C152 Laboratory Exercise 3

Professor: Krste Asanovic
TA: Andrew Waterman
Department of Electrical Engineering & Computer Science
University of California, Berkeley

March 9, 2010

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 Microarchitectural Interface and an out–of–order 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 open–ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open–ended 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, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students can work individually or in groups of two or three. Any open-ended lab assignment completed as a group should be written up and handed in separately. Students are free to take part in different groups for different lab assignments.

You are only required to do one of the open ended assignments. These assignments are in general starting points or suggestions. Alternatively, you can propose and complete your own open ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the TA or professor.

It is also important to stress that how concise the report is and how the data is presented will be taken into account when grading. Problems usually are 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 the earlier laboratory assignments. However, we will re-include all the relevant files from past labs in this lab's distribution bundle for your convenience. Furthermore, we will assume that you remember all the commands used in earlier labs for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the first lab guide 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 previous labs, 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 instruction's execution to be delayed according to a microarchitectural 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. Microarchitectural models interact with Simics execution via the Micro-Architectural Interface.

Microarchitecturally 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 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 load–store queue is also simulated.

The Simics microarchitecture 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 microarchitecture of the simulated processor.

A point worthy of note is that `steps` and `cycles` are no longer 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.

1.2 Graded Items

You will turn a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. Problem 2.3: IPC statistics for each benchmark and answers
2. Problem 2.4: IPC statistics for all configurations and answers
3. Problem 2.5: IPC statistics for all memory configurations and answers
4. Problem 3.1/3.2/3.3 modifications and evaluations (include source code if required)

2 Directed Portion

2.1 General methodology

While you must ensure you are capturing a representative portion of the program's 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. 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 fast 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 `cory353-{2,4,6,...,24}l.eecs` instructional 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

Start Simics in `-fast` mode, using the `targets/sunfire/bagel-ma-common.simics` script. Make sure the host filesystem or workspace is mounted and that the benchmark binaries and input files are copied into the target machine. To save time in the following sections, you should create a checkpoint that has all the files loaded, and probably a checkpoint at each of the initial magic breakpoints in the benchmark programs. Unfortunately, while the target machine being simulated is the same as in some previous labs, the underlying Simics modules used in the simulation are different (they use the MAI), so you will **not** be able to use checkpoints created in past labs.

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 if you don't already have a checkpoint saved, run the benchmark program (`bzip_sparc, mcf_sparc, soplex_sparc`).
  ```
  host$ ./simics -stall -c bzip_bagel_files_loaded.conf
  simics> c
  target# ./<benchmark>_sparc input.<jpeg/in/mps>
  ```
- It will 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 Lab 2. 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 checkpoint. This will be useful to you for the remaining sections of the lab assignment.
  ```
  write-configuration bzip_bagel_warmed_caches.conf
  ```
- Start Simics in -ma mode. When you start from a checkpoint you may see an error relating to the `'last_cache'` component. Disregard this error. Set the OoO parameters to the desired values:
  ```
  host$ ./simics -ma -c bzip_bagel_warmed_caches.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 contest!

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 would be arduous, but luckily a completely separate framework for such an exploration already exists. It was created for a branch predictor contest run by the MICRO conference and the Journal of Instruction-Level Parallelism. The contest provided entrants with C++ framework for implementing and testing their submissions, which is what you will use for our in-class study. Information and code can be found at:

`http://cava.cs.utsa.edu/camino/cbp2`

A description of the available framework can be found at:
`http://cava.cs.utsa.edu/camino/cbp2/cbp2-infrastructure-v2/doc/index.html`

You can compile and run this framework on essentially any machine with a decently modern version of `gcc/g++`. So, while the TA will not be able to help you with setup problems on your personal machine, you may choose to compile and experiment there to avoid server contention. You will only have to modify one .h file to complete the assignment! Just follow the directions at the above link.

Just like the original contest, we will allow your submissions to be in one of two categories (or both). The categories are realistic predictors (the size of the data structures used by your predictor are capped) or idealistic predictors (no limits on the resources used by your predictor). Even for realistic predictors, we are only concerned about the memory used by the simulated branch predictor structures, not the memory used by the simulator itself. Follow the original contest guidelines.

In the interests of time, you can pick 3-5 benchmarks from the many included with the framework to test iterations of your predictor design on. If you want to submit to the contest, make sure you leave **at least one** benchmark from the whole set that you **do not** test the predictor on!

A final rule: you can browse textbooks/technical literature for ideas for branch predictor designs, but don't get code from the internet.

For the lab report: Submit the 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.

For the contest: We will take the code you submit with the lab, and test its performance on a set of benchmarks chosen by us. Please email your code in a tar file to the TA.

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 4–width 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. Remember to use the full Simics path on `/share/instsww/...` when compiling on the target machine.

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.

3.3 Collecting data about the limits of ILP

The goal of this open-ended assignment is to test the limits of ILP achievable for the three benchmarks included in the lab. As suggested in the directed portion of the lab, there are diminishing returns provided by increasing the width of the processor and the size of the reorder buffer. Your

job for this project is to determine what these limits are using the same procedures applied in the directed portion of the lab. Use the OoO Bagle machine with a cache access delay of 1 and memory access delay of 10. For each benchmark, make a recommendation of processor width and reorder buffer size, and provide as evidence data which demonstrate that your choice maximizes IPC while minimizing on-chip overhead.