

EN292-S10: Advanced Computer Architecture
Homework 2 — Simulating a Superscalar Processor
Due Tuesday, October 17, 2006 by 5:00pm

This homework problem consists of running some benchmark applications on an out-of-order issue simulator called SIMPLESCALAR. The idea of this assignment is to understand how different branch predictor schemes can influence processor performance. In addition, I've included a short exercise to help you get acquainted with how SIMPLESCALAR is organized and how it may be modified.

Part I: Setting Up Your Directory and Script Files

Since this software only runs on the LEMS cluster (for now), you'll need a LEMS account to do this assignment. Please see Arpie if you don't have one.

Everything you need to run the simulations can be found in the following directory on the LEMS cluster:

```
/lems/cadpro/simplescalar
```

- You will need to do the following to set up your simulations:

1. In your home directory, create the directory path:

```
/HOMEDIR/simplescalar/RUNS
```

Where HOMEDIR is your home directory path (e.g. /home/iris)

2. Copy the following files to /HOMEDIR/simplescalar/RUNS:

```
/lems/cadpro/simplescalar/scripts/baseline.cfg  
/lems/cadpro/simplescalar/scripts/template-2000.ss
```

3. Create your own directory for the simulator code, creating links to most of the files. From your simplescalar directory, /HOMEDIR/simplescalar type:

```
mkdir simplesim-3.0  
ln -s /lems/cadpro/simplescalar/simplesim-3.0/* simplesim-3.0/.  
rm simplesim-3.0/sim-outorder.*  
cp /lems/cadpro/simplescalar/simplesim-3.0/sim-outorder.* simplesim-3.0/.
```

4. Compile your own version of *sim-outorder*:

```
make sim-outorder
```

5. The file `baseline.cfg` contains all the default configuration values for the processor.

For instance a bimodal (i.e., 2-bit) branch predictor is defined as follows:

```
-bpred:bimod 4096
```

where 4096 is the number of entries in the pattern history table. These values can be overwritten at the command line or via a separate script file. More on this later.

Part II: Modifying the Source Code

In this part, I want you add a new counter to *sim-outorder.c* to keep track of executed instructions. In addition, include new instrumentation code to keep track of the average and maximum number of consumers of each integer or floating point result produced by the processor. A “consumer” is any other instruction that uses the result as one or more of its operands, i.e. it is the sink of a true data dependency.

Testing Out Your Simulator

Running real benchmarks on your simulator is necessary to get an idea of the real impact of your implementation on the performance of the program. However, these benchmarks can take a long time to run and if all you want to do it test out your code to make sure it didn’t break and/or it’s giving you proper output, this can make debugging take much longer than necessary. Instead, you can run some simple test programs whose statistical output is not meaningful, but will allow you to debug more easily. These test programs can be found in the directory `/lems/cadpro/simplescalar/simplesim-3.0/tests/bin.big`. In particular, `test-fmath` is a simple fast program that will usually do the trick. To run it, from your `simplescalar` directory simply type, all on the same line:

```
sim-outorder -config ../RUNS/baseline.cfg
/lems/cadpro/simplescalar/simplesim-3.0/tests/bin.big/test-fmath
```

Many lines of data are output to the screen. Some of the lines that may be of particular interest to you are:

```
sim_num_insn          12345678 # total number of instructions committed
sim_total_insn        12398765 # total number of instructions decoded
sim_cycle             123454569 # total simulation time in cycles
sim_IPC                1.2356 # instructions per cycle
bpred_bimod.bpred_addr_rate 0.8825 # branch address-prediction rate
bpred_bimod.bpred_dir_rate  0.9115 # branch direction-prediction rate
```

Two statistics in particular, *sim_num_insn* and *sim_total_insn* report the number of instructions committed and decoded, respectively. In addition to these, I would like you to add another counter called *sim_exec_insn* that keeps track of the number of instructions actually executed.

1. Follow the constructs in the file *sim-outorder.c* for adding new counters. In particular, see the use of routine *stat_reg_counter* for initializing and declaring the counters *sim_total_insn* or *sim_num_insn*. You also have to figure out where your new counter should be incremented (i.e., somewhere when the code actually sends off an instruction to be executed (or issued in SIMPLESCALAR terminology)).
2. Recompile *sim-outorder*, and run it with *test-fmath* to make sure you are getting the expected output.
3. Comment on the value of *sim_exec_insn* relative to *sim_num_insn* and *sim_total_insn*.
4. Comment on the values *ave_num_consumers* and *max_num_consumers*.

Part III: Simulations using Different Branch Predictors

To get more meaningful statistics from our simulator, we need to run simulations on much bigger benchmarks than *test-fmath*. You will be using sample benchmarks taken from the SPEC2000 benchmark suite to do this. But first a few more words on your setup:

- As mentioned earlier, the file `baseline.cfg` contains all the default configuration values for the processor. You will need to overwrite some of these values, but you will do this via `template-2000.ss` rather than modifying `baseline.cfg`

The file `template-2000.ss` is a script file you will use for running your experiments. Edit it as follows:

- Change the `SIMDIR` variable on line 8 to point to the directory you created to build your own version of `simplescalar` (e.g., `/HOMEDIR/simplescalar/simplesim-3.0`).
 - Change the `RES` and `CONFIG` variables on lines 28 and 33 respectively to point to your own area (e.g. `/HOMEDIR/simplescalar/RUNS`).
 - Change the `TESTDIR` variable on line 37 to point to the directory within your `RUNS` directory where this round of results will reside. That is, for every set of experiments you run, you will need to edit this line to point to a different directory. Make sure you also create this directory before you run your experiments.
 - Create the directory `.scripts` in your home directory (e.g., `/home/iris/.scripts`) and change line 43 of `template-2000.ss` so `SCRIPTDIR` points to this directory. This directory will hold input and output information for running your scripts. Each experiment will have a unique input and output file created. In general, you will not need to access these files, but they may come in handy if you had an error running your experiments.
 - You can change the configuration of any parameterized component in the simulator. For instance, in lines 51–64 of the script file I’m setting up a different test whereby adding line 56 and 57, I’ve changed the branch predictor to use the 2 level **GA**g scheme with a history size of 8 (and 2^8 entries). Lines 51–64 can be copied and modified as necessary. Notice that there are added flags for fast forwarding the simulation and limiting the number of instructions simulated (`-fastfwd` and `-max:inst`). These lines should not be modified.
- The script file `template-2000.ss` is set up to run experiments on two benchmark circuits, *parser* and *wupwise*. You will need to modify line 4 to include only those benchmarks which you were assigned (see below). Each simulation can take anywhere from 20 minutes to 2 hours to run.

Running Your Experiments

- By executing the file `template-2000.ss` your experiments will automatically be queued to run on one of the ULTRASparc computers in the LEMS lab. We have 10 machines each with 2–4 processors making it possible to run 22 jobs simultaneously on the cluster. The job

distribution program submits jobs for execution according to when they were submitted to the queue (not according to any “fair” distribution among users).

- Once the simulations complete, results will be placed in the directory specified in `template-2000.ss`. For instance, if my result directory is specified as

```
-directory ${RES}${TESTDIR}/test1,
```

all results will appear in `/home/iris/simplescalar/RUNS/latest/test1/Out/`. The results are in compressed format; you can read them using `gunzip -c` or uncompress them first. Note that if you try to run a new batch of tests using the previously generated result directory, you will get an error message such as

```
The Ran/parser file exists, not running it again.
```

This is to protect you from accidentally overwriting your old results. If you really want to do this, delete everything in the `TESTDIR/Ran` directory first.

Setting Up Your Experiments

- The first set of studies evaluates the effect of the branch hit table on the branch prediction accuracy. Next we are going to compare different predictor schemes to see what effect they have on program performance. In general, you should leave all unspecified options as the default value, *i.e.*, *only change the parameter in question*.
 - a) Start with a bimodal branch predictor (this is the default case). Vary the size of the predictor table, using values of 512, 1024, 2048, and 4096. Note the effect on branch prediction accuracy and program performance. When does the prediction accuracy level off?
 - b) Compare the results you get in part (a), with those using a perfect predictor (`-bpred perfect`).
 - c) Using a table size of 1024, try a 2-level branch prediction of your choosing (describe what you’re using and why in your writeup).
 - d) Using a bimodal branch predictor with table size 1024, change the DL1 cache size from 8KB to 64KB (doubling each time) to see the effect of cache size on performance.
 - e) Repeat part (d), but this time set the DL1 to 32KB and change the latency from 1, 3, 5 cycles.
- If you have any problems setting up your experiments, a good reference is *The SimpleScalar Tool Set*, by Burger and Austin. You can access this paper from:
`/lems/cadpro/simplescalar/simplescalar.pdf`
- As stated earlier, since each simulation can take a long time to run, and you’ll need to make several runs, I’m only having you run four benchmarks (two integer, and two floating point) in order to reduce the total simulation time.

| Name | Benchmarks |
|---------|------------------------------|
| Yiwen | bzip2, gcc, mgrid, wupwise |
| Elif | gzip, parser, swim, equake |
| Hua | vpr, gcc, mesa, applu |
| Chia-En | twolf, parser, mesa, ammp |
| Greg | vortex, vpr, equake, wupwise |
| Nuno | twolf, gcc, art, apsi |
| Cesare | vortex, gzip, apsi, mgrid |
| Brendan | bzip2, vpr, applu, swim |

Evaluating Your Results

- This evaluation should be fairly elaborate. That is, I'm not just interested in the numbers you get in your output files (though you should report these numbers in your writeup). More specifically, I would like a detailed analysis of your results, explaining why your results came out the way they did, how they compare with experimental results discussed in class and why there may be discrepancies. Don't forget to include some mention of instruction counts *sim_num_insn*, *sim_total_insn*, *sim_exec_insn*, *ave_num_consumers*, and *max_num_consumers*

Include a graph or table of your results, comparing values to those obtained to a base case (e.g., under bi-modal with 1024 table, and 8KB DL1 with 1 cycle latency).

Since you are only running four benchmarks in your experiments, it may not be fair to generalize your results over a broad range of applications. Nonetheless, since it not reasonable to everyone to run a whole suite of benchmarks, you can take some liberties here. Include in your report an explanation of the kinds of implications your results may have on the hardware implementation of the processor. What kind of recommendations can you make for a superscalar processor design based on these results? Would it be different for floating point or integer benchmarks? Include a printout of your `template-2000.ss` file in your writeup.