work-item.



b) Local memory size per work-group.



c) Work-group size.

Figure 4.9: Output plots generated by the GPU Occupancy Calculator

aspect of this curve depends on the wavefront size, the number of registers available on the compute unit, and the number of registers used by each kernel instance (work-item).

The number of registers per work-item is exclusively decided at compile time, and does not depend on any runtime configuration by the OpenCL host program.

4.6.2 Local Memory Used per Work-Group

A compute unit has also a limited amount of local memory. When the compute unit allocates a work-group that increases the amount of used local memory, it will reduce the total number of wavefronts that can be allocate to the same compute unit (Figure 4.9b). The aspect of this curve depends on the local memory available on the compute unit, the local memory used by each work-group, the wavefront/work-group sizes, and the memory allocation chunk size.

The local memory used by a work-group is potentially decided both at compile time (static local variables) and run-time (dynamic local variables).

4.6.3 Work-Group Size

Finally, the work-group size also determines the total number of wavefronts that can be allocated (Figure 4.9c). Since the allocation unit is an entire work-group (set of wavefronts), this plot shows peaks instead of a continuous descent. The work-group size is exclusively decided at run-time by the OpenCL host program.

4.7 Trying it out

In this section, a quick guide is shown to try a simulation of an OpenCL program for Evergreen GPUs. After downloading, unpacking, and building Multi2Sim (see Section 1.4.1), you will need to download the package containing the pre-compiled OpenCL benchmarks in the *Benchmarks* section of the Multi2Sim website [11]. The package is called `m2s-bench-amdapp-2.5-evg.tar.gz` and can be unpacked with the following command:

```
tar -xzvf m2s-bench-amdapp-2.5-evg.tar.gz
```

One of the included benchmarks corresponds to the classical matrix multiplication algorithm for GPUs (`MatrixMultiplication` directory), which is a pre-compiled version of the OpenCL sample included in AMD's Accelerated Parallel Processing (APP) software kit [8]. Two important files can be found for this benchmark:

- `MatrixMultiplication`: this is a statically linked x86 executable file, embedding a specific implementation of the OpenCL library required by Multi2Sim (more about this later). This executable has been generated from the `MatrixMultiplication.cpp` and `MatrixMultiplication.hpp` sources, not included in the package.
- `MatrixMultiplication_Kernels.bin`: this is a binary file containing the matrix multiplication OpenCL kernels compiled for the AMD Evergreen architecture. It was generated by compiling the `MatrixMultiplication_Kernels.cl` source file, which is not included in the package either.

4.7.1 First Executions

First, let us try to run the x86 program natively to obtain a list of its possible command-line options, like this:

```
$ ./MatrixMultiplication -h
```

From the listed options, let us use `-q` to avoid a dump of big input matrices, and try to run the OpenCL program with its default matrix sizes:

```
$ ./MatrixMultiplication -q
```

The output is an error message, telling that the program cannot be run natively on your machine, because it was statically linked with the Multi2Sim OpenCL library. The result is that the first call to an OpenCL function (here `clGetPlatformIDs`) causes the library to detect that it is not being run on top of the simulator, and the program stops. Let us check out the functional simulation of the program with `m2s --evg-sim functional` then:

```
$ m2s --evg-sim functional MatrixMultiplication -q
```

Now the program reaches a few more steps in its execution, but the simulation stops again with an error message, notifying that the program called function `clCreateProgramWithSource`. This is an OpenCL function used to compile OpenCL kernels' source code at runtime, performing calls to platform-dependent lower levels of the OpenCL stack. Currently, Multi2Sim does not support the runtime compilation of OpenCL code, so we need the program to use the pre-compiled Evergreen kernel binary provided in the package. Fortunately, the samples included in the APP software kit [8] provide a command-line argument `--load <file>`, that allows the user to specify an off-line

compiled binary. If this option is given, the program will use the function `clCreateProgramWithBinary` instead:

```
$ m2s --evg-sim functional MatrixMultiplication --load MatrixMultiplication_Kernels.bin -q
```

This should have been a correct execution for the default input matrix sizes of 64×64. Since no output matrix is shown in this case, not much can be really appreciated in this execution, but you can add option `-e` to the sample command-line to make a self-test of the result matrix. When this option is provided, the benchmark repeats the computation using x86 code, compares the result matrix with the one obtained from the Evergreen kernel, and dumps `Passed` or `Failed` if they match or not, respectively.

4.7.2 The Evergreen GPU Statistics Summary

At the end of an simulation, Multi2Sim presents a summary of statistics in the standard error output (see Section 1.4.3) that follows the INI file format. If an Evergreen functional or detailed simulation took place, a section named `[Evergreen]` is included in this report, including the following variables:

- `SimType`. Simualtion type, as specified in the command line. Possible values are `Functional` and `Detailed`.
- `Time`. Total Evergreen simulation time in seconds. This value includes the time in which the Evergreen functional/detailed simulator was active either in exclusive execution, or in parallel with other architecture simulations. It does not include the time that only other simulation modules were running.
- `NDRangeCount`. Total number of OpenCL NDRanges enqueued to the Evergreen GPU during this simulation.
- `Instructions`. Total number of emulated instructions. In a detailed simulation, this value can be higher than the effective number of instructions committed in the guest program, since it also includes instructions emulated through speculative execution paths.
- `InstructionsPerSecond`. Number of instructions emulated per second, calculated as the quotient of `Instructions` and `Time`. This value is not a performance metric for the guest program. It is used to measure simulation speed.

The following variables are present in the statistic summary only when detailed simulation is selected in the command line:

- `Cycles`. Number of simulation cycles during which an Evergreen detailed simulation was active, either exclusively or in parallel with simulation of other architectures.
- `CyclesPerSecond`. Number of simulation cycles per second, calculated as the quotient of `Cycles` and `Time`. This metric does not measure performance of the guest program. It measures simulation speed.
- `IPC`. Instructions committed per cycle, calculated as the quotient of `CommittedInstructions` and `Cycles`.

Additionally, the Evergreen model and its associated command-line options can cause the simulation to end. This cause is recorded in variable `SimEnd` in section `[General]` of the statistics summary. Besides those values presented in Section 1.4.3, the following additional values are possible for `SimEnd`:

- `EvergreenMaxInst`. The maximum number of Evergreen instructions has been reached, as specified in command-line option `--evg-max-inst <num>`. In functional simulation, this is the maximum number of emulated instructions, as represented in section `[Evergreen]`, variable `Instructions`; in detailed simulation, it is the maximum number of committed instructions, as shown in variable `CommittedInstructions` of the same section.
- `EvergreenMaxCycles`. The maximum number of Evergreen simulation cycles has been reached, as specified in command-line option `--evg-max-cycles <num>`.
- `EvergreenMaxKernels`. The maximum number of Evergreen kernels has been reached, as specified in command-line option `--evg-max-kernels <num>`.

4.7.3 The OpenCL Trace

The Multi2Sim OpenCL library interfaces in such a way with Multi2Sim, that allows it to perform a detailed trace of all OpenCL calls performed by the program. Since version 3.1, Multi2Sim provides the command-line option `--evg-debug-opencl <file>` for this purpose, where `<file>` is the name of the file where to dump the trace. If `stdout` is specified, the OpenCL trace will be dumped in the standard output:

```
$ m2s --evg-sim functional --evg-debug-opencl stdout MatrixMultiplication \
      --load MatrixMultiplication_Kernels.bin -q

clGetPlatformIDs
  'libm2s-opencl' version: 1.0.0
  num_entries=1, platforms=0x8180af8, num_platforms=0x0, version=0x10000
clGetPlatformInfo
  platform=0x10000, param_name=0x903, param_value_size=0x64,
  param_value=0xfffdfd98, param_value_size_ret=0x0
    [...]
```

Notice that this command-line option is added before the x86 program name, since it refers to a simulator option, rather than an option for the benchmark. For each OpenCL function call, the argument values are dumped, including some additional description of special arguments, such as flags or strings. The format of this output is exactly the same as that used for dumping system calls information in previous Multi2Sim versions, using the `--x86-debug-syscall` option. A longer output can be observed after the `clCreateKernel` call:

```
clCreateKernel
  program=0x50007, kernel_name=0x8131e32, errcode_ret=0xfffdfd7c
    kernel_name='mmmKernel_local'

CAL ABI analyzer: parsing file '/tmp/m2s.PT0RPS'
  Parsing encoding dictionary
2 entries

  [...]
  Encoding dictionary entry 1:
    d_machine = 0x9
    d_type    = 0x4
    d_offset  = 0x4d20
    d_size    = 0x2444
    d_flags   = 0x0

  Encoding dictionary entry selected for loading: 1
    pt_note: type=2 (ELF_NOTE_ATI_INPUTS), descsz=0
    pt_note: type=10 (ELF_NOTE_ATI_CONSTANT_BUFFERS), descsz=16
      Note including number and size of constant buffers (2 entries)
      constant_buffer[1].size = 5 (vec4f constants)
      constant_buffer[0].size = 9 (vec4f constants)
    [...]

    arg 0: 'matrixA', pointer to float values (16-byte group) in global memory
    arg 1: 'matrixB', pointer to float values (16-byte group) in global memory
    arg 2: 'matrixC', pointer to float values (16-byte group) in global memory
    arg 3: 'widthA', value of type i32
    arg 4: 'blockA', pointer to float values (16-byte group) in local memory
  kernel 'mmmKernel_local' using 0 bytes local memory
```

When the call to `clCreateKernel` is performed by the program, the requested kernel is loaded from the OpenCL kernel binary file `MatrixMultiplication_Kernels.bin`, which is a file using the Executable and Linkable Format (ELF). In this case, the requested kernel is `mmmKernel_local` (notice that there can be more than one kernel embedded in an Evergreen binary file). Multi2Sim uses its ELF file parser to locate the kernel, and extracts all its information from the file.

The portion of the Evergreen binary associated with a kernel is in turn another embedded ELF file, which includes the kernel code with different representations, such as LLVM, AMD IL, assembly language, and most importantly, Evergreen instructions. Multi2Sim extracts the latter, jointly with other kernel information, such as the number and type of arguments for the kernel, amount of local memory used, etc.

4.7.4 The Evergreen ISA Trace

After the x86 program has finished setting up the OpenCL environment and the kernel input parameters, it performs a call to `clEnqueueNDRangeKernel`. This call launches the execution of the OpenCL kernel and transfers control to the GPU emulator, which is able to interpret AMD Evergreen binary code [9]. Command-line option `--evg-debug-isa <file>` can be used to obtain the trace of Evergreen instructions executed by the GPU functional simulator, where `<file>` is the name of the file where to dump the trace.

Let us try the execution of the matrix multiplication kernel again, this time dumping into the standard output the Evergreen instruction trace. To make the problem simpler, let us change the default input sizes of the matrices to 8×4 and 4×4 for *MatrixA* and *MatrixB*, respectively, and the local block size to 1 single 4×4 element. This problem size generates an ND-Range containing just 2 work-items. Figure 4.10 shows a fragment of the ISA trace obtained for the execution of the

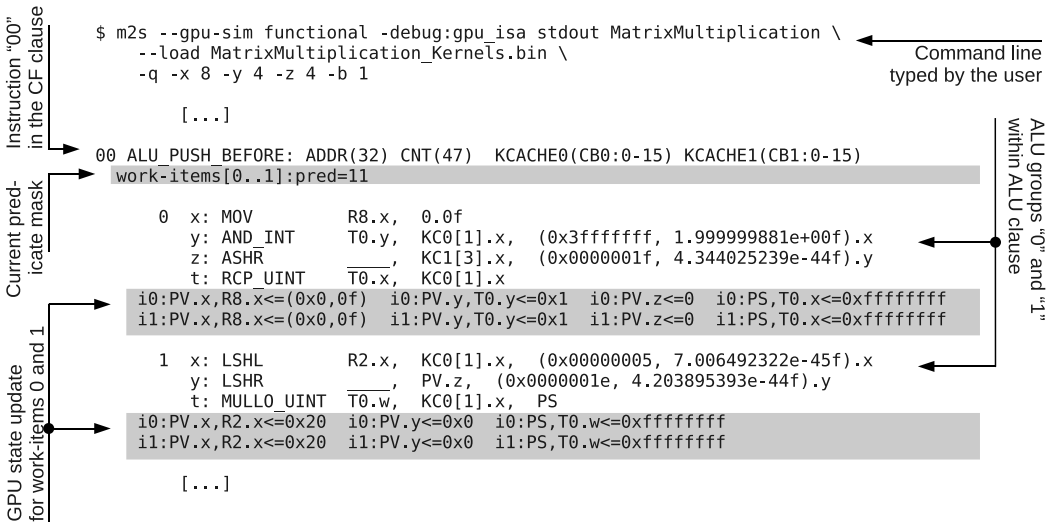


Figure 4.10: ISA trace excerpt for the MatrixMult OpenCL kernel emulation. The command line specified generates an ND-Range with 2 work-items in total.

MatrixMult kernel, including the command line typed by the user.

An Evergreen kernel is usually formed of a main *control flow* (CF) clause. A clause is a set of instructions placed contiguously in memory. In a CF clause, assembly instructions are preceded by a two-digit identifier (e.g., `00 ALU_PUSH_BEFORE`). After this instructions, the `pred` (*predicate*) property of the work-item is dumped. This property is a bitmap containing as many bits as number of threads, where each bit is set to one if the corresponding thread is active. Active threads dump their arithmetic computations into their destination registers, while inactive (or masked) threads do not. Predicate bitmaps are used to handle control-flow divergence among threads within the same work-item. For more on thread divergence see section 4.2.2.

The `ALU_PUSH_BEFORE` CF instruction initiates a so-called ALU (*arithmetic-logic-unit*) clause. An ALU clause is composed of ALU groups, which in turn are formed of ALU instructions. Each ALU group is labeled with a number (`0` and `1` above), and contains at the most five instructions (labeled as `x`, `y`, `z`, `w`, and `t`). The ALU instruction label determines the hardware unit where the instruction will be executed. Labels `x` through `w` represent the simple arithmetic-logic units, while label `t` stands for the transcendental unit, used to execute complex operations. An ALU group can contain at the most one transcendental operation. All ALU instructions within a group are executed in parallel, and they can be viewed as a single VLIW instruction.

Also, a work-item execute ALU clauses in a SIMD (single-instruction multiple-data) fashion, that is, all of them execute exactly the same instructions at a time. The instruction trace shows after each ALU group the values written into each thread's register. For example, `i0:PV.x,R8.x<=(0x0,0f)` means that the ALU group was executed by thread 0, and the value 0x0 (equals to 0 interpreted a floating-point number) is written into component `x` of both register `R8` and `PV`, being `PV` a special register storing always the result of the last operation on each component.

Chapter 5

The AMD Southern Islands GPU Model

First, at Version 3.0, Multi2Sim introduced a model for the Evergreen Family of AMD graphics processing units (GPUs) as presented in chapter 4. As new technology appeared on the market interest grew for the latest AMD GPU architecture, Southern Islands. Since Version 4.0, thanks to continued collaboration with AMD Multi2Sim introduces a model for the Southern Islands GPU architecture.

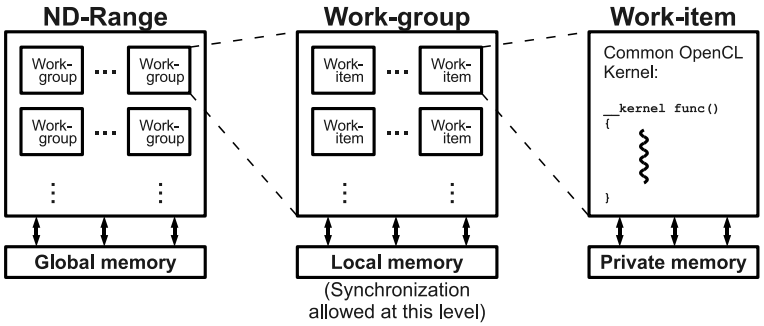
5.1 Mapping the OpenCL Model into an AMD Southern Islands GPU

The Southern Islands family of GPUs (a.k.a., Radeon HD 7000 series) is the latest in AMD's APP lineup, targeted for graphics applications as well as data-intensive general-purpose applications. Figure 5.1 presents a block diagram of the Radeon HD 7970 GPU, a high-end device in the Southern Islands family. The architecture provides a conceptual mapping to the OpenCL programming model (see Chapter 3).

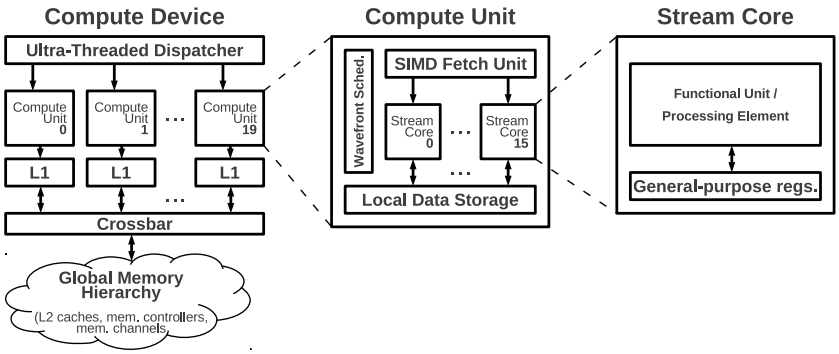
When an OpenCL kernel is launched on a Southern Islands compute device the ND-Range is initially transfered. A global front-end (*ultra-threaded dispatcher*) processes the ND-Range, arbitrarily assigning work-groups to any of the 32 available *compute units*. Each compute unit has access to the *global memory*.

Each compute unit contains a set of 64 vector *stream cores*, devoted to the execution of one work-item. All stream cores within the compute unit have access to a common 64KB *local data share* (LDS), used by the work-items to share data at the work-group level. The LDS is the implementation of the *local memory* concept as defined in OpenCL. Stream cores also are given access to files of *general-purpose registers*, which provide the support for the *private memory* concept as defined in OpenCL.

There are a few important novelties in a Southern Islands device which are not exposed to the programming model. First, considering the mapping between a work-group and a compute unit, the number of stream cores (64) in the compute unit is much lower than the maximum number of work-items (1024) in a work-group. To resolve this, work-groups are broken down into *wavefronts* of 64 work-items. A wavefront, also called a *scheduling unit*, is a chunk of work-items which must execute at the same time, and is executed in a *single instruction multiple data* (SIMD) fashion. In other words, a shared instruction fetch unit provides the same machine instruction to all stream cores executing a wavefront.



a) Elements defined in the OpenCL programming model. Work-items running the same code form work-groups, which in turn, compose the whole ND-Range. (see Section 3.1)



b) Simplified block diagram of the Radeon HD 7970 hardware architecture. This GPU belongs to the Southern Islands family of AMD devices.

Figure 5.1: OpenCL Programming Model and Southern Islands Hardware Architecture.

The stream cores are organized into 4 SIMD units of 16 stream cores each. A single instruction for an entire wavefront is virtually executed at once on 16 stream cores by time-multiplexing them into four slots. Furthermore, each wavefront in the compute unit is assigned to a particular SIMD unit, and must be scheduled only to that unit. This is because the private vector register files are associated with the SIMD units.

A second distinctive feature of the Southern Islands family is the *vector-scalar* design. Machine instructions are separated into two categories, vector instructions which function in the normal SIMD manner, and scalar instructions which need only execute once per wavefront. With the addition of a scalar unit, wavefronts which do not need to utilize all the stream cores of a SIMD unit for the current instruction can execute in less time on the scalar unit, and allow another wavefront to use the SIMD unit instead.

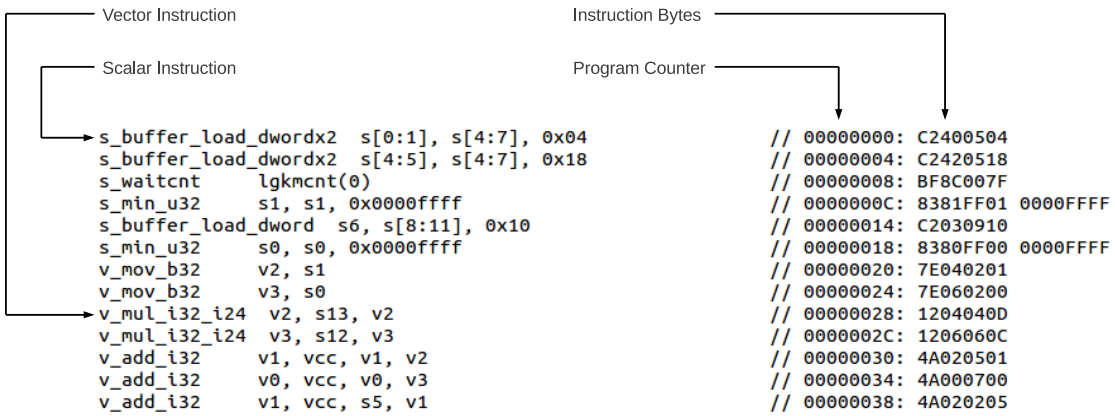


Figure 5.2: Example of Southern Islands Assembly.

5.2 The Southern Islands Instruction Set Architecture (ISA)

5.2.1 Southern Islands Assembly

When the Southern Islands GPU emulator receives the OpenCL kernel to execute, an emulation loop starts in which Southern Islands instructions are fetched, decoded, and executed. In this section, the basic format and characteristics of the AMD Southern Islands instruction set are discussed, based on the sample assembly code shown in Figure 5.3.

Southern Islands is a vector-scalar architecture, and the ISA contains both vector instructions and scalar instructions. Scalar instructions are preceded by a `s_`; vector instructions by a `v_`. Register files are also separated into vector registers (`v0, v1, v2` ...) and scalar registers (`s0, s1, s2` ...). Vector registers are associated with the SIMD units, and hold different values for each work-item. Scalar registers are associated with the scalar unit and hold one value per wavefront.

There is a maximum of 104 scalar general purpose registers, and 256 vector general purpose registers. The actual register limit is dependent on the number of wavefronts assigned to the compute unit, as the private memory must be shared among them. All registers are 32 bits, and all 64 bit instructions use 2 consecutive registers to store 64-bit values.

As well as general purpose registers, there are also a couple of special scalar registers, used implicitly in many instructions. There are however other forms of these instructions in which you can specify a specific register explicitly in place of the special registers if desired. Special registers also may be handled directly, in the same way as general purpose registers. For example, one of the most common of the special registers is the vector condition code (VCC). This is actually composed of two registers and is used as a 64-bit mask. Vector instructions which make comparisons often implicitly store the result of the operation in these registers, which can be used later with the name vcc. A compare instruction could also specify any two consecutive scalar general purpose registers as the destination.

5.2.2 Control Flow and Thread Divergence

The SIMD execution model used by the compute units present on a Southern Islands GPU causes the same machine instruction to be executed concurrently by all work-items belonging to the same wavefront (see Section 5.1). This implementation simplifies hardware by allowing a common instruction fetch engine to be shared among stream cores, but becomes problematic when a conditional branch instruction is resolved differently in any pair of work-items, causing *thread divergence*.

```
float a,b,c;
if (a>b)
        c = a;
else
        c = b;

// Register v0 contains a, v1 contains b, v2 contains c.

                v_cmp_gt_f32            v0, v1                  // a>b, saves into vcc

                s_and_saveexec_b64      s0, vcc                 // Save old exec to s0 and perform ''if''
                s_cbranch_vccz          label0                  // Branch if all lanes fail
                v_mov_b32               v2, v0                  // c = a

label0          s_andn2_b64             exec, s0, exec          // Perform ''else''
                s_cbranch_execz         label1                  // Branch if all lanes fail
                v_mov_b32               v2, v1                  // c = b

label1          s_mov_b64               exec, s0                // Restore exec mask
```

Figure 5.3: Sample Southern Islands thread divergence.

The Southern Islands ISA utilizes a series of execution masks to address thread divergence. The execution mask is a 64-bit map, where each bit represents the active status of an individual work item in the wavefront. If a work-item is labeled as inactive, the result of any arithmetic computation performed in its associated stream core is ignored, preventing it from changing the kernel state. The strategy to support thread divergence consists in bringing all work-items together through all possible execution paths, while keeping active only those work-items whose conditional execution matches the currently fetched instruction flow. To support nested branches and procedure calls, a series of execution masks are used to keep track of the active state at each level.

In Southern Islands, the execution mask is a set of two consecutive special registers named EXEC. The execution mask is handled directly by software, and nested execution masks must be stored in scalar general purpose registers. Examples of thread divergence in Southern Islands assembly can be seen in Figure 5.3.

5.3 The Southern Islands GPU Device Architecture

Since Multi2Sim 4.0, the processor model includes the architectural simulation of a Southern Islands GPU. This option can be activated by using the command-line argument `--si-sim detailed`. Similarly to the other architectural simulators, the Southern Islands architectural model is based on calls to the Southern Islands functional simulation, which provides traces of executed machine instructions.

5.3.1 Work-Group Scheduling and Configuration

The GPU device can be seen as the hardware unit aimed at running an OpenCL ND-Range. Each GPU compute unit executes one or more OpenCL work-groups at a time. When the CPU launches an OpenCL kernel into the GPU, work-groups are initially mapped into compute units until all of them reach their maximum occupancy. When a work-group finishes execution, the associated compute unit allocates a new waiting work-group, and this process is repeated until the entire ND-Range is executed.

The main Southern Islands GPU architectural parameters can be tuned in the Southern Islands GPU configuration INI file using option `--si-config <file>`. This option should always be used together with option `--si-sim detailed` for a detailed Southern Islands GPU simulation. Section `[Device]` in this file can contain any of the following main configuration variables:

- `NumComputeUnits`. Number of compute units in the GPU. Each compute unit executes one work-group at a time.
- `NumStreamCores`. Number of stream cores in a compute unit. This value must divide evenly by the number of wavefront pools, and that quotient must evenly divide into the wavefront size. These restrictions will become clearer after Section 5.4.
- `NumRegisters`. Number of registers in a compute unit, also referred to as private memory. The register file is shared among all work-items and work-groups executing in the compute unit at a time.
- `WavefrontSize`. Number of work-items within a wavefront.

A statistics report of the Southern Islands architectural simulation can be obtained with option `--si-report <file>`. Like the configuration files, this report follows a plain text INI file format, and provides the following variables in the `[Device]` section:

- `NDRangeCount`. Number of OpenCL kernels scheduled into the GPU with calls to `clEnqueueNDRangeKernel` performed by the OpenCL host program.
- `Instructions`. Total number of Southern Islands machine instructions executed in the GPU. This counter is incremented by one for each instruction executed by a whole wavefront, regardless of the number of work-items forming it.
- `ScalarALUInstructions`. Total number of scalar arithmetic-logic instructions run in the device.
- `ScalarMemInstructions`. Total number of scalar memory instructions run in the device.
- `BranchInstructions`. Total number of branch instructions run in the device.
- `VectorALUInstructions`. Total number of vector arithmetic-logic instructions run in the device.
- `LocalMemInstructions`. Total number of local memory instructions run in the device.
- `VectorMemInstructions`. Total number of vector memory instructions run in the device.
- `Cycles`. Number of cycles the GPU has been active. The device is considered active as long as any of its compute units has a work-group mapped to it.
- `InstructionsPerCycle`. Quotient of `Instructions` and `Cycles`.

5.3.2 Mapping Work-Groups to Compute Units

A GPU compute unit can run several OpenCL work-groups at a time. However, the specific number of work-groups depends on several architectural and run-time parameters, given by the GPU configuration files, the launched OpenCL kernel binary, and the ND-Range global and local sizes. The architectural factors that limit the number of work-groups mapped to a compute unit are listed next, together with the associated configuration variables in the Southern Islands simulator (option `--si-config <file>`, section `[Device]`):

- **Limit in number of work-groups**. In a real GPU, each work-group needs a hardware structure to store information related to it. Since the total architectural storage in a compute unit devoted to this is limited, there is a maximum predefined number of work-groups that can be mapped. Variable `MaxWorkGroupsPerComputeUnit` in the GPU configuration file controls this limit.

- **Limit in number of wavefronts**. There is also a limited amount of total wavefronts whose state can be held at a time by a compute unit, specified by variable `MaxWavefrontsPerWavefrontPool` in the configuration file (section `[ComputeUnit]`). The number of wavefronts forming a work-group is determined by the OpenCL host program, during the call to `clEnqueueNDRangeKernel` that specifies the local (work-group) size. Depending on this runtime parameter, the actual limit in work-groups per compute unit is limited by `MaxWorkGroupsPerWavefrontPool` and `MaxWavefrontsPerWavefrontPool`, whichever is reached first (both in section `[ComputeUnit]`).
- **Limit in number of registers**. Each work-item needs a specific amount of registers to execute, which can be found out from encoded metadata in the OpenCL kernel binary. Since there is a limited amount of registers in the compute unit, specified with variable `NumRegisters`, the number of work-groups can be also constrained by this.

  Registers are allocated in chunks, whose size and granularity can be tuned with two additional configuration variables. Variable `RegisterAllocSize` defines the minimum amount of registers that can be allocated at a time, while variable `RegisterAllocGranularity` defines the granularity of these allocations. If the latter is equal to `Wavefront`, the number of registers allocated per wavefront is the first multiple of `RegisterAllocSize` equal or greater than the number of registers needed by all its work-items. In contrast, if `RegisterAllocGranularity` is set to `WorkGroup`, a multiple of the chunk size if allocated at the granularity of the whole work-group. The latter is a more efficient register allocation.
- **Limit in local memory size**. Finally, each work-group uses a specific amount of local memory, which is determined by the sum of the static local variables encoded in the OpenCL kernel binary, and the dynamic local memory specified by the OpenCL host program at runtime. The total amount of local memory used by all work-groups allocated to a compute unit cannot exceed the size of the physical local memory (section `[LocalMemory]`, variable `Size`). Thus, this imposes an additional limit in number of allocated work-groups per compute unit.

  Similarly to registers, local memory bytes are allocated in chunks (section `[LocalMemory]`, variable `AllocSize`). The amount of local memory allocated by a work-group is the first multiple of `AllocSize` equal or greater than the actual local memory required by a work-group.

Notice that the runtime global and local sizes must allow at least one single work-group to be mapped to a compute unit. If this condition is not satisfied, for example because the number of registers allocated by a single work-group exceeds `NumRegisters`, the simulator will stop with an error message reporting this problem.

The final limit in number of work-groups per compute unit is determined by Multi2Sim right after the OpenCL function `clEnqueueNDRangeKernel` is executed by the simulated OpenCL host program. This value is computed as the minimum of the four limiting factors presented above.

5.4 The Compute Unit Architecture

The compute unit architecture of a Southern Islands GPU is represented in Figure 5.4. There are five main components in a compute unit, called SIMD Units, Scalar Unit, Branch Unit, Local Data Share (LDS) Unit, and Vector Memory Unit.

When an OpenCL work-group is initially mapped to the compute unit, the wavefronts of the work-group are mapped onto one of the compute unit's *wavefront pools*. A Southern Islands compute unit is designed with multiple wavefront pools, in which wavefronts wait to be fetched for execution. Each wavefront pool is mapped to one particular SIMD Unit, which is responsible for the vector

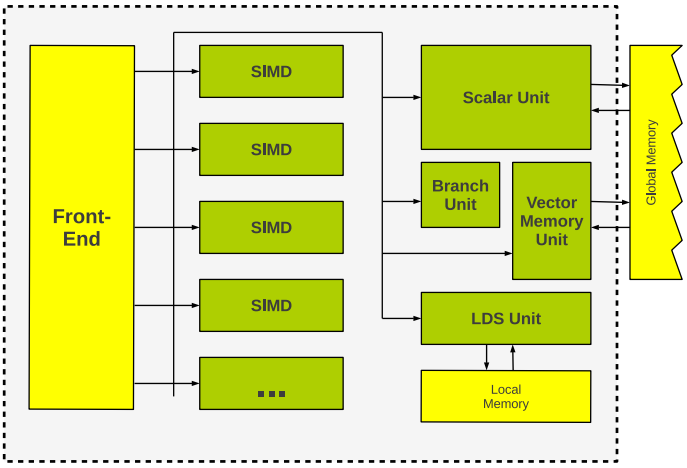


Figure 5.4: Block Diagram of a Southern Islands Compute Unit.

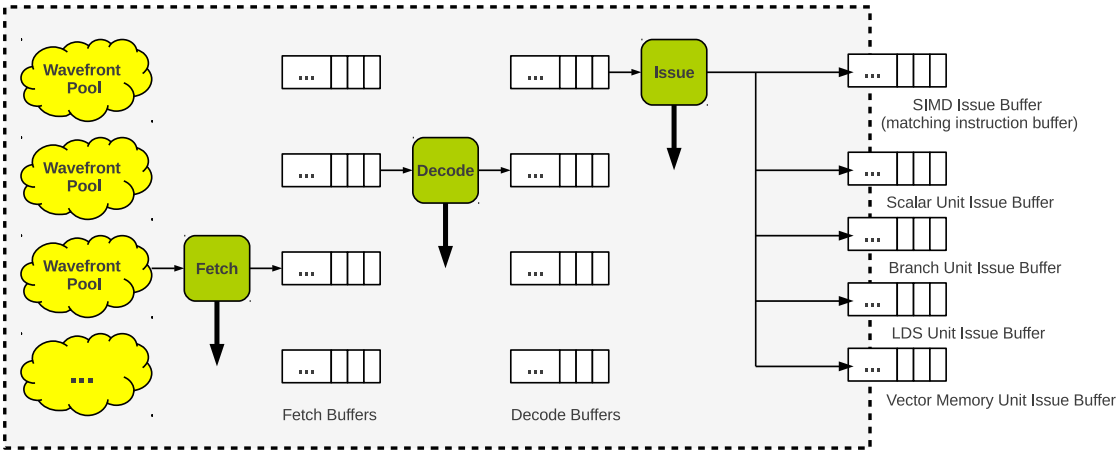


Figure 5.5: Block Diagram of a Southern Islands Compute Unit Front-End pipeline.

operations of all work-groups mapped to its own wavefront pool. All other execution units are independent of which particular wavefront pool the instruction belongs to.

The compute unit front-end is in charge of scheduling wavefronts from the wavefront pools to the execution units. The major pipeline stages in the compute unit front-end are *fetch*, *decode*, and *issue*. The pipeline stages are represented in Figure 5.5. In a single cycle one wavefront pool will be fetched from, while another wavefront pool which has previously been fetched will be decoded, and yet another previously decoded wavefront pool will be issued. The compute unit will fetch from wavefront pools in a round-robin manner and will decode and issue similarly, but shifted behind the previous stage by the previous stage's latency.

The fetch and decode stages are very simple. The fetch stage takes the oldest wavefront available in the wavefront pool and brings it to a fetch buffer. The decode stage then looks through this buffer, decodes the oldest wavefront, and places it into a decode buffer. The issue stage looks through this buffer. When a wavefront is issued, it must be issued to the appropriate execution unit. If the instruction is a vector arithmetic-logic instruction, it must be issued to the SIMD unit mapped to its wavefront pool. Otherwise, the instruction is issued to its associated shared execution unit based on the instruction type, and regardless of the wavefront pool it belongs to.

The compute unit front end looks through the available decoded instructions to issue and issues to appropriate available execution units. Instructions are issued by age, but if an instruction's execution unit is full, the compute unit looks to the next ready instruction, and always issues if possible.

Each execution unit handles a specific class of instructions, aimed at performing a set of particular types of operations.

- **The SIMD Unit**. Executes vector arithmetic-logic instructions. Instructions are executed in a Same Instruction Multiple Data manner for the wavefront. Vector private memory is held in this unit.
- **The Scalar Unit**. Executes scalar arithmetic-logic and scalar global memory instructions. Instructions are executed once per wavefront.
- **The Branch Unit**. Handles a special class of scalar instructions which perform simple control flow operations.
- **The Local Data Share Unit**. Handles all local memory operations.
- **The Vector Memory Unit**. Handles all vector global memory operations.

The configuration parameters of the compute unit can be specified in a section named `[ComputeUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `NumWavefrontPools`. Number of wavefront pools and SIMD units in each compute unit.
- `FetchLatency`. Number of cycles it takes to fetch a wavefront from its wavefront pool.
- `FetchWidth`. Maximum number of instructions which can be fetched in a single cycle.
- `FetchBufferSize`. Size of the buffer which holds instruction bytes that have been fetched and are waiting to be decoded.
- `DecodeLatency`. Number of cycles it takes to decode a wavefront's current instruction.
- `DecodeWidth`. Maximum number of instructions which can be fetched in a single cycle.
- `DecodeBufferSize`. Size of the buffer which holds wavefronts that have been decoded and are waiting to be issued.
- `IssueLatency`. Number of cycles it takes to issue a wavefront to its execution unit.
- `IssueWidth`. Maximum number of instructions which can be issued in a single cycle.

The report dumped with simulator option `--si-report <file>` includes detailed statistics for every compute unit in the device. Each compute unit has an associated section named `[ComputeUnit <id>]`, where `<id>` is a number between 0 and `NumComputeUnits` - 1. The included variables and their meaning are:

- `WorkGroupCount`. Number of work-groups mapped to the compute unit.
- `Instructions`. Total number of instructions run in the compute unit. For each wavefront executing an instruction in a SIMD manner, this counter is incremented once.
- `ScalarALUInstructions`. Total number of scalar arithmetic-logic instructions run in the compute unit.
- `ScalarMemInstructions`. Total number of scalar memory instructions run in the compute unit.
- `VectorALUInstructions`. Total number of vector arithmetic-logic instructions run in the compute unit. For each wavefront executing an instruction in a SIMD manner, this counter is incremented once.

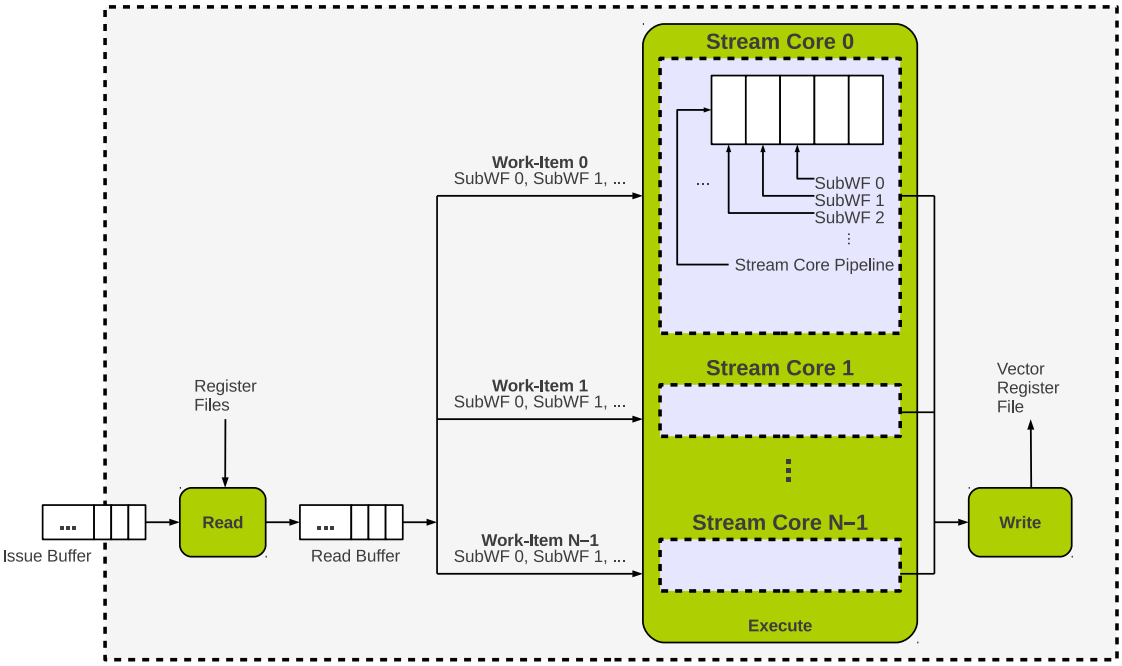


Figure 5.6: Block Diagram of a Southern Islands SIMD Unit pipeline.

- `VectorMemInstructions`. Total number of vector memory instructions run in the compute unit. For each wavefront executing an instruction in a SIMD manner, this counter is incremented once.
- `Cycles`. Number of cycles that the compute unit had some work-group mapped to it.
- `InstructionsPerCycle`. Quotient of `Instructions` and `Cycles`.

The architecture of the execution units is based on pipeline stages. All of them consist of three main stages, read, execute, and write-back. The read and write-back stages are quite similar in all units. The read stage reads instructions from an instruction memory, and the write-back stage returns the wavefront to its wavefront pool. The execution stage does vary between units. The next sections detail the architecture of each execution unit.

5.4.1 The SIMD Unit

The SIMD unit is responsible for executing vector arithmetic-logic instructions. The architecture of the SIMD unit is shown in Figure 5.6, modeled as a pipelined unit with the following stages:

- **Read Stage**. The SIMD unit reads an instruction's current operands from the register files. The instruction is then placed in the read buffer to wait for execution.
- **Execute Stage**. This is the core stage of a GPU, where arithmetic-logic computations are carried out in each stream core. When the instruction's operands are ready, the execution stage starts to issue the operations into the stream cores. Each stream core accepts one work-item every cycle. However, notice that the number of available stream cores does not necessarily match (i.e., might be smaller than) the number of work-items in the current wavefront.

  The solution for this was discussed earlier in Section 5.1 and results in the time-multiplexed execution of wavefronts. The wavefront is split into sub-wavefronts at the execute stage, where

each sub-wavefront contains as many work-items as available stream cores. The number of sub-wavefronts in a wavefront is therefore the quotient of the size of the wavefront and the number of stream cores per SIMD unit. Sub-wavefronts are executed in a pipeline through the stream cores. When a stream core receives a work-item, its execution latency is several cycles (configurable), and the result of the operation is available only after this latency. However, the stream core is ready to accept a new work-item in the next cycle. This results in the execution stage only accepting one wavefront every N cycles, N being the number of sub-wavefronts per wavefront, as the wavefront is broken down and issued to the stream cores one sub-wavefront per cycle.

In the Radeon HD 7970 GPU, wavefronts are composed of 64 work-items, while there are 16 stream cores per SIMD. Thus, there are 4 sub-wavefronts. Notice also there are 4 SIMD units in the compute unit. This allows the latency of the execution to be hidden as instructions are only issued to an individual SIMD unit once every 4 cycles.

The division of a wavefront into sub-wavefronts is an architectural decision that allows the wavefront size and the number of stream cores per SIMD to be chosen as independent parameters. Stream cores are expensive resources forming the bulk of the GPU area, and an increase of their number has a significant hardware cost impact. However, increasing the wavefront size reduces the need for fetch resources (more work-items execute one common instruction), although it might increase thread divergence. Thus, the hardware cost versus thread divergence trade-off can be handled as a separate problem, without involving the number of stream cores in the design decision.

- **Write Stage**. The SIMD unit removes all completed wavefronts, writes their results to the vector register file, and returns them to their wavefront pools in which they will wait to execute the next instruction.

The configuration parameters of the SIMD unit can be specified in a section named `[SIMDUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `Width`. Number of instructions processed by the SIMD unit per cycle.
- `IssueBufferSize`. Size of the buffer which holds instructions that have been issued and are waiting to be read.
- `ReadLatency`. Number of cycles it takes to read operands from the register files.
- `ReadBufferSize`. Size of the buffer which holds instructions that have completed the read stage and are waiting to be executed.
- `StreamCoreLatency`. Number of cycles it takes to execute a work-item in a stream core. Stream cores are pipelined and accept a new work-item every cycle. Wavefronts are executed in `NumSubWavefronts` + `StreamCoreLatency` - 1 cycles and a new wavefront may begin execution every `NumSubWavefronts` cycles as the wavefront is broken down and issued to the stream cores one sub-wavefront per cycle.

5.4.2 The Scalar Unit

The scalar unit is responsible for executing scalar arithmetic-logic and global memory instructions. The architecture of the scalar unit is shown in Figure 5.7.

- **Read Stage**. The scalar unit reads an instruction's current operands from the scalar register file. The instruction is then placed in the read buffer to wait for execution.

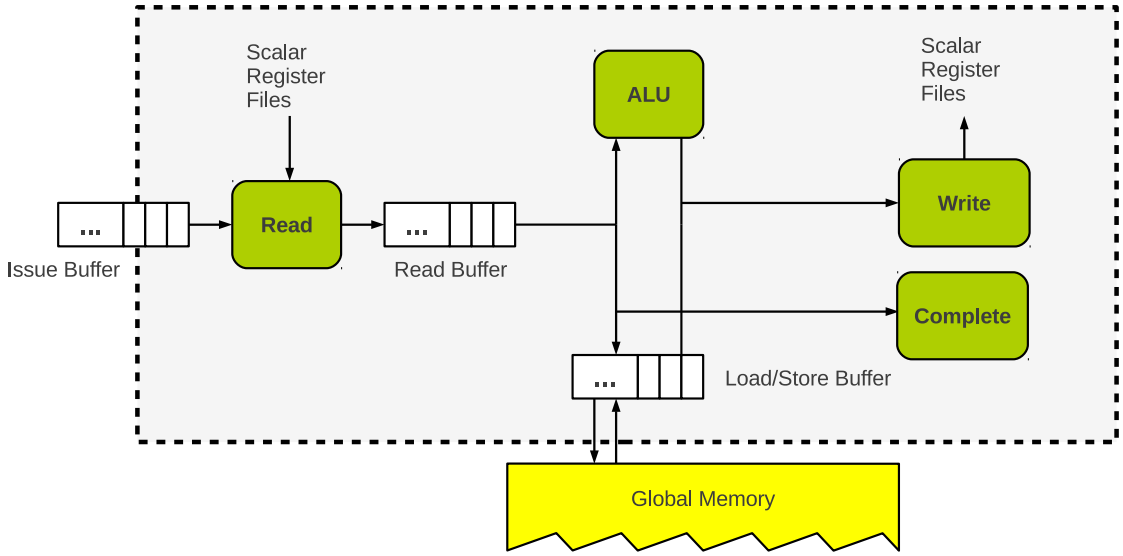


Figure 5.7: Block Diagram of a Southern Islands Scalar Unit pipeline.

- **Execute Stage**. The scalar unit executes two distinct types of instructions, arithmetic-logic and global memory. The execution of arithmetic-logic instructions is a fixed latency operation; There is no SIMD execution or complicated pipelining. Scalar global memory instructions are handled by the memory hierarchy model and are variable latency, depending on the memory configuration and state. A configurable limit is imposed on the number of in-flight memory operations at any time.
- **Write Stage**. The scalar unit removes all completed wavefronts, writes their results to the scalar register file, and returns them to their wavefront pools in which they will wait to execute the next instruction.

The configuration parameters of the scalar unit can be specified in a section named `[ScalarUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `Width`. Number of instruction processed by the scalar unit per cycle.
- `IssueBufferSize`. Size of the buffer which holds instructions that have been issued and are waiting to be read.
- `ReadLatency`. Number of cycles it takes to read operands from the register file.
- `ReadBufferSize`. Size of the buffer which holds instructions that have completed the read stage and are waiting to be executed.
- `ALULatency`. Number of cycles it takes to execute a scalar arithmetic-logic instruction.
- `MaxInflightMem`. Maximum number of of in-flight scalar memory operations at any time.

5.4.3 The Branch Unit

The branch unit is responsible for control flow instructions. The advantage of having a branch unit separate from the scalar unit is that the branch unit has a lower latency. The separation allows branching instructions to not get slowed down or stalled along with more costly arithmetic-logic or global memory instructions in the scalar unit. The architecture of the branch unit is shown in Figure 5.8.

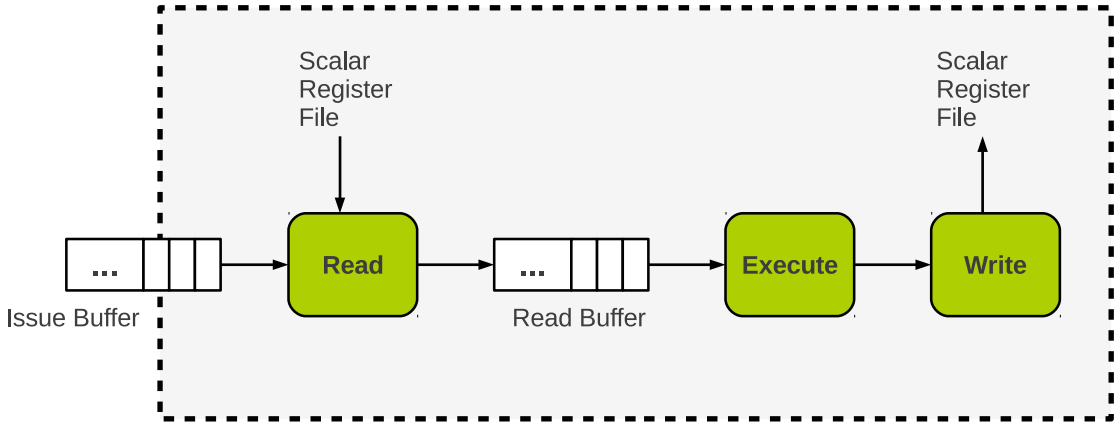


Figure 5.8: Block Diagram of a Southern Islands Branch Unit pipeline.

- **Read Stage**. The branch unit reads an instruction's current operands from the scalar register file. The instruction is then placed in the read buffer to wait for execution.
- **Execute Stage**. The branch unit executes instructions with a fixed configurable latency.
- **Write Stage**. The branch unit removes all completed wavefronts, writes their results to the scalar register file, and returns them to their wavefront pools in which they will wait to execute the next instruction.

The configuration parameters of the branch unit can be specified in a section named `[BranchUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `Width`. Number of instruction processed by the branch unit per cycle.
- `IssueBufferSize`. Size of the buffer which holds instructions that have been issued and are waiting to be read.
- `ReadLatency`. Number of cycles it takes to read operands from the register file.
- `ReadBufferSize`. Size of the buffer which holds instructions that have completed the read stage and are waiting to be executed.
- `BranchLatency`. Number of cycles it takes to execute a branch instruction.

5.4.4 The Local Data Share (LDS) Unit

The Local Data Share unit is responsible for handling all local memory instructions. The architecture of the LDS unit is shown in Figure 5.9.

- **Read Stage**. The LDS unit reads an instruction's current operands from the register files. The instruction is then placed in the read buffer to wait for execution.
- **Execute Stage**. The LDS unit executes instructions with a variable latency dependent on the local memory configuration and state. The instructions are handled by the memory hierarchy model. A configurable limit is placed on the number of in-flight memory operations at any time.
- **Write Stage**. The LDS unit removes all completed wavefronts, writes their results to the register files, and returns them to their wavefront pools in which they will wait to execute the next instruction.

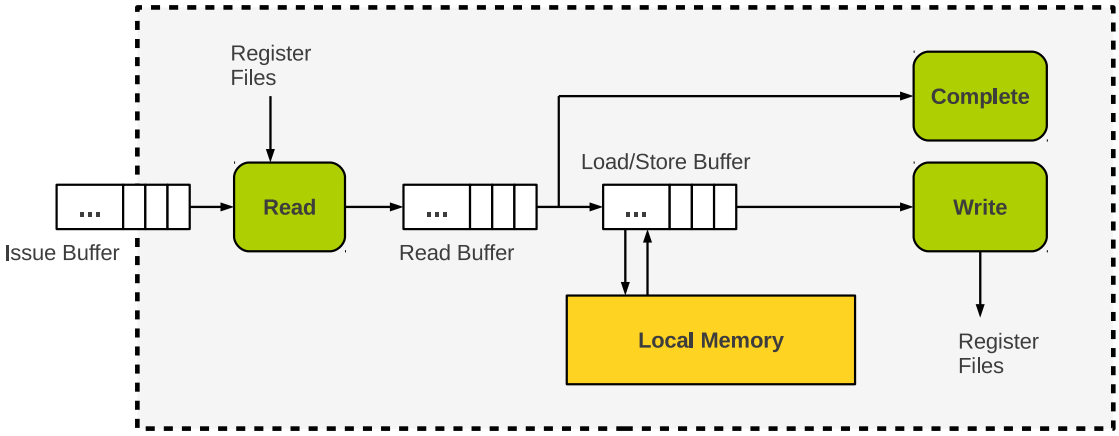


Figure 5.9: Block Diagram of a Southern Islands LDS Unit pipeline.

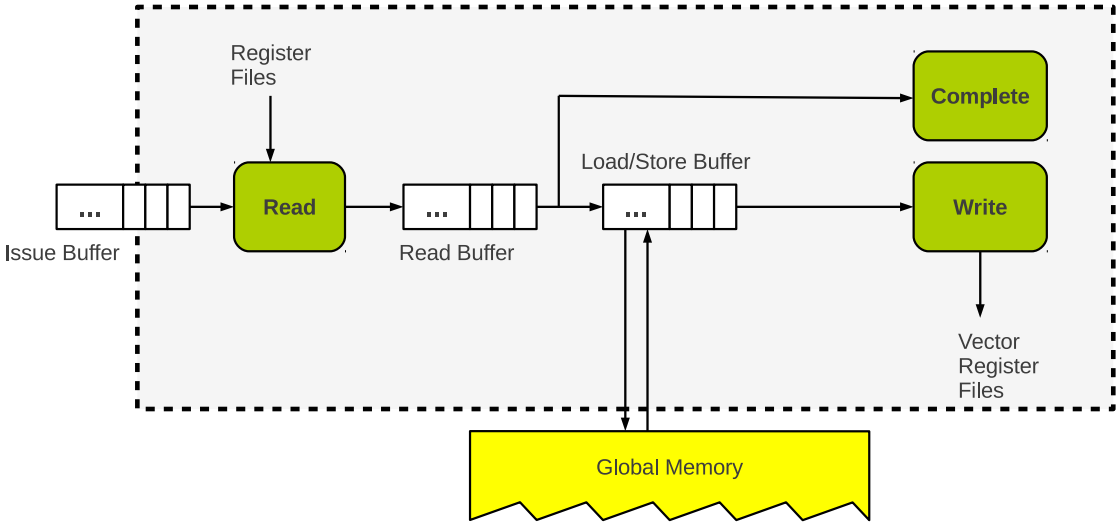


Figure 5.10: Block Diagram of a Southern Islands Vector Memory Unit pipeline.

The configuration parameters of the LDS unit can be specified in a section named `[LDSUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `Width`. Number of instruction processed by the LDS unit per cycle.
- `IssueBufferSize`. Size of the buffer which holds instructions that have been issued and are waiting to be read.
- `ReadLatency`. Number of cycles it takes to read operands from the register file.
- `ReadBufferSize`. Size of the buffer which holds instructions that have completed the read stage and are waiting to be executed.
- `MaxInflightMem`. Maximum number of of in-flight local memory operations at any time.

5.4.5 The Vector Memory Unit

The Vector Memory Unit is responsible for handling all vector global memory operations. The architecture of the vector memory unit is shown in Figure 5.10.

- **Read Stage**. The vector memory unit reads an instruction's current operands from the register files. The instruction is then placed in the read buffer to wait for execution.
- **Execute Stage**. The vector memory unit executes instructions with a variable latency dependent on the global memory configuration and state. The instructions are handled by the memory hierarchy model. A configurable limit is placed on the number of in-flight memory operations at any time.
- **Write Stage**. The vector memory unit removes all completed wavefronts, writes their results to the vector register file, and returns them to their wavefront pools in which they will wait to execute the next instruction.

The configuration parameters of the vector memory unit can be specified in a section named `[VectorMemUnit]` in the Southern Islands configuration file (option `--si-config <file>`). The allowed configuration variables are:

- `Width`. Number of instruction processed by the vector memory unit per cycle.
- `IssueBufferSize`. Size of the buffer which holds instructions that have been issued and are waiting to be read.
- `ReadLatency`. Number of cycles it takes to read operands from the register file.
- `ReadBufferSize`. Size of the buffer which holds instructions that have completed the read stage and are waiting to be executed.
- `MaxInflightMem`. Maximum number of of in-flight vector global memory operations at any time.

5.5 The Southern Islands Memory Architecture

The GPU software state is divided into three *memory scopes*, called *private memory*, *local memory*, and *global memory*. The access to each memory scope is defined by software, so there are different instructions or instruction fields specifying which memory scope is targeted in a given memory access. Private memory is accessible per work-item, local memory is shared by a work-group, and global memory is common for the whole ND-Range.

Global memory has a significantly higher latency than local and private memory. To improve its performance, Southern Islands GPUs use multiple levels of caches in the global memory scope, forming the *global memory hierarchy*. As opposed to the GPU memory scopes, accesses to different components within the global memory hierarchy are decided by hardware, transparently to the programmer, in a similar way as the cache hierarchy works on a CPU.

This section describes the model and configuration used in Multi2Sim for the private and local memory scopes. Since private and local memory are on chip and accessed by the different execution units, their configuration is discussed in this chapter. Global memory is accessed by the vector memory unit and is discussed in Chapter 6 in detail.

5.5.1 Private Memory

GPU private memory is a different way to refer to the compute unit's register files. The vector register file provides a private copy of register values for each work-item of the work-group mapped to a compute unit at a given time. The scalar register file provides register values for each wavefront mapped to a compute unit at a given time. These files are accessed by the SIMD and scalar units respectively, (see Sections 5.4.1 and 5.4.2), during the corresponding read and write stages in their respective pipelines.

Multi2Sim provides a model with no contention for register file accesses. In the worst case, a wavefront mapped to the scalar unit is accessing the register file at the same time as a wavefront mapped to the SIMD unit is trying to access it. Even in this case, the involved wavefronts are different, and their work-items will access separate regions of the register file. A per-wavefront banked organization of the register file with enough ports to feed each wavefront's work-items is the equivalent hardware implementation for a model without register file access contention.

5.5.2 Local Memory

There is one local memory module in each compute unit of the GPU, accessible to all work-items of the current work-group running on it. Local memory is accessed by LDS instructions. In a GPU, the local memory latency is higher than private memory because its capacity is higher, and each work-item has access to its entire contents. In the case of local memory, accesses happen in the read or write stages of the LDS engine pipeline. When an instruction accesses local memory, each work-item in the wavefront mapped to the LDS engine issues an access to a potentially different memory location.

Every memory instruction causes each work-item executing it to provide memory addresses based on their private copies of the registers containing them. This causes a chunk of memory address to be enqueued in an access buffer associated to the accessed memory. However, all enqueued addresses need not be translated into actual memory accesses in most of the cases. Since it is likely for adjacent work-items to access also adjacent memory locations, some contiguous accesses in the access queue might fall within the same memory block, and thus they can be coalesced into one single memory access. The coalescing degree depends on the memory block size and the generated memory addresses. The algorithm discussing the coalescing of addresses is discussed in Chapter 6.

Multi2Sim allows a flexible configuration of the local memory parameters in the `[LocalMemory]` section of the GPU configuration file, using option `--si-config <file>`. The following variables can be used:

- `Latency`. Number of cycles since the time a read/write port is allocated until the access to the memory component completes.
- `BlockSize`. Size of the block. This is the minimum access unit for a memory component. For any memory component, it determines the coalescing degree among concurrent accesses.
- `Banks`. Number of banks (N) in which the memory contents are organized. Given a continuous memory address space, bank 0 stores memory blocks starting at address 0, N, $2N$, ..., bank 1 stores memory blocks 1, $N+1$, $2N+1$, ..., and bank $N-1$ stores memory blocks $N-1$, $2N-1$, $3N-1$, etc.
- `ReadPorts`, `WritePorts`: Number of read and write ports per bank.

For each local memory element, a set of statistics is dumped in the simulation reports. The statistics can be found in the report associated with the `--si-report <file>` option, using variables prefixed with `LocalMemory` under sections `[ComputeUnit <id>]`. The set of statistics related to local memory elements and their meaning is the following. Since local memory is explicitly managed by the OpenCL program, every read/write access to local memory is a hit in the local memory.

- `Accesses`. Total number of accesses requested from a compute unit.
- `Reads`, `Writes`. Number of read requests received from a compute unit.
- `CoalescedReads`, `CoalescedWrites`. Number of reads/writes that were coalesced with previous accesses. These are requested accesses that never translated into an effective memory access,

since they matched a block targeted by a previous access in the same cycle. See Chapter 6 for more details on coalescing.

- `EffectiveReads`. Number of reads actually performed (= `Reads` − `CoalescedReads`).
- `EffectiveWrites`. Number of writes actually performed (= `Writes` − `CoalescedWrites`).

5.5.3 Global Memory

The GPU global memory, as modeled in Multi2Sim, is structured as a cache hierarchy completely configurable by the user. A dedicated configuration file is used for this purpose, passed to the simulator with the `--mem-config <file>` option. The configuration of the CPU and the GPU memory hierarchy is done in a similar manner and is discussed in Chapter 6 The statistics report of the accesses performed on each component of the global memory hierarchy can be obtained at the end of a simulation by using option `--mem-report <file>`. See Chapter 6 for details on the reported statistics.

5.6 Trying It Out

In this section, a quick guide is shown to try a simulation of an OpenCL program for Southern Islands GPUs. After downloading, unpacking, and building Multi2Sim (see Section 1.4.1), you will need to download the package containing the pre-compiled OpenCL benchmarks in the Benchmarks section of the Multi2Sim website [11]. The package is called `m2s-bench-amdapp-2.5-si.tar.gz` and can be unpacked with the following command:

```
tar -xzvf m2s-bench-amdapp-2.5-si.tar.gz
```

One of the included benchmarks corresponds to the classical matrix multiplication algorithm for GPUs (`MatrixMultiplication` directory), which is a pre-compiled version of the OpenCL sample included in AMD's Accelerated Parallel Processing (APP) software kit [8]. Two important files can be found for this benchmark:

- `MatrixMultiplication`: this is a statically linked x86 executable file, embedding a specific implementation of the OpenCL library required by Multi2Sim (more about this later). This executable has been generated from the `MatrixMultiplication.cpp` and `MatrixMultiplication.hpp` sources, not included in the package.
- `MatrixMultiplication_Kernels.bin`: this is a binary file containing the matrix multiplication OpenCL kernels compiled for the AMD Southern Islands architecture. It was generated by compiling the `MatrixMultiplication_Kernels.cl` source file, which is not included in the package either.

5.6.1 First Executions

First, let us try to run the x86 program natively to obtain a list of its possible command-line options, like this:

```
$ ./MatrixMultiplication -h
```

From the listed options, let us use `-q` to avoid a dump of big input matrices, and try to run the OpenCL program with its default matrix sizes:

```
$ ./MatrixMultiplication -q
```

The output is an error message, telling that the program cannot be run natively on your machine, because it was statically linked with the Multi2Sim OpenCL library. The result is that the first call to an OpenCL function (here `clGetPlatformIDs`) causes the library to detect that it is not being run on top of the simulator, and the program stops. Let us check out the functional simulation of the program with `m2s --si-sim functional` then:

```
$ m2s --si-sim functional MatrixMultiplication -q
```

Now the program reaches a few more steps in its execution, but the simulation stops again with an error message, notifying that the program called function `clCreateProgramWithSource`. This is an OpenCL function used to compile OpenCL kernels' source code at runtime, performing calls to platform-dependent lower levels of the OpenCL stack. Currently, Multi2Sim does not support the runtime compilation of OpenCL code, so we need the program to use the pre-compiled Southern Islands kernel binary provided in the package. Fortunately, the samples included in the APP software kit [8] provide a command-line argument `--load <file>`, that allows the user to specify an off-line compiled binary. If this option is given, the program will use the function `clCreateProgramWithBinary` instead:

```
$ m2s --si-sim functional MatrixMultiplication --load MatrixMultiplication_Kernels.bin -q
```

This executes the default input matrix sizes of 64×64. Since no output matrix is shown in this case, not much can be really appreciated in this execution, but you can add option `-e` to the sample command-line to make a self-test of the result matrix. When this option is provided, the benchmark repeats the computation using x86 code, compares the result matrix with the one obtained from the Southern Islands kernel, and dumps `Passed` or `Failed` if they match or not, respectively.

5.6.2 The Southern Islands GPU Statistics Summary

At the end of a simulation, Multi2Sim presents a summary of statistics in the standard error output (see Section 1.4.3) that follows the INI file format. If a Southern Islands functional or detailed simulation took place, a section named `[SouthernIslands]` is included in this report, including the following variables:

- `SimType`. Simualtion type, as specified in the command line. Possible values are `Functional` and `Detailed`.
- `Time`. Total Southern Islands simulation time in seconds. This value includes the time in which the Southern Islands functional/detailed simulator was active either in exclusive execution, or in parallel with other architecture simulations. It does not include the time that only other simulation modules were running. It is important to note that this is not a performance metric, it is only the real time for the simulation duration.
- `NDRangeCount`. Total number of OpenCL NDRanges enqueued to the Southern Islands GPU during this simulation.
- `Instructions`. Total number of emulated instructions.
- `InstructionsPerSecond`. Number of instructions emulated per second, calculated as the quotient of `Instructions` and `Time`. This value is not a performance metric for the guest program. It is used to measure simulation speed.

The following variables are present in the statistic summary only when detailed simulation is selected in the command line:

- `Cycles`. Number of simulation cycles during which an Southern Islands detailed simulation was active, either exclusively or in parallel with simulation of other architectures.
- `CyclesPerSecond`. Number of simulation cycles per second, calculated as the quotient of `Cycles` and `Time`. This metric does not measure performance of the guest program. It measures simulation speed.
- `IPC`. Instructions committed per cycle, calculated as the quotient of `CommittedInstructions` and `Cycles`. This value measures the performance of the guest program.

Additionally, the Southern Islands model and its associated command-line options can cause the simulation to end. This cause is recorded in variable `SimEnd` in section `[General]` of the statistics summary. Besides those values presented in Section 1.4.3, the following additional values are possible for `SimEnd`:

- `SouthernIslandsMaxInst`. The maximum number of Southern Islands instructions has been reached, as specified in command-line option `--si-max-inst <num>`. In functional simulation, this is the maximum number of emulated instructions, as represented in section `[SouthernIslands]`, variable `Instructions`; in detailed simulation, it is the maximum number of committed instructions, as shown in variable `CommittedInstructions` of the same section.
- `SouthernIslandsMaxCycles`. The maximum number of Southern Islands simulation cycles has been reached, as specified in command-line option `--si-max-cycles <num>`.
- `SouthernIslandsMaxKernels`. The maximum number of Southern Islands kernels has been reached, as specified in command-line option `--si-max-kernels <num>`.

5.6.3 The OpenCL Trace

The Multi2Sim OpenCL library interfaces with Multi2Sim in such a way that allows it to perform a detailed trace of all OpenCL calls performed by the program. Since version 4.0, Multi2Sim provides the command-line option `--si-debug-opencl <file>` for this purpose, where `<file>` is the name of the file where to dump the trace. If `stdout` is specified, the OpenCL trace will be dumped in the standard output:

```
$ m2s --si-sim functional --si-debug-opencl stdout MatrixMultiplication \
      --load MatrixMultiplication_Kernels.bin -q

clGetPlatformIDs
  'libm2s-opencl' version: 1.0.0
  num_entries=1, platforms=0x8180af8, num_platforms=0x0, version=0x10000
clGetPlatformInfo
  platform=0x10000, param_name=0x903, param_value_size=0x64,
  param_value=0xfffdfd98, param_value_size_ret=0x0

  [...]
```

Notice that this command-line option is added before the x86 program name, since it refers to a simulator option, rather than an option for the benchmark. For each OpenCL function call, the argument values are dumped, including some additional description of special arguments, such as flags or strings. The format of this output is exactly the same as that used for dumping system calls information with option `--x86-debug-syscall` option. A longer output can be observed after the `clCreateKernel` call:

```
clCreateKernel
  program=0x50008, kernel_name=0x812d132, errcode_ret=0xbffefce8
    kernel_name='mmmKernel_local'
Kernel Metadata:
;ARGSTART:__OpenCL_mmmKernel_local_kernel
;version:3:1:104
;device:tahiti
;uniqueid:1025
;memory:uavprivate:0
;memory:hwregion:0
;memory:hwlocal:0
;pointer:matrixA:float:1:1:0:uav:10:16:RO:0:0
;pointer:matrixB:float:1:1:16:uav:11:16:RO:0:0
;pointer:matrixC:float:1:1:32:uav:12:16:RW:0:0
;value:widthA:i32:1:1:48
;pointer:blockA:float:1:1:64:hl:1:16:RW:0:0
;function:1:1035
;privateid:8
;reflection:0:float4*
;reflection:1:float4*
;reflection:2:float4*
;reflection:3:int
;reflection:4:float4*
;ARGEND:__OpenCL_mmmKernel_local_kernel
```

When the call to `clCreateKernel` is performed by the program, the requested kernel is loaded from the OpenCL kernel binary file `MatrixMultiplication_Kernels.bin`, which is a file using the Executable and Linkable Format (ELF). In this case, the requested kernel is `mmmKernel_local` (notice that there can be more than one kernel embedded in a Southern Islands binary file). Multi2Sim uses its ELF file parser to locate the kernel, and extracts all its information from the file.

The portion of the Southern Islands binary associated with a kernel is in turn another embedded ELF file, which includes the kernel code with different representations, such as LLVM, AMD IL, assembly language, and most importantly, Southern Islands instructions. Multi2Sim extracts the latter, jointly with other kernel information, such as the number and type of arguments for the kernel, amount of local memory used, etc.

5.6.4 The Southern Islands ISA Trace

After the x86 program has finished setting up the OpenCL environment and the kernel input parameters, it performs a call to `clEnqueueNDRangeKernel`. This call launches the execution of the OpenCL kernel and transfers control to the Southern Islands GPU emulator, which is able to interpret AMD Southern Islands binary code. Command-line option `--si-debug-isa <file>` can be used to obtain the trace of Southern Islands instructions executed by the Southern Islands GPU functional simulator, where `<file>` is the name of the file where to dump the trace.

Let us try the execution of the matrix multiplication kernel again, this time dumping into the standard output the Southern Islands instruction trace. To make the problem simpler, let us change the default input sizes of the matrices to 8×4 and 4×4 for *MatrixA* and *MatrixB*, respectively, and the local block size to 1 single 4×4 element. This problem size generates an ND-Range containing just 2 work-items.

```
$ m2s --si-sim functional --si-debug-isa stdout MatrixMultiplication \
--load MatrixMultiplication_Kernels.bin \
-q -x 8 -y 4 -z 4 -b 1

[...]

local_size = 1 (1,1,1)
global_size = 2 (1,2,1)
group_count = 2 (1,2,1)
wavefront_count = 2
wavefronts_per_work_group = 1
 tid tid2 tid1 tid0   gid gid2 gid1 gid0   lid lid2 lid1 lid0  wavefront            work-group
   0    0    0    0     0    0    0    0     0    0    0    0  wavefront[i0-i0].0   work-group[i0-i1].0
   1    0    1    0     1    0    1    0     0    0    0    0  wavefront[i1-i1].0   work-group[i1-i2].0

 s_mov_b32     m0, 0x00008000                              // 00000000: BEFC03FF 00008000
S124<=(32768)

 s_mov_b32     m0, 0x00008000                              // 00000000: BEFC03FF 00008000
S124<=(32768)

 s_buffer_load_dwordx2  s[0:1], s[4:7], 0x04               // 00000008: C2400504
S0<=(1,1.4013e-45f) S1<=(1,1.4013e-45f)

 s_buffer_load_dwordx2  s[0:1], s[4:7], 0x04               // 00000008: C2400504
S0<=(1,1.4013e-45f) S1<=(1,1.4013e-45f)

[...]

 v_mul_f32     v2, 0x4f800000, v2                          // 0000001C: 100404FF 4F800000
t0: V2<=(4.29497e+09f)

 v_mul_f32     v2, 0x4f800000, v2                          // 0000001C: 100404FF 4F800000
t1: V2<=(4.29497e+09f)

[...]
```

At the beginning of the Southern Islands ISA trace, information about the setup of the NDRange is dumped. The local and global sizes for each dimension are listed, along with the total number of wavefronts and the number of wavefronts per work-group. A table is then given of each work-item, and includes identification flags for the three-dimensional work-item identifier (tid0, tid1, tid2), global identifier (gid0, gid1, gid2), and local identifier (lid0, lid1, lid2), and includes the wavefront and work-group identifiers. Each three dimensional identifier set also includes a fourth dimensionless identifier used by the simulator (tid, gid, lid).

After the dump of the initial setup, dumps of emulated instructions appear. These instructions correspond to the emulation of an instruction in a single wavefront and appear in order of execution. The assembly corresponding to the instruction is printed, and then a print of the updates to the device's virtual memory space. Scalar instructions act on scalar registers common to the wavefront, while vector instructions act on vector registers which hold unique values for each thread. Each active thread which makes changes to its vector registers is shown under an instruction dump, using labels "t0: V0<=(0) t1: V0<=(0) ...". Values which are stored in registers may also be shown in multiple formats if it is unknown how they should be interpreted (e.g., integer or float).

Chapter 6

The Memory Hierarchy

Multi2Sim provides a very flexible configuration of the memory hierarchy. Any number of cache levels can be used, with any number of caches in each level. Caches can be unified or separate for data and instructions, private or shared per CPU core, CPU thread, or GPU compute unit, and they can serve specific physical address ranges. In this chapter, it is shown how the memory hierarchy is modeled, configured and implemented in Multi2Sim, including caches, main memory, and interconnection networks.

6.1 Memory Hierarchy Configuration

The configuration of the memory hierarchy is specified in a plain-text INI file, passed to the simulator with option `--mem-config <file>`. Each section in the file represents a component of the memory hierarchy, formed of a set of cache modules, main memory modules, and interconnects.

Interconnects can be defined in two possible ways. The simplest way is using sections `[Network <name>]` within the memory hierarchy configuration file. In this case, a default network topology is created, consisting of a central switch with bidirectional links connected to each of the nodes (memory modules) attached to the network. Networks defined within the memory configuration file are referred to hereafter as *internal networks*.

Alternatively, interconnects can be defined externally in a network configuration file, passed to the simulator with command-line option `--net-config <file>`. This approach should be used to create networks with custom topologies with full configuration flexibility (see Chapter 7). Networks defined externally in the network configuration file are referred to as *external networks*[1].

6.1.1 Sections and Variables

In the memory configuration file, section `[General]` is used to define global parameters affecting the entire memory system. The possible variables included under this section are:

- `PageSize`. Memory page size. Virtual addresses are translated into new physical addresses in ascending order at the granularity of the page size.

Section `[Module <name>]` defines a generic memory module. This section is used to declare both caches and main memory modules accessible from CPU cores or GPU compute units. These are the possible variables in this section:

[1]The names *internal* and *external* networks do not specify any quality of the modeled network. They only refer to the configuration file where the network was defined, i.e., whether it was internally in the memory configuration file, or externally in the network configuration file.

- `Type`. Type of the memory module, where possible values are `Cache` or `MainMemory`. From the simulation point of view, the difference between a cache and a main memory module is that the former contains only a subset of the data located at the memory locations it serves.
- `Geometry`. Cache geometry, defined in a separate section of type `[Geometry <geo>]`. This variable is required when `Type` is set to `Cache`.
- `LowNetwork`. Network connecting the module with other lower-level modules, i.e., modules closer to main memory. This variable is mandatory for caches, and should not appear for main memory modules. Value `<net>` can refer to an internal network defined in a `[Network <net>]` section, or to an external network defined in the network configuration file.
- `LowNetworkNode`. If `LowNetwork` points to an external network, this variable should specify the network node that the module is mapped to. For internal networks, this variable should be omitted.
- `HighNetwork`. Network connecting the module with other higher-level modules, i.e., modules closer to CPU cores or GPU compute units. For modules that are directly accessible by CPU/GPU requesting devices, this variable should be omitted.
- `HighNetworkNode`. If `HighNetwork` points to an external network, node that the module is mapped to.
- `LowModules`. List of lower-level modules, separated by spaces. For a cache module, this variable is required. If there is only one lower-level module, it serves the entire address space for the current module. If there are several lower-level modules, each should serve a disjoint subset of the physical address space. This variable should be omitted for main memory modules.
- `BlockSize`. Block size in bytes. This variable is required for a main memory module. It should be omitted for a cache module (in this case, the block size is specified in the corresponding cache geometry section).
- `Latency`. Memory access latency in number of cycles. This variable is required for a main memory module, and should be omitted for a cache module (the access latency is specified in the corresponding cache geometry section in this case).
- `Ports`. Number of read/write ports. This variable is only allowed for a main memory module. The number of ports for a cache is specified in a separate cache geometry section.
- `DirectorySize`. Size of the directory in number of blocks. The size of a directory limits the number of different blocks that can reside in upper-level caches. If a cache requests a new block from main memory, and its directory is full, a previous block must be evicted from the directory, and all its occurrences in the memory hierarchy need to be first invalidated. This variable is only allowed for a main memory module.
- `DirectoryAssoc`. Directory associativity in number of ways. This variable is only allowed for a main memory module.
- `AddressRange`. Physical address range served by the module. If not specified, the entire address space is served by the module. There are two possible ways of defining the address space, using alternative syntax:
  - `BOUNDS <low> <high>`

    This format is used for *ranged addressing*. The module serves every address between `low` and `high`. The value in `<low>` must be a multiple of the module block size, and the value in `<high>` must be a multiple of the block size minus 1. The default value for `AddressRange` is `BOUNDS 0x0 0xffffffff`.

— `ADDR DIV <div> MOD <mod> EQ <eq>`

This format is used for *interleaved addressing*. The address space is split between different modules in an interleaved manner. If dividing an address by `<div>` and modulo `<mod>` makes it equal to `<eq>`, it is served by this module. The value of `<div>` must be a multiple of the block size. When a module serves only a subset of the address space, the user must make sure that the rest of the modules at the same level serve the remaining address space.

Section `[CacheGeometry <geo>]` defines a geometry for a cache. Caches can then be instantiated with `[Module <name>]` sections, and point to the geometry defined here. These are the possible variables:

- `Sets`. Number of sets in the cache.
- `Assoc`. Cache associativity. The total number of blocks contained in the cache is given by the product `Sets` × `Assoc`.
- `BlockSize`. Size of a cache block in bytes. The total size of the cache in bytes is given by the product `Sets` × `Assoc` × `BlockSize`.
- `Latency`. Hit latency for a cache in number of cycles.
- `Policy`. Block replacement policy. Possible values are `LRU`, `FIFO`, and `Random`.
- `MSHR`. Miss status holding register (*MSHR*) size in number of entries. This value determines the maximum number of accesses that can be in flight for the cache, including the time since the access request is received, until a potential miss is resolved.
- `Ports`. Number of ports. The number of ports in a cache limits the number of concurrent access hits. If an access is a miss, it remains in the MSHR while it is resolved, but releases the cache port.

Section `[Network <net>]` defines an internal interconnect, formed of a single switch connecting all modules pointing to the network. For every module in the network, a bidirectional link is created automatically between the module and the switch, together with the suitable input/output buffers in the switch and the module.

- `DefaultInputBufferSize`. Size of input buffers for end nodes (memory modules) and switch.
- `DefaultOutputBufferSize`. Size of output buffers for end nodes and switch.
- `DefaultBandwidth`. Bandwidth for links and switch crossbar in number of bytes per cycle. See Chapter 7 for a description of the modeled architecture for switches, buffers, and links.

Section `[Entry <name>]` creates an entry into the memory system. An entry is a connection between a CPU or GPU requesting device and a module in the memory system.

- `Type`. Type of processing node that this entry refers to. Possible values are `CPU` and `GPU`.
- `Core`. CPU core identifier. This is a value between 0 and the number of cores minus 1, as defined in the CPU configuration file (option `--x86-config <file>`). This variable should be omitted for GPU entries.
- `Thread`. CPU thread identifier. Value between 0 and the number of threads per core minus 1, as specified in the CPU configuration file. Omitted for GPU entries.
- `ComputeUnit`: GPU compute unit identifier. Value between 0 and the number of compute units minus 1, as defined in the active GPU's configuration file (options `--evg-config <file>`, `--si-config`, etc.). This variable should be omitted for CPU entries.

- `DataModule`: Module in the memory system that will serve as an entry to a CPU core/thread when reading/writing program data. The value in `<mod>` corresponds to a module defined in a section `[Module <mod>]`. Omitted for GPU entries.
- `InstModule`: Module serving as an entry to a CPU core/thread when fetching program instructions. Omitted for GPU entries.
- `Module`: Module serving as an entry to a GPU compute unit when reading/writing program data in the global memory scope. Omitted for CPU entries.

6.1.2 Memory Hierarchy Commands

In the memory hierarchy configuration file, section `[Commands]` can be used to initialize parts of the state of the memory hierarchy, as well as to perform sanity checks on its final state. This section of the configuration file is only used for debugging purposes in the verification process of the memory coherence protocol implementation, and should not be used for standard simulations based on benchmarks execution. Each variables in section `[Commands]` represents one command. Commands are represented with consecutive and bracketed indexes starting at 0:

```
[ Commands ]
Command[0] = SetBlock mod-l1-0 16 1 0x1000 E
Command[1] = Access mod-l1-0 1 LOAD 0x1000
Command[2] = CheckBlock mod-l1-0 0 0 0x0 I
Command[3] = SetOwner mod-l2-0 1 0 0 mod-il1-0
...
```

Each command is a string formed of a set of tokens separated with spaces. The possible commands can be split into three different categories, depending on whether they initialize state, schedule events, or perform sanity checks. The following commands perform state initialization:

- `SetBlock <mod> <set> <way> <tag> <state>`. Set the initial tag and state of a block in a cache memory or in the directory of a main memory module. Token `mod` is the name of the memory module, as defined in a previous section `[Module <name>]` of the memory hierarchy configuration file. Tokens `set` and `way` identify the block in the memory module. Token `tag` is the initial tag for the block, given as an hexadecimal memory address. The user should make sure that the memory address represented by `tag` is actually served by module `mod`, maps to set `set`, and is a multiple of the module's block size. Finally, `state` sets the initial MOESI state of the block, given as a capital initial letter.
- `SetOwner <mod> <set> <way> <sub_block> <owner>`. Set the owner of block at {`set`, `way`} in the directory of memory module `mod`. Token `sub_block` specifies the sub-block index when there are higher-level modules with smaller block sizes (0 if there are no sub-blocks). Finally, `owner` sets the initial owner of the block among any of the higher-level caches, as set up in the memory hierarchy configuration file. Token `owner` is a module name, as defined in a previous section `[Module <name>]`, or `None` if the sub-block should have no owner.
- `SetSharers <mod> <set> <way> <sub_block> <sharer1> [<sharer2> [<sharer3> ...]]`. Set the sharers bit-map of block {`set`, `way`} in the directory of module `mod`. Each of the given sharers corresponds to a bit in the sharers bit-map that will be initially set. Each sharer is a higher-level module, referred to by the name assigned in a previous section `[Module <name>]`. The list of sharers can be replaced by `None` if the sub-block should have no sharer.

The following commands schedule events on the memory hierarchy:

- `Access <mod> <cycle> <type> <addr>`. Perform an access on memory module `mod` (must be an L1 cache) of type `type` at address `addr`. Token `type` can be `Load`, `Store`, or `NCStore`. The access will happen in cycle `cycle`, where 1 is the first simulation cycle.

Finally, the following commands perform sanity checks at the end of the simulation:

- `CheckBlock <mod> <set> <way> <tag> <state>`. At the end of the simulation, check that block at {`set`, `way`} in module `mod` contains tag `tag` and MOESI state `state`. If it does, the simulation finishes successfully. Otherwise, Multi2Sim reports and error and terminates with exit code 1. The exit code can be checked by scripts to analyze the sanity check results. The meaning and syntax of each token is the same as in command `SetBlock`.
- `CheckOwner <mod> <set> <way> <sub_block> <owner>`. At the end of the simulation, check that the directory entry contained in sub-block `sub_block` of block at {`set`, `way`} in module `mod` has higher-level module `owner` as an owner. The owner can also be set to `None`. The syntax is the same as in command `SetOwner`.
- `CheckSharers <mod> <set> <way> <sub_block> <sharer1> [<sharer2> [<sharer3> ...]]`. At the end of the simulation, check that the directory entry contained in sub-block `sub_block` of block at {`set`, `way`} in module `mod` has a sharers bit-map where only those bits associated with `sharer1`, `sharer2`, etc. are set to 1. The list of sharers can also be replaced `None` to specify no sharer. The syntax is the same as in command `SetSharers`.

6.2 Examples of Memory Hierarchy Configurations

Some examples are shown next to help understand the format of the memory hierarchy configuration file. The presented configuration files can be found under the `samples/memory` directory of the Multi2Sim distribution package (starting at version 4.0.1). Each subdirectory contains the CPU, GPU, memory, and network configuration files needed to reproduce examples, as well as a `README` file showing the command line to run.

6.2.1 Cache Geometries

Sections describing cache geometries are needed in the memory configuration files (option `--mem-config`) of all examples. For the sake of brevity, these sections are shown in the following listing, and omitted from all memory hierarchy configuration files presented later. Geometries labeled `geo-l1` and `geo-l2` will be used throughout for L1 and L2 caches, respectively.

```
[CacheGeometry geo-l1]
Sets = 128
Assoc = 2
BlockSize = 256
Latency = 2
Policy = LRU
Ports = 2
```

```
[CacheGeometry geo-l2]
Sets = 512
Assoc = 4
BlockSize = 256
Latency = 20
Policy = LRU
Ports = 4
```

6.2.2 Example: Multicore Processor using Internal Networks

Figure 6.1 presents an example of a multicore processor with three cores. To start from this processor model, a CPU configuration file must be initially created with the following contents, passed to the simulator with option `--x86-config <file>`:

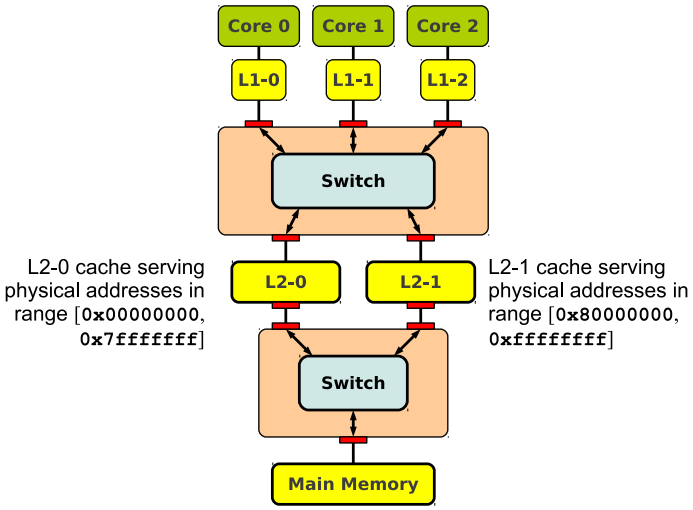


Figure 6.1: Multicore processor using internal networks.

```
[ General ]
Cores = 3
```

The memory hierarchy for this example shows a private L1 cache per core, unified for data and instruction requests. There are two L2 caches shared between all three cores. The following listing shows the memory hierarchy configuration file, where geometries `geo-l1` and `geo-l2` are reused for L1 and L2 caches, respectively:

```
[Module mod-l1-0]
Type = Cache
Geometry = geo-l1
LowNetwork = net-l1-l2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-1]
Type = Cache
Geometry = geo-l1
LowNetwork = net-l1-l2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-2]
Type = Cache
Geometry = geo-l1
LowNetwork = net-l1-l2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l2-0]
Type = Cache
Geometry = geo-l2
HighNetwork = net-l1-l2
LowNetwork = net-l2-mm
LowModules = mod-l2-mm
AddressRange = BOUNDS 0x00000000 0x7FFFFFFF

[Network net-l1-l2]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Network net-l2-mm]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256
```

```
[Module mod-l2-1]
Type = Cache
Geometry = geo-l2
HighNetwork = net-l1-l2
LowNetwork = net-l2-mm
LowModules = mod-l2-mm
AddressRange = BOUNDS 0x80000000 0xFFFFFFFF

[Module mod-mm]
Type = MainMemory
BlockSize = 256
Latency = 200
HighNetwork = net-l2-mm

[Entry core-0]
Arch = x86
Core = 0
Thread = 0
DataModule = mod-l1-0
InstModule = mod-l1-0

[Entry core-1]
Arch = x86
Core = 1
Thread = 0
DataModule = mod-l1-1
InstModule = mod-l1-1

[Entry core-2]
Arch = x86
Core = 2
Thread = 0
DataModule = mod-l1-2
InstModule = mod-l1-2
```

Additionally, two internal interconnection networks are used, one connecting L1 caches with L2 caches, defined in section `[net-l1-l2]`, and another connecting L2 caches with main memory, defined in section `[net-l2-mm]`. The processor cores are connected to the L1 caches using the `[Entry <name>]` sections. Memory modules are connected to lower networks (i.e., networks closer to main memory) using the `LowNetwork` variable, and to higher networks (i.e., closer to the CPU) using the `HighNetwork` variable. Variable `LowNetworkModules` is used to specify which lower memory modules serve misses on a cache. For example, cache `mod-l1-0` specifies two lower modules (`mod-l2-0` and `mod-l2-1`) to be accessed upon a miss, each of which provides a different subset of the entire physical address space.

6.2.3 Example: Multicore with External Network

In this example, the default L1-to-L2 network configuration provided in the previous 3-core processor example is replaced with a custom interconnect (external network), declared in a separate network configuration file. The block diagram for this example is shown in Figure 6.2. Reusing the cache geometries defined in Section 6.2.1, the listing below shows the contents of the memory hierarchy configuration file:

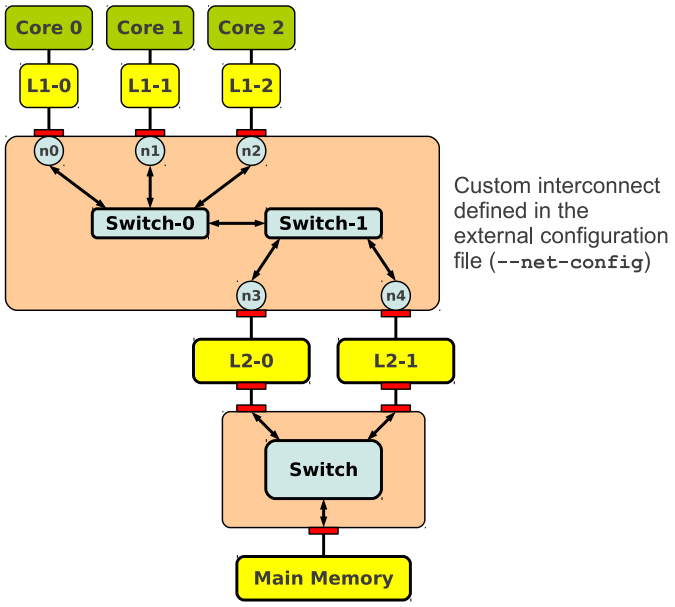


Figure 6.2: Multicore with external networks.

```
[Module mod-l1-0]
Type = Cache
Geometry = geo-l1
LowNetwork = net0
LowNetworkNode = n0
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-1]
Type = Cache
Geometry = geo-l1
LowNetwork = net0
LowNetworkNode = n1
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-2]
Type = Cache
Geometry = geo-l1
LowNetwork = net0
LowNetworkNode = n2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l2-0]
Type = Cache
Geometry = geo-l2
HighNetwork = net0
HighNetworkNode = n3
LowNetwork = net-l2-mm
AddressRange = BOUNDS 0x00000000 0x7FFFFFFF
LowModules = mod-mm

[Module mod-l2-1]
Type = Cache
Geometry = geo-l2
HighNetwork = net0
HighNetworkNode = n4
LowNetwork = net-l2-mm
AddressRange = BOUNDS 0x80000000 0xFFFFFFFF
LowModules = mod-mm

[Module mod-mm]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net-l2-mm

[Network net-l2-mm]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Entry core-0]
Arch = x86
Core = 0
Thread = 0
DataModule = mod-l1-0
InstModule = mod-l1-0

[Entry core-1]
Arch = x86
Core = 1
Thread = 0
DataModule = mod-l1-1
InstModule = mod-l1-1

[Entry core-2]
Arch = x86
Core = 2
Thread = 0
DataModule = mod-l1-2
InstModule = mod-l1-2
```

The three `[Entry <name>]` sections define the entry points to the memory hierarchy, i.e., the connections between the CPU cores and their associated L1 caches. Network `net-l2-mm` is defined within

the memory hierarchy configuration file, so it is automatically created with a default topology, using a single switch and one bidirectional link per node. On the contrary, network `net-0` referenced in section `[Module mod-l1-0]` is not defined in the same file. Thus, the simulator will expect to find this network definition in the network configuration file passed with option `--net-config <file>`, listed below.

The L1-to-L2 network consists of two switches and five end nodes, each associated with an L1 or L2 cache module. The nodes associated with L1 caches are connected to one switch (`sw0`) and the L2 nodes are connected to another switch (`sw1`). Three bidirectional links are defined between nodes `n0`...`n2` and switch `sw0`, and two more bidirectional links are created between nodes `n3`...`n4` and switch `sw1`. Finally, an additional bidirectional link is defined between the two switches `sw0` and `sw1`. The following code shows the contents of the network configuration file:

```
[Network.net0]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Network.net0.Node.sw0]
Type = Switch

[Network.net0.Node.n0]
Type = EndNode

[Network.net0.Node.n1]
Type = EndNode

[Network.net0.Node.n2]
Type = EndNode

[Network.net0.Node.sw1]
Type = Switch

[Network.net0.Node.n3]
Type = EndNode

[Network.net0.Node.n4]
Type = EndNode

[Network.net0.Link.sw0-n0]
Source = sw0
Dest = n0
Type = Bidirectional

[Network.net0.Link.sw0-n1]
Source = sw0
Dest = n1
Type = Bidirectional

[Network.net0.Link.sw0-n2]
Source = sw0
Dest = n2
Type = Bidirectional

[Network.net0.Link.sw0-sw1]
Source = sw0
Dest = sw1
Type = Bidirectional

[Network.net0.Link.sw1-n3]
Source = sw1
Dest = n3
Type = Bidirectional

[Network.net0.Link.sw1-n4]
Source = sw1
Dest = n4
Type = Bidirectional
```

When an external network is given, memory components defined in the memory hierarchy configuration file connected to that network need to specify the network node that they are mapped to. For example, notice in the listings above how memory module `mod-l1-0` (defined in section `[Module mod-l1-0]` of the memory hierarchy configuration file) is connected to external network `net-0` (variable `LowNetwork`) and is mapped with node `n0` of that network (variable `LowNetworkNode`). Node `n0` is a member of network `net-0` defined in section `[Network.net-0.Node.n0]` of the network configuration file.

6.2.4 Example: Multicore with Ring Network

A more complex example is represented in Figure 6.3, using a 4-core processor. The cores have private L1 data caches, and a common L1 instruction cache is shared every two cores. The `[Entry <name>]` sections in the memory hierarchy configuration file are responsible for doing the association between CPU cores and data or instruction caches, by assigning values to the `InstructionModule` and `DataModule` variables. The network declared between the L1 and L2 is an internal network with default topology, while the network between the L2 caches and the main memory modules is an

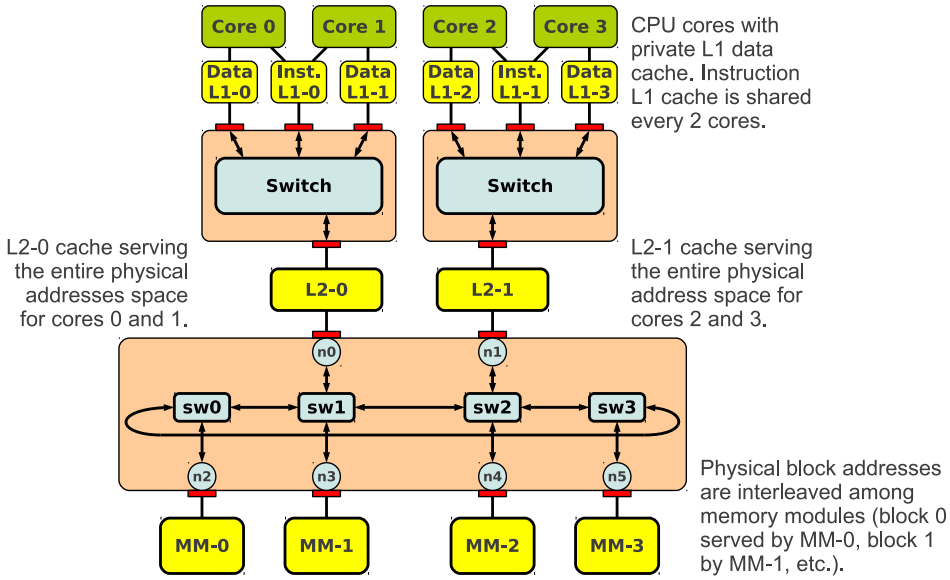


Figure 6.3: Multicore with ring network.

external network with custom topology.

In this example, the two L2 caches serve independent sets of higher-level (L1) caches, so each L2 can serve the entire address space. Thus, the `AddressRange` variable is not given for the L2 modules. Main memory is configured using a banked organization with four banks, using the alternative syntax for the value of `AddressRange` in the main memory modules. The memory hierarchy configuration file is listed next.

```
[Module mod-l1-0]
Type = Cache
Geometry = geo-d-l1
LowNetwork = net-l1-l2-0
LowModules = mod-l2-0

[Module mod-l1-1]
Type = Cache
Geometry = geo-d-l1
LowNetwork = net-l1-l2-0
LowModules = mod-l2-0

[Module mod-l1-2]
Type = Cache
Geometry = geo-d-l1
LowNetwork = net-l1-l2-1
LowModules = mod-l2-1

[Module mod-l1-3]
Type = Cache
Geometry = geo-d-l1
LowNetwork = net-l1-l2-1
LowModules = mod-l2-1

[Module mod-il1-0]
Type = Cache
Geometry = geo-i-l1
LowNetwork = net-l1-l2-0
LowModules = mod-l2-0

[Module mod-il1-1]
Type = Cache
Geometry = geo-i-l1
LowNetwork = net-l1-l2-1
LowModules = mod-l2-1

[Network net-l1-l2-0]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Network net-l1-l2-1]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Module mod-l2-0]
Type = Cache
Geometry = geo-l2
HighNetwork = net-l1-l2-0
LowNetwork = net0
LowNetworkNode = n0
LowModules = mod-mm-0 mod-mm-1 mod-mm-2 mod-mm-3

[Module mod-l2-1]
Type = Cache
Geometry = geo-l2
HighNetwork = net-l1-l2-1
LowNetwork = net0
LowNetworkNode = n1
LowModules = mod-mm-0 mod-mm-1 mod-mm-2 mod-mm-3
```

```
[Module mod-mm-1]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net0
HighNetworkNode = n3
AddressRange = ADDR DIV 256 MOD 4 EQ 1

[Module mod-mm-2]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net0
HighNetworkNode = n4
AddressRange = ADDR DIV 256 MOD 4 EQ 2

[Module mod-mm-3]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net0
HighNetworkNode = n5
AddressRange = ADDR DIV 256 MOD 4 EQ 3

[Entry core-0]
Arch = x86
Core = 0
Thread = 0
DataModule = mod-l1-0
InstModule = mod-il1-0

[Entry core-1]
Arch = x86
Core = 1
Thread = 0
DataModule = mod-l1-1
InstModule = mod-il1-0

[Entry core-2]
Arch = x86
Core = 2
Thread = 0
DataModule = mod-l1-2
InstModule = mod-il1-1

[Entry core-3]
Arch = x86
Core = 3
Thread = 0
DataModule = mod-l1-3
InstModule = mod-il1-1
```

The ring network `net0` is defined to connect L2 cache modules with main memory modules. Two end nodes required for the L2 cache modules, and four additional end nodes are associated with the four main memory modules. Four switches are connected forming a ring, where each of them is connected to one main memory module, and only two of them are connected to L2 caches. The listing below shows the contents of the network configuration file.

```
[Network.net0]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Network.net0.Node.sw0]
Type = Switch

[Network.net0.Node.sw1]
Type = Switch

[Network.net0.Node.sw2]
Type = Switch

[Network.net0.Node.sw3]
Type = Switch

[Network.net0.Node.n0]
Type = EndNode

[Network.net0.Node.n1]
Type = EndNode

[Network.net0.Node.n2]
Type = EndNode

[Network.net0.Node.n3]
Type = EndNode

[Network.net0.Node.n4]
Type = EndNode

[Network.net0.Node.n5]
Type = EndNode

[Network.net0.Link.sw0-n2]
Source = sw0
Dest = n2
Type = Bidirectional

[Network.net0.Link.sw1-n3]
Source = sw1
Dest = n3
Type = Bidirectional

[Network.net0.Link.sw2-n4]
Source = sw2
Dest = n4
Type = Bidirectional

[Network.net0.Link.sw3-n5]
Source = sw3
Dest = n5
Type = Bidirectional

[Network.net0.Link.sw1-n0]
Source = sw1
Dest = n0
Type = Bidirectional

[Network.net0.Link.sw2-n1]
Source = sw2
Dest = n1
Type = Bidirectional

[Network.net0.Link.sw0-sw1]
Source = sw0
Dest = sw1
Type = Bidirectional

[Network.net0.Link.sw1-sw2]
Source = sw1
Dest = sw2
Type = Bidirectional

[Network.net0.Link.sw2-sw3]
Source = sw2
Dest = sw3
Type = Bidirectional

[Network.net0.Link.sw3-sw0]
Source = sw3
Dest = sw0
Type = Bidirectional
```

Notice that a ring topology contains contains a cycle of network links, which can cause deadlocks when routing packets between input and output buffers or intermediate switches and end nodes. See Chapter 7 for more details regarding routing tables and deadlocks.

6.2.5 Heterogeneous System with CPU and GPU cores

This example presents an heterogeneous system with one x86 CPU core and four Evergreen GPU compute units, as represented in Figure 6.4. It also illustrates how each thread of the multi-threaded x86 core can have its own entry into the memory hierarchy through a private L1. The Evergreen compute units share one single L1 cache. The `[Entry <name>]` sections in the memory hierarchy configuration file are responsible for these associations. Main memory modules and global memory modules are declared in a similar manner for both the CPU and the GPU, using `[Module <name>]` sections where variable `Type` is equal to to `MainMemory`. The following listing shows the memory hierarchy configuration file.

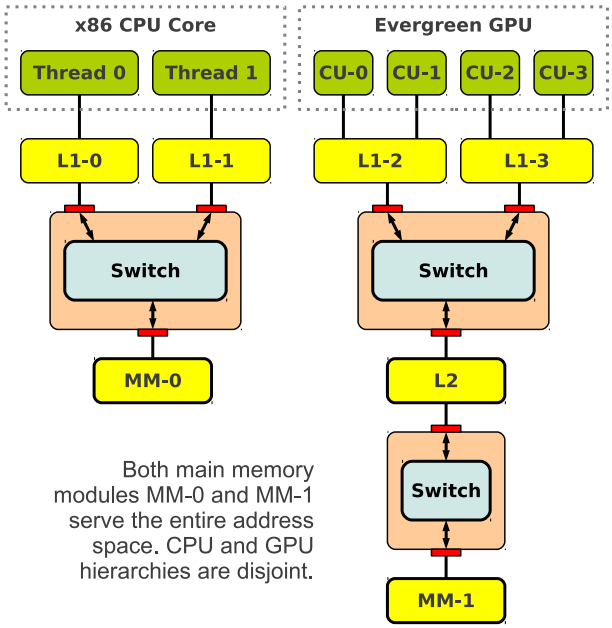


Figure 6.4: Heterogeneous system with one multithreaded x86 CPU core and four Evergreen GPU compute units.

```
[Module mod-gpu-l1-0]
Type = Cache
Geometry = geo-gpu-l1
LowNetwork = net-gpu-l1-l2
LowModules = mod-gpu-l2-0

[Module mod-gpu-l1-1]
Type = Cache
Geometry = geo-gpu-l1
LowNetwork = net-gpu-l1-l2
LowModules = mod-gpu-l2-0

[Module mod-gpu-l2-0]
Type = Cache
Geometry = geo-gpu-l2
HighNetwork = net-gpu-l1-l2
LowNetwork = net-gpu-l2-mm
LowModules = mod-gpu-mm

[Network net-gpu-l1-l2]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Network net-gpu-l2-mm]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Module mod-gpu-mm]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net-gpu-l2-mm

[Entry gpu-cu-0]
Arch = Evergreen
ComputeUnit = 0
Module = mod-gpu-l1-0

[Entry gpu-cu-1]
Arch = Evergreen
ComputeUnit = 1
Module = mod-gpu-l1-0

[Entry gpu-cu-2]
Arch = Evergreen
ComputeUnit = 2
Module = mod-gpu-l1-1

[Entry gpu-cu-3]
Type = GPU
ComputeUnit = 3
Module = mod-gpu-l1-1

[Module mod-cpu-l1-0]
Type = Cache
Geometry = geo-cpu-l1
LowNetwork = net-cpu-l1-mm
LowModules = mod-cpu-mm

[Module mod-cpu-l1-1]
Type = Cache
Geometry = geo-cpu-l1
LowNetwork = net-cpu-l1-mm
LowModules = mod-cpu-mm

[Network net-cpu-l1-mm]
DefaultInputBufferSize = 1024
DefaultOutputBufferSize = 1024
DefaultBandwidth = 256

[Module mod-cpu-mm]
Type = MainMemory
BlockSize = 256
Latency = 100
HighNetwork = net-cpu-l1-mm

[Entry core-0]
Arch = x86
Core = 0
Thread = 0
DataModule = mod-cpu-l1-0
InstModule = mod-cpu-l1-0

[Entry core-1]
Arch = x86
Core = 0
Thread = 1
DataModule = mod-cpu-l1-1
InstModule = mod-cpu-l1-1
```

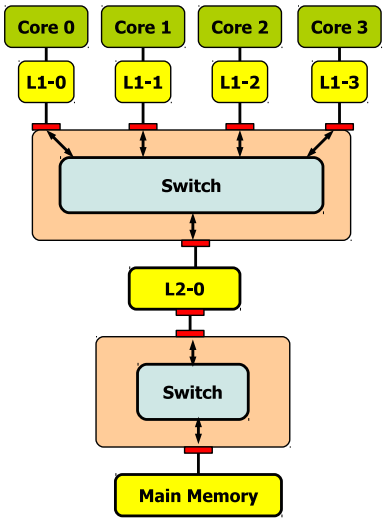


Figure 6.5: Default memory hierarchy configuration

All networks shown in this example are internal networks, so no additional network configuration file is required. The code for the x86 CPU configuration file, passed with option `--x86-config <file>`, is shown below:

```
[ General ]
Cores = 1
Threads = 2
```

Finally, the following listing shows the code for the Evergreen GPU configuration file, passed with option `--evg-config <file>`.

```
[ Device ]
NumComputeUnits = 4
```

6.3 Default Configuration

When the memory hierarchy configuration file is omitted for a detailed simulation, Multi2Sim provides the default hierarchy shown in Figure 6.5. It is composed of individual L1 caches per CPU core, unified for instructions and data, and shared for every hardware thread if the cores are multithreaded. A default interconnect based on a single switch connects all L1 caches with a common L2 cache, and another default network connects the L2 cache with a single main memory module. A similar configuration is created automatically for the GPU memory hierarchy, using private L1 caches per compute unit, and a single L2 cache and global memory module.

6.4 Cache Coherence

Multi2Sim implements the MOESI protocol [12] to maintain coherence between caches in the same level of the memory hierarchy. No matter what flexible configuration is chosen for the memory

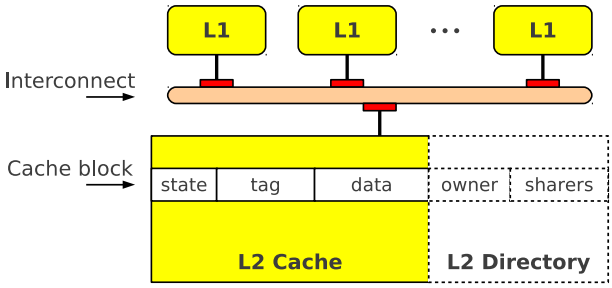


Figure 6.6: Directory attached to an L2 cache to enforce coherence between itself and all L1 caches sharing it.

hierarchy, it all can stay coherent by guaranteeing that a set of caches in a given cache level are coherent with their lower-level cache, i.e., the next cache closer to main memory. The MOESI protocol enforces coherence with the concept of write exclusiveness; before writing into a cache block, the protocol first ensures that the target cache owns an exclusive copy of the block.

6.4.1 Cache Directories

To enforce coherence between a cache in level N (where $N > 1$) and a cache closer to the processor in level $N-1$, a directory is attached to the level-N cache, as represented in Figure 6.6. A directory has one entry for each block present in its corresponding cache. Together, a cache block and its associated directory entry contain the following fields:

- `State`. The state of the block can be any of the five MOESI states (Modified, Owned, Exclusive, Shared, or Invalid). A real cache needs 3 bits to encode these five possible states.
- `Tag`. If the block is in any state other than I, this field uniquely identifies the address of the block in the entire memory hierarchy. In a system with 32-bit physical addresses and a set-associative cache with `Sets` sets and blocks of `BlockSize` bytes, the number of bits needed for the tag is equal to $32 - log_2(\texttt{BlockSize}) - log_2(\texttt{Sets})$.
- `Data`. Block data of `BlockSize` bytes.
- `Owner`. This field contains the identifier of the higher-level cache that owns an exclusive copy of this block, if any. A special value is reserved to refer to the fact that no exclusive copy of the block exists. For an L2 cache with n L1 caches connected to it, this field has $\lceil log_2(n+1) \rceil$ bits.
- `Sharers`. Bit mask representing the higher-level caches sharing a copy of the block, either exclusive or non-exclusively. This field has n bits for an L2 cache with n L1 caches connected to it.

The first three fields are contained in a cache block, while the three last fields comprise a directory entry. L1 caches do not attach a directory, since they lack higher-level caches to keep coherence among. Thus, the presence of a block in a first-level cache only requires storing fields `State`, `Tag`, and `Data`.

6.4.2 Main Memory Directories

A special case of directories are those associated with main memory modules, in the sense that they are not attached to a cache whose number of blocks can now be used to deduce the directory size. If a memory module is configured to serve the entire 32-bit physical address space, its size is

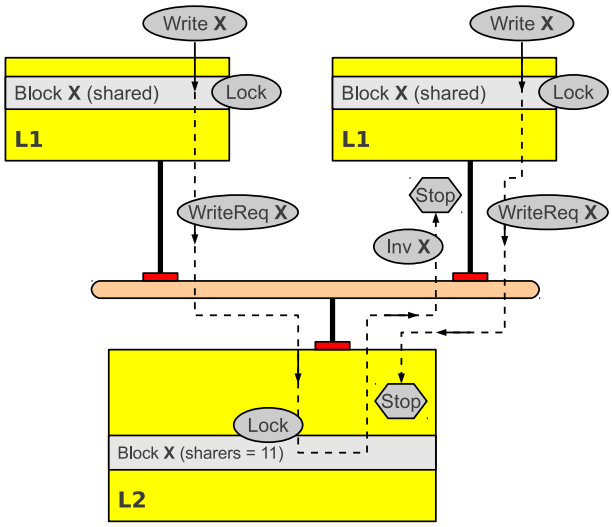


Figure 6.7: Example of a deadlock situation.

assumed to be 4GB. In general, a main memory module is assumed to contain all data associated with the addresses that it serves, without possibly causing a miss that would require disk swapping on a real system.

However, Multi2Sim accurately models the size of a directory associated with main memory, with the difference that this size has to be explicitly given in this case, instead of just deduced from an associated cache size. In the memory configuration file, a module of `Type = Memory` defines the properties of its associated directory using variables `DirectorySize` and `DirectoryAssoc`. The number of blocks present in this directory at a given time limits the blocks that can be contained in any higher-level cache directly or indirectly connected to the main memory module. Please refer to Section 6.1 for more details on the format of the memory configuration file.

6.4.3 Deadlocks

When a cache access involves several coherence actions, more than one cache block may be updated at once. For this aim, the associated directory entries must be first locked, and only after all of them are safely held, the actual block states are changed. This might lead to a deadlock situation when two coherence transactions are in flight, and one of them is requesting some directory entry that has been previously locked by the other, and vice versa.

Figure 6.7 shows an example of a deadlock situation in a memory hierarchy portion composed by two L1 caches where coherence is maintained with respect to a shared L2 cache. In this example, a block `x` is contained in both L1 caches in a *shared* state (read-only), so the directory entry associated with `x` in the L2 cache contains a sharer mask with both sharers set to 1. The deadlock occurs when a write access is issued to blocks `x` in both L1 caches at the same time.

After the writes are issued, the L1 directory entries are locked, and one write request per L1 cache is sent to L2. The interconnect controller serializes both requests, and eventually one of them reaches L2 first (assumed is that the request sent by the L1 cache on the left is first). This request locks the L2 directory entry associated with `x`, and reads the sharer mask; after observing that there is another sharer of the block, an invalidation is sent upwards toward the L1 cache on the right.

When this invalidation reaches the L1 cache, it tries to lock the directory entry for `x`. The

attempt fails, so the invalidation waits until the directory entry is released. At the same time, the write request sent by the L1 cache on the right is trying to lock the L2 directory entry for `x`, which is already locked by the coherence transaction triggered by the L1 on the left.

The solution to this deadlock situation implemented by Multi2Sim consists in prioritizing those requests traveling from lower to upper cache levels, and canceling downward requests whenever they enter in conflict with some other coherence transaction. In other words, if a down-up request stumbles upon a locked directory entry, it waits until it is released. On the contrary, an up-down request unable to lock a directory entry gives up and travels backwards, releasing all directory entries locked before.

In the example shown in Figure 6.7, the write request on the right is canceled, and the directory entry for `x` in the L1 cache on the right is released. Thus, the invalidation coming from the L2 cache can proceed, and the cache access started on the left can eventually finish. The canceled L1 access is then resumed after a specific time. This time is chosen as a variable random number of cycles in order to guarantee forward progress when multiple L1 caches contend for the same block.

6.5 Statistics Report

A detailed report of the memory hierarchy simulation is dumped in the file specified by the `--mem-report` option. The memory hierarchy statistics report follows the INI file format, containing one section per cache, main memory module, and interconnect. For each interconnect, the statistics report includes those sections and variables specified in the description for the network statistics report (Section 7.8).

For each cache or main memory module, the statistics report shows a section `[<name>]`, where `<name>` is the module name specified in section `[Module <name>]` of the memory hierarchy configuration file. The following variables are provided under this section:

- `Accesses`. Number of accesses to the cache.
- `Hits`, `Misses`. Block hits and misses. Their sum is equal to `Accesses`.
- `HitRatio`. Number of hits divided by the number of accesses.
- `Evictions`. Number of blocks evicted from the cache due to a block replacement (block was selected by the local block replacement policy), or due to a remote block invalidation (a remote write request is received).
- `Reads`, `Writes`. Block reads and writes. Their sum is equal to `Accesses`.
- `ReadHits`, `WriteHits`. Hit reads and hit writes.
- `ReadMisses`, `WriteMisses`. Missed reads and missed writes.
- `NonBlockingReads`, `NonBlockingWrites`. Non-blocking accesses occur when the read/write request comes from an upper-level (up→down) element (i.e., L1 requesting an L2 block, or a processor requesting an L1 block). These statistics track their occurrences.
- `BlockingReads`, `BlockingWrites`. Blocking accesses occur when a read/write request comes from a lower-level (down→up) element (i.e., main memory requesting an L2 block, or L2 block invalidating an L1 block). The sum of blocking reads (writes) and non-blocking reads (writes) is equal to the values reported by `Reads` (`Writes`).

To prevent deadlocks, a cache access might be canceled and retried after a random number of cycles (see Section 6.4.3). The simulation statistics distinguish the retried accesses by reporting the following values:

- `Retries`. Number of retried accesses. This value is always 0 for a cache other than L1.
- `ReadRetries`, `WriteRetries`. Number of retried reads/writes. The sum of these values is equal to `Retries`.
- `NoRetryAccesses`. Number of successful accesses, equal to `Accesses−Retries`.
- `NoRetryHits`, `NoRetryMisses`. Successful accesses resulting in hits/misses. Their sum is equal to `NoRetryAccesses`.
- `NoRetryHitRatio`. Hit ratio for successful accesses, equal to `NoRetryHits` divided by `NoRetryAccesses`.
- `NoRetryReads`, `NoRetryWrites`. Successful reads/writes. Their sum is equal to `NoRetryAccesses`.
- `NoRetryReadHits`, `NoRetryWriteHits`. Successful read/write hit. Directory entries or blocks were successfully locked and the searched cache line was found.
- `NoRetryReadMisses`, `NoRetryWriteMisses`. Directory entries or blocks were successfully locked, but the searched cache line was not present and it had to be requested somewhere else.

Chapter 7

Interconnection Networks

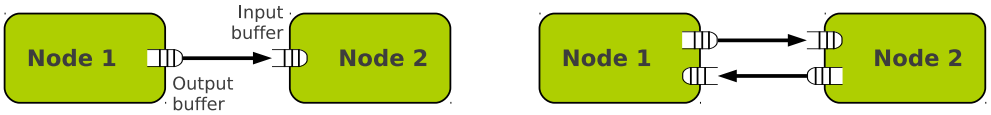
Multi2Sim provides a flexible model of interconnection networks between different cache levels in the memory hierarchy. The network library in Multi2Sim gives users the ability to manage these connections. This chapter shows the interconnect model capabilities and configuration, as well as the description of statistic reports.

7.1 Model Description

In Multi2Sim, interconnects can be defined in two different configuration files: the memory hierarchy configuration file passed with option `--mem-config <file>`, or the network configuration file itself, passed with option `--net-config <file>`. In either case, the network model includes a set of end nodes, a set of switch nodes, a set of links connecting input with output buffers of pairs of nodes, and a two-dimensional routing table. For networks defined in the memory configuration file, please see Chapter 6. The rest of this chapter focuses on custom networks defined within the network configuration file.

To configure a new network, the user needs to enumerate the nodes in the network configuration file. Nodes in the network are classified as end nodes or switch nodes. An end node can send and receive packets, using another end node as a source or destination node. Switches can only forward packets between end nodes and other switches.

The user must also specify a set of links between pairs of nodes. A link is used to connect one end node with a switch node (i.e., two end nodes cannot be connected together), and can be defined as unidirectional or bidirectional. A unidirectional link connects one output buffer of a source node to one input buffer of a destination node, while a bidirectional link creates an additional connection between one output buffer of the destination node and one input buffer of the source node. Input and output buffers are created implicitly for the nodes every time a new link is defined, as shown in Figure 7.1.



a) Unidirectional link. b) Bidirectional link.

Figure 7.1: Connection of two nodes with a unidirectional or bidirectional link, with an implicit creation of input and output buffers in source and destination nodes.

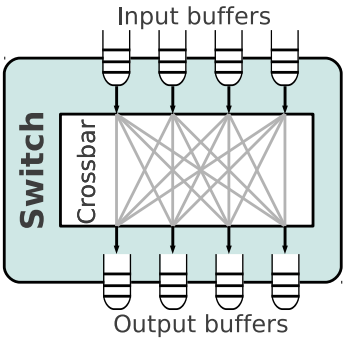


Figure 7.2: Model of the switch architecture.

The internal architecture of a switch is shown in Figure 7.2. For each incoming connection, the switch incorporates an input buffer, and an output buffer is created for each outgoing link. The switch includes a crossbar that communicates all input buffers with all output buffers. Any packet at the head of an input buffer can be routed by the crossbar into the tail of any output buffer of the switch. If several input buffers try to reach the same output buffer, an arbiter routes packets from each input buffer in a round-robin fashion.

7.2 Communication Model

Information exchange between nodes is in form of packets. A packet transmission starts at an end node and finishes at another end node, possibly traversing a set of intermediate switches. The transmitted packet can meet different contention points during its lifetime in its route. *Permanent* contentions are those that will lead to unsuccessful transmission and simulation failure, while *temporary* contentions just delay the successful transmission of a packet.

A permanent contention occurs for two main reason. The first reason is the lack of a possible path from source to destination. When configuring a network topology, the user must make sure that an end node can reach every other end node that it needs to communicate with. The network should include enough links to allow for this communication. The second reason for a permanent contention is an insufficient capacity for any of the input or output buffers of any intermediate switch or end node. All buffers involved in the packet transmission should be equal or larger than the packet attempted to be sent. Three effects can cause temporary contention delaying packet transfer time. First, a packet is retained in a buffer if it is not located at its head. This effect is well known as head-of-line blocking. Second, a packet located at the head of a buffer can experiment contention if the link it is requesting is currently occupied with another packet transmission. Finally, a packet also has to wait at a buffer's head when the following buffer on the route is full. Only when new entries are released in the destination buffer, the packet is able to continue its way to the destination node.

7.3 Routing

Routing tables can be configured in Multi2Sim automatically or manually. If no routes are specified in the network configuration file, the simulator uses the Floyd-Warshall algorithm to initialize the routing table based on the existing links between the nodes. The routing table contains the shortest paths for every pair of nodes in the network. The manual mode for the routing table configuration

is activated if the presence of an additional section `[Network.<name>.Routes]` is found in the network configuration file, where `<name>` is the name of a previously defined network. In this case, only those routes specified by the user are eligible for a packet. For every pair of nodes that can be reached from one another, the user needs to specify what is the next hop traversed in the network. Nodes that are at a 1-hop distance are implicitly accessible.

For example, let us assume a network with two end nodes `n1` and `n2`, and one switch `sw`. Node `n1` is connected to `sw` through a unidirectional link, and `sw` is in turn connected to `n2`. In the manual configuration, the user must specify that node `n1` needs to route a packet through `sw` if its final destination is `n2`. This is done by including this entry in the configuration file: `n1.to.n2 = sw`. Once the packet is in `sw`, it knows that the next step is traversing the link that connects it to `n2`, since it is placed at a 1-hop distance. A more sophisticated example is presented in Section 7.6.

When a route is specified between two nodes, there must be a link connecting the source node and the next hop. For example, the routing entry `n1.to.n2 = sw` relies on a previously created link between `n1` and `sw`. The user must also make sure that all possible routes potentially followed by packets are specified. The ability to route packets between every pair of nodes is not checked by Multi2Sim at start up. Instead, execution will stop with an error message if a packet is sent between unreachable nodes during simulation.

Multi2Sim also provides the potential to use Virtual Channels (VCs) between two nodes. A virtual channel is a communication path built on a time-multiplexed physical link and consuming a portion of its total bandwidth but are nowadays also leveraged to improve network latency and throughput. Each unidirectional virtual channel is realized by its own private pair of input and output buffers. Virtual channels were originally presented to solve the problem of deadlocks in networks.

Deadlock is a network state where no messages can advance because each message requires to be forwarded to a buffer which is occupied by another message. When a network contains cycles, the occurrence of deadlock is probable.It is the responsibility of Multi2Sim's user to make sure that the routing table does not contain cycles, which could lead to deadlocks while transmitting packets during simulation time. When the routing table is created automatically, the network topology should be cycle-free, that is, no node should be reachable from itself after initially traversing one of its output links. When the routing table is manually configured, the only requirement is to avoid cycles in the routes. Even if the topology contains cycles in this case, the network is deadlock-free if the routing table omits them.

Multi2Sim checks for cycles in the routing table at the beginning of the simulation. If one is found, a warning is shown, but the simulation still continues. Whether the routing cycle entails an effective deadlock, then, depends on the specific combination of transferred packets. Note that a simulation silently stalls upon a deadlock occurrence, potentially leaving the user wondering about unexpectedly long and seemingly endless simulation times.

7.4 Network Configuration

The network configuration file is a plain-text INI file (see Section 10.1), passed to the simulator with option `--net-config <file>`. Multiple interconnects can be defined in the same configuration files, together with its nodes, and their connections. A new network is created for each section named `[Network.<name>]`. The string specified in `<name>` is used later in the network configuration file, as well as the memory hierarchy configuration file, to refer to the defined network. The variables contained in this section are:

- `DefaultInputBufferSize` (required). Default size for input buffers in nodes and switches, specified in number of packets. If the section creating a new end node or switch does not specify a size for its input buffers, this default size will be used instead.
- `DefaultOutputBufferSize` (required). Default size for output buffers in nodes and switches in number or packets. Upon creation of a switch or node in the network, the simulator uses this size, unless a different value is given in the node section.
- `DefaultBandwidth` (required). Default bandwidth for links in the network, specified in number of bytes per cycle. If a link's bandwidth is not specified, the simulator uses this value.

A network node is created with a new section following the pattern `[Network.<network>.Node.<node>]`, where `<network>` is a previously defined interconnect, and `<node>` is the name of the node. The string in `node` is used to refer to this node from other sections in the network configuration file, the memory hierarchy configuration file, and statistic reports. The following variables are associated with each node:

- `Type`. Type of the node. Possible options are `EndNode` or `Switch`.
- `InputBufferSize` (optional). Size of the input buffer in number of packets. If not present, the input buffer size is set to the value specified in `DefaultInputBufferSize` of the corresponding network section.
- `OutputBufferSize` (optional). Size of output buffers in number of packets. If not present, the output buffer size is set to the value specified in `DefaultOutputBufferSize` of the corresponding network configuration.
- `Bandwidth` (optional). For switches, bandwidth of internal crossbar communicating input with output buffers. If not present, the value is set to the value specified in `DefaultBandwidth` of the corresponding network section. For end nodes, this variable is ignored.

New links are created with sections following the pattern `[Network.<network>.Link.<link>]`, where `<network>` is the network name, and `<link>` is the name of the link. The string in `link` is used to refer to this link from other sections in the network configuration file, memory hierarchy configuration file, and statistic reports. A link connects an output buffer of a source node with an input buffer of a destination node. These buffers are created automatically for each link.

- `Source` (required). Source node connected to the link. The value for this variable should be the name of a node defined using a section of type `Network.<network>.Node.<node>`, and only the string specified in `<node>` should be used.
- `Dest` (required). Destination node for the link.
- `Type` (optional). This variable defines the link direction. Possible values are `Unidirectional` (default) and `Bidirectional`. A bidirectional link is equivalent to two unidirectional links in opposite directions, with the corresponding additional input and output buffers created for each link.
- `Bandwidth` (optional). Bandwidth of the link in bytes per cycle. If not present, the value is taken from variable `DefaultBandwidth` in the corresponding network section.
- `VC` (optional). Number of Virtual channels. The default value is 1, meaning that the link is not split into virtual communication paths at all.

The routing table can be manually configured with a section `[Network.<network>.Routes]`. For a single route between two end nodes, every *route step* from source to destination should be identified. In other words, a route step must be given between every node in the network and every end

node that it should be able to reach. Defining each unidirectional route step follows pattern `<node_A>.to.<node_C> = <node_B>[:<VC>]`.

- `node_A`. Source node of a route step. This node can be either an end node or a switch.
- `node_C`. Destination node of a route step. It must be an end node.
- `node_B`. Immediate next node where a packet must be routed when its current location is `node_A` and its final destination is `node_C`. A link must exist connecting `node_A` to `node_B`.
- `VC` (optional). It is the virtual channel's identifier that is being used for the certain route-step. When a link is not split into multiple virtual channels, this field should be omitted. If omitted, for a link that does contain virtual channels a default value of 0 is considered.

7.5 Example of Network Configuration

To illustrate the network configuration file format, an example network is shown in Figure 7.3. This network is composed of 4 end nodes and 3 switches, connected with each other using unidirectional or bidirectional links, as reflected by the single or double-arrow connectors, respectively. The following listing shows the network configuration file associated with this example. In this listing, the routing between nodes is done automatically. Additional examples of network configurations can be found in Section 6.2, also illustrating their integration with the rest of the memory hierarchy.

```
[ Network.mynet ]
DefaultInputBufferSize = 16
DefaultOutputBufferSize = 16
DefaultBandwidth = 1

[ Network.mynet.Node.N1 ]
Type = EndNode

[ Network.mynet.Node.N2 ]
Type = EndNode

[ Network.mynet.Node.N3 ]
Type = EndNode

[ Network.mynet.Node.N4 ]
Type = EndNode

[ Network.mynet.Node.S1 ]
Type = Switch

[ Network.mynet.Node.S2 ]
Type = Switch

[ Network.mynet.Node.S3 ]
Type = Switch

[ Network.mynet.Link.N1-S1 ]
Type = Bidirectional
Source = N1
Dest = S1

[ Network.mynet.Link.N2-S2 ]
Type = Bidirectional
Source = N2
Dest = S2

[ Network.mynet.Link.S3-S1 ]
Type = Bidirectional
Source = S3
Dest = S1

[ Network.mynet.Link.S3-S2 ]
Type = Bidirectional
Source = S3
Dest = S2

[ Network.mynet.Link.S3-N3 ]
Type = Unidirectional
Source = S3
Dest = N3

[ Network.mynet.Link.N4-S3 ]
Type = Unidirectional
Source = N4
Dest = S3
```

7.6 Example of Manual Routing

In this section, an example of a network configuration file with automatic and manual routing is presented. Figure 7.4(a) shows an indirect network with six nodes, where each of them is connected to a separate switch, and switches are connected to each other forming a 2×3 mesh topology. The following listing shows the configuration file associated with this network.

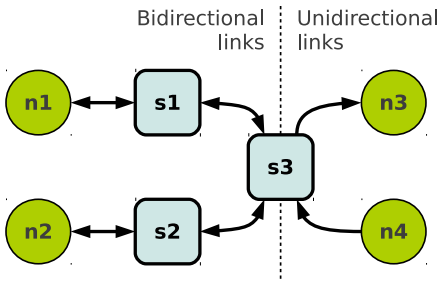
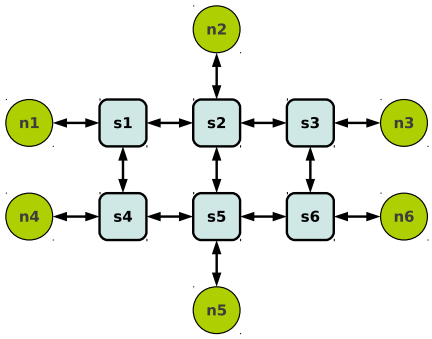


Figure 7.3: Example of a network with 4 end nodes, 3 switches, 4 bidirectional links, and 2 unidirectional links.

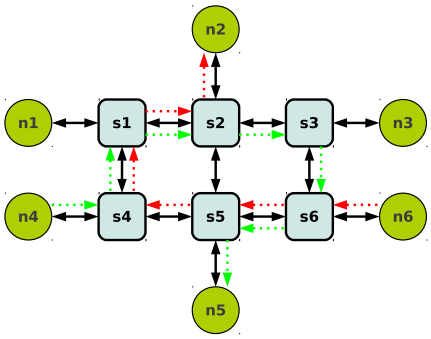
If a routing section is not present in the configuration file, the simulator automatically calculates the shortest paths between each pair of end nodes. Automatic routes are safe when the topology does not include connectivity cycles. However, the presence of link cycles could cause deadlocks if packets follow those routes. The mesh is an example of such topologies, where the user needs to enter a custom routing table to safely route packets without deadlocks.

```
[ Network.mynet ]
DefaultInputBufferSize = 4
DefaultOutputBufferSize = 4
DefaultBandwidth = 1

[ Network.mynet.Node.N1 ]
Type = EndNode

[ Network.mynet.Node.N2 ]
Type = EndNode

[ Network.mynet.Node.N3 ]
Type = EndNode

[ Network.mynet.Node.N4 ]
Type = EndNode

[ Network.mynet.Node.N5 ]
Type = EndNode

[ Network.mynet.Node.N6 ]
Type = EndNode

[ Network.mynet.Node.S1 ]
Type = Switch

[ Network.mynet.Node.S2 ]
Type = Switch

[ Network.mynet.Node.S3 ]
Type = Switch

[ Network.mynet.Node.S4 ]
Type = Switch

[ Network.mynet.Node.S5 ]
Type = Switch

[ Network.mynet.Node.S6 ]
Type = Switch

[ Network.mynet.Link.N1-S1 ]
Type = Bidirectional
Source = N1
Dest = S1

[ Network.mynet.Link.N2-S2 ]
Type = Bidirectional
Source = N2
Dest = S2

[ Network.mynet.Link.N3-S3 ]
Type = Bidirectional
Source = N3
Dest = S3

[ Network.mynet.Link.N4-S4 ]
Type = Bidirectional
Source = N4
Dest = S4

[ Network.mynet.Link.N5-S5 ]
Type = Bidirectional
Source = N5
Dest = S5

[ Network.mynet.Link.N6-S6 ]
Type = Bidirectional
Source = N6
Dest = S6

[ Network.mynet.Link.S1-S2 ]
Type = Bidirectional
Source = S1
Dest = S2

[ Network.mynet.Link.S1-S4 ]
Type = Bidirectional
Source = S1
Dest = S4

[ Network.mynet.Link.S2-S3 ]
Type = Bidirectional
Source = S2
Dest = S3

[ Network.mynet.Link.S2-S5 ]
Type = Bidirectional
Source = S2
Dest = S5

[ Network.mynet.Link.S3-S6 ]
Type = Bidirectional
Source = S3
Dest = S6

[ Network.mynet.Link.S4-S5 ]
Type = Bidirectional
Source = S4
Dest = S5

[ Network.mynet.Link.S5-S6 ]
Type = Bidirectional
Source = S5
Dest = S6
```

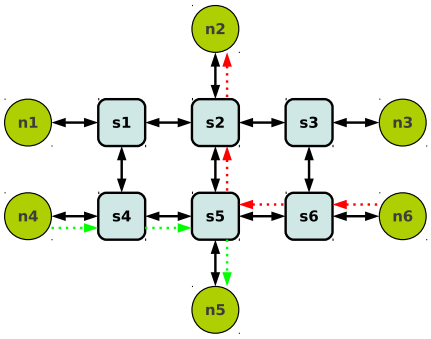
As an initial example illustrating Multi2Sim's configuration potential for routing tables, Figure 7.4(b) shows two routes used for packets going from `n4` to `n5` (green) and from `n6` to `n2` (red). The following listing shows the additional code that should be added to the network configuration file to support these routes.



a) Indirect network with 6 end nodes and a 2x3 Mesh of Switches.



b) Example of two routes showing capability of the manual Routing.



c) Example of two routes in the network with manual X-Y routing Configuration.

Figure 7.4: Example of a network with 6 end nodes, a 2×3 mesh of switches, all bidirectional links.

```
[Network.mynet.Routes]
N4.to.N5 = S4
S4.to.N5 = S1
S1.to.N5 = S2
S2.to.N5 = S3
S3.to.N5 = S6
S6.to.N5 = S5
N6.to.N2 = S6
S6.to.N2 = S5
S5.to.N2 = S4
S4.to.N2 = S1
S1.to.N2 = S2
```

As a realistic example, X-Y routing is a popular routing scheme used in mesh networks, using a unique shortest path between each pair of end nodes in a deadlock-free manner. Figure 7.4(c) presents the routes for the same two pairs of nodes using X-Y routing. Table 7.1 completes all possible routes for the 2×3 mesh, and the following listing shows the additional code required in the network configuration file.

```
[Network.mynet.Routes]
N1.to.N2 = S1
S1.to.N2 = S2

N1.to.N3 = S1
S1.to.N3 = S2
S2.to.N3 = S3

N1.to.N4 = S1
S1.to.N4 = S4

N1.to.N5 = S1
S1.to.N5 = S2
S2.to.N5 = S5

N1.to.N6 = S1
S1.to.N6 = S2
S2.to.N6 = S3
S3.to.N6 = S6

N2.to.N1 = S2
S2.to.N1 = S1

N2.to.N3 = S2

N2.to.N4 = S2
S2.to.N4 = S1

N2.to.N5 = S2

N2.to.N6 = S2

N3.to.N1 = S3
S3.to.N1 = S2

N3.to.N2 = S3
S3.to.N2 = S2

N3.to.N4 = S3
S3.to.N4 = S2
S2.to.N4 = S1

N3.to.N5 = S3
S3.to.N5 = S2

N3.to.N6 = S3

N4.to.N1 = S4
S4.to.N1 = S1

N4.to.N2 = S4
S4.to.N2 = S5
S5.to.N2 = S2

N4.to.N3 = S4
S4.to.N3 = S5
S5.to.N3 = S6
S6.to.N3 = S3

N4.to.N5 = S4
S4.to.N5 = S5

N4.to.N6 = S4
S4.to.N6 = S5
S5.to.N6 = S6

N5.to.N1 = S5
S5.to.N1 = S4

N5.to.N2 = S5

N5.to.N3 = S5

N5.to.N4 = S5
S5.to.N4 = S4

N5.to.N6 = S5

N6.to.N1 = S6
S6.to.N1 = S5

N6.to.N2 = S6
S6.to.N2 = S5

N6.to.N3 = S6

N6.to.N4 = S6
S6.to.N4 = S5

N6.to.N5 = S6
S6.to.N5 = S5
```

Table 7.1: Manual X-Y Routing for every pair of end nodes in the network

Source	Switches				Dest	Source	Switches				Dest
N1	S1	S2			N2	N4	S4	S1			N1
	S1	S2	S3		N3		S4	S5	S2		N2
	S1	S4			N4		S4	S5	S6	S3	N3
	S1	S2	S5		N5		S4	S5			N5
	S1	S2	S3	S6	N6		S4	S5	S6		N6
N2	S2	S1			N1	N5	S5	S4	S1		N1
	S2	S3			N3		S5	S2			N2
	S2	S1	S4		N4		S5	S6	S3		N3
	S2	S5			N5		S5	S4			N4
	S2	S3	S6		N6		S5	S6			N6
N3	S3	S2	S1		N1	N6	S6	S5	S4	S1	N1
	S3	S2			N2		S6	S5	S2		N2
	S3	S2	S1	S4	N4		S6	S3			N3
	S3	S2	S5		N5		S6	S5	S4		N4
	S3	S6			N6		S6	S5			N5

7.7 Example Using Virtual Channels

Figure 7.5 shows an indirect network with four nodes, where each of them is connected to a corresponding switch (i.e. end node `n0` to switch `s0`) with a physical link. Switches are connected to each other with unidirectional links, forming a 4-node ring topology. Two routes are shown in this example: One from end node `n0` to end node `n3`, and another from end node `n2` to end node `n1`. Packets from node `n0` to node `n3` should go through switches `s0`, `s1`, `s2`, and `s3` and packets from node `n2` to node `n1` should go through switches `s2`, `s3`, `s0`, and `s1`, respectively. There is a cycle between switches in this network because routes in the network go through the same switches while two route-steps in these two routes use same physical links and buffers.

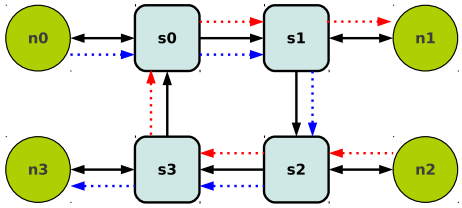


Figure 7.5: Example of a network with 4 end nodes and four switches. Switches are connected forming a ring topology.

A deadlock scenario is presented for this network in Figure 7.6. As shown in the figure, end node `n0` is sending its packets (marked with blue rhombuses) to `n3` and end node `n2` is sending its packets (marked with red squares) to `n1`. Blue and red packets share two physical links through their path, i.e. physical link between `s2` and `s3` and link between `s0` and `s1`. The deadlock occurs in this scenario because blue packets cannot advance due to the input buffer in switch `s3` being occupied with red packets; and simultaneously, red packets cannot advance due to the input buffer in switch `s1` being fully occupied with blue packets.

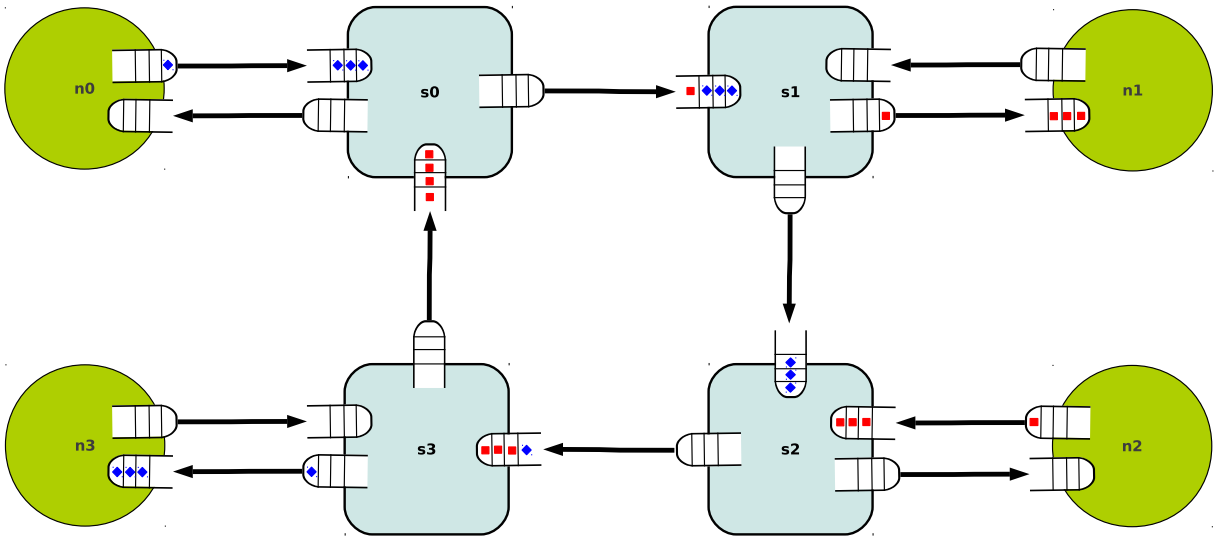


Figure 7.6: Example of a deadlock potential in the network.

To solve the deadlock in this example, virtual channels can be used on either of two links, shared between these routes. Two virtual channels are introduced on top of the physical link between `s2` and `s3`, and each of them assigned to one route. This way blue packets from `n0` to `n3` advance through

one of the virtual channels and red packets from `n2` to `n1` advance through the other. Figure 7.7 illustrates the resolution of the deadlock by means of the additional virtual channel.

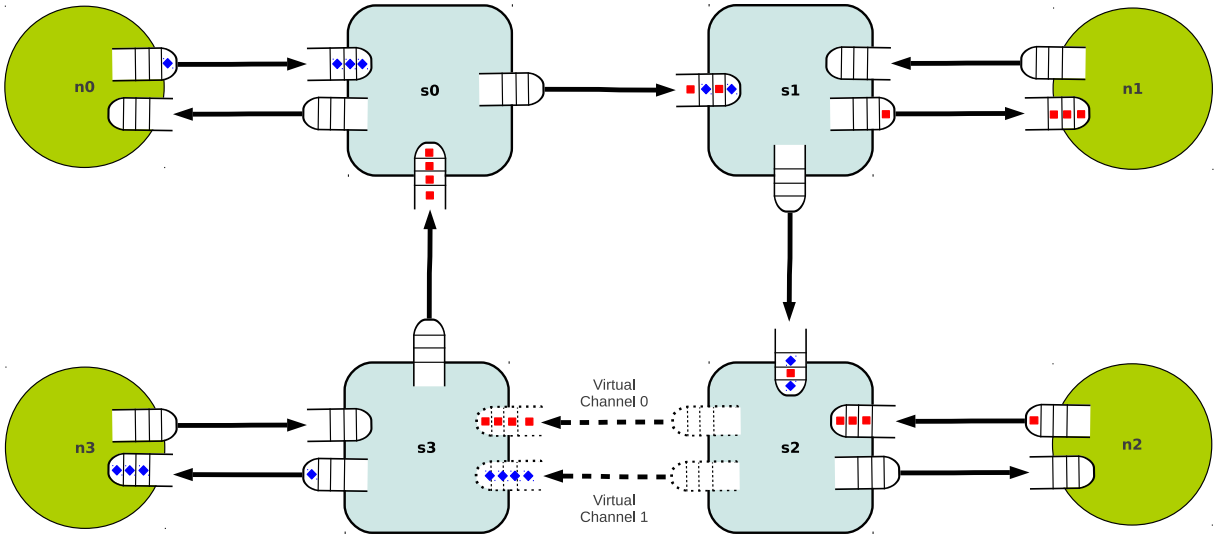


Figure 7.7: Resolution of the deadlock condition.

The following listing shows the configuration file associated with this network. For the link between switches `s2` and `s3`, variable `VC` in section `[Network.mynet.Link.s2-s3]` is used to define the number of virtual channels on top of the physical link. The virtual channels are then used in the routes defined in section `[Network.mynet.Routes]` by using suffices `:0` and `:1` in the corresponding route-steps.

```
[ Network.mynet ]
DefaultInputBufferSize = 4
DefaultOutputBufferSize = 4
DefaultBandwidth = 1

[ Network.mynet.Node.n0 ]
Type = EndNode

[ Network.mynet.Node.n1 ]
Type = EndNode

[ Network.mynet.Node.n2 ]
Type = EndNode

[ Network.mynet.Node.n3 ]
Type = EndNode

[ Network.mynet.Node.s0 ]
Type = Switch

[ Network.mynet.Node.s1 ]
Type = Switch

[ Network.mynet.Node.s2 ]
Type = Switch

[ Network.mynet.Node.s3 ]
Type = Switch
```

```
[ Network.mynet.Link.n0-s0 ]
Type = Bidirectional
Source = n0
Dest = s0

[ Network.mynet.Link.n1-s1 ]
Type = Bidirectional
Source = n1
Dest = s1

[ Network.mynet.Link.n2-s2 ]
Type = Bidirectional
Source = n2
Dest = s2

[ Network.mynet.Link.n3-s3 ]
Type = Bidirectional
Source = n3
Dest = s3

[ Network.mynet.Link.s0-s1 ]
Type = Unidirectional
Source = s0
Dest = s1

[ Network.mynet.Link.s1-s2 ]
Type = Unidirectional
Source = s1
Dest = s2
```

```
[ Network.mynet.Link.s2-s3 ]
Type = Unidirectional
Source = s2
Dest = s3
VC = 2

[ Network.mynet.Link.s3-s0 ]
Type = Unidirectional
Source = s3
Dest = s0

[ Network.mynet.Routes ]
n2.to.n1 = s2
s2.to.n1 = s3:0
s3.to.n1 = s0
s0.to.n1 = s1

n0.to.n3 = s0
s0.to.n3 = s1
s1.to.n3 = s2
s2.to.n3 = s3:1
```

7.8 Statistics Report

A detailed report of the interconnection network simulation is dumped in the file specified by the `--report-net <file>` option. The statistic report follows the INI file format, and includes one separate section `[Network.<name>]` for every network created in the system. The following variables are used:

- `Transfers`. Total number of packets received by all nodes in the network.
- `AverageMessageSize`. The average message (packet) size of total packets transferred in the network.
- `AverageLatency`. The average latency of packets that are transferred throughout the network in cycles.

The set of statistics related to each link in the network is presented in sections following the pattern `[Network.<network>.Link.link_<source>.out_buf_<bufferid> >_<dest>.in_buf_<bufferid>]`, where `bufferid` fields represent integer buffer identifiers created automatically for the link. A bidirectional link shows two sections in the statistic report, each corresponding to one of the equivalent unidirectional links.

- `Config.Bandwidth`. Link bandwidth, as specified in the network configuration file.
- `TransferredMessages`. Number of transferred packets through the specific link in number of packets.
- `TransferredBytes`. Amount of data transferred through this link in bytes.
- `BusyCycles`. Number of cycles where the link was busy transferring data.
- `BytesPerCycle`. Number of bytes transfered per cycle, considering the entire simulation time.
- `Utilization`. Link utilization, calculated as the ratio of `BytesPerCycle` over the link bandwidth.

For each node, section `[Network.<network>.Node.<node>]` provides the following statistics:

- `Config.InputBufferSize`. Input buffer size, as specified in the network configuration file.
- `Config.OutputBufferSize`. Output buffer size, as specified in the network configuration file.
- `SentMessages`. Number of packets sent by the node.
- `SentBytes`. Amount of data sent by node in bytes.
- `SendRate`. Bytes per cycle sent by the node.
- `ReceivedMessages`. Number of packets received from the network.
- `ReceivedBytes`. The amount of data received by the node in bytes.
- `ReceiveRate`. Bytes per cycle received by the node.
- `In/out_buf_<number>.MessageOccupancy`. Average occupancy of input and output buffers in number of packets per cycle. A separate variable is reported for each buffer created for the node.
- `In/out_buf_<number>.ByteOccupancy`. Average occupancy of input and output buffers in number of bytes per cycle.
- `In/out_buf_<number>.Utilization`. Average buffer occupancy in bytes as a fraction of the maximum buffer capacity.

7.9 Stand-Alone Network Simulation

Multi2Sim provides a tool to stress an interconnect given in the network configuration file using synthetic traffic patterns. This tool is referred to as the *stand-alone network simulator*, and is integrated within the rest of the simulator functionality. It can be invoked using command-line option `--net-sim <network>`, where `network` is the name of a network defined in the network configuration file passed with option `--net-config <file>`. The following additional command-line options are available for stand-alone network simulation:

- `--net-injection-rate <rate>` (Default = 0.01). Packet injection rate for every end node in the network. Each end node injects packets in the network using random delays with exponential distribution, where *lambda* = `rate`. The destination end node of the packet is also chosen randomly among all reachable destination end nodes.
- `--net-max-cycle <cycle>` (Default = 1M). Number of simulation cycles.
- `--net-msg-size <size>` (Default = 1 byte). Packet size in bytes. An entire packet should fit in the smallest buffer created in the network. The transfer latency of a packet will depend on the bandwidth of the links it has to traverse, as well as the contention it experiences in every intermediate node.

Assuming that the example configuration file shown in Section 7.5 is stored in file `net-config`, the following command line can be used to stress network `mynet` during 1M cycles, having each node inject one packet to the network every 10 cycles on average:

```
m2s --net-config net-config --net-sim mynet --net-max-cycles 1000000 \
       --report-net report-net --net-injection-rate 0.1
```

Chapter 8

M2S-Visual: The Multi2Sim Visualization Tool

8.1 Introduction

M2S-Visual is a visualization tool, integrated in the Multi2Sim simulation framework, that provides visual representations for analysis of architectural simulations. The primary function of M2S-Visual is to observe the state of the CPU and GPU pipeline and the state of the memory, providing features such as simulation pausing, stepping through cycles, and viewing properties of in-flight instructions and memory accesses.

The state of CPU and GPU software entities (contexts, work-groups, wavefronts, and work-items) and memory entities (accesses, sharers, owners) are represented, along with the state of CPU and GPU hardware resources(cores, compute units) and memory hierarchy(L1 cache, L2 cache and the main memory). The main window shows the overview of the CPU, GPU and memory. Secondary windows can be opened, representing time or block diagram that the user can navigate through.

8.1.1 Compilation of M2S-Visual

M2S-Visual is integrated in the same executable file as the rest of Multi2Sim's simulation features. The tool requires the GTK 3.0 development packages to be installed in your system for correct compilation. If this library is missing, Multi2Sim will still compile successfully, but without support for the visualization features. During the compilation of Multi2Sim, the `configure` script will detect the presence of this package, and output a warning message in case it is not found.

8.1.2 Running M2S-Visual

M2S-Visual acts as an off-line analysis tool. In a first execution of an architectural simulator, Multi2Sim generates a trace file containing a detailed report of all actions occurring in the modeled hardware. The generation of the trace file is configured with command-line option `--trace <file>`, where `<file>` is a plain-text file compressed with the *gzip* format. The name of the file should have the `.gz` extension (e.g., `output-trace.gz`).

When launching a detailed simulation with activated traces, one needs to be careful with the computational weight of simulated programs. The simulator dumps several lines of text in every execution cycle, and even if the output format is compressed, its size can reach gigabytes of information very quickly.

An example is given next on how to run the `test-threads` application[1] using a CPU configuration with 4 cores. This mini-benchmark is an x86 program that takes a value n as an argument, and spawns $n-1$ child threads. Each thread, including the parent, dumps its identifier, and exits. The first step is creating a CPU configuration file (`x86-config`) specifying the number of CPU cores:

```
[ General ]
Cores = 4
```

The CPU detailed simulation is launched with the following command:

```
m2s --x86-sim detailed --x86-config x86-config --trace my-trace.gz test-threads.i386 4
```

The following code is an excerpt of file `my-trace.gz`, generated during the simulation (command `gunzip my-trace.gz` can be used to uncompress the trace):

```
...
c clk=144
x86.inst id=16 core=0 stg="wb"
x86.inst id=18 core=0 stg="wb"
x86.inst id=19 core=0 stg="i"
x86.inst id=21 core=0 stg="i"
mem.access name="A-3" state="cpu-l2:find_and_lock_action"
mem.access name="A-3" state="cpu-l2:find_and_lock_finish"
mem.access name="A-3" state="cpu-l2:read_request_action"
mem.access name="A-5" state="cpu-l2:read_request_receive"
mem.access name="A-3" state="cpu-l2:read_request_updown"
mem.access name="A-5" state="cpu-l2:find_and_lock"
mem.new_access_block cache="cpu-l1-0" access="A-1" set=4 way=1
...
```

The trace file is formed of a set of header specifying the configuration of the processor model for this specific simulation. For each simulation cycle, a set of lines represent all actions occurring in processor pipelines, GPU compute units, or memory hierarchy components. The specific format of the simulation trace out of the scope of this guide, but is documented instead in code comments on the M2S-Visual source code.

Once the trace has been generated, the next step is launching M2S-Visual consuming it (using the compressed version, as generated by the previous `m2s` execution). To launch M2S-Visual, option `--visual <file>` is used, where `<file>` is the compressed trace file.

```
m2s --visual my-trace.gz
```

Before the main window shows up, the trace is uncompressed, and simulation checkpoints are created. The goal of the checkpoints is to allow for fast navigation through cycles. For example, if the user wants to navigate from cycle 1040 back to cycle 1039, a checkpoint-less implementation would need to reset the visualized machine state, and process the trace lines for all 1039 again. On the contrary, assuming a checkpoint frequency of 500 simulation cycles, M2S-Visual loads the visualized machine state at cycle 1000, and then processes only those trace lines for the following 39 cycles. In simulations with millions of cycles, this feature is indispensable.

As an additional example, the following two lines of code show the code for the execution and visualization of a GPU program, using the OpenCL matrix multiplication implementation available in the AMD SDK benchmark suite (Section *Benchmarks* on the website):

[1] This program is available on Multi2Sim's website, under Section *Benchmarks*, as part of the *Mini-Benchmarks* suite

```
$ m2s --evg-sim detailed --trace my-trace.gz MatrixMultiplication \
    --load MatrixMultiplication_Kernels.bin -x 32 -y 32 -z 32 -q

$ m2s --visual my-trace.gz
```

8.2 Main Window

The main window of M2S-Visual has four components: a cycle bar, a CPU panel, a GPU panel and a memory system panel. The availability of CPU and GPU panel depends on the type of detailed simulation was run. The CPU panel activates for detailed CPU simulations, while the GPU panel becomes available for GPU detailed simulations. The memory panel is shown in either case.

8.2.1 The Cycle Bar

The cycle navigation bar features both a scrollable bar and navigation buttons to step through simulation cycles at desired increments (1, 10, or 100) or to jump to a specific cycle number (Figure 8.1. This cycle bar synchronizes the main window with all secondary windows and diagrams opened for this simulation, as described below.

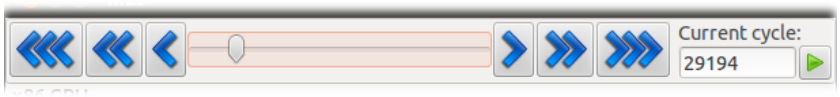


Figure 8.1: Cycle navigation bar.

8.2.2 Panels

The outlook of the rest of the panels depends on the simulation settings. For example, the number of elements in the CPU panel depends on the number of cores set up for the simulation. They show a summary of the current state of the CPU cores, GPU compute units, and memory system at the cycle selected with the cycle navigation bar.

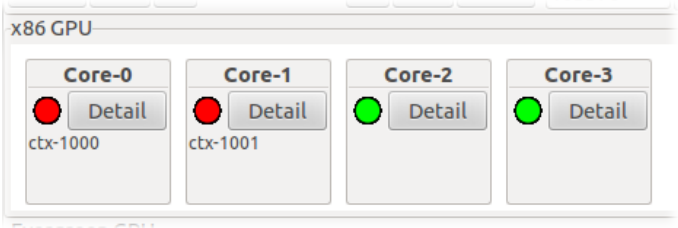


Figure 8.2: Panel representing a multi-core x86 CPU, as part of M2S-Visual's main window.

In Figure 8.2, an example of the CPU panel state is shown for a 4-core processor model. Each CPU core is represented with a gray board, and contains a *busy* indicator (red or green light), the list of contexts running on the CPU core and a *Detail* button. Clicking on a context label opens a pop-up window showing detailed information about the context, while the *Detail* button opens a time diagram for the core.

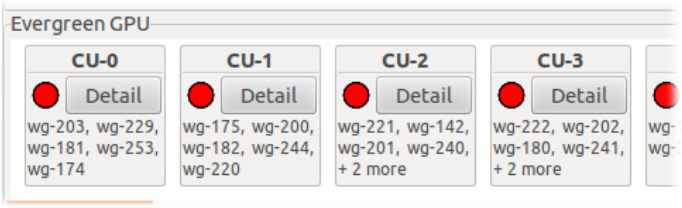


Figure 8.3: Panel representing an Evergreen GPU, as part of M2S-Visual's main window.

Figure 8.3 shows an excerpt of the GPU panel state, where each gray board represents a compute unit. Each GPU compute unit contains a *busy* indicator, a list of work-groups running on it, and a *Detail* button. Clicking on a work-group label opens an information pop-up window for the work-group, while the *Detail* button provides a time diagram for the compute unit.

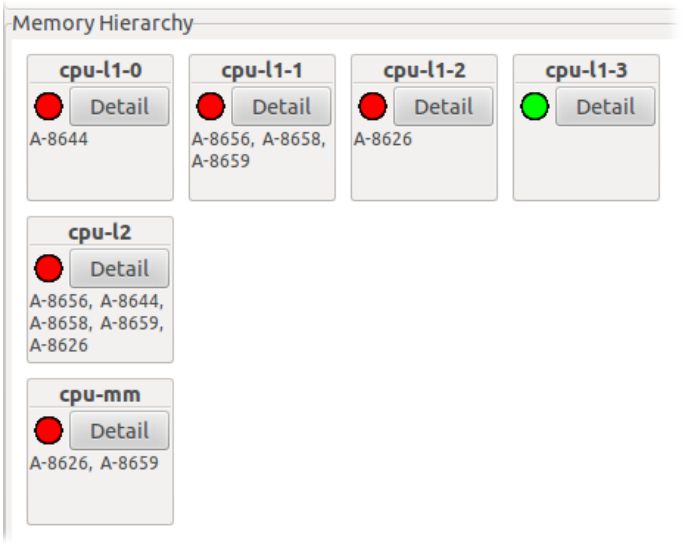


Figure 8.4: Panel representing the memory hierarchy. This panel is part of M2S-Visual's main window.

Finally, Figure 8.4 shows an example of the state of a memory hierarchy for a 4-core processor, using private L1 caches, and a shared L2 cache. Each gray board represents a cache or a main memory module (directory). A *busy* indicator shows whether the module is currently serving any access. The list of accesses currently being served is shown underneath. The label representing an access can be clicked on to observe the access properties. A *Detail* button shows the detailed state of the module.

8.3 The x86 CPU Visualization

For each CPU core, a time diagram can be generated and navigated through by pressing the *Detail* button on the CPU panel. Multiple time diagrams can be opened at the same time for different CPU cores, and all of them are synchronized when the cycle navigation controls on the main window are updated.

As shown in Figure 8.5, a CPU time diagram contains columns. The left column is the x86

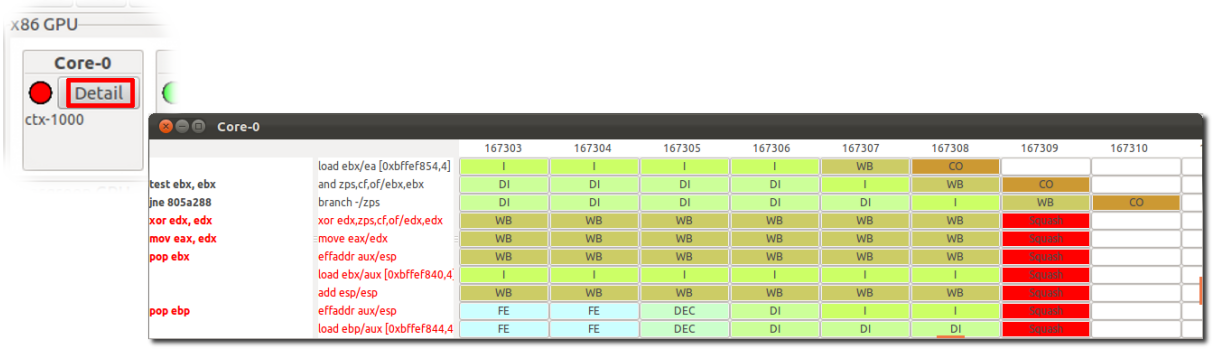


Figure 8.5: Time diagram for the x86 CPU pipeline. The time diagram is opened by clicking on the *Detail* button on a specific x86 core board.

instructions in flight. The middle column contains the micro-code generated for x86 instructions. The micro-code is generated following the rules in Section 2.7. Though the micro-instructions are effectively generated in the *decode* stage on a real machine, the time diagram represents them starting from the *fetch* stage, as soon as the cache block containing the associated x86 instruction is fetched from the instruction cache. The right column contains a table, representing the pipeline stage where a micro-instruction is located in each cycle. Speculative x86 instructions and micro-code in the two left-most columns are colored red. These instructions are squashed at the time a mispeculated branch reaches the *writeback* or *commit* stage, depending on the configuration for branch resolutions.

8.4 The Evergreen GPU Visualization

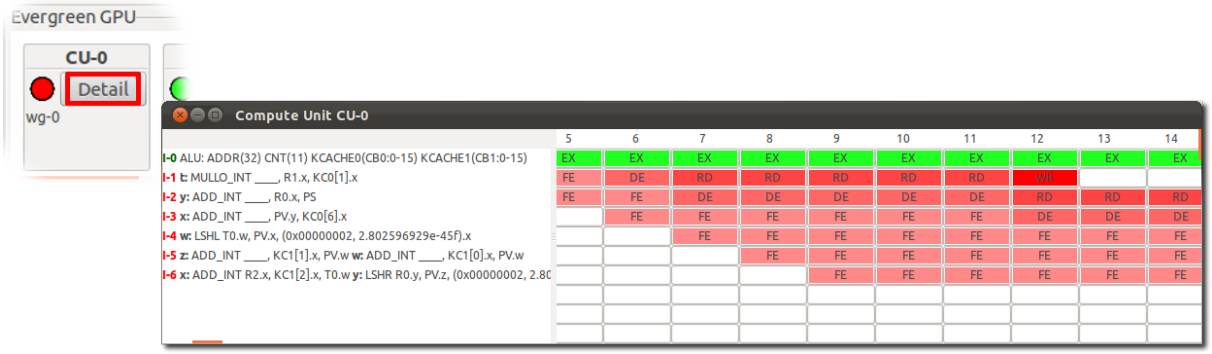


Figure 8.6: Time diagram for the Evergreen GPU pipeline, opened by clicking on the *Detail* button on a specific Evergreen compute unit board.

For each GPU compute unit, a time diagram can be generated and navigated through by pressing the *Detail* button on the main window, as shown in Figure 8.6. Multiple time diagrams can be opened at the same time for different compute units, and all of them will be synchronized when the cycle navigation controls on the main window are updated.

The left column shows the assembly code of Evergreen instructions running on the compute unit, in the same order that they were fetched. The columns in the right table represent cycles,

while the rows correspond to instructions in flight. The contents of each table cell represents the pipeline stage where instructions are located. Instructions are colored green, red, or blue, depending on whether they belong to control-flow (CF), arithmetic-logic (ALU), or texture (TEX) clauses, respectively (see Section 4.2.1 for more information on the Evergreen ISA). Darker color tonalities are used for later stages in the pipelines.

8.5 Memory Hierarchy Visualization

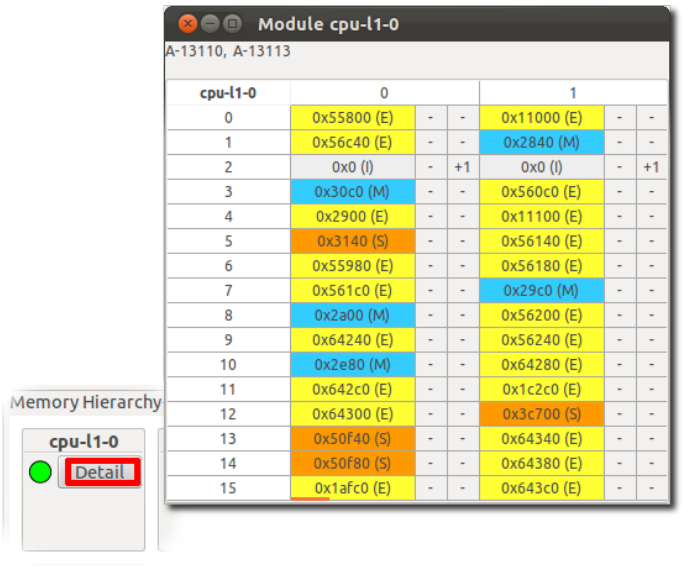


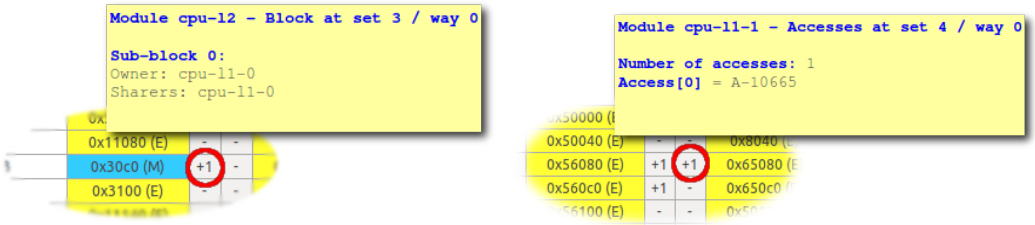
Figure 8.7: Visual representation of a cache.

The simulated memory hierarchy panel is structured in levels of caches or memory modules, where a higher module is a module closer to the processor. Figure 8.4 shows an example for a memory hierarchy representation composed of four L1 caches, one shared L2 cache, and one shared main memory. Pressing the *Detail* button opens the visual representation of a cache.

Figure 8.7 shows an example of a 2-way set-associative L1 cache with 16 sets. The top panel contains in-flight accesses in the module, labeled A-x, where x is an access identifier assigned in order of creation. Each cell is a cache block, containing a tag and a state. The state can be one of the five states in the MOESI cache coherence protocol, and it determines the color of the cell. The two smaller cells on the right of the tag represent the number of sharers in the upper-level cache for this block, and the number of in-flight accesses for the block, respectively.

For highest-level modules, the number of sharers of a block is always 0. For lowest-level modules (i.e., main memory modules), the table represents only the directory, organized as a cache structure, no different from upper-level cache structures. Notice that the size of the directory in a memory module determines the maximum number of in-flight blocks in the rest of the memory hierarchy.

Figure 8.8(a) shows the sharers for L2 block with tag `0x30c0`, obtained by clicking on the left "+1" label. The pop-up window shows that the upper-level cache `cpu-l1-0` is the only sharer of the block. The owner is set also to `cpu-l1-0`, meaning that this cache has an exclusive copy of it that can be used for write access. Figure 8.8(b) shows the set of in-flight accesses for the block, botained by clicking on the right "+1" label. In the example, only access `A-10665` is in-flight for this block. The access also appears in the top panel of in-flight accesses.



a) Sharers and owner of the block in the higher-level cache.

b) In-flight accesses on the block.

Figure 8.8: Pop-up windows showing sharers of a block in the higher cache level, and in-flight accesses. Windows show up after clicking on the labels in the columns adjacent to each block's tag and state.

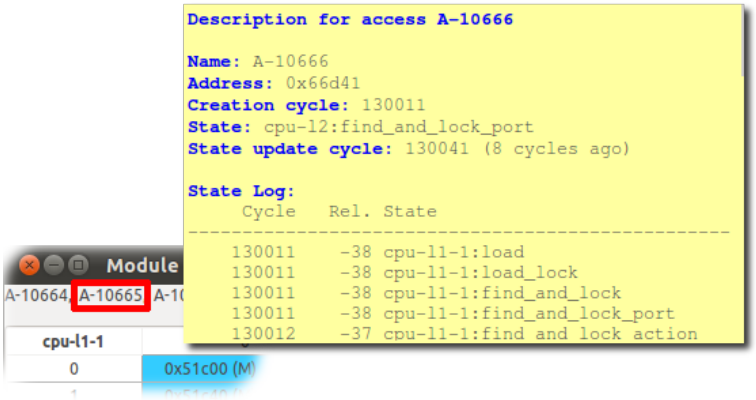


Figure 8.9: Pop-up window showing the properties and log of an in-flight memory access. The window shows up after clicking on the access label located in the access panel.

Detailed information of an access can be obtained by clicking on its corresponding label on the top panel of a memory module, as shown in Figure 8.9. The history of the access is completely shown in the *State log* field, including the absolute cycle number for each access event occurrence, cycle number relative to the current cycle, as selected in the main cycle bar, and current event transition for the access.

Chapter 9

M2S-Cluster: Launching Massive Simulations

9.1 Introduction

M2S-Cluster is a system to launch simulations automatically using a set of benchmarks on top of Multi2Sim. The tool works on an infrastructure composed of a client Linux-based machine and a server formed of several compute nodes, using the *condor* framework [13] as a task scheduling mechanism. M2S-Cluster simplifies the routine task of launching sets of simulations with a single command-line tool that communicates the client and server.

9.2 Requirements

Figure 9.1 shows a block diagram representing a system with the ability to run M2S-Cluster. The system has the following configuration and hardware/software requirements:

- Client machine used to manage simulation executions. This is the Linux-based user's local working machine. Package *subversion* needs to be installed for availability of command-line tool `svn`. The name of the client machine is referred to as `CLIENT` hereafter.
- Server machine (cluster of machines) composed of a front-end and several back-ends, all of

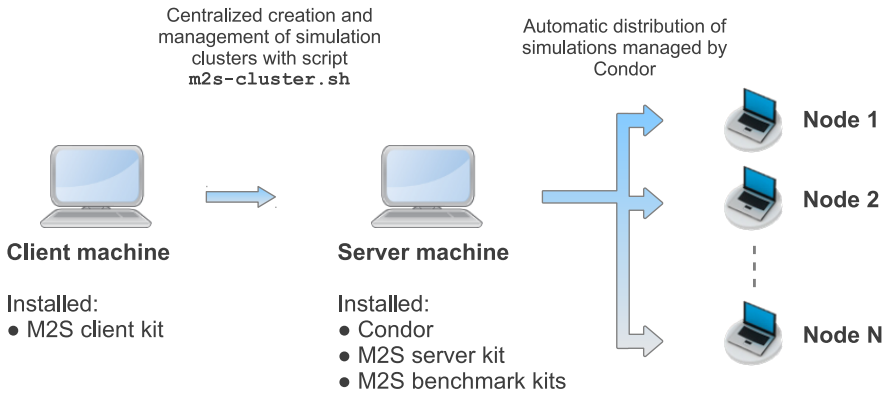


Figure 9.1: Representation of a system running M2S-Cluster.

them sharing the same file system through, for example, NFS (network file system). The server front-end needs an installation of the GNU Autotools (commands `aclocal`, `autoconf`, and `automake`), the *subversion* tool (command `svn`), and the distributed task scheduler *condor* (commands `condor_submit`, `condor_q`, `condor_status`). The server machine acting as a front-end for all server computing nodes is referred to as `SERVER` below.

- The user login name must be the same in `CLIENT` and `SERVER`. Also, the user in `CLIENT` needs to have a key-less access to `SERVER` through an `ssh` connection, based on a private/public key configuration. The following sequence of commands can be used to set up key-less login through `ssh`:

```
user@CLIENT:~$ ssh-keygen -t rsa
user@CLIENT:~$ scp .ssh/id_rsa.pub SERVER:
user@CLIENT:~$ ssh SERVER

user@SERVER:~$ mkdir -p .ssh
user@SERVER:~$ cat id_rsa.pub >> .ssh/authorized_keys
user@SERVER:~$ rm id_rsa.pub
user@SERVER:~$ exit
```

  After running this code, the user should make sure that command `ssh SERVER`, run at `CLIENT`, connects to the server machine without prompting for a password.

9.3 Installation

M2S-Cluster is composed of three different types of packages, called the *client kit*, the *server kit*, and the *benchmark kits*. The client kit is installed in `CLIENT`, while the server and benchmark kits are installed in `SERVER`. Installations only involve downloading SVN repositories on the home folders, without the need for root access privilege in the client or server machines.

9.3.1 Installation Steps

Follow the indicated steps to install each component of M2S-Cluster:

- On the client machine, type the following command on the home folder to install the client kit:

```
user@CLIENT:~$ svn co http://multi2sim.org/svn/m2s-client-kit
```

- On the server machine, type the following command to install the server kit:

```
user@SERVER:~$ svn co http://multi2sim.org/svn/m2s-server-kit
```

- Finally, the benchmark kits need to be installed also on `SERVER`, using one separate command per benchmark kit that will be used for simulations. The complete list of available benchmark kits is listed on the Multi2Sim website. For example, the AMDAPP 2.5 benchmark suite for Evergreen can be installed with the following command:

```
user@SERVER:~$ svn co http://multi2sim.org/svn/m2s-bench-amdapp-2.5-evg
```

  Only those benchmarks provided for free by their developers are publicly available on the Multi2Sim SVN server. Commercial benchmarks, such as SPEC2006, only include statically pre-compiled executables, omitting data files. Table 9.1 lists all benchmark kits publicly available on M2S-Cluster.

`amdapp-2.5-evg`	AMD OpenCL benchmarks for Evergreen (v. 2.5)
`amdapp-2.5-si`	AMD OpenCL benchmarks for Southern Islands (v. 2.5)
`amdapp-2.7-evg`	AMD OpenCL benchmarks for Evergreen (v. 2.7)
`minibench`	Small benchmarks to test basic Multi2Sim functions
`parsec-2.1`	PARSEC parallel benchmarks based on `pthread` and `OpenMP` (v. 2.1)
`rodinia`	OpenCL Rodinia benchmark suite
`spec2006`	SPEC 2006 benchmarks (restricted)
`splash2`	SPLASH-2 parallel benchmarks based on `pthread`
`x86-sse`	Regression tests for x86 SSE instructions

Table 9.1: List of benchmark suites available on M2S-Cluster.

9.3.2 Keeping Repositories up to Date

The M2S-Cluster project is updated progressively by several developers, who might require all new changes to be applied to all M2S-Cluster repositories simultaneously for compatibility. To guarantee correct behavior, scripts launching new simulations on servers first check that all repositories are up to date by retrieving the latest version information from the SVN server. If any repository is out of date in the home folder of either `CLIENT` or `SERVER`, a warning message will notify this fact. To update an already downloaded repository, command `svn update` should be run on all repositories, including client, server, and benchmark kits.

```
user@CLIENT:~/m2s-client-kit$ svn update
user@SERVER:~/m2s-server-kit$ svn update
user@SERVER:~/m2s-bench-amdapp-2.5-evg$ svn update
[...]
```

9.4 The Client Tool `m2s-cluster.sh`

M2S-Cluster is managed in a centralized manner by means of shell script `CLIENT:~/m2s-client-kit/bin/m2s-cluster.sh`, which is part of M2S-Cluster's client kit. If run without arguments, the tool shows a help message listing all possible command-line options, for quick reference. The first expected argument is a command, followed by a variable sequence of arguments dependent on the command itself. The following sections define the behavior of `m2s-cluster.sh` for each possible command.

9.4.1 Command `create`

Syntax:

```
m2s-cluster.sh create <cluster>
```

Create a new cluster of simulations (also called *jobs*). Creating a cluster is the first step to run simulations; then the cluster is populated with jobs, which will be later submitted to the server. The name given in argument `<cluster>` is used in subsequent commands to refer to this cluster. A directory hierarchy is created in the server machine to store data for the cluster execution at `SERVER:~/m2s-server-kit/run/<cluster>/`. The initial state of the cluster is `Created` (see Section 9.4.4 for a list of possible states).

9.4.2 Command `add`

Syntax:

```
m2s-cluster.sh add <cluster> <job_name> [<benchmarks>] [<options>]
```

Add a new job to the cluster, where `<job_name>` is the identifier of the new job. The job name can contain slash (`/`) characters. The name determines the location in the server kit where the job's temporary files are stored, as expressed by path `SERVER:~/m2s-server-kit/run/<cluster>/<job_name>/`.

Argument `<benchmarks>` is the list of workloads to be run as different contexts on top of the same Multi2Sim simulation. Each workload will be passed automatically by M2S-Cluster as a different section `[Context XXX]` of the context configuration file (see Section 1.4.4). Each benchmark is given as a pair `<suite>/<benchmark>` (e.g., `splash2/fft`). Notice that the name given in `<suite>` omits the `m2s-bench-` prefix used in the corresponding benchmark kit. For each benchmark suite used here, the corresponding benchmark kit needs to be installed in the server machine (e.g., `SERVER:~/m2s-bench-splash2`).

The following optional arguments can be given:

- `-p <num_threads>`

  If the first benchmark in the list accepts a variable number of threads, the value given in `<num_threads>` is used. More specifically, this value replaces variable `$NTHREADS` in file `benchmark.ini` for the corresponding workload and benchmark kit (see Section 9.6).

- `--send <file>`

  Send an additional file to be included in the working directory of the simulation execution. This option is useful to send configuration files for Multi2Sim. To send multiple files, use double quotes (e.g., `--send "mem-config x86-config"`).

- `--sim-args <args>`

  Additional arguments for the simulator. This option can be used, for example, to make Multi2Sim consume the configuration files previously sent with option `--send`. Use double quotes if the command sent to Multi2Sim contains any space character (e.g., `--sim-args "--mem-config mem-config"`).

  When adding options that make Multi2Sim generate output reports, the file name of these reports should start with prefix `report-` (e.g., `--sim-args "--x86-report report-pipeline"`). The reason is that a subsequent call to command `m2s-cluster.sh import` will automatically import all files generated with this prefix.

- `--bench-args <args>`

  Additional arguments for the first benchmark in the list. The `Args` variable in section `[Context 0]` of the automatically created context configuration file will be composed of the default arguments for the benchmark followed by the additional arguments specified in this option. Use double quotes when there is more than one argument (e.g., `--bench-args "-x 16 -y 16"`).

- `--data-set <set>`

  For those benchmarks providing several datasets, this argument specifies which one to use. The dataset is `Default` if this option is not specified. All datasets supported for a benchmark are listed as sections of the `benchmark.ini` file in the corresponding benchmark kit (see Section 9.6).

9.4.3 Command `submit`

Syntax:

```
submit <cluster> <server>[:<port>] [<options>]
```

Submit the cluster to the server and start its execution, automatically fetching and building a copy of Multi2Sim from its official SVN repository. Optional argument `<port>` specifies the port for SSH connections (22 is assumed by default). After running this command, a cluster transitions to state `Submitted`. A cluster must be in state `Created`, `Completed`, or `Killed` for it to be (re-)submitted (see Section 9.4.4). The following additional arguments can be used:

- `-r <revision>`

  Multi2Sim SVN revision to use for the cluster execution. If this option is omitted, the latest SVN update will be fetched automatically from the Multi2Sim server.

- `--tag <tag>`

  If this option is specified, the simulator source code is fetched from the `multi2sim/tags/multi2sim-<tag>` directory, containing a stable distribution. If the option is omitted, the code is fetched from the development trunk at `multi2sim/trunk`.

- `--configure-args <args>`

  Arguments to be passed to the `configure` script when building the fetched Multi2Sim source code. Use double quotes for multiple arguments. For simulations using non-tested code, it is recommended to at least activate the debug mode with option `--configure-args "--enable-debug"`.

- `--exe <file>`

  Multi2Sim executable file to be used for simulations. This option overrides the default behavior of fetching a Multi2Sim version for the SVN repository. Instead, it allows the user to specify a custom version of the simulator. Options `-r`, `--tag`, and `--configure-args` are ignored if option `--exe` is used.

  The user should make sure that the executable can run correctly on the server environment. Preferably, the executable should be created through a compilation on the server, or the executable should be the cached file generated with a previous call to option `--exe-dir`.

- `--exe-dir <dir>`

  Directory in the local machine containing the Multi2Sim source code to be used for simulations, instead of a version of the official SVN repository. Before launching the cluster, this code is sent to the server and compiled. A copy of the binary created in the server is also imported and cached in the client machine. To avoid the remote compilation overhead, a future cluster can reuse this binary by means of option `--exe` instead. Options `-r`, `--tag`, and `--configure-args` are ignored if option `--exe-dir` is used.

9.4.4 Command `state`

Syntax:

```
m2s-cluster.sh state <cluster> [-v]
```

Print the current state of a cluster. Additional information about the cluster is printed if optional flag `-v` is specified. The cluster can be in any of the following states:

- `Invalid`. The cluster does not exist.
- `Created`. The cluster has been created, but not submitted to the server yet.
- `Submitted`. The cluster has been submitted to the server, and is currently running. Additional information is attached to this state when flag `-v` is used.
- `Completed`. All jobs associated with the cluster have completed execution in the server.
- `Killed`. The cluster has been killed before completing execution with command `m2s-cluster.sh kill`.

9.4.5 Command `wait`

Syntax:

```
m2s-cluster.sh wait <cluster1> [<cluster2> [...]]
```

Wait for a list of clusters to finish execution. The command finishes once all clusters are in state `Created`, `Completed`, `Killed`, or `Invalid`. The server is queried periodically to obtain the latest state for all clusters.

9.4.6 Command `kill`

Syntax:

```
m2s-cluster.sh kill <cluster>
```

Kill all jobs associated with the cluster in the server. The cluster must be in state `Submitted` for this operation to be valid. After killing the cluster, it transitions to state `Killed`.

9.4.7 Command `import`

Syntax:

```
m2s-cluster.sh import <cluster> [-a]
```

Import all output files and simulation reports generated by Multi2Sim for all jobs in the cluster. A directory hierarchy is created in the client machine at `CLIENT:~/m2s-client-kit/result/<cluster>/`, with one subdirectory per job containing its associated files. This directory hierarchy is identical to that created in the server for the execution of the cluster at `SERVER:~/m2s-server-kit/run/<cluster>/`, but the former only contains simulation output files. The selection of which files need to be imported is done by analyzing their name and selecting the following:

- `sim.out` – Output of the benchmark running on Multi2Sim.
- `sim.ref` – Reference output of the benchmark. This file is originally provided by the benchmark in some cases, and can be useful to check simulation correctness by comparing it with `sim.out`.
- `sim.err` – Simulator output, usually containing a summary of the statistic reports.
- `XXX-report` – All files with the `-report` suffix are imported.
- `XXX-config` – Configuration files with the `-config` suffix are also imported.

If optional flag `-a` is specified, all files in the running directory are imported, including benchmark executable and data files, and regardless of their names. The cluster must be in state `Submitted`, `Completed`, or `Killed`.

9.4.8 Command `remove`

Syntax:

```
m2s-cluster.sh remove <cluster>
```

Remove all information about the cluster and its jobs. The entire directory hierarchy associated with the cluster, both in the server and client, is deleted at the following locations:

```
SERVER:~/m2s-server-kit/run/<cluster>
CLIENT:~/m2s-client-kit/result/<cluster>
```

A cluster must be in state `Created`, `Completed`, or `Killed`. Querying the cluster state after it is removed returns a virtual state `Invalid`.

9.4.9 Command `list`

Syntax:

```
m2s-cluster.sh list [<cluster>]
```

If no value is given for argument `<cluster>`, a list of all existing clusters is printed. If the name of a cluster is given, all jobs added to `<cluster>` are listed. Following the listing, the standard error output shows a summary of the printed clusters or jobs.

9.4.10 Command `list-bench`

Syntax:

```
m2s-cluster.sh list-bench <server> [<suite>]
```

If optional argument `<suite>` is omitted, this command lists the benchmark kits available in the server. If a benchmark suite is given, the command lists the benchmarks available in the server for that suite.

9.4.11 Command `server`

Syntax:

```
m2s-cluster.sh server <cluster>
```

Print the server where a cluster is or was running. The syntax of the output string is `<server>[:<port>]`, where the port is only specified if other than 22. The cluster must be in state `Submitted`, `Completed`, or `Killed`.

9.5 Automatic Creation of Clusters: Verification Scripts

Clusters of simulations can be created automatically using scripts that launch Multi2Sim verification tests, architectural exploration experiments, or any other predefined set of simulations. These scripts are referred to in this section as *verification scripts*. They use a set of calls to `m2s-cluster.sh` to create, add jobs to, and submit a cluster, in such a way that the specific set of jobs forming the cluster is abstracted from the user.

To ease usage compatibility of verification scripts, this section presents a standard interface to all of them. By following this common interface, a verification script can either spawn its own threads, launch secondary verification scripts, or in turn be called by other verification scripts. Ultimately, any verification script is associated with one or more clusters, directly or indirectly created, whose state can be easily queried or modified through the proposed interface.

A set of standard verification scripts is currently available in the client kit at `CLIENT:~/m2s-client-kit/remote-tests`. For example, script `test-amdapp-2.5-evg-emu.sh` launches the emulation of the AMDAPP-2.5 benchmark suite for Evergreen, runs a validations on the benchmark outputs, and plots emulation time and other statistics.

Similarly to script `m2s-cluster.sh`, the interface of a verification script includes a command as its first argument—an execution without arguments will just print a help message. The following sections define the behavior for each command.

9.5.1 Command `submit`

Syntax:

```
<script_name>.sh submit <server>[:<port>] [<options>]
```

Create the set of one of more clusters associated with the verification script, fill them with the corresponding jobs, and launch the clusters in the server machine specified in `<server>`. The optional arguments for this command do not differ from command `m2s-cluster.sh submit`. In fact, these options are internally passed to `m2s-cluster.sh` once it is time to submit the cluster. The options are:

- `-r <revision>`. Multi2Sim SVN revision to be used in `SERVER`.
- `--tag <tag>`. Tag directory to be used (Multi2Sim trunk used if not specified).
- `--configure-args <args>`. Arguments for the `configure` script.
- `--exe <file>`. Multi2Sim custom executable.
- `--exe-dir <dir>`. Multi2Sim custom directory.

9.5.2 Command `kill`

Syntax:

```
<script_name>.sh kill
```

Kill all clusters associated with the verification script. This command recursively calls the `kill` command of secondary verification scripts, and runs `m2s-cluster.sh kill` for those clusters explicitly created by this script that are in state `Submitted`.

9.5.3 Command `state`

Syntax:

```
<script_name>.sh state
```

Return the state of the verification script. This state is computed as a combination of the cluster states and the states of secondary scripts, using the following recursive definitions:

- `Invalid`. All clusters associated with this verification script, as well as all secondary verification scripts, are in state `Invalid` (i.e., do not exist).
- `Submitted`. At least one of the clusters associated with this verification script, or alternatively a secondary verification script, is in state `Submitted`.
- `Completed`. All clusters associated with this verification script, as well as all secondary verification scripts are in state `Completed`.
- `Killed`. At least one of the clusters associated with this verification script, or alternatively a secondary verification script, is in state `Killed`. An exception is the fact that any associated cluster or secondary verification script is in state `Submitted`; in this case, the reported state is `Submitted` instead.

9.5.4 Command `wait`

Syntax:

```
<script_name>.sh wait
```

Wait for all associated clusters and secondary verification scripts to reach state `Completed`. A summary of the global state is reported in a string periodically updated in real time.

9.5.5 Command `process`

Syntax:

```
<script_name>.sh process [-f]
```

Import output files generated during the execution of clusters associated with this and secondary verification scripts. File importation is performed by calling command `m2s-cluster.sh import` for every associated cluster, and command `<sec_script_name>.sh process` for every secondary script. Output files are then processed by generating plots or analyzing results, and a verification error code is return (0 for passed, non-zero for failed verification). Failed verifications are propagated recursively through secondary verification scripts. Unless flag `-f` is specified, output files are imported only when a local copy of the results is not present in the client (i.e., the clusters and secondary scripts are imported for the first time).

9.5.6 Command `remove`

Syntax:

```
<script_name>.sh remove
```

Remove all clusters created by this verification script, as well as secondary verification scripts. The server and client copies of the execution and result directories are deleted, using command `m2s-cluster.sh remove` for each cluster. The verification script state must be `Completed` or `Killed`.

9.6 Benchmark Kits

Each benchmark suite in M2S-Cluster is provided as a separate repository with prefix `m2s-bench-`. When simulation of a given benchmark is launched, the benchmark kit containing that workload

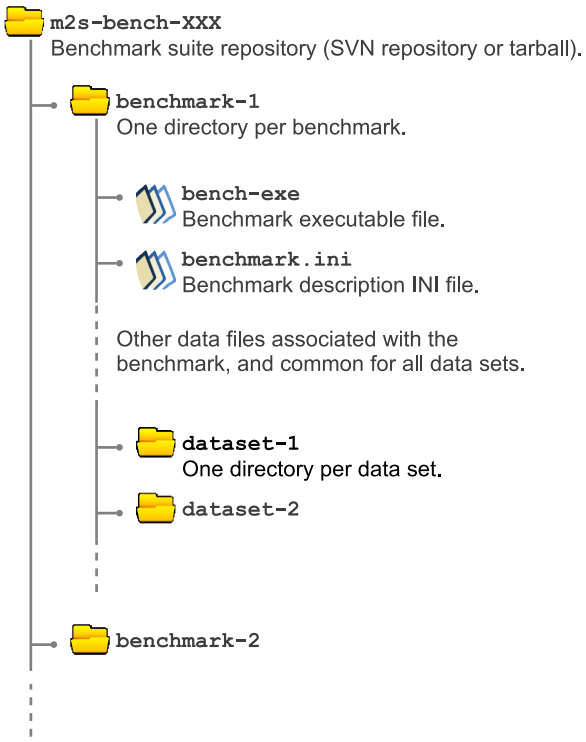


Figure 9.2: Structure of a directory containing a benchmark kit.

needs to be installed, and updated to the latest SVN version, at the top level of the user's home directory on `SERVER`. Benchmark kits are available as public SVN repositories, as well as software packages on Multi2Sim's website. For those benchmarks holding commercial licenses, only statically compiled binaries are provided.

Figure 9.2 shows the directory structure followed by all benchmark suites in M2S-Cluster. Using PARSEC-2.1 as an example, the parent directory `m2s-bench-parsec-2.1` contains one subdirectory per workload: `blackscholes`, `bodytrack`, `canneal`, etc. The regular (non-directory) files in the benchmark directory are those files required for the benchmark execution that are irrespective of the selected dataset. These files are automatically copied by M2S-Cluster to a local execution directory when simulations are submitted.

The benchmark directory also contains a set of subdirectories, one for each dataset supported by the benchmark. For PARSEC-2.1, datasets `small`, `medium`, and `large` are provided. When M2S-Cluster launches a benchmark, the contents of the dataset directory, as requested by the user through command `m2s-cluster.sh add --data-set`, are copied together with the rest of the workload's regular files, and at the same level as the benchmark executable.

Finally, a file named `benchmark.ini` is present in every benchmark's directory, containing additional information for its execution. An example of the `benchmark.ini` file contents is presented in the following listing, for benchmark `blackscholes`:

```
[ Default ]
Exe = blackscholes
Args = $NTHREADS in_16K.txt prices.txt
Data = data-medium

[ Small ]
Exe = blackscholes
Args = $NTHREADS in_4K.txt prices.txt
Data = data-small

[ Medium ]
Exe = blackscholes
Args = $NTHREADS in_16K.txt prices.txt
Data = data-medium
```

The file follows the INI file format, where each section corresponds to a dataset, with the same name as the corresponding subdirectory in the benchmark directory hierarchy. A section named `[Default]` is present for all benchmarks, used as the description for the benchmark execution when option `--data-set` is omitted from command `m2s-cluster.sh add`. The possible variables for each section are:

- `Exe`. Name of the executable file of the benchmark, as found at the top-level of the benchmark directory.
- `Args`. Arguments used for the benchmark execution. Variable `$NTHREADS` can be used as an argument, which will be replaced during the execution of command `m2s-cluster.sh add` by the value specified in option `-p`, or by 1 if this option is absent.
- `Env`. List of environment variables added for the benchmark simulation. Variable `$NTHREADS` can be used here as well with the same purpose.
- `Data`. Dataset used for the benchmark execution. The contents of the directory with the same name are copied during the execution of command `m2s-cluster.sh submit` to a temporary execution directory for the benchmark, together with the benchmark executable and other regular files.

9.7 Usage Examples

9.7.1 Tying It Out

As a first example, and assuming a successful installation of M2S-Cluster as presented in Section 9.3.1, let us target the distributed execution of a set of two simulations composed of benchmark *MatrixMultiplication* from the Evergreen AMDAPP-2.5 suite and benchmark *fft* from the SPLASH2 suite. In this context, the term *cluster* will be used hereafter referring to a group of independent executions of Multi2Sim on `SERVER`, each of which will be called, in turn, a *job*.

For all following examples, it is also assumed that all benchmark kits (directories with prefix `m2s-bench-`) needed by simulations are present and up to date on `SERVER`. In this particular case, directories `m2s-bench-amdapp-2.5-evg` and `m2s-bench-splash2` must be present under the user's home directory on the server machine. Besides these considerations, the management of the distributed simulation is fully administrated locally at the user's client machine, by means of the shell script `m2s-cluster.sh` located in `user@CLIENT:~/m2s-client-kit/bin`. For simplicity, the code listings below assume that this directory is part of the `PATH` environment variable, allowing the script to be easily invoked through a command line that omits its absolute path in the command line.

First, a cluster of simulations, named `my-cluster`, is created.

```
user@CLIENT:~$ m2s-cluster.sh create my-cluster
creating cluster 'my-cluster' - ok
```

The two mentioned benchmarks are added to the cluster as two different jobs, named `job-0` and `job-1`, respectively. Benchmarks are identified by the name of the benchmark suite (omitting the `m2s-bench-` prefix), followed by the benchmark name itself, and separated by a forward slash "/".

```
user@CLIENT:~$ m2s-cluster.sh add my-cluster job-0 amdapp-2.5-evg/MatrixMuliplication
queuing job 'job-0' to cluster 'my-cluster' - job 0 - ok
user@CLIENT:~$ m2s-cluster.sh add my-cluster job-1 splash2/fft
queuing job 'job-1' to cluster 'my-cluster' - job 1 - ok
```

Once the cluster and its components have been defined, it is submitted for remote execution in `SERVER`.

```
user@CLIENT:~$ m2s-cluster.sh submit my-cluster SERVER
Checking Multi2Sim trunk, SVN Rev. 1001 - up to date - ok
submitting cluster 'my-cluster' - sending files - condor id 14349 - 2 jobs submitted - ok
```

Upon the absence of additional options in the execution of command `m2s-cluster.sh submit`, the latest SVN revision of Multi2Sim is fetched from its official repository and built in the server. Despite an initially lasting compilation of the simulator, the submission of subsequent clusters is sped up as the generated binary is cached in `SERVER`.

The state of the submitted cluster can be checked from the client machine. If the returned state has not transitioned from `Submitted` to `Completed` yet, the user can wait for its finalization through a blocking call.

```
user@CLIENT:~$ m2s-cluster.sh state my-cluster
Submitted
user@CLIENT:~$ m2s-cluster.sh wait my-cluster
1 total, 0 created, 1 submitted, 0 completed, 0 killed, 0 invalid (as of 11:01am)
```

When the cluster completes, simulation results are obtained.

```
user@CLIENT:~$ m2s-cluster.sh import my-cluster
importing cluster output - create package - import - ok
```

These results are automatically made available in path `CLIENT:~/m2s-client-kit/result/my-cluster`. A listing of the directory contents shows two internal subdirectories, one for each job in the cluster. Further exploring the contents of the subdirectory associated with a job, for example `job-0`, the standard output and standard error output are found in files `sim.out` and `sim.err`, respectively.

Once results become useless and can be safely discarded, the cluster is removed, completely clearing the associated directory in `CLIENT` as well as all temporary files originated in `SERVER` during its execution.

```
user@CLIENT:~$ m2s-cluster.sh remove my-cluster
removing cluster 'my-cluster' - removing cluster in server - ok
```

9.7.2 Using a Modified Copy of Multi2Sim

By default, M2S-Cluster uses a copy of Multi2Sim's source code from the official SVN repository. It is common, however, that a user modifies the source code, and then launches a set of simulations

running on the generated private binary `m2s`. Command-line options `--exe-dir` and `--exe` in script `m2s-cluster.sh` serve this specific purpose. In the following example, let us assume that a private modified copy of Multi2Sim is located at `user@CLIENT:~/project/multi2sim`, used to run a cluster with benchmarks `fft`, `lu`, and `ocean` from the SPLASH2 benchmark suite. The following commands create the cluster:

```
user@CLIENT:~$ m2s-cluster.sh create my-cluster
creating cluster 'my-cluster' - ok

user@CLIENT:~$ m2s-cluster.sh add my-cluster job-0 splash2/fft
queuing job 'job-0' to cluster 'my-cluster' - job 0 - ok

user@CLIENT:~$ m2s-cluster.sh add my-cluster job-1 splash2/lu
queuing job 'job-1' to cluster 'my-cluster' - job 1 - ok

user@CLIENT:~$ m2s-cluster.sh add my-cluster job-2 splash2/ocean
queuing job 'job-2' to cluster 'my-cluster' - job 2 - ok
```

The cluster is submitted to the server using option `--exe-dir`, with its argument pointing to the parent directory of the private copy of Multi2Sim.

```
user@CLIENT:~$ m2s-cluster.sh submit my-cluster SERVER --exe-dir /home/user/project/multi2sim
submitting cluster 'my-cluster' - building
    [ cached in '/home/user/m2s-client-kit/tmp/m2s-remote-exe' ]
    - sending files - condor id 14352 - 3 jobs submitted - ok
```

If building the simulator fails on the server, the cluster is not submitted, and the compilation log is dumped in the standard output. Upon success, a copy of the executable generated in the server is cached locally in the temporary directory of the client kit, and an output message shows the path where this executable can be accessed (`/home/ubal/m2s-client-kit/tmp/m2s-remote-exe`). This executable can be passed later with another cluster submission (option `--exe`) to avoid the overhead of rebuilding the code in the server.

```
user@CLIENT:~$ m2s-cluster.sh wait my-cluster
1 total, 0 created, 0 submitted, 1 completed, 0 killed, 0 invalid (as of 11:04am)

user@CLIENT:~$ m2s-cluster.sh submit my-cluster SERVER \
    --exe /home/user/m2s-client-kit/tmp/m2s-remote-exe
submitting cluster 'my-cluster' - sending files - condor id 14353 - 3 jobs submitted - ok
```

9.7.3 Transferring Configuration Files and Reports

The input files consumed by Multi2Sim, as well as the output files generated by it, need to be transferred from `CLIENT` to `SERVER` and vice versa. In the following example, a cluster with one single simulation is created, using benchmark `fft` from the SPLASH2 suite, where input file `user@CLIENT:~/Documents/x86-config` is used as the x86 pipeline configuration file, and a detailed x86 pipeline report is obtained from the simulation. The example also illustrates how to pass arguments to Multi2Sim's command line by using the `--sim-arg` option in M2S-Cluster.

```
user@CLIENT:~$ m2s-cluster.sh create my-cluster
creating cluster 'my-cluster' - ok

user@CLIENT:~$ m2s-cluster.sh add my-cluster job-0 splash2/fft \
    --sim-arg "--x86-sim detailed --x86-config x86-config --x86-report report-pipeline" \
    --send /home/user/Documents/x86-config
queuing job 'job-0' to cluster 'my-cluster' - job 0 - ok

user@CLIENT:~$ m2s-cluster.sh submit my-cluster SERVER
Checking Multi2Sim trunk, SVN Rev. 1001 - up to date - ok
submitting cluster 'my-cluster' - sending files - condor id 14354 - 1 jobs submitted - ok
```

In the `m2s-cluster.sh add` command above, three options are specified as command-line arguments for Multi2Sim: one option to activate a detailed simulation, a second option to use the given x86 pipeline configuration file, and a third option to dump the x86 pipeline report to a file called `report-pipeline`. For the last option, notice the need to make the output file have the `report-XXX` prefix; this allows it to be later imported automatically with command `m2s-cluster.sh import`. Option `--send` points to the x86 pipeline configuration file to be sent to the server before simulation starts.

Once the cluster has been submitted, we wait for its finalization, and import the simulation results.

```
user@CLIENT:~$ m2s-cluster.sh wait my-cluster
1 total, 0 created, 0 submitted, 1 completed, 0 killed, 0 invalid (as of 03:54pm)

user@CLIENT:~$ m2s-cluster.sh import my-cluster
importing cluster output - create package - import - ok
```

Simulation results are retrieved in `CLIENT:~/m2s-client-kit/result`, under a subdirectory named after the imported cluster (`my-cluster`). The latter directory contains, in turn, one subdirectory for each job that is part of the cluster (in this case, only `job-0`). Finally, the job directory contains its associated simulation results imported from the server, as observed next:

```
user@CLIENT:~$ cd m2s-client-kit/result/my-cluster/job-0
user@CLIENT:~/m2s-client-kit/result/my-cluster/job-0$ ls
ctx-0  report-pipeline  sim.err  sim.out
```

Chapter 10

Tools

10.1 The INI file format

An INI file is a plain text file used to store configuration information and statistic reports for programs. Multi2Sim uses this format for all of its input and output files, such as context configuration or cache hierarchy configuration files. This format is also used in Multi2Sim for output files, such as detailed simulation statistics reports. This is an example of a text file following the INI file format:

```
; This is a comment
[ Button Accept ]
Height = 20
Width = 40
Caption = 'OK'

[ Cancel ]
State = Disabled
```

Each line of an INI file can be a comment, a section name, or a variable-value pair. A comment is a line starting with a semicolon; a section name is given as a string set off by square brackets (e.g., `[ Button Accept ]`); and a variable-value pair is represented by separating the variable name and its value with an `=` sign. Section and variable names are case-sensitive in Multi2Sim.

The user can specify the values for an integer variable in decimal, hexadecimal, and octal formats. The latter two formats use the `0x` and `0` prefixes, respectively. Integer variables can also include suffixes `K`, `M`, and `G` to multiply the number by 10^3, 10^6, and 10^9, respectively. Lower-case suffixes `k`, `m`, and `g` multiply the number by 2^{10}, 2^{20}, and 2^{30}, respectively.

10.1.1 The inifile.py tool

The `inifile.py` tool can be found in the `tools/inifile` directory within the Multi2Sim distribution package. It is a Python script aimed at automatically analyzing and modifying INI files, avoiding their manual edition. The command-line syntax of the program can be obtained by executing it without arguments. To illustrate its functionality by an example, let us run a simulation of the `test-args` and `test-sort` benchmarks on a 2-threaded processor model, by using the files provided in the `samples` directory. Run the following command under the `samples/x86` directory:

```
m2s --x86-sim detailed --ctx-config ctx-config-args-sort --x86-config x86-config-args-sort \
        --x86-report x86-report
```

This command uses the `ctx-config-args-sort` context configuration file, which allocates benchmark `test-args` in context 0, and benchmark `test-sort` in context 1. Likewise, it uses the `x86-config-args-sort` to set up 2 threads, and dumps a detailed pipeline statistics report into file `x86-report`. All `ctx-config-args-sort`, `x86-config-args-sort`, and `x86-report` files follow the INI file format. After running this simulation, let us analyze the obtained results with the `inifile.py` script.

10.1.2 Reading INI files

As shown in Section 2.21, the pipeline statistics report includes one section per core, thread, and complete processor. Type the following commands:

```
inifile.py x86-report read c0t0 Commit.Total
inifile.py x86-report read c0t1 Commit.Total
inifile.py x86-report read c0 Commit.Total
```

These commands return the number of committed instructions in thread 0, thread 1, and core 0. Since threads 0 and 1 are contained in core 0, the third output value is equal to the sum of the two first values.

10.1.3 Writing on an INI file

To show how to modify the contents of an INI file, the following example changes the context configuration file using `inifile.py`:

```
inifile.py ctx-config-args-sort remove "Context 0" StdOut
m2s --x86-sim detailed --ctx-config ctx-config-args-sort --x86-config x86-config-args-sort \
        --x86-report x86-report
inifile.py ctx-config-args-sort write "Context 0" StdOut context-0.out
```

The first line removes the parameter `StdOut` in the `[Context 0]` section. Then, the second line reruns the simulation with the new context file. Since the redirection of the standard output has been removed, the `test-args` benchmark dumps its output to screen. Finally, the third line restores the original contents of the context file, by adding the `StdOut` parameter again.

10.1.4 Using scripts to edit INI files

Every time the `inifile.py` tool is called, it analyzes the complete structure of the INI file before performing the requested action on it. For large INI files, this can entail some costly work, which becomes redundant when several actions are performed on the same file. In this case, it is possible to parse the INI file only the first time, by using an `inifile.py` script, as follows:

```
script=$(mktemp)
echo "read c0 Commit.Total" >> $script
echo "read c0t0 Commit.Total" >> $script
echo "read c0t1 Commit.Total" >> $script
inifile.py x86-report run $script
rm -f $script
```

The code above creates a temporary file (command `mktemp`), whose name is stored in variable `script`. Then, three `read` actions are stored into the script to retrieve the number of committed instructions in core 0, thread 0, and thread 1. Finally, the `inifile.py` tool is executed with the `run`

command, using the script file name as last argument. The result obtained by each `read` command is presented in a new line on screen.

10.2 McPAT: Power, Area, and Timing Model

McPAT is an integrated power, area, and timing modeling framework that supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond [14]. The tool and the technical report describing it can be downloaded in the following link:

`http://www.hpl.hp.com/research/mcpat`

McPAT provides a flexible XML interface that can interact with a properly modified performance simulator. Through a plain-text file in XML format, McPAT receives the architectural parameters and technological features of the modeled processor, as well as some structure access statistics provided by a performance simulator. With this information, the tool dumps power, area, and timing statistics for the computed design.

McPAT uses the Cacti tool [15] to obtain the models for the on-chip data arrays, CAM tables, and cache structures. It also uses its own power, area, and timing models for combinational logic, such as functional units, results broadcast buses, instruction wakeup logic, or interconnection networks.

10.2.1 McPAT input file

The following is en excerpt of an XML McPAT input file specifying the characteristics and activity of the instruction cache:

```
<component id="system.core0.icache" name="icache">
   <param name="icache_config" value="131072,32,8,1,8,3,32,0"/>
   <param name="buffer_sizes" value="16,16,16,0"/>
   <stat name="read_accesses" value="1000"/>
   <stat name="read_misses" value="56"/>
   <stat name="conflicts" value="3"/>
</component>
```

In this example, the `<param>` entries specify the cache geometry and cache controller buffer sizes, respectively. The `<stat>` entries give the read accesses, read misses, and cache line conflicts occurred during a performance simulation carried out previously on top of Multi2Sim.

By analyzing this data, jointly with some other global technological data such as feature size or clock frequency, McPAT can estimate the area, access time, and both dynamic and leakage energy dissipated during the execution of the simulated benchmarks.

10.2.2 Interaction with Multi2Sim

Multi2Sim has been adapted to provide those statistics that McPAT requires in its input file. Though the processor models provided by McPAT and Multi2Sim are not exactly the same, still some common configurations can be obtained to estimate the global energy dissipated for a given benchmark execution. The correspondence between the Multi2Sim statistics and McPAT input parameters is given in Appendix II at the end of this document.

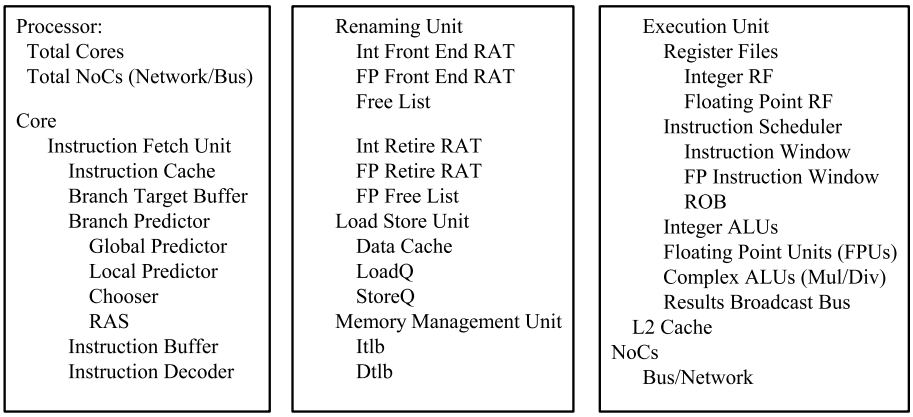


Table 10.1: Hardware components reported by McPAT

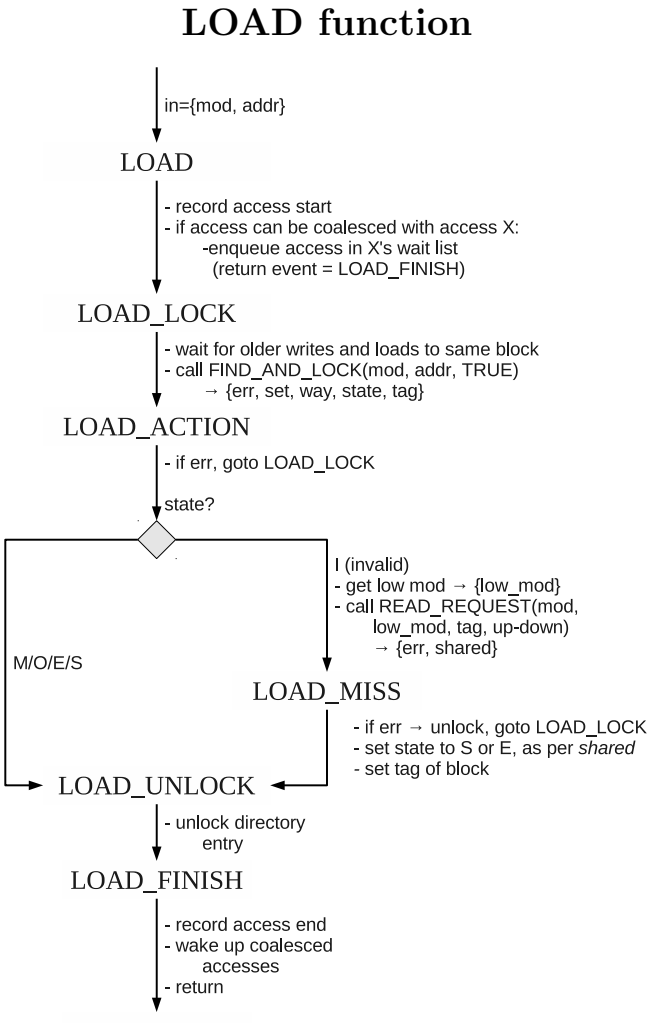
10.2.3 McPAT output

Table 10.1 shows a list of the hardware components whose area, access time, and energy is detailed in the McPAT output. These components are classified hierarchically, and statistics are given both individually and in groups. For each of the listed elements, the following values are reported:

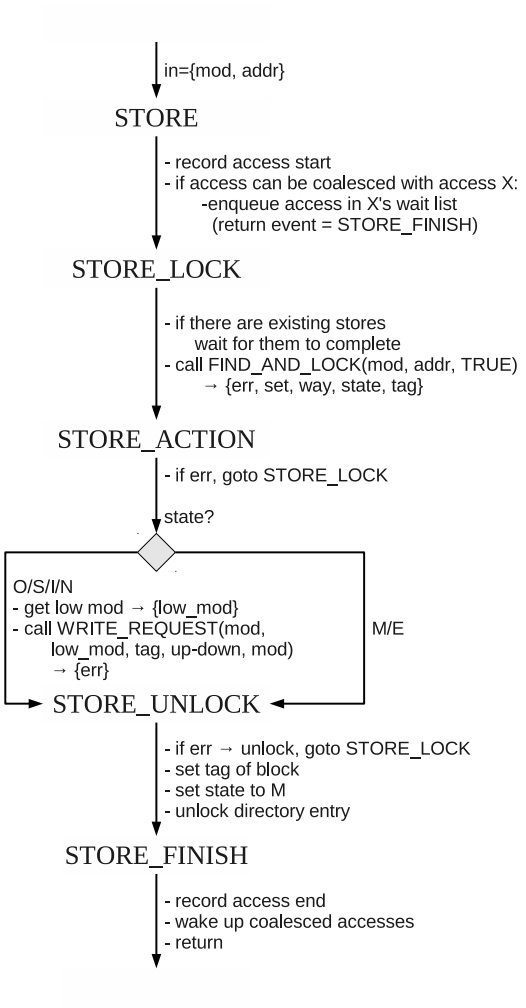
- Area (mm^2).
- Peak dynamic power (W).
- Subthreshold and gate leakage power (W).
- Average runtime dynamic power (W).

Since these values are given separately for each processor structure, the energy dissipation associated to each component can be independently evaluated. For example, an alternative design can be proposed for a given processor structure (such as a register file or data cache), and its physical properties can be evaluated with the Cacti tool [15]. Then, the Cacti results can be combined with the McPAT output, and the global power, energy, and area can be obtained for the proposed design.

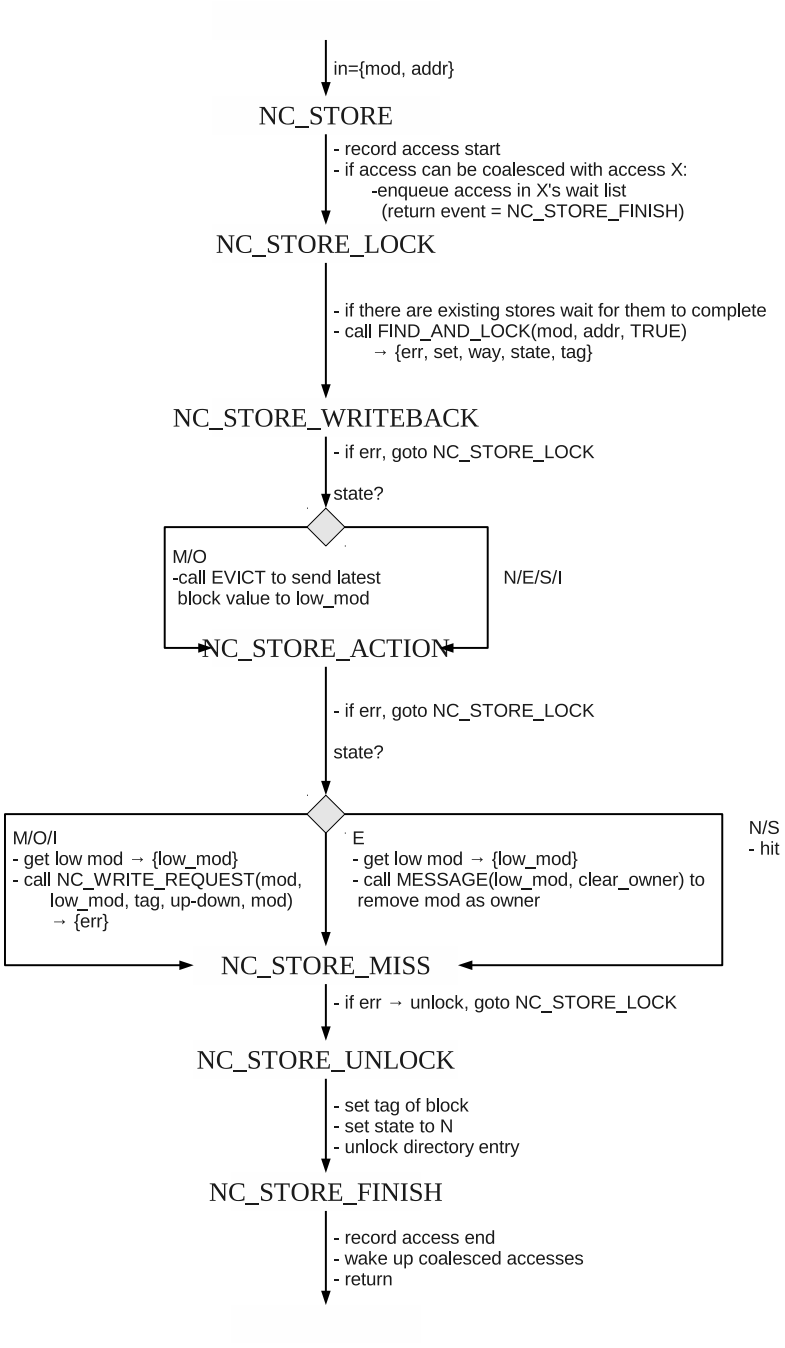
Appendix I. Event Flow Diagrams for the MOESI Protocol Implementation



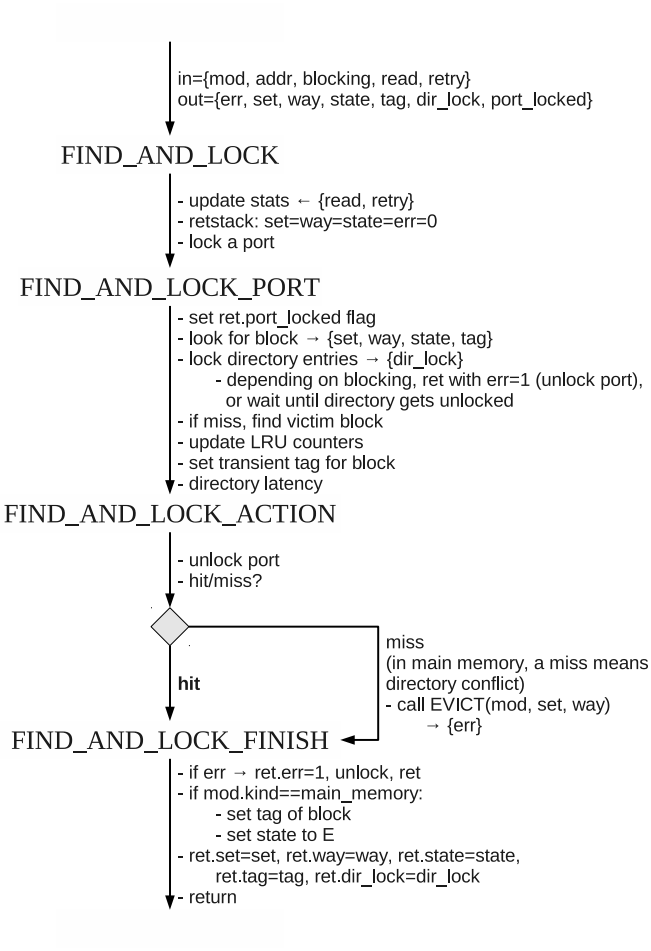
STORE function



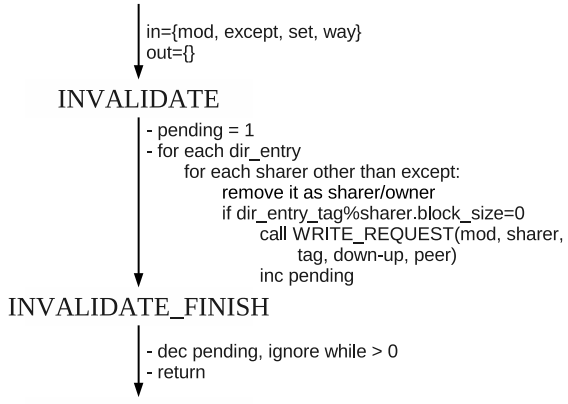
NC_STORE function



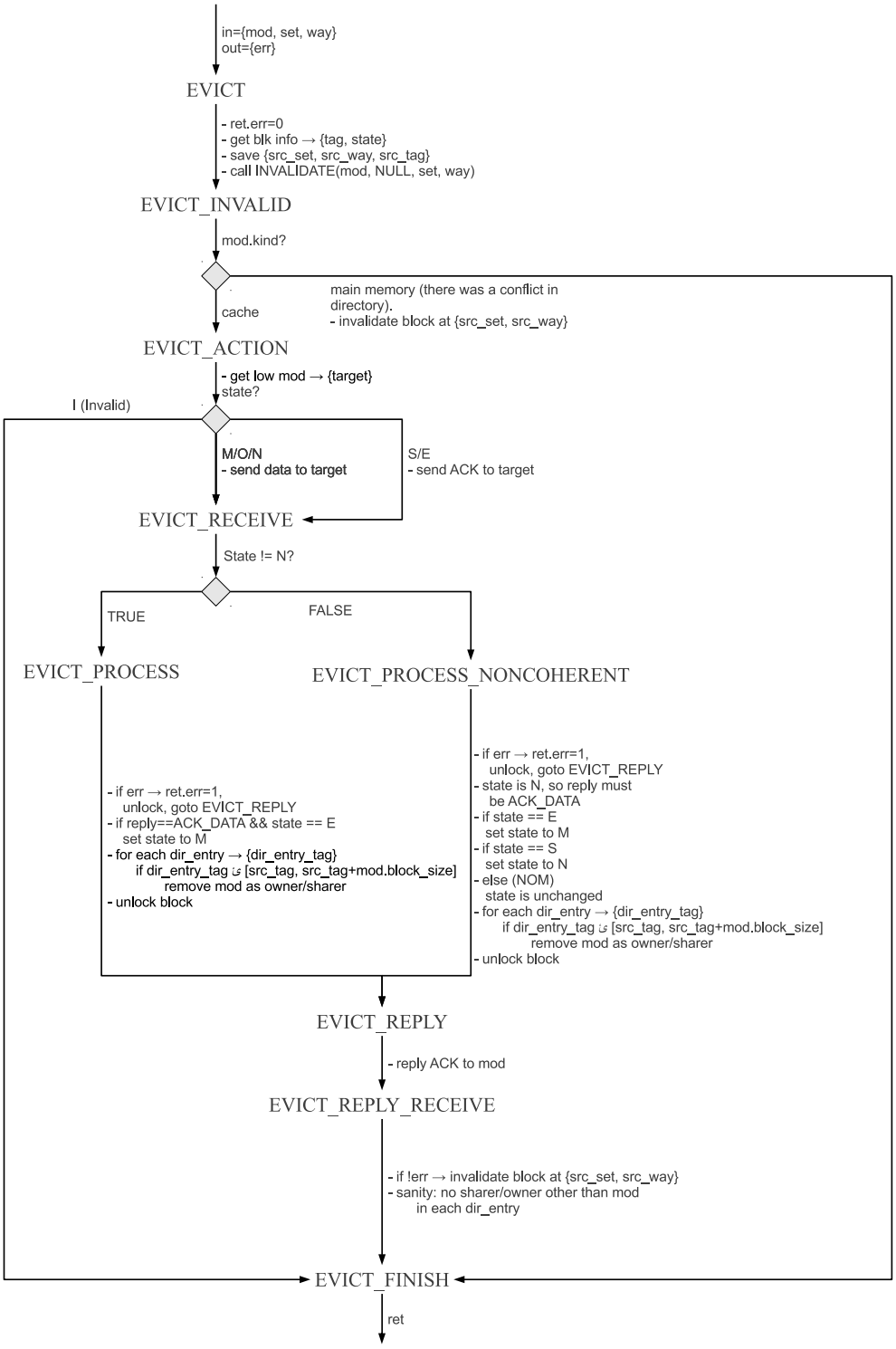
FIND_AND_LOCK function



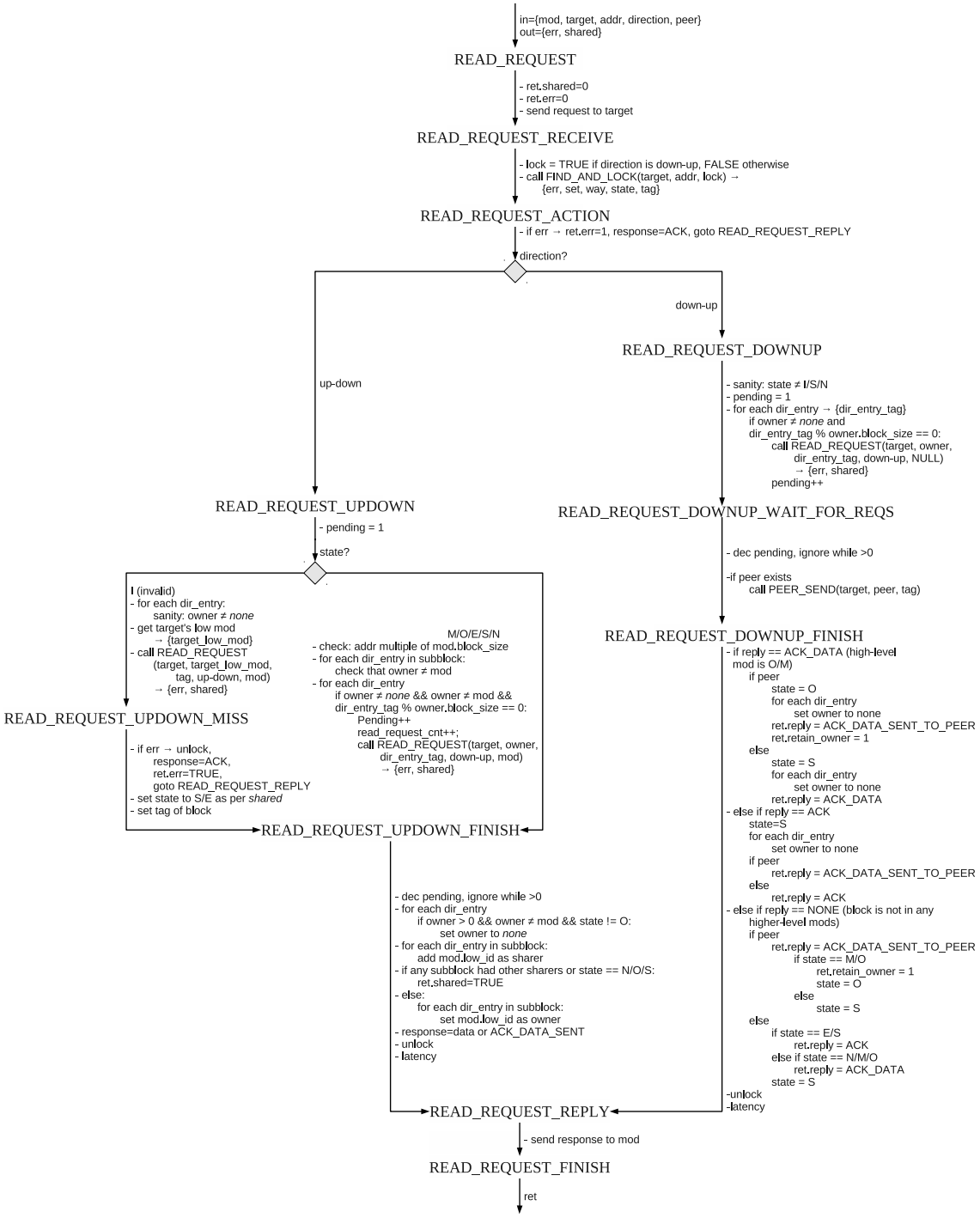
INVALIDATE function



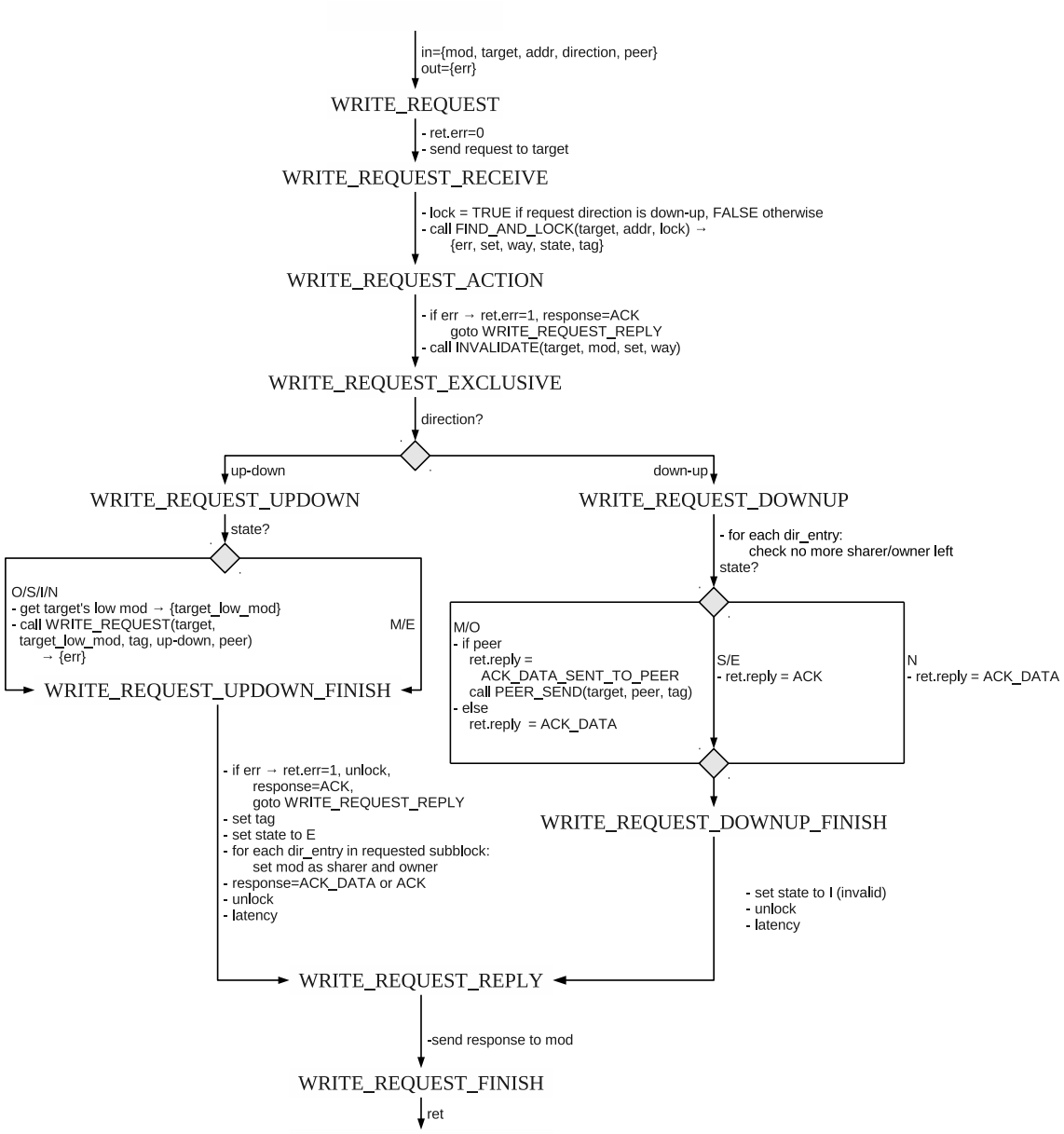
EVICT function



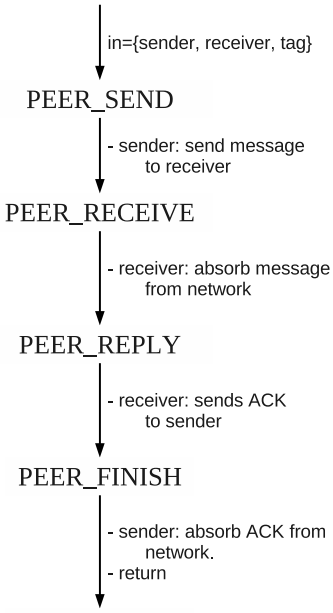
READ_REQUEST function



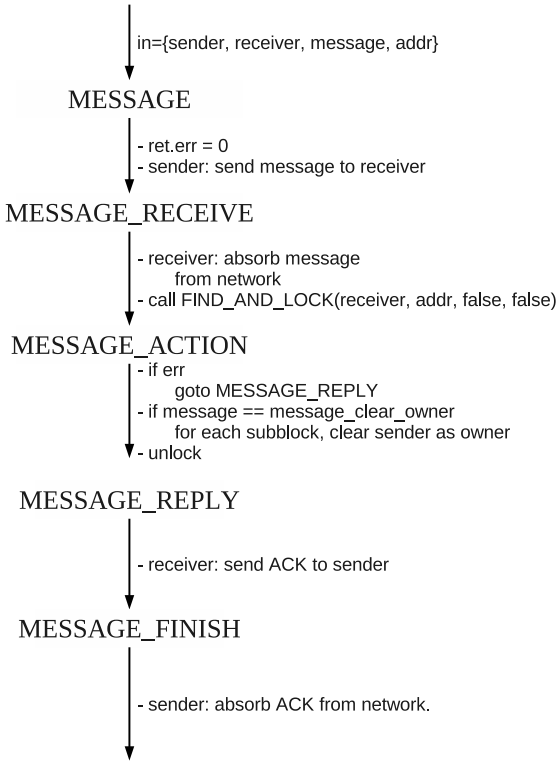
WRITE_REQUEST function



PEER function



MESSAGE function



Appendix II. Correspondence between McPAT and Multi2Sim Statistics

McPAT		Multi2Sim	
XML component	**Statistic**	**Section**	**Statistic**
system	total_cycles	global	Cycles
	idle_cycles	—	—
	busy_cycles	global	Cycles
system.core0	total_instructions	c0	Dispatch.Total
	int_instructions	c0	Dispatch.Integer
	fp_instructions	c0	Dispatch.FloatingPoint
	branch_instructions	c0	Dispatch.Ctrl
	branch_mispredictions	c0	Commit.Mispred
	load_instructions	c0	Dispatch.Uop.load
	store_instructions	c0	Dispatch.Uop.store
	committed_instructions	c0	Commit.Total
	committed_int_instructions	c0	Commit.Integer
	committed_fp_instructions	c0	Commit.FloatingPoint
	pipeline_duty_cycle	c0	Commit.DutyCycle
	total_cycles[1]	—	—
	idle_cycles	—	—
	busy_cycles	—	—
	ROB_reads	c0(t0)	ROB.Reads
	ROB_writes	c0(t0)	ROB.Writes
	rename_reads	c0t0	RAT.Reads
	rename_writes	c0t0	RAT.Writes
	fp_rename_reads[2]	—	—
	fp_rename_writes	—	—
	inst_window_reads	c0(t0)	IQ.Reads
	inst_window_writes	c0(t0)	IQ.Writes
	inst_window_wakeup_accesses	c0(t0)	IQ.WakeupAccesses
	fp_inst_window_reads[2]	—	—
	fp_inst_window_writes	—	—
	fp_inst_window_wakeup_accesses	—	—
	int_regfile_reads	c0(t0)	RF.Reads
	int_regfile_writes	c0(t0)	RF.Writes
	float_regfile_reads[2]	—	—
	float_regfile_writes	—	—
	function_calls	c0	Dispatch.Uop.call
	context_switches	c0	Dispatch.WndSwitch
	ialu_accesses	c0	Issue.SimpleInteger
	fpu_accesses	c0	Issue.FloatingPoint
	mul_accesses	c0	Issue.ComplexInteger
	cdb_alu_accesses[3]	—	—
	cdb_fpu_accesses[3]	—	—
	cdb_mul_accesses[3]	—	—
	IFU_duty_cycle[3]	—	—
	LSU_duty_cycle[3]	—	—
	MemManU_I_duty_cycle[3]	—	—
	MemManU_D_duty_cycle[3]	—	—
	ALU_duty_cycle[3]	—	—
	MUL_duty_cycle[3]	—	—

Continued on Next Page...

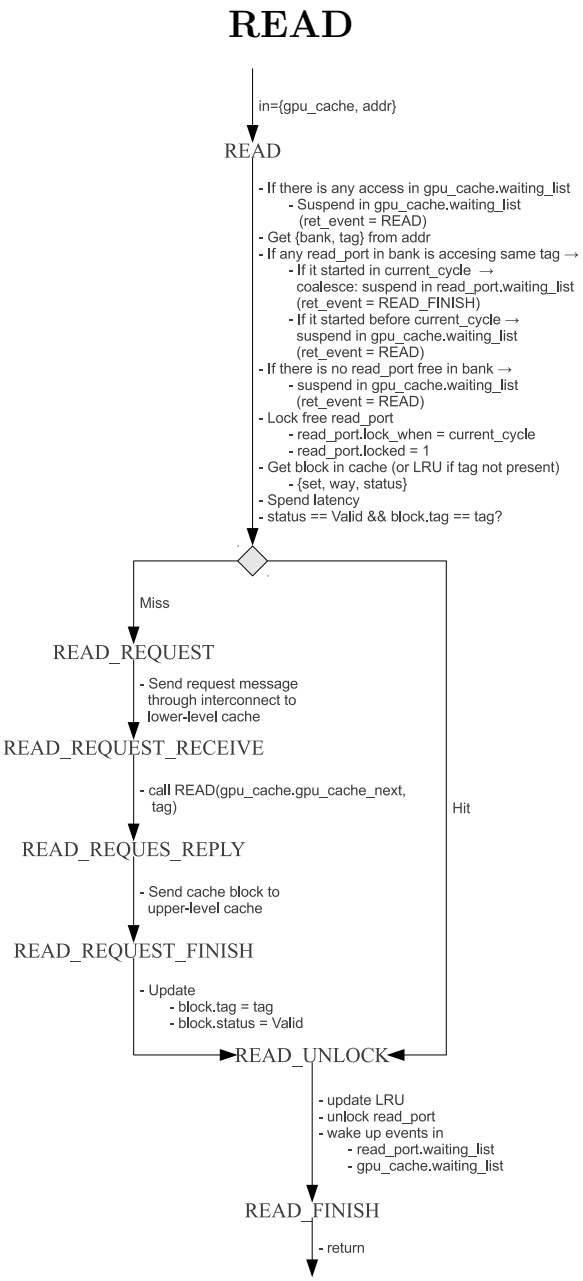
XML component	Statistic	Section	Statistic
	FPU_duty_cycle[3]	—	—
	ALU_cdb_duty_cycle[3]	—	—
	MUL_cdb_duty_cycle[3]	—	—
	FPU_cdb_duty_cycle[3]	—	—
system.core0.BTB	read_accesses	c0t0	BTB.Reads
	write_accesses	c0t0	BTB.Writes
system.core0.itlb	total_accesses	itlb.0.0	Accesses
	total_misses	itlb.0.0	Misses
	conflicts	itlb.0.0	Evictions
system.core0.icache	read_accesses	il1-0	Reads
	read_misses	il1-0	ReadMisses
	conflicts	il1-0	Evictions
system.core0.dtlb	total_accesses	dtlb.0.0	Accesses
	total_misses	dtlb.0.0	Misses
	conflicts	dtlb.0.0	Evictions
system.core0.dcache	read_accesses	dl1-0	Reads
	write_accesses	dl1-0	Writes
	read_misses	dl1-0	ReadMisses
	write_misses	dl1-0	WriteMisses
	conflicts	dl1-0	Evictions

[1]McPAT uses these values for heterogeneous cores only, which are not supported under Multi2Sim.

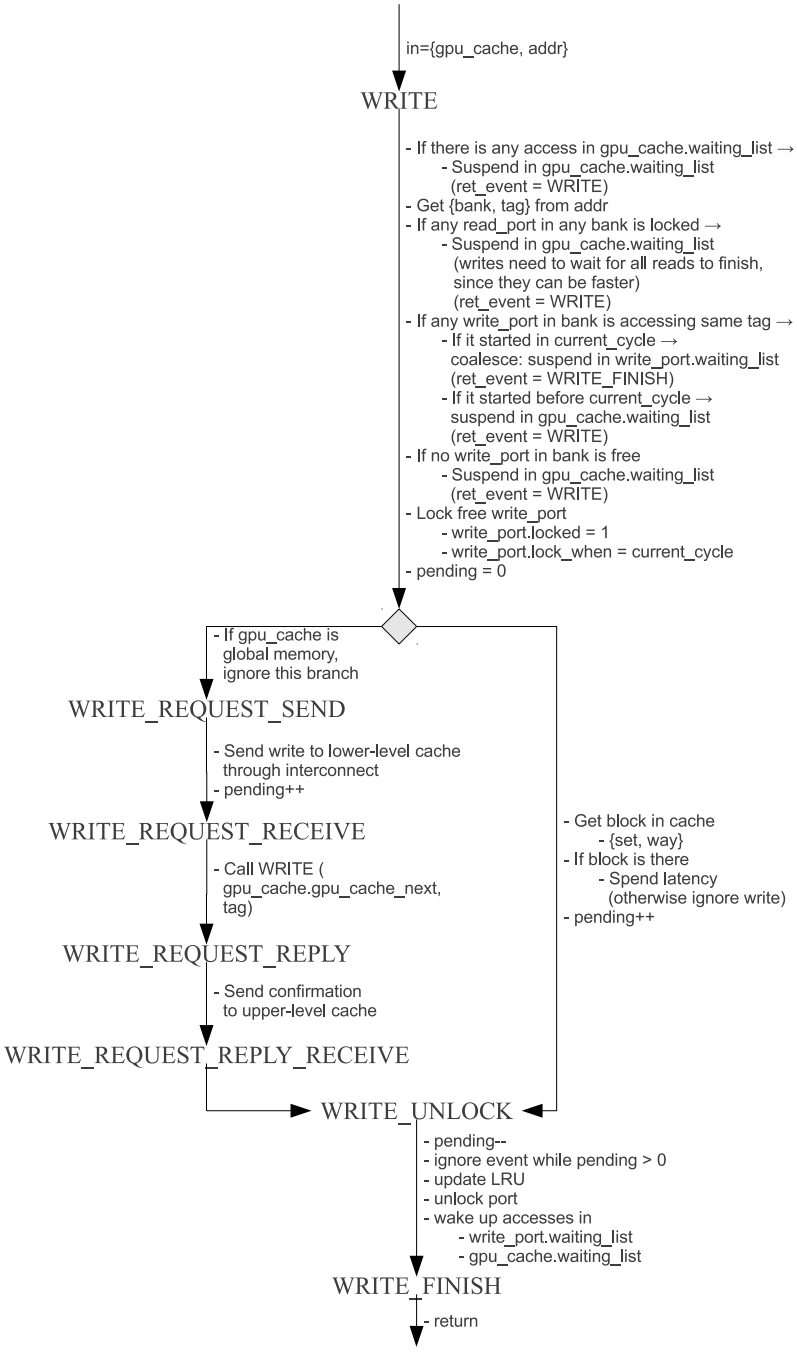
[2]Multi2Sim does not model separate renaming tables, register files, or instruction queues for integer and floating-point registers.

[3]The meaning of these stats is unclear or is not documented by the McPAT developers, so the statistics do not have a correspondence yet.

Appendix III. GPU Global Memory Hierarchy Events



WRITE


in={gpu_cache, addr}
WRITE
- If there is any access in gpu_cache.waiting_list →
- Suspend in gpu_cache.waiting_list
(ret_event = WRITE)
- Get {bank, tag} from addr
- If any read_port in any bank is locked →
- Suspend in gpu_cache.waiting_list
(writes need to wait for all reads to finish,
since they can be faster)
(ret_event = WRITE)
- If any write_port in bank is accessing same tag →
- If it started in current_cycle →
coalesce: suspend in write_port.waiting_list
(ret_event = WRITE_FINISH)
- If it started before current_cycle →
suspend in gpu_cache.waiting_list
(ret_event = WRITE)
- If no write_port in bank is free
- Suspend in gpu_cache.waiting_list
(ret_event = WRITE)
- Lock free write_port
- write_port.locked = 1
- write_port.lock_when = current_cycle
- pending = 0
- If gpu_cache is
global memory,
ignore this branch
WRITE_REQUEST_SEND
- Send write to lower-level cache
through interconnect
- pending++
WRITE_REQUEST_RECEIVE
- Call WRITE (
gpu_cache.gpu_cache_next,
tag)
WRITE_REQUEST_REPLY
- Send confirmation
to upper-level cache
WRITE_REQUEST_REPLY_RECEIVE
- Get block in cache
- {set, way}
- If block is there
- Spend latency
(otherwise ignore write)
- pending++
WRITE_UNLOCK
- pending--
- ignore event while pending > 0
- update LRU
- unlock port
- wake up accesses in
- write_port.waiting_list
- gpu_cache.waiting_list
WRITE_FINISH
- return

Appendix IV. X86 Micro-Instruction Set

Micro-instruction:	`move`
Functional unit:	None

Move the contents of an integer register into another. Source and destination operands of this micro-instruction can only be integer general purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`), segment registers (`es`, `cs`, `ss`, `ds`, `fs`, `gs`), or status flags (`zps`, `of`, `cf`, `df`).

Micro-instruction:	`add, sub`
Functional unit:	Integer adder

Integer addition and subtraction. Both types of micro-instructions are executed on the integer adder functional unit.

Micro-instruction:	`mult`
Functional unit:	Integer multiplier

Integer multiplication.

Micro-instruction:	`div`
Functional unit:	Integer divider

Integer division and modulo.

Micro-instruction:	`effaddr`
Functional unit:	Address computer

Effective address computation for memory accesses. For those instruction where an operand is a memory location, the effective address needs to be computed before. For example, the operand `[ebx+0x4]` in an x86 CISC instruction will generate an initial `effaddr` micro-instruction calculating the memory address by adding the contents of register `ebx` plus 4.

Micro-instruction:	`and, or, xor, not`
Functional unit:	Logic unit

Bitwise AND, OR, XOR, and NOT. All of them are executed on a functional unit devoted to logic operations.

Micro-instruction:	`shift`
Functional unit:	Logic unit

This micro-instruction is used for bit shifts or bit rotation operations. It is executed on the logic functional unit.

Micro-instruction:	`sign`
Functional unit:	Logic unit

Sign change. This micro-instruction is used for integer operations that just involve the sign bit of an integer number, such as absolute value computation, or sign switch. Since it involves a simple bit alteration, it is assumed to be executed on the logic unit.

Micro-instruction:	`fadd, fsub, fcomp`
Functional unit:	Floating-point adder

Floating-point addition, subtraction, and comparison. These micro-instructions are all executed on the floating-point adder functional unit. A comparison of two floating-point number involves subtracting them and checking the properties of the result.

Micro-instruction:	`fmult`
Functional unit:	Floating-point multiplier

Floating-point multiplication.

Micro-instruction:	`fdiv`
Functional unit:	Floating-point divider

Floating-point division.

Micro-instruction:	`fexp, flog, fsin, fcos, fsincos, ftan, fatan, fsqrt`
Functional unit:	Floating-point complex operator

Floating-point computation of an exponential value, floating-point logarithm, sine, cosine, combined sine/cosine, tangent, arctangent, and square root, respectively. All these operations are assumed to be implemented in hardware on a complex floating-point functional unit.

Micro-instruction:	`fpush, fpop`
Functional unit:	None

Push/pop a value into/from the floating-point stack. These two micro-instructions affect the floating-point stack pointer, which in turn causes the operants of floating-point operations (`st0`, `st1`, ...) to be interpreted differently before being mapped to physical registers. See Section 2.12 for further information about floating-point register renaming.

Micro-instruction:	`x-and, x-or, x-xor, x-not, x-shift, x-sign`
Functional unit:	XMM logic unit

XMM micro-instructions performing logic operations on 128-bit values.

Micro-instruction:	`x-add, x-sub, x-comp, x-mult, x-div`
Functional unit:	XMM integer unit

XMM micro-instructions performing integer operations on 128-bit double quad-words.

Micro-instruction:	`x-fadd, x-fsub, x-fcomp, x-fmult, x-fdiv, x-fsqrt`
Functional unit:	XMM floating-point unit

XMM micro-instruction performing floating-point operations on 128-bit XMM registers.

Micro-instruction:	`x-move, x-shuf`
Functional unit:	XMM logic unit

Move and shuffle bits between XMM registers, or between XMM and general-purpose x86 registers.

Micro-instruction:	`x-conv`
Functional unit:	XMM floating-point unit

Conversion between integer and floating-point values in 128-bit XMM registers.

Micro-instruction:	`load, store`
Functional unit:	Memory hierarchy

Memory read and write to the data cache. These micro-instructions usually follow an effective address computation. As opposed to the rest of micro-instructions, the latency of memory operations is variable, depending on the presence of blocks in data caches of the contention in the memory hierarchy interconnects.

Micro-instruction:	`call, ret`
Functional unit:	None

Call to and return from a function. These micro-instructions affect the control flow of the program. For branch prediction, they affect the return address stack (RAS), where function return addresses are pushed and popped (see Section 2.9). These micro-instructions do not require any functional unit to execute.

Micro-instruction:	`jump, branch`
Functional unit:	None

Unconditional jump and conditional branch, affecting the control flow of the program. In conditional branches, the branch predictor will provide a direction prediction for the branch in the fetch stage. The actual branch condition usually depends on the value of x86 flags (`CF`, `OF`, etc.). These micro-instructions do not require any functional unit to execute.

Micro-instruction:	`ibranch`
Functional unit:	None

Internal branch into microcode. This micro-instruction is used when decoding an x86 string operation to jump into an intermediate location within the sequence of generated micro-instructions. See Section 2.7 for more details on string operations decoding.

Micro-instruction:	`syscall`
Functional unit:	None

Micro-instruction for a system call. This instruction can only be executed in non-speculative mode. Since the operating system is not modeled in Multi2Sim, all actions performed during a system call execution are modeled as a single `syscall` micro-instruction. This is the principle of the *application-only* simulation model. On one hand, it is less acurate than a full-system simulation with an operating system running on the simulator. On the other hand, simulation speed is much higher, while it allows us to focus just on the benchmarks simulation.

Bibliography

[1] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that Use two Levels of Branch History. In *Proc. of the 20th Int'l Symposium on Computer architecture*, 1993.

[2] Tse-yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proc. of the 7th ACM Conference on Supercomputing*, 1993.

[3] Intel Corporation. *Intel Architecture – Software Developer's Manual, Volume 2: Instruction Set Reference*.

[4] E. Rotenberg, J. Smith, and S. Bennett. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th Int'l Symposium on Microarchitecture*, 1996.

[5] X. Qian, H. Huang, Z. Duan, J. Zhang, N. Yuan, Y. Zhou, H. Zhang, H. Cui, and D. Fan. *Optimized Register Renaming Scheme for Stack-Based x86 Operations*, volume 4415 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[6] The Khronos Group The OpenCL Standard. *www.khronos.org/opencl*.

[7] AMD. *Compute Abstraction Layer – Programming Guie*. www.amd.com, Dec. 2010.

[8] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK). *www.amd.com/stream*.

[9] AMD. *Evergreen Family Instruction Set Architecture: Instructions and Microcode*. www.amd.com, Feb. 2011.

[10] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide*. http://developer.amd.com/GPU/AMDAPPSDK/, Jan. 2011.

[11] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[12] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *Proc. of the 13th International Symposium on Computer architecture*, June 1986.

[13] Condor – High Throughput Computing. `http://research.cs.wisc.edu/condor/`, 2012.

[14] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of the 42nd Int'l Symposium on Microarchitecture*, Dec. 2009.

[15] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, School of Computing, University of Utah, 2007.