CS5513 Computer Architecture

Individual Project 2: Simple Scalar Simulator

Due: Nov 13th, 2020, Friday, 11:59pm

1 Objectives

In this project, **you are to experiment with difference configurations of a CPU simulator to study the performance impact of these configurations**. The goal of this project is to help you understand the impact of various CPU design choices and learn the process of modern computer architecture design and research.

2 General Project Descriptions

2.1 Individual Project

This project is an individual project. Each student is expected to conduct her/his own experiments and submit her/his own report.

2.2 The SimpleScalar CPU Simulator

As we have discussed in the class, architecture simulators are critical software tools used by computer architects and researchers to evaluate the potential of their new ideas. Almost all new architecture designs have to be validated in some simulator(s) first.

In this project, we will the use the famous SimpleScalar CPU simulator. SimpleScalar is a cycle-accurate microarchitectural simulator. It was written by Dr. Todd Austin and Dr. Doug Burger when they were graduate students at University of Wisconsin. SimpleScalar implements the internal components of a modern CPU with Alpha ISA and PISA ISA.[1] SimpleScalar allows its users to tune the design parameters of the simulated CPU components to study the impact of these parameters. Additionally, as an opensource software, users can modify SimpleScalar to add their new components or new CPU management algorithms. For this project, we will focus on the branch predictors and the caches.

However, a simulator cannot be used by itself – it always have to be executed with benchmarks. Or, more accurately speaking, a simulator is always instructed by its user to execute a benchmark. Therefore, for a simulator, a corresponding set of benchmarks are also published to accompany the simulator. A standard set of benchmarks also make it easier to reproduce a research result.

2.3 Experiment Environment

SimpleScalar simulator executes in Linux or Unix-like environments. If you choose to conduct the experiments on the CS department's fox servers, there is no extra actions to take. If you choose to conduct the experiments on your own computers, please ensure that you have the compiler tool chain (GCC and Make at least) to build SimpleScalar.

[1] Alpha was a legacy ISA used in DEC/HP servers, and PISA was a teaching ISA. Many CPU simulators implement Alpha ISA due to its simplicity.

The addresses for the fox servers are $fox01.cs.utsa.edu$, ..., $fox06.s.utsa.edu$. You can ssh to any one of the six servers. Your user name should be your *abcid*, and your default password should be your banner id with the "@".

2.4 Obtaining SimpleScalar

SimpleScalar can be obtained with the following steps:

- For those who want to conduct the experiments on fox servers, you can copy the compiled SimpleScalar, benchmarks, and the configuration file with this command on any one of the fox servers.

```
Architect $> cp -r /home/wwang/share/Comp_Arch/SimpleScalar/ .
```

- For those who want to compile their own SimpleScalar, you can download it from the SimpleScalar's official site: go to the website, choose "Tools" and download *simplesim-3v0e.tgz*. The downloaded file can be uncompressed, and the SimpleScalar can be built, with the following commands,

```
Architect $> tar -xvzf simplesim-3v0e.tgz
Architect $> cd simplesim-3.0
Architect $> make config-alpha
Architect $> make sim-outorder
```

  In addition to the simulator, you also need to download the benchmarks for this project. The benchmarks can be download at this location. This *tar.gz* can be download and uncompressed with commands,

```
Architect $> wget http://www.eecs.umich.edu/~taustin/eecs573_public/\
             instruct-progs.tar.gz
Architect $> tar -xvzf instruct-progs.tar.gz
```

  At last, please download the CPU configuration file with this link. This configuration file is based on the Intel Nehalem processor.[2] A simple $wget$ can download this configuration file,

```
Architect $> wget https://wwang.github.io/teaching/Spring2020/CS3853/\
             /extra/nehalem.cfg
```

With the SimpleScalar built and the benchmarks downloaded, you should be able to test it out with this command (do not forget to cd into your SimpleScalar folder if you copied it),

```
Architect $> ./simplesim-3.0/sim-outorder -config nehalem.cfg \
             ./benchmarks/perl.alpha ./benchmarks/perl-tests.pl
```

[2]This configuration was designed by Dr. Daniel A. Jiménez from TAMU. Dr. Jiménez was a UTSA graduate and professor.

Note that, the simulator may take some time to execute (about 20 seconds on fox servers). If the simulator executes correctly, you should see an output of the statistics of the execution. The last few lines of the execution should be,

```
-tlb:itlb    itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb    dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat                   30 # inst/data TLB miss latency (in cycles)
-res:ialu                   8 # total number of integer ALU's available
-res:imult                  4 # total number of integer multiplier/dividers available
-res:memport                2 # total number of memory system ports available (to CPU)
-res:fpalu                  4 # total number of floating point ALU's available
-res:fpmult                 4 # total number of floating point multiplier/dividers avail
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat              false # operate in backward-compatible bugs mode (for testing on
```

2.5 Usage of SimpleScalar

2.5.1 Commandline Usage

SimpleScalar can be called with the following syntax,

```
Architect $> PATH/sim-outorder -config nehalem.cfg options benchmark input
```

The *PATH* represents the path to your SimpleScalar directory (e.g., the "./simplesim-3.0" in the above test run). The *benchmark* represents the benchmark to be executed in the simulator (e.g., the "./benchmarks/perl.alpha" in the above test run). The *input* represents the input file used by the benchmark (e.g., the "./benchmarks/perl-tests.pl" in the above test run). The *options* allows us to alter the configurations of the CPU. I will give you specific options to use later. Alternatively, you can also modify the *nehalem.cfg* file to change CPU configurations.

2.5.2 Benchmarks

There are five benchmarks (along with their inputs) in the *benchmarks* directory or from the download *instruct-progs.tar.gz* file. These benchmarks and their inputs are listed in Table 1. Note that, since we only build an ALPHA ISA SimpleScalar, we will only use ALPHA benchmarks. Some of these benchmarks may take tens of seconds to a few minutes to execute (recall that the major downside of simulators is their slow speed). You can also download additional benchmarks from here.[3] These additional benchmarks do not have input files.

[3]Again, thanks to Dr. Jiménez.

	Benchmark	Input file	Example Commands and Notes
1	anagram.alpha	anagram.in	sim-outorder -config nehalem.cfg anagram.alpha words <anagram.in
2	compress95.alpha	compress95.in	sim-outorder -config nehalem.cfg compress95.alpha <compress95.in >OUTFILE This benchmark is a compressing program and can take any text file as input.
3	go.alpha	2stone9.in	sim-outorder -config nehalem.cfg go.alpha 50 9 2stone9.i
		5stone21.in	sim-outorder -config nehalem.cfg go.alpha 50 21 5stone21.in
4	perl.alpha	perl-tests.pl	sim-outorder -config nehalem.cfg perl.alpha perl-tests.pl
		bigint.pl	sim-outorder -config nehalem.cfg perl.alpha bigint.pl
5	cc1.alpha	1stmt.i	sim-outorder -config nehalem.cfg cc1.alpha -O 1stmt.i

Table 1: List of Benchmarks. It is your responsibility to ensure the paths to the inputs files are proper in your commands.

3 Project Procedure

The steps for this project are described as following.

1. **Step 1:** Obtain and test SimpleScalar following the instructions in Section 2.5.
2. **Step 2:** Evaluate the performance impact of different branch predictors.

   (a) In the *nehalem.cfg*, the branch predictor was configured as a (two-level) correlated branch prediction with a 14-bit Global Branch History Register (GBHR) and Branch History Tables (BHT) with 16,384 two-bit entries. This configuration is specified in the configuration file as

   ```
   -bpred:2lev              1 16384 14 1
   ```

   If we want to change the branch predictor to be 16-bit GBHR with BHTs of 4096 entries, we can run the simulator with the following command,

   ```
   Architect $> PATH/sim-outorder -config nehalem.cfg \
                -bpred 2lev -bpred:2lev 1 4096 16 1 \
                benchmark input
   ```

   Effectively, the option "-bpred:2lev 1 4096 16 1" tells the simulator to use a new branch prediction configuration. Besides 2-level branch predictor, SimpleScalar also support the following branch predictors.

Branch Predictor	Description
nottaken	Assume not taken
taken	Assume taken
perfect	prefect predictor (no mispredictions)
bimod	2-bit saturating counter
2lev	(2-level) correlated predictor
comb	combining predictor

   Except *taken*, *nottaken* and *perfect*, the other three branch predictors also accept additional parameters (configurations). You can run command *sim-outorder -h* to get a list of additional parameters.
   Some additional examples: to use nottaken predictor,

```
Architect $> PATH/sim-outorder -config nehalem.cfg \
             -bpred nottaken \
             benchmark input
```

To use 1024 entries in BHT with 2-bit saturating counter,

```
Architect $> PATH/sim-outorder -config nehalem.cfg \
             -bpred bimod -bpred:bimod 1024 \
             benchmark input
```

(b) Your task is to evaluate the performance of different branch predictors on each benchmark. W will use two performance metrics, the branch misprediction rate and the instructions-per-cycle (IPC). Both metrics can be computed from the output statistics from SimpleScalar. More specifically, you should see the following outputs on your screen after executing the simulator,

```
sim_num_insn                  709628 # total number of instructions committe
        ...
sim_num_branches              124435 # total number of branches committed
        ...
sim_IPC                       1.2896 # instructions per cycle
        ...
bpred_2lev.misses              14959 # total number of misses
```

Clearly, *sim_IPC* gives us the IPC. To compute the branch misprediction rate, you can use equation

$$\frac{bpred\_2lev.misses}{sim\_num\_branches} \tag{1}$$

Note that, when you use other predictors (e.g., *bimod*), you will see slightly different output name for the branch prediction misses (e.g., *bpred_bimod.misses*).

(c) After executing the benchmarks with different configurations, please record the outputs and compute the IPCs and misprediction rates. You should at least experiment with all the predictors using their default settings. After you have finished the experiments, please report your results using a figure, such as Figure 1 (the numbers are of course fake).

You should have at least two figures, one for IPC and one for misprediction rate.

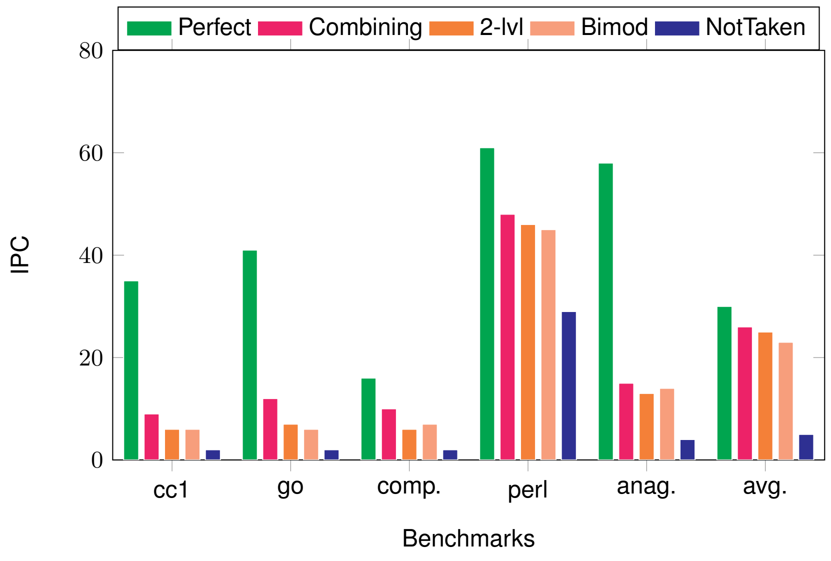

Figure 1: Example figure for plotting IPC changes with branch predictor configurations.

3. **Step 3:** Evaluate the performance impact of out-of-order execution.

   (a) You can change the configuration of the inorder/out-of-order executions with the following options

   ```
   -issue:inorder           true # run pipeline with in-order execution
   ```

   To configure the change the processor to use in-order execution, you can specify

   ```
   Architect $> PATH/sim-outorder -config nehalem.cfg \
                -issue:inorder true \
                benchmark input
   ```

   (b) Please compare the performance of in-order and out-of-order execution for each of the benchmarks. You can use IPC and total number of cycles as the performance metrics.

4. **Step 4:** Evaluate the performance impact of different instruction decoding width and issue width on out-of-order execution.

   (a) You can change the configuration of the inorder/out-of-order executions with the following options

   ```
   -decode:width                4 # instruction decode B/W (insts/cycle)
   -issue:width                 4 # instruction issue B/W (insts/cycle)
   ```

   The default decode and issue widths are both 4. The widths should be power of 2.

   (b) For each benchmark, please increase the decode and issues widths until there is no performance (total execution cycles) improvement. Then report the lowest decode and issue widths that provide the best performance.

4 Deliverables and Submission Guidelines

Please summarize your experiment steps, results and findings in an experiment report. The report should clearly describe the experiments you have conducted and give the results (including figures) of these experiments. Each set of experiments should have its own section, and the report should have at least three pages, excluding the title page if you have one. Please submit the report to Blackboard as a single PDF file.

5 Grading

The report will be graded based on the completeness of the experiments, the clarity of the reported results, and the quality of the writing.