CMSC 22200 Computer Architecture
The University of Chicago Autumn 2015
Lab Assignment 1
Due: Thursday, October 22 at the beginning of class by hardcopy

In this lab assignment, we explore how to execute and profile (collect performance data on) programs within the Marss simulator. Then, we analyze the collected data to understand the mix of instructions in each of the programs.

Marss is a full-system simulator based on the Intel X86 Architecture. You can visit http://www.marss86.org/~marss86/index.php/Home to learn more about Marss.

Please run Marss on a Linux machine in CSIL. To get it set up:

1. Download the file "marssx86.bash" that is included with this assignment to your home directory on the CSIL machine.
2. Place the file in a directory you want to use for your installation.
3. Run the command "bash marssx86.bash". This will retrieve the software, compile it, and copy files that contain an installation of Linux and some benchmarks (to run within the simulator) to your machine. Note that while the Marss installation will be in your home directory, and thus available on any CSIL machine, the other files will be on the machine's hard drive (in "/local/your_cnet_id/"). Thus, you must repeat this step if you switch to a different CSIL machine.

Part 1. Running Marss

Change to the directory that has been created for you and enter the following command to run Marss:

```
$ ./qemu/qemu-system-x86_64 -m 512 -hda /local/your_cnet_id/marssx86/
disk-images/ubuntu-natty-lab1.qcow2
```

The "-m" option is to set the memory size of the simulated machine. Here we set it to 512 MB.

A Qemu window will open and show the console of the emulated Linux machine as it boots. To login, enter “root” for both the username and password.

Q1. Provide a screenshot of the command you ran to start QEmu and the Qemu window. We should be able to see your CNetID as part of the prompt from the CSIL machine; please position windows so it is not

obstructed.

Part 2. Profiling Simple Programs

Computer architects profile programs to measure and analyze important characteristics of how they run. Particularly when a program is complex, profiling allows these characteristics to be determined much faster and more simply than by inspecting the assembly code. Architects are interested in knowing about performance characteristics such as instruction count, cycle count, number of cache hits and misses, mix of different instruction types, among others.

Marss is capable of reporting a vast number of different performance characteristics including those we mentioned above. (When you see the output from Marss, you will better appreciate just how much data it collects. Virtually all of it is not relevant to this particular assignment, and much of it is beyond the scope of this course. So, you'll need to pick through the output to extract only the information we need today.)

For this part, we provided two simple programs to profile and analyze in terms of performance.

I. Bubble Sort
a. You can find the bubble_sort.c and bubble_sort.o files in the main directory of the Qemu image.
b. Run Marss based on part 1. In order to simulate a program on Marss, first you need to set the configuration. Use Ctrl-alt-2 to switch into the Qemu configuration console. Type the following command:

```
simconfig -stats bubble_sort.st -logfile bubble_sort.log -machine atom_core
```

The "-stats" option specifies the statistics output file name; "-logfile" specifies the log file name; and "-machine" specifies the processor model that you want to run on. To learn more about "atom_core," see: http://en.wikipedia.org/wiki/Intel_Atom
c. Use control-alt-1 to switch back to the main Qemu window. Run the following command line:

```
./start_sim; ./bubble_sort.o; ./stop_sim
```

The "start_sim" and "stop_sim" programs notify Marss to start or stop simulation profiling.
d. Run bubble sorting for input sizes of 100 and 1000 elements. (When you run the program, it will ask what size you want.) You can either go back to the QEmu console to assign a different statistics file name for the second (larger) run, or just perform two runs and later separate them out from within the single file.

Now, on your real CSIL machine, check the statistics file, which has been saved within the marss directory: bubble_sort.st. It has three (long) sections, separated by lines with a lot of dashes. The three sections represent statistics for kernel mode (operations performed by the Linux OS); user mode (operations performed by the bubble sort program); and the combination of the two. In this lab we are only interested in user mode statistics, since they reflect the program we run. (If you saved both runs of the bubble sort into the same file, it actually has six sections: the first three are for the first run.)

Note that this file contains a lot of information. Part of the challenge of this lab is to determine what information to pull out to answer specific questions.

Q2.
a. How many instructions were executed for each input size?
How many cycles were taken for each input size?
(There are many instruction counts in the statistics file. Use the one that is under the heading of "issue:". Note that there are sections tabbed over hierarchically under "issue:" before the tabbing gets back to the level immediately under "issue:". Skip over these and find the instruction count that is just one tab level to the right of the tab level of "issue:". And watch out for abbreviations....)
b. Given the complexity of bubble sort (O(n^2)), input size 1000 should have 100 times more executed instructions than input size 100. Look at the C code and explain why this doesn't happen. (The answer may be very counterintuitive to you. Hint: think of a third experiment you can run to confirm your hypothesis; if your hypothesis is correct, you will have enough data across the three runs to account for the discrepancy very closely....)
c. Look at the "opclass" section for user mode in the statistics file.
i. Which operation types are related to load and store?
ii. Which operation types are related to control flow?
iii. Which operation types are related to procedure calls?
Plot a pie chart showing the percentage of operations for load and store vs. other operations.
Plot a pie chart showing the percentage of operations for control flow vs. other operations.
Plot a pie chart showing the percentage of operation for procedure calls vs. other operations.

II. Fibonacci
Repeat the above steps for the Fibonacci program and input sizes 10,000 and 100,000.
Q3.
a. How many instructions were executed for each input size?
How many cycles were taken for each input size?
b. Given the complexity of Fibonacci is O(n) (for this "memoization" version that only calculates the value of a given input once and

remembers it for other calls to the same function with the same input), input size 100,0000 should have 10 times more executed instructions than input size 10,000. Look at the C code and explain why this doesn't happen.
c. Plot a pie chart showing percentage of operations for load and store vs. other operations.
Plot a pie chart showing percentage of operations for control flow vs. other operations.
Plot a pie chart showing percentage of operations for procedure calls vs. other operations.

Part 3. Profiling Real Benchmarks

Now we will run more sophisticated benchmarks.
The system we have provided also contains a directory: parsec-3.0, which is a copy of the parsec benchmark suite. Parsec has various types of benchmarks. In this lab we want to run blackscholes, streamcluster, and swaptions.

Since these benchmarks have long runtimes, run them for 200 million instructions. You can specify an instruction limit for simulations as below:

simconfig -stopinsns 200m -stats name.st -logfile name.log -machine atom_core

To run a Parsec benchmark, enter the following command:

$ ./start_sim; ./parsec-3.0/bin/parsecmgmt -a run -p BenchmarkName; ./stop_sim

Q4.
a. How many instructions were executed for each benchmark?
How many cycles were taken for each program?
c. Plot a pie chart showing percentage of operations for load and store vs. other types of operations.
Plot a pie chart showing percentage of operations for control flow vs. other
operations.
Plot a pie chart showing percentage of operations for procedure calls vs.
other operations.
d. How different are these benchmarks in terms of operation types? Explain
what you infer from it.