IS2202 Lab 2: *Micro-architectural exploration*

Professor: Mats Brorsson
TA: Ananya Muddukrishna, Georgios Varisteas

May 5, 2014

1 Introduction and goals

The goal of this laboratory assignment is to allow you to conduct some simple virtual experiments in the Simics simulation environment. Using the Simics Micro-Architectural Interface and an outoforder execution processor model, you will collect statistics and make some architectural recommendations based on the results.

The lab has two sections, a directed portion and an openended portion. Unless specified, all sections of this document belong to the directed portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The openended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas. Students are encouraged to discuss solutions to the lab assignments with other students. You can work alone or in a group of two. Submit one report per group which contains your solutions for the directed portion and any open-ended portion which you have explored. Both students need to be identified in the report if you have worked in a group.

Pay attention to the conciseness and presentation quality of the report, for these factors will be taken into account while grading. Problems given in the lab are usually specific about what statistics they want, so there is no need to give them all. Tables and especially graphs are much more efficient and effective ways to communicate data.

This lab assumes you have completed assignments under Lab1. We also assume that you remember all the commands used in Lab1 for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the guide for Lab1 or the Simics User Guide.

1.1 Simics MAI Overview

The Simics simulator by default assumes that every instruction completes instantaneously in a single cycle. This allows for speedy simulations that are useful for software/firmware correctness testing. As we saw in Lab1, Simics can be extended with memory hierarchy timing modules to model realistic performance effects of a user-defined memory hierarchy. These extensions increase simulation realism at the expense of simulation speed.

In this lab, we will make use of further extensions which allow an instructions execution to be delayed according to a micro-architectural model. This model can be programmed to simulate the timing behavior of the instructions as if they were being run on an in-order or out-of-order execution (OoO) processor. Micro-architectural models interact with Simics execution via the Micro-Architectural Interface (MAI).

Micro-architecturally accurate models are coded using C or the Simics Device Modeling Language. For this lab, we will use MAI modules included with Simics, rather than code our own. Specifically, we will use the MAI extension for the Sunfire UltraSPARC II processor (the Bagle machine). This extension allows for out-of-order execution, branch target speculation, and includes a memory hierarchy as well.

Several new variables are exposed to the user when working with MA models. The user can configure the width of the pipeline (the number of instructions allowed to fetch, execute, retire or commit in a single cycle), the size of the reorder buffer, whether instructions must retire in-order, and several memory hierarchy parameters related to OoO. The variables will be discussed as they are needed in the following lab sections.

When running in mirco-architecture (MA) mode, Simics tracks dependencies of several varieties (register, control and memory) that exist in the instruction stream. It then places the instructions in a structure called an *instruction tree*. For the machine we are simulating, the instruction tree tracks the same state as the reorder buffer and associated structures would in an actual processor. This tree can be examined with the command `print-instruction-queue`. A loadstore queue is also simulated.

The Simics micro-architecture simulator we will use in this lab speculates on branches by filling the instruction tree with instructions from both branch paths. Speculated instructions may only commit when their preceding branches have resolved. This behavior is notably different from the actual execution of many real out-of-order processors, but produces similar performance effects. The simulator we are using speculatively predicts branch target addresses.

The execution of instructions is divided into 6 stages (init, fetch, decode, execute, retire, commit) and instructions are only allowed to advance to the next stage when all applicable dependencies have been satisfied. In this way, instructions are allowed to execute out of order but may accumulate a multi-cycle delay appropriate to the underlying micro-architecture of the simulated processor.

In OoO execution, Simics `steps` and `cycles` are not necessarily equivalent. A step occurs whenever an instruction commits. Advancing the simulation by one cycle may mean that multiple steps occur, or that none do. Similarly, advancing the simulation by one instruction may pause the simulation in the middle of a cycle. For this reason, it is advisable to use the `step-cycle` or `run-cycles` command to advance simulation, and then simply measure the number of steps that have occurred.

See the Simics MAI User Guide included with this lab for more information about any of these topics.

2 Directed Portion

2.1 General methodology

While you must ensure you are capturing a representative portion of the programs execution, you can measure instruction execution statistics whenever you like with `ptime` or logging. For maximum efficiency, you should make sure that you are making use of all three operation modes of Simics namely `-fast` (default mode), `-stall` (memory hierarchy simulation) and `-ma` (micro-architecture simulation). You only want to run in the slow, highly detailed modes when it is necessary to do so in order to collect accurate data.

The general methodology of this lab is:

1. Start Simics in normal mode, but use an MA-extended machine.
2. Mount the host file system and load the appropriate files.
3. Checkpoint the system.
4. Restart Simics in `stall` mode and begin executing benchmark code to warm the caches.
5. Checkpoint the system.
6. Restart Simics in MA mode and collect OoO data.

During this process Simics may report errors depending on which mode you are using and whether or not you are starting from a checkpointed simulation. If your simulation still runs after an error is reported, then simply disregard it.

You can use any of the `[avril|columbiana|atlantis|subway|shell |malavita]@it.kth.se` servers to complete this lab assignment. Do not wait until the night before the assignment is due, because you will face resource contention that may significantly increase the time it takes to complete the assignment.

2.2 Setup

To begin with, set up the Simics license environment by defining the environment variable `LM_LICENSE_FILE`.

```
host$ export LM_LICENSE_FILE=27005@lic03.ug.kth.se
```

First create a workspace called `lab-2` in your home directory. If you do not have space in your home directory, create the workspace in the /nobackup/<your-login-name> directory. Remember to transfer important files such as simulation results and modified files into your home directory as /nobackup undergoes routine cleanup and is not backed up.

For this lab we will use the Simics target called Bagle which models a single UltraSPARC-II processor.

Navigate to your Simics workspace and into the subdirectory `targets/sunfire`. Copy the image files for this lab (bagle5*.craff) from the course web into this directory. Start Simics in `-fast` mode using the Bagle MA mode script.

```
host$ cd targets/sunfire
host$ Copy all bagle5*.craff files found in Simics-Bagle-disc-dumps.tar
(course web) to here
host$ ../../simics -fast bagle-ma-common.simics
```

This will boot the Bagle machine. If you see a boot error saying "`[ma_cpu0 error] cpu0 is not an MAI compatible processor, make sure Simics is started with the -ma flag.`", just pause the simulation and hit continue. Once the machine has booted, create a post-boot checkpoint.

```
simics> write-configuration bagle-ma-after_boot.conf
```

Create the working directory on the target.

```
target# mkdir -p /home/lab-2
```

Mount the host file system on the target and copy all the benchmark files found in the benchmarks directory of lab2-efiles.tar.gz into the target's /home/lab-2/ directory. Create a post-copy checkpoint now.

```
simics> write-configuration bagle-ma-efiles_copied.conf
```

2.3 Collecting IPC statistics using the MAI

In this section you will collect information on the instruction level parallelism inherent to the benchmark programs. You will do this by running the benchmarks on a simulated superscalar out-of-order processor and measuring the average number of instructions executed per cycle.

Remember to enable magic breakpoints. When you start from a checkpoint you may see an error relating to the *last_cache* component. Disregard this error.

For each benchmark: Start Simics in `-stall` mode, and run the benchmark programs. Benchmark specific invocation instructions are given in Table 1. The benchmarks in this lab are the same as those from Lab1.

Benchmark	**Execution instructions, assuming you are in /home/lab-2 on the target**
vortex	`target# cd ./255.vortex/data/test/input` `target# ./../../../vortex bendian.raw`
equake	`target# cd ./183.equake/data/test/input` `target# ../../../quake < inp.in`
parser	`target# cd ./197.parser/` `target# ./parser data/all/input/2.1.dict < data/test/input/test.in`

Table 1: Benchmark execution instructions

An example sequence of instructions for the *vortex* benchmark is given below:

```
host$ ../../simics -stall -c bagle-ma-efiles_copied.conf
simics> magic-break-enable
simics> c
target# cd ./255.vortex/data/test/input
target# ./../../../vortex bendian.raw
```

Once the benchmark is invoked, it will soon reach a magic breakpoint and the simulation will pause. Run for at least 100,000,000 instructions to warm the cache. You can check its statistics the same way we did in Lab1. By default, for this lab, instruction accesses are instantaneous and not cacheable, and only data accesses are stored in the cache:

```
simics> c 100_000_000
simics> cache_cpu0.statistics
```

Create a cache-warmed checkpoint. This will be useful to you for the remaining sections of the lab assignment.

```
simics> write-configuration bagle-ma-vortex_warmed_cache.conf
```

Create such cache-warmed checkpoints for the other benchmarks as well.

Start Simics in `-ma` mode. When you start from a cache-warmed checkpoint you may see an error relating to the *last_cache* component. Disregard this error. Set the OoO parameters to the desired values, as shown in the example sequence for the vortex benchmark below:

```
host$ ../../simics -ma -c bagle-ma-vortex_warmed_cache.conf
simics> ma_cpu0->fetches_per_cycle = 4
simics> ma_cpu0->execute_per_cycle = 4
```

```
simics> ma_cpu0->retires_per_cycle = 4
simics> ma_cpu0->commits_per_cycle = 4
simics> cpu0->reorder_buffer_size = 32
```

Run for at least 10,000,000 cycles and count the number of instructions that commit in this time frame. The MAI-enabled processor automatically reports the number of steps that have occurred every million cycles (this count is cumulative). The step count is incremented every time an instruction commits.

```
simics> run-cycles 10_000_000
```

When you have collected enough data, halt Simics and proceed to the next benchmark. For each benchmark, record the number of cumulative instructions executed in the span of cycles that you measured. What is the recorded IPC for each benchmark? Which benchmark had the best IPC, and which had the worst?

2.4 Collecting data about the effect of superscalar pipeline width on IPC

In this section, you will pick one benchmark and examine the effects of superscalar issue width on IPC for that benchmark. To do this, vary the parameters of the `ma_cpu0` object.

```
simics> ma_cpu0->fetches_per_cycle = <width>
simics> ma_cpu0->execute_per_cycle = <width>
simics> ma_cpu0->retires_per_cycle = <width>
simics> ma_cpu0->commits_per_cycle = <width>
simics> cpu0->reorder_buffer_size = 32
```

Vary all widths together though $\{1, 2, 4, 8, 16\}$, while keeping the `reorder_buffer_size` at 32. Then repeat with a `reorder_buffer_size` of 64. Are there diminishing returns on increasing pipeline width? How does reorder buffer size affect this performance? What factors might limit the effectiveness of increasing pipeline width?

2.5 Collecting data about the effect of memory latency on OoO efficiency

In this section you will investigate the effect of the memory hierarchy on OoO machine performance. To accomplish this you will vary the access time of the data cache and the access time of main memory. The cache access delay is controlled by the `penalty_read and penalty_write` parameters of the `cache_cpu0` object (default value is 1). The main memory access delay is controlled by the `stall_time` attribute of the `staller_cpu0` object.

Pick one benchmark, and record the IPC in the same fashion as the previous sections for the following memory hierarchy timings (cache read, cache write, memory):

```
{ (1, 1, 10), (2, 2, 10), (5, 5, 10), (1, 1, 20), (1, 1, 50) }
```

Use a superscalar width of 4 and a `reorder_buffer_size` of 32. The commands used to change the parameters are:

```
simics> cache_cpu0->penalty_read = <cache access delay>
simics> cache_cpu0->penalty_write = <cache access delay>
simics> staller_cpu0->stall_time = <memory access delay>
```

Record the IPC measured for each memory configuration. What impact does increased memory latency have on performance? To what degree does out-of-order execution mask the increased memory hierarchy delays?

3 Open-ended Portion

3.1 Branch predictor study

For this open-ended project, you will design your own branch predictor and test it on some realistic benchmarks. Changing the operation of branch prediction in Simics is painstaking work, but luckily for us, a completely separate framework for such an exploration already exists. It was created for a branch predictor contest called Championship Branch Prediction organized by the *MICRO* conference and the *Journal of Instruction-Level Parallelism*. The contest provided entrants with a C++ framework for implementing and testing their submissions, which is what you will use for our in-class study.

Description of the BP exploration framework, including its usage and download instructions code can be found at:

http://www.jilp.org/cpb2014/

You can compile and run this framework on essentially any machine with a decently modern version of `gcc/g++`. You can also use one of the servers listed for this lab. You will only have to modify predictor.cc and predictor.h. file to complete the assignment! Just follow the directions at the above web link.

Just like the original contest, we will allow your submissions to be in one of two categories (or both). The categories are conditional branches and indirect branches. A storage memory budget of 65K bytes applies to both categories. Follow the original competition rules. In the interest of time, you can pick 3-5 benchmarks (traces) from the many included with the framework to test iterations of your predictor design on. The contest is unfortunately over or else you could also have submitted to it! This must not however stop you from

comparing your predictor to the ones created by the winners as listed on the site.

Note: You can browse textbooks/technical literature for ideas for branch predictor designs, but please do not get code from the internet.

For the lab report: Submit well-commented source code for your predictor, an overall description of its functionality, and a summary of its performance on 3-5 of the benchmarks provided with the framework. Report which benchmarks you tested your predictor out on.

3.2 Create code that performs no better on an OoO machine

The goal of this open-ended assignment is to purposefully design code which does not perform any better when run on a superscalar out-of-order processor. Such code will demonstrate poor ILP, as shown by the measurable IPC. The goal is to have IPC of the code on the out-of-order processor be as close as possible to the IPC of the code on the in-order processor.

You should compare the code when run on a 4width OoO core (i.e. using the targets/sunfire/bagle-ma-common.simics machine) with the code when run on a single-issue in-order core (i.e. using the targets/sunfire/bagle-gcache-common.simics machine). However, make sure the parameters and configuration of the memory hierarchies are identical for both machines!

There is no line limit for the code used in this lab. Your code must run for at least one million cycles, and it does not have to terminate. Use the magic breakpoint code from the GEMM benchmark (gemm.c;lines:6-11,91,95) mentioned in Section 3.6 of Lab1 instructions document. You can also refer to the Simics User Guide (page 143) to understand how magic breakpoints are written and used.

For compiling your code, you need to use the compiler `gcc` within the target machine unless you have a Sparc/Solaris machine available. Note that if you want to include Simics headers while using the target compiler, add a `-I` flag as shown below:

```
target# gcc -I/host/pkg/simics/3.0.31/simics-3.0.31/src/include
<other_options>
```

Submit your source code, an explanation how it operates and how it restricts ILP, and the record you made of the code's IPC on the in-order and out-of-order cores.

4 Acknowledgment

A large part of this lab has been adapted with permission from the organizers of CS152 Spring 2010 course at University of California, Berkeley.