C152 Laboratory Exercise 1

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

January 30, 2009

1 Introduction and goals

The goal of this laboratory assignment is to familiarize you with the Simics simulation environment while also allowing you to conduct some simple experiments. Using a modified instruction tracer module, you will collect instruction mix 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.

1.1 Terminology and conventions

Host machine refers to the remote server on which you are running Simics. *Target machine* refers to the simulated machine running inside of Simics. `[workspace]` refers to the directory in which you create as your Simics workspace in 2.1.

If you should type something on the host machine's command line we will note it like this:

```
host$ this is what you type
```

When on the Simics command line we will note it like this:

```
simics> this is what you type
```

And when on the target machine's command line we will note it like this:

```
target# this is what you type
```

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:

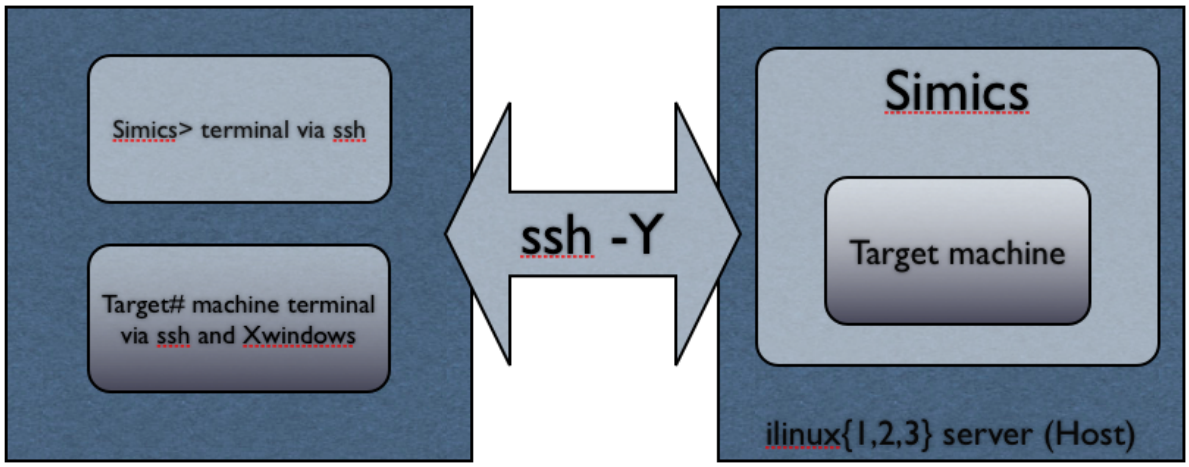

Figure 1: Lab assignment setup

1. Problem 2.3: recorded instruction mixes for each benchmark and answers
2. Problem 2.4: thought problem answers
3. Problem 2.5: design problem answers
4. Problem 3.1: source code and recorded ratio
5. Problem 3.2 and Problem 3.3 design proposals

2 Directed Portion

2.1 Setting Up Your Simics Workspace

To complete this lab you will log in to an instructional server, which is where you will run Simics. We will provide you with an instructional computing account for this purpose. However, you will also need a client machine which can display X11 windows. See `http://inst.eecs.berkeley.edu/connecting.html#xwindows` for a guide on setting this up on Windows or OSX. Make sure you use `ssh -Y` to ensure that X11 packets are forwarded to your machine by the server. Simics can be run on any of the instructional Linux servers `ilinux1,2,3.eecs` (see `http://inst.eecs.berkeley.edu/cgi-bin/clients.cgi?choice=servers` for more information about available machines).

Although Simics is installed in only one place, every student will create their own Simics workspace to contain their personal scripts and checkpoints. In order to create a workspace called `simics-workspace` in your home directory, run:

```
host$ /share/instsww/pkg/virtutech/simics-3.0.30/bin/workspace-setup ~/simics-workspace
```

2.2 Simics First Steps

Now that your workspace has been created, you are ready to begin simulation. Navigate to your workspace and load the Sun UltraSPARC III Sunfire machine (specifically the configuration called Bagle):

```
host$ cd ~/simics-workspace
host$ ./simics targets/sunfire/bagle-common.simics
```

Simics will start up and present you with a command line, and a new terminal window should also appear. The new window is the terminal of the target machine being simulated by Simics. The simulation begins paused. Type

```
simics>  continue
```

to begin the simulation and watch the machine boot up. After a short while you should be logged in to the target machine as `root`. This simulated machine is running Linux, so all the usual command line utilities should work.

Pressing `[control-C]` in the terminal with the Simics command line interface will pause the simulation and make the `simics>` prompt reappear. While the simulation is paused, you will not be able to interact with the simulated machine via its terminal. Remember to restart the simulation with `continue` before attempting to execute commands on the simulated machine.

Checkpointing

Once the machine has booted, it would be best to save the state of the simulation so that we can resume from this point the next time we start Simics, without having to sit and watch the machine boot up again. This process is called checkpointing, and it will be very useful to you at times when you wish to run from a certain single point multiple times to collect different types of data. To create a checkpoint called `after_boot.conf`, simply pause the simulation and run

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

The checkpoint is now saved. To use the checkpoint in the future, simply start Simics with the `-c` flag, like this:

```
host$ ./simics -c after_boot.conf
```

One important thing to remember is that Simics checkpoints only save the differences between the current checkpoint and the previous checkpoint. This is good because it saves lots of space. The problem is that if you create one checkpoint, keep running, create a second checkpoint, and then go and delete the first checkpoint, then the second one won't work any more. Keep this in mind when creating and deleting checkpoints.

Mounting the host filesystem

The next thing to be done is to mount the file system of the host machine on the target machine. This will allow you to access any files you have stored on the host machine inside the simulation you are running. To mount the host file system, all you have to do is type

```
target#  mount /host
```

This will mount the `/` directory of the host machine on the `/host` directory of the target machine. This command mounts the host machine's file system read only. Mounting with write permissions is possible but experimental, and will not be required for this lab.

Now you can just use `cp` to copy files from the `/host` directory tree into `/root` or `/home` or wherever you like inside the simulated machine. Remember that these files will be lost forever if you quit Simics without checkpointing (though of course you could re–add them in a new simulation if necessary).

Simulation commands

The Simics simulation can be controlled with a variety of simple commands. You have already learned how to run and suspend the simulation with `[Control-C]` and `continue`. Other useful commands include instructions to step forward a certain number of instructions or cycles (in this lab all instructions have a CPI of 1, so the two commands are equivalent):

```
simics> step-instruction 100_000
simics> step-cycle 100_000
```

Long numbers can include `_` separators for clarity. Many instructions also have abbreviated short forms, such as `si` for `step-instruction` or `c` for `continue`. `continue` can also be followed by an integer to advance the simulation by that many instructions without printing the intermediate ones.

Other useful commands print information about the current state of the simulated system. `pregs` and `pfregs` display the contents of all the machine registers or floating point registers. `read-reg` and `write-reg` can be used to view or modify the contents of specific registers, and `get` and `set` will do the same for memory locations. `ptime` will display how much simulated time has elapsed over the course of the simulation, and `pstats` will report statistics for the individual processor itself.

The command `display` can be used to cause one of the other commands to be executed whenever the simulation is suspended. For example

```
simics> display ptime
```

will cause the `ptime` command to be run every time the simulation is paused. Keep track of the display id reported when a `display` command is first executed, as this is how you will specify which one to turn off when you are ready to stop displaying a certain command:

```
simics> undisplay 1
```

Finally, adding a `!` character at the beginning of a command entered at the `simics>` prompt will cause that command to be executed by the outside shell, and not by Simics. However, it is often easiest to have multiple `ssh` sessions open, one of which is running Simics and one of which is used to execute commands on the host machine.

Other useful Simics commands can be found in the Simics User Guide distributed with this Lab.

Generic tracing

Create an instruction tracer using the command

```
simics> new-tracer
```

This will create a generic instruction tracer that displays instructions and data accesses on the Simics console as they are about to execute. It is also possible to make the trace go to a file – this is specified when the trace is started. Start and stop the trace using the commands

```
simics> trace0.start
simics> trace0.stop
```

When the tracer is active, it will record and display the instructions executed by your `continue` or `step-instruction` commands. The basic trace module provided with Simics is simplistic; you will use and improve a modified version of the module in the following lab sections.

2.3 Instruction Mix Tracing

For this section of the lab you will use a modified version of the `trace` module to track the instruction mixes of several benchmark programs provided to you. The first step is to copy the source code of the original `trace` module into your personal Simics workspace:

```
host$ cd ~/simics-workspace
host$ /share/instsww/pkg/virtutech/simics-3.0.30/bin/workspace-setup --copy-device trace
```

Copy the files `Makefile`, `gcommands.py`, `trace.c`, `trace.h`, and `mix.h` provided with this lab into the `~/simics-workspace/modules/trace` directory. This will replace the original source code with code that tracks the frequency of certain types of instructions. Compile the new tracer module with `make`.

```
ilinux3$ cd ~/simics-workspace
ilinux3$ make
```

You should also untar the file `benchmarks-1.tar` distributed with this Lab (`tar -xvf benchmarks-1.tar`) and make sure the three binary and three input files are available in your `simics-workspace` directory. The binary files for this lab are `bzip2_sparc, mcf_sparc, soplex_sparc`.

Quit and restart Simics, using either the `targets/sunfire/bagle-common.simics` script or a checkpoint you created in the previous section. Make sure the host filesystem or workspace is mounted, and copy the 3 benchmark binaries and 3 benchmark input files into the target machine.

For each benchmark:

- Create an instruction tracer. Remember that it will need a new name. Turn on magic breakpoints.

  ```
  simics> new-tracer [pick a name for the tracer]
  simics> magic-break-enable
  simics> continue
  ```

- Run one of the benchmark programs.

  ```
  target# ./bzip2_sparc -z input.jpg
  ```
  or
  ```
  target# ./mcf_sparc input.in
  ```
  or
  ```
  target# ./soplex_sparc input.mps
  ```

- It will quickly reach a breakpoint and the simulation will pause.

- Run for 100,000,000 instructions in order to get to the heart of benchmark execution.
  ```
  simics> c 100_000_000
  ```

- Turn on the instruction tracer.
  ```
  simics> [tracer name].start
  ```

- And trace for 1,000,000 instructions.
  ```
  simics> c 1_000_000
  ```

- Turn off the instruction tracer and record the reported general instruction mix.
  ```
  simics> [tracer name].stop
  ```

Note the mix of different types of instructions vary between benchmarks. Record the mix you observed for each benchmark. Which benchmark is the floating–point based scientific code? Which benchmark seems most likely to be memory bound? Which benchmark seems most likely to be dependent on branch predictor performance?

2.4 Thought problem

Suppose the Bagle machine is designed such that the average CPI of loads and stores is 2 cycles, integer arithmetic instructions take 1 cycle, and other instructions take 1.5 cycles on average. What is the overall CPI of the machine for each benchmark?

What is the relative performance for each benchmark if loads/stores are sped up to have an average CPI of 1 cycle? Is this still a worthwhile modification if it means that arithmetic instructions now take 1.5 cycles? Is it worthwhile for all benchmarks, or only some? Explain.

2.5 Design problem

Imagine that you are being asked by your employer to evaluate a potential modification to the design of a 5–stage pipeline SPARC similar to the 5–stage MIPS one that has been discussed in lecture. The proposed modification is that the Execute/Address Calculation stage and the Memory Access stage be merged into a single pipeline stage. In this combined stage, the ALU and Memory will operate in parallel. Data access instructions will use memory while leaving the ALU idle, and arithmetic instructions will use the ALU while leaving memory idle. These changes are beneficial in terms of area and power efficiency. Think to yourself why this is the case, and if you are still unsure, ask about it in Section or OH.

In standard SPARCv9, the effective address of a load or store is calculated by either summing the contents of the two registers specified by the instruction, or by summing the contents of one register with an immediate value. The problem with the new design is that there is is now no way to perform any address calculation in the middle of a load or store instruction, since loads and stores do not get to access the ALU. Proponents of the new design advocate changing the ISA to allow only one addressing mode: register direct addressing. Only one source register is used, and the value it contains is the memory address to be accessed. No offset can be specified.

With the new design, any load or store instruction making use of double register address calculation, or which add a non–zero offset to a single register's contents will take two instructions. First the register or immediate values must be summed with an add instruction, and then this calculated address can be loaded from or stored to in the next instruction. Load and store instructions which currently use an offset of zero will not require extra instructions on the new design.

Your job is to determine the percentage increase in the total number of instructions that would have to be executed under the new design. This will require a more detailed analysis of the different types of loads and stores executed by our benchmark codes.

Modify the tracing module, specifically the file `mix.h` to detect the percentage of instructions that are register–register loads and stores, the percentage that are register–immediate loads and stores with non–zero offsets, and the percentage that are register–immediate loads with zero offsets.

Follow the directions in the `mix.h` file to accomplish this task. Use the provided excerpts from the SPARCv9 specification to determine which bits of the instruction correspond to which fields. After modifying `mix.h`, remember to recompile the module with `make` in your `simics-workspace` directory.

What percentages of the instruction mix do the various types of load and store instructions make up? Evaluate the new design in terms of the percentage increase in the number of instructions that will have to be executed. Which design would you advise your employer to adopt? (Justify your position.)

3 Open-ended Portion

3.1 Mix Manufacturing

The goal of this section is to investigate how effectively (or ineffectively) the compiler will handle complicated C code created by you.

Using no more than 15 lines of C code, attempt to produce SPARC assembly code with the maximum ratio of branch to non–branch instructions. In other words, try to produce as many branch instructions as possible. Your C code can contain as many poor coding practices as you like, but limit yourself to one statement per line and do not cheat by calling functions or executing any code not contained within the 15 line block. Your code must terminate. You can use code that creates jumps, but jump instructions do not count; only conditional branches count.

Copy the file `mix_manufacturing.c` into your home directory. Modify it to add your custom code. To test for correctness you can just compile on the host machine:

```
host$ gcc -I/share/instsww/pkg/virtutech/simics-3.0.30/src/include
mix_manufacturing.c -o mix
host$ ./mix
```

However, in order to run and trace the code in Simics to determine how successful you were, you will have to compile the code on the simulated machine. Start up a Bagle simulation, mount the host machine, and copy in the source file. Then, all you will need to do is run the slightly different command (note `/host`):

```
target# gcc -I/host/share/instsww/pkg/virtutech/simics-3.0.30/src/include
mix_manufacturing.c -o mix
```

to create a binary that can run inside the simulation. Create a new tracer, and use it to report the ratio of branches to non–branches that you achieved in your code. You will submit this mix report, and your lines of C code.

3.2 Processor Design

Propose a processor modification of your own to a 5–stage pipeline like the one presented in section 2.5. Justify your design modification's overhead, cost, or motivation by explaining which instructions are affected by the changes you propose and in what way. You may have to draw a diagram explaining your proposed changes for clarity, and you may have to modify the trace module (but probably only `mix.h`) to track specific types of instructions not previously traced. A further tactic might be to show that while some instructions are impacted negatively, these instructions are not a significant portion of certain benchmarks. Feel free to be creative.

3.3 ISA Design

Imagine you are modifying the SPARCv9 ISA to allow for two sizes of instructions. With this modification, you will be able reduce the total code size of programs compiled for your machine

by allowing some instructions to be encoded in a more compact form. Pick an instruction format (they are listed in the SPARCv9 handout) and make a proposal for why it should be the one which is allowed to be reduced in size. What is your proposal for reducing the size of this format, and how long is this smaller encoding? By how much does including the variable length instruction reduce code size for the benchmarks we looked at earlier? You can use the full SPARCv9 manual (developers.sun.com/solaris/articles/sparcv9.html) if you want to find additional instruction types which use your proposed format. You might also get some ideas by looking at the variable length branch instruction problem included in the Section 2 notes.