M-SIM: A Flexible, Multithreaded Architectural Simulation Environment

Joseph J. Sharkey, Dmitry Ponomarev, Kanad Ghose
{jsharke, dima, ghose}@cs.binghamton.edu

Abstract

M-Sim is a multi-threaded microarchitectural simulation environment with a detailed cycle-accurate model for the key pipeline structures. M-Sim extends the SimpleScalar 3.0d toolset with accurate models of the pipeline structures, including explicit register renaming, and support for the concurrent execution of multiple threads according to the Simultaneous Multithreading (SMT) model. For power estimations, M-Sim includes the Wattch framework as applied to SimpleScalar. This technical report provides an overview of M-Sim, including a detailed description of the simulated processor as well as instructions for the installation and use of the M-Sim environment. The description is focused only on the changes made with respect to the Simplescalar.

1. Introduction / Overview

Simultaneous Multithreading (SMT) [TE+ 95] is an effective technique to increase the throughput of a traditional superscalar processor with relatively little additional hardware. This is achieved via simultaneous sharing of the key datapath components, such as the issue queue and the physical register file, among multiple threads. In this report, we describe the details of M-Sim – a multi-threaded microarchitectural simulation environment with a detailed cycle-accurate model for the key pipeline structures. The goal of M-Sim is to provide a flexible simulation framework in which to conduct academic research related to SMT microprocessors.

M-Sim is based on the SimpleScalar 3.0d toolset [BA 97], supporting the unmodified, statically linked Alpha AXP binaries as well as the power estimation as supplied by the Wattch framework [BTM 00]. M-Sim extends SimpleScalar by accurately modeling the key pipeline structures, such as the issue queue, the register file and the reorder buffer. M-Sim also explicitly models register renaming. Section 2 describes the pipeline structures modeled by M-Sim. A detailed discussion of the multi-threading model implemented by M-Sim is given in Section 3. The installation and use of M-Sim is described in Section 4.

2. Cycle-Accurate Modeling of Pipeline Structures

M-sim explicitly models the Reorder Buffer (ROB), the Issue Queue (IQ), the Load/Store Queue (LSQ), separate integer and floating-point register files, register renaming, and the associated rename table.

The ROB is modeled as a FIFO buffer (ROB), with pointers to the head (ROB_head) and tail (ROB_tail) and a counter of the number of instructions currently resident in the ROB (ROB_num). Entries are allocated at the tail during instruction dispatch() and are freed from the head during instruction commit(). The ROB consists of several ROB_entry's, which replace the RUU_station's of the original SimpleScalar. The structure of an ROB_entry can be found in the sim-outorder.c file.

The IQ is modeled as an array of iq_entry's. Each iq_entry can be in one of two states: IQ_ENTRY_FREE or IQ_ENTRY_ALLOC, indicating that the corresponding entry is free or already allocated, respectively. IQ entries are allocated at instruction dispatch() and are released at issue() and on branch mispredictions in rob_recover().

The integer and floating point physical register files are modeled separately as int_reg_file and fp_reg_file, respectively. Both register files contain an array of physreg_t, which in turn contain the register state (reg_state). The register can be in one of four states: 1) the register is free (REG_FREE), 2) the register has been allocated to an instruction, but has not yet been written to (REG_ALLOC), 3) the register is allocated to an instruction and the value has been written (REG_WB), 4) the register is in the architectural state (REG_ARCH). Physical registers are allocated at the dispatch() stage, and deallocated at commit() and, in the case of branch mispredictions, rob_recover().

The rename tables are modeled as an array of integers. They maintain the current mapping of each architectural register to a physical register. Architectural registers 0 to 31 are integer registers, and registers 32 to 63 are floating point registers.

3. Details of the SMT Model and Implementation

M-Sim supports both single threaded execution (superscalar mode) and the simultaneous execution of multiple threads according to the SMT [TE+ 95] model. In the SMT mode, some processor structures are shared between threads, and others are private to each thread. This section describes the SMT model in some more detail as well as its implementation in the code.

Threads maintain separate PC counters, but share the fetch unit and I-Cache. Threads share the available bandwidth in the front end, including fetch, decode, and renaming. Recent literature describes several policies for efficient fetching of instructions among multiple threads [FC+ 03, TE+ 96]. M-Sim implements the well known ICOUNT fetch policy [TE+ 96], by default, fetching from up to 2

threads per cycle. M-Sim implements separate branch predictors per thread, which was shown to provide the best performance [RFL 03] for multithreaded processors. Shared branch prediction tables with per-thread global history buffers were shown to provide the best performance/complexity trade-off [RFL 03], providing accuracy slightly less than completely separate predictors but with less complexity. The integration of such predictors is left for future versions of M-Sim.

Each thread maintains its own rename table because it has its own set of architectural registers). After renaming, instructions from all threads are dispatched into the shared issue queue. Separate issue queues can be used for integer and floating point instructions, but the current version of M-Sim models the case where one shared issue queue is used for all instructions..

In the issue queue, instructions from all threads participate in instruction wakeup and compete for the issue bandwidth in selection. Instructions that are selected for issue continue to register file access. There are separate physical register files for integer and floating-point registers. Both register files are shared among the threads. After register file access is complete, instructions begin execution on the functional units, which are also shared among the threads.

Loads and stores access the shared data cache. In order to maintain the correct ordering of memory accesses, the load / store queue (LSQ) is used. M-Sim maintains separate LSQ's per thread, so that an unresolved address from one thread does not prevent loads in other threads from issuing.

After execution, instructions write back to the register files. Commitment (or retirement) is done in order for each thread out of the re-order buffers (ROBs). It is possible to share the ROB among the threads, but separate ROBs have several advantages. A shared ROB may complicate the recovery from a branch misprediction, since only instructions from the thread with the misprediction need to be squashed. Secondly, since the fetch policy determines the order of instructions between the threads, there is no need to commit these instructions in the same order they were fetched. The only necessity is to commit them in the correct order *within* each thread. Finally, a shared ROB would need to be sufficiently large, possibly having a large access delay. For these reasons, we have chosen to implement separate ROBs per thread in M-Sim. Furthermore, M-Sim assumes that each ROB can commit *W* instructions (where *W* is the machine width). This should not be a problem because each thread has its own rename table and ROB, so the commitment from each thread can occur in parallel.

To implement the SMT model in the simulator, we have added the file smt.h. This file defines a *thread context*, which maintains all the private structures for each thread. In addition, an array (contexts) has been added to sim-outorder.c to provide access to these contexts. The number of concurrent threads supported in

the simulator is defined by MAX_CONTEXTS, the default value of which is set to 8. If more than 8 threads are needed, this value may be increased.

4. Installation and Use

To install M-Sim, first un-tar the archive file. This should create a directory called "m-sim_v1.0". This directory contains a Makefile that should be used in building the simulator. Be sure to edit the Makefile as necessary for the host computing platform.

Issuing the make command should generate an executable called sim-outorder, which is the main executable for the simulator. Running M-Sim with no parameters will produce a list of command line arguments that the simulator accepts. A list of these arguments, along with their descriptions, is also supplied in Appendix A.

In order to support the concurrent execution of multiple programs, which may each have their own command line arguments and input/output redirection, M-Sim relies on the use of *argument files*. Argument files were also used in the implementation of SMTSIM [Tul 96]. An argument file contains only one line of the following format:

fastfwd_interval # executable arguments… < infile > outfile

The fastfwd_interval is the number of instructions to execute in the fast forward mode. This parameter is only used if the –fastfwd option on the command line is not specified. This is followed by the '#' symbol. Next, the executable (which should be in the correct binary format) is specified, followed by the (optional) list of arguments to pass in to the executable. Finally, one input redirection and one output redirection may be specified for this executable. Below are some examples of argument files:

1000000000 # ammp < ammp.in > ammp.out

40000000 # gcc integrate.i –o integrate.S > gcc.out

100000000 # gap –l ./gap/ -q –m 192M < ./ref.in > gap.out

Table 1: Sample commands for running single threaded, dual threaded, and 4 threaded simulations.

Mode	Command
Single-Threaded .	./sim-outorder gcc.arg
2-threaded SMT	./sim-outorder gcc.arg ammp.arg
4-threaded SMT	./sim-outorder gcc.arg ammp.arg wupwise.arg gap.arg

When only one argument file is specified, the simulator executes in the single-threaded (superscalar) mode. Alternatively, the multithreaded mode is invoked when more than one argument is specified to the simulator. Table 1 provides examples for running one, two, and four threaded workloads.

5. Limitations and Future Work

This section describes some of the limitations of M-Sim. This list is not intended to be comprehensive, but rather aims to provide the user with an initial understanding of the capabilities of the simulation environment. These capabilities are subject to change in future releases.

Only the Alpha AXP binaries are supported by M-Sim. While the original Simplescalar also supports the PISA binaries, M-Sim does not. This was a design choice made by the authors to allow for the simplification of some simulator code. Specifically, support for the PISA binaries requires support for longs and doubles in the simulator, which can significantly complicate sections of the code, especially with the explicit modeling of register renaming.

This version of M-Sim does not support the simultaneous execution of dependent threads. Only independent processes, with private memory spaces, are supported. Support for dependent threads is left for future versions.

The Wattch power model [BTM 00] has been incorporated to provide a first-order approximation of power within the simulator. The accuracy of the power model is not guaranteed by the authors of M-Sim, and is subject to the limitations of Wattch. For more details on Wattch, please refer to the Wattch documentation [BTM 00].

Appendix A: Simulation Parameters

M-Sim accepts many command line parameters. Most of these parameters stem from the original SimpleScalar. Below is a detailed list of all the command line parameters, with their descriptions.

Parameter	Description
-config *filename*	Load configuration from a file
-dumpconfig *filename*	Dump configuration to a file
-h <*true*|*false*>	Print help message
-v <*true*|*false*>	Verbose operation

-d <*true*|*false*> Enable debug message

-i <*true*|*false*> Start in Dlite debugger

-seed <*int*> Random number generator seed (0 for timer seed)

-q <*true*|*false*> Initialize and terminate immediately

-redir:sim *filename* Redirect simulator output to file

-redir:prog *filename* Redirect simulated program output to file

-nice <*int*> Simulator scheduling priority

-max:inst <*uint*> Maximum number of instruction's to execute

-fastfwd <*int*> Number of insts skipped before timing starts

-ptrace <*string list..*> Generate pipetrace, i.e., <fname|stdout|stderr> <range>

-fetch:ifqsize <*int*> Instruction fetch queue size (in insts)

-fetch:mplat <*int*> Extra branch mis-prediction latency

-fetch:speed <*int*> Speed of front-end of machine relative to execution core

-bpred <*string*> Branch predictor type
{nottaken|taken|perfect|bimod|2lev|comb}

-bpred:bimod <*int*> Bimodal predictor config (<table size>)

-bpred:2lev <*int list*> 2-level predictor config
(<l1size> <l2size> <hist_size> <xor>)

-bpred:comb <*int*> Combining predictor config (<meta_table_size>)

-bpred:ras <*int*> Return address stack size
(0 for no return stack)

-bpred:btb <*int list*> BTB config
(<num_sets> <associativity>)

-bpred:spec_update <*string*>
Speculative predictors update in {ID|WB}
(default non-spec)

-decode:width <*int*> Instruction decode B/W (insts/cycle)

-issue:width <*int*> Instruction issue B/W (insts/cycle)

-issue:inorder <*true*|*false*>
Run pipeline with in-order issue

-issue:wrongpath <*true*|*false*>
Issue instructions down wrong execution paths

-commit:width <*int*> Instruction commit B/W (insts/cycle)

-cache:dl1 <*string*> L1 data cache config, i.e., {<config>|none}

-cache:dl1lat <*int*> L1 data cache hit latency (in cycles)

-cache:dl2 <*string*> L2 data cache config, i.e., {<config>|none}

-cache:dl2lat <*int*> L2 data cache hit latency (in cycles)

-cache:il1 <*string*> L1 inst cache config, {<config>|dl1|dl2|none}

-cache:il1lat <*int*> L1 instruction cache hit latency (in cycles)

-cache:il2 <*string*> L2 instruction cache config{<config>|dl2|none}

-cache:il2lat <*int*> L2 instruction cache hit latency (in cycles)

-cache:flush <*true*|*false*>
Flush caches on system calls

-cache:icompress <*true*|*false*>
Convert 64-bit inst addresses to 32-bit inst equivalents

-mem:lat <*int list...*> Memory access latency
(<first_chunk> <inter_chunk>)

-mem:width <*int*> Memory access bus width (in bytes)

-tlb:itlb <*string*> Instruction TLB config {<config>|none}

-tlb:dtlb <*string*>Data TLB config, i.e., {<config>|none}

-tlb:lat <*int*> Inst/data TLB miss latency (in cycles)

-res:ialu <*int*> Total number of integer ALU's available

-res:imult <*int*> Total number of integer multiplier/dividers available

-res:memport <*int*> Total number of memory system ports available (to CPU)

-res:fpalu <*int*> Total number of floating point ALU's available

-res:fpmult <*int*> Total number of floating point multiplier/dividers available

-pcstat <*string list*> Profile stat(s) against text addr's
(mult uses ok)

-bugcompat <*true*|*false*> Operate in backward-compatible bugs mode
(for testing only)

-rob:size <*int*> Reorder buffer (ROB) size (per thread)

-iq:size <*int*> Issue queue (IQ) size (shared between threads)

-rf:size <*int*> Physical Register file (RF) size for each the INT and FP physical register file (shared between threads)

-lsq:size <*int*> Load/store queue (LSQ) size (per thread)

References

[BTM 00] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *in the Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000

[BA 97] D. Burger, T. Austin, “The SimpleScalar tool set: Version 2.0”, Technical Report, Department of Computer Science, University of Wisconsin-Madison, June 1997 and documentation for all SimpleScalar releases.

[CF+ 03] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, “Improving Memory Latency Aware Fetch Policies for SMT processors”, *in the Proceedings of the 5th International Symposium on High Performance Computing*, October 2003.

[RFL 03] M. Ramsay, C. Feucht, M. Lipasti, “Exploring Efficient SMT Branch Predictor Design”, *in the Workshop on Complexity Effective Design*, 2003.

[TE+ 95] D.M. Tullsen, S.Eggers, H. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, *in the Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[TE+ 96] D.M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, “Exploiting Choic: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”, *in the Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[Tul 96] D. M. Tullsen, “Simulation and Modeling of a Simultaneous Multithreading Processor” *in the 22nd Annual Computer Measurement Group Conference,* December 1996.