

CS 6290: High-Performance Computer Architecture

Fall 2007

Homework 3

Report Due: October 25th before class

This homework is intended to help you set up the simulator for the project and to help you understand the performance of multiple-issue out-of-order processors.

Part 1 [5 points]: Setting up the simulator

This part carries only 5 points, but it is needed in order to complete Part 2. First, create a directory for the simulator and all the other stuff you will need. We will use a “sim” directory within our home directory, but you can use any directory name you want.

```
mkdir ~/sim
cd ~/sim
```

Now download the simulator source code from the class web page, put the downloaded file in this sim directory, and then unpack it:

```
< put sesc-20071002.tar.bz2 in the sim directory>
tar xvjf sesc-20071002.tar.bz2
```

Now you should have a “sesc” directory within your sim directory. To build a simulator, it is recommended that you create a separate build directory (e.g. “runsim”). We will create our runsim directory in our sim directory:

```
mkdir runsim
cd runsim
```

Now that we have a build directory, we need to configure it for building sesc. To do this, use the configure script provided with SESC:

```
../sesc/configure --enable-smp --enable-mipsemul
```

The first option enables multiprocessor support in the simulator (this will be needed for the project). The second option enables MIPS emulation support that you will need to run the applications used in this homework. Another useful configure option is “--enable-debug”, which turns on debugging checks within the simulator. This can be useful for the project when you are debugging your changes to the simulator. However, this debugging support slows the simulator down considerably and should only be used when debugging.

After configuring the simulator, we are ready to build it using the Makefile created by the configure script:

```
make
```

This may take a while, but when it is complete you should have the simulator executable “sesc.smp” in your build directory. Before we can run the simulator, we need to tell it what kind of computer to simulate. This is specified in the configuration file. SESC already comes with a variety of configuration files, and to begin with we can simply use the default configuration for the multiprocessor simulator we built. Make this default configuration:

```
make sesc.conf
```

Now we need some application to run inside the simulator. As mentioned above, the simulator uses the MIPS ISA, so we need an application compiled for a MIPS-based Linux machine. Since these are hard to find, we will download a pre-compiled application from the class web-site. We will also need some input files in order for this application to work. To avoid clutter in our simulator’s build directory, let’s create another directory for the application and put the application executable there:

```
cd ..  
mkdir crafty  
cd crafty  
<put crafty.mipseb and test.in in this directory>
```

Now we are ready to test our simulator. Use the following command line:

```
../runsim/sesc.smp -itest.in -c../runsim/sesc.conf crafty.mipseb
```

This runs the simulator and tells it to simulate execution of crafty.mipseb on the simulated machine, which is specified in the configuration file sesc.conf. **This simulation will take some time!** The simulator is not only emulating the instructions in a different ISA, it is also modeling the behavior of the simulated processor cycle by cycle.

In general, the command line for running the simulator has four parts. First, the name of the simulator. We are running sesc.smp we have build in our runsim directory. Second, we specify simulator options. Here, we are using the “-I” option, which tells the simulator to use the “test.in” as the standard input for the simulated application. Other related options are “-o” which puts the simulated application’s standard output to a file and “-e” which puts the application’s standard error to a file.

The third part of the simulator’s command line (after simulator options) is the executable to simulate. Here we are using the “crafty.mipseb” application. After this, we can list any command line options for the simulated application itself. Since our crafty.mipseb application takes no command line parameters, we have specified no such arguments in its command line.

Once the simulation is done, there will be a report file in the application’s directory. The report file has a name that begins with “sesc_crafty.mipseb.” and then has a unique identifier. This identifier makes the report file name unique, so you can do several simulations and get different reports for them. Let’s say our report file name is “sesc_crafty.mipseb.V7BFX9”. This report file contains the command line we used, a copy of the configuration we used, and then lots of information about the simulated execution we have just completed. You can read the report file yourself, but since the processor we are simulating is very sophisticated, there are many statistics about its

various components in the report file. To get the most relevant statistics out of the report file, you can use the report script that comes with SESC:

```
../sesc/scripts/report.pl -last
```

The “-last” option tells the script to use the latest (by file time) report in the current directory. Instead of “-last”, you can specify the name of the report file you want to use, or use “-a” which makes the script process all reports in the current directory.

The printout of the report script tells us how fast the simulation was going (Exe Speed), how much real time elapsed (Exe Time) for the simulation, and how much time on the simulated machine we simulated (Sim Time). Note that we have simulated an execution that would take only about one tenth of a second on the simulated machine, and it took many seconds to do this.

The printout of the report script also tells us how accurate the simulated processor’s branch predictor was overall, and how its components did (RAS, direction predictor, and BTB). Next, it tells us how many instructions were executed and the breakdown of these instructions. Finally, it tells us the overall IPC achieved by the simulated processor on this application.

To complete Part 1 of the homework, answer the following:

- A) What is the Exe Speed for this simulation?
- B) What is the Exe Time for this simulation?
- C) What is the Sim Time for this simulation?
- D) What is the accuracy of the branch direction predictor (BPred) in this simulation?
- E) What is the IPC achieved by the processor in this simulation?

Part 2 [5 points]: Effect of branch prediction

Now we can change some parameters of the processor to see how that affects its performance. First, we will change the branch predictor. The branch predictor for the processor is specified in section [BPredIssueX] of the configuration file (./runsim/sesc.conf). This is a hybrid predictor with a 16K-entry meta-predictor, where each entry of the meta-predictor is a 2-bit counter. The meta-predictor chooses between 1) a global predictor with 11-bit history and 16K 1-bit entries, and 2) a “normal” predictor with 16K 2-bit counter entries.

Change this configuration (save the original!) to model a simple table of 16K 2-bit predictors. To do this, in the [BPredIssueX] section change the type from “hybrid” to “2bit”, and add the following fields:

```
size=16*1024  
bits=2
```

Now re-run the simulation.

- A) What is the new IPC?
- B) What is the accuracy of the direction predictor?

Now let's use a static predictor that always predicts that a branch is not taken. To use this predictor, change the predictor type to "NotTaken" and re-run the simulator.

C) What is the new IPC?

D) What is the accuracy of the direction predictor?

Part 3 [10 points]: Effect of ROB size

Before we go on to changing other processor parameters, restore the `sesc.config` file to use the default hybrid predictor. Now we will change some of the parameters that describe the processor core. These parameters are located in the `[issueX]` section. Find the "robSize" parameter. Note how it is specified as a function of the processor's issue width (which is 4 by default, so robSize is 176). Also note that SESC specifies the number of physical registers available for renaming separately from ROB size (see `intRegs` and `fpRegs`), it models separate load and store queues (`maxLoads` and `maxStores`), and it also specifies the maximum number of fetched-but-uncommitted branches. First, change only the ROB size to 32 and re-run the simulation.

A) What is the new IPC?

Now (with ROB size 32) change `maxLoads` to 32 (the default for a 4-wide processor is 56). Re-run the simulation.

B) What is the new IPC?

C) Why did the change in load queue size have this effect?

Now let's have 512 in each of `robSize`, `maxLoads`, `maxStores`, and `maxBranches`, but leave `intRegs` and `fpRegs` at their default values. Re-run the simulation.

D) What is the new IPC?

E) Compare the new IPC to the IPC in the default configuration. We have many more resources, but the IPC is not that much better. Why?

Finally, let's have 256 for each of `robSize`, `maxLoads`, `maxStores`, `intRegs` and `fpRegs`. Re-run the simulation.

F) What is the IPC now?

G) This configuration is less expensive than the one in Part 3D. Why does this new configuration perform better?