Simulation Project For Computer Architecture*

Xun Tang[1], Tong Wu[2], Yi Wu[3] and Song Yang[4]

***Abstract*— The main goal of this project is to run a set of simulations using the SimpleScalar simulator, which helps you understand the basic concepts introduced in the lectures, such as instruction set architecture, quantitative performance analysis, memory hierarchy, and instruction-level parallelism.**

I. Experimental Setups

Since different hardwares effect the result of experiment, the setups are listed down below. The data is collected with setup 1 if it is not mentioned.

Setup 1: intel Core i5 6200U 12GB memory

Setup 2: intel Core i5 2.3 GHz 8GB memory

II. Lab Experiment 1 (General Instructions)

A. Part1: Results and descriptions

As for Part1 of the lab experiment1, we first set up the simplescalar to Alpha ISA, using the command below:
-make clean
-make config-alpha
-make
Then according to the readme file in the Benchmarks folder, we type several commands to do the test:
for anagram.alpha:
./sim-profile -iclass Benchmarks/anagram.alpha words Benchmarks/anagram.in
for go.alpha:
./sim-profile -iclass Benchmarks/go.alpha 50 9 2stone9.in
for compress95.alpha:
./sim-profile -iclass Benchmarks/compress95.alpha Benchmarks/compress95.in
for cc1.alpha:
./sim-profile -iclass Benchmarks/cc1.alpha -0 1stmt.i

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
anagram.alpha	4960	18.17	19.66	6.33	11.57	43.78	0.20
go.alpha	718274	14.73	13.18	2.65	14.97	54.14	0.32
compress95.alpha	4845	15.50	11.48	4.83	14.22	53.65	0.00
cc1.alpha	760637	21.26	14.17	4.74	11.18	48.61	0.03

TABLE I

Part 1 Setup 1 Song Yang

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
anagram.alpha	4957	18.16	19.67	6.34	11.56	43.79	0.2
go.alpha	718271	14.73	13.18	2.65	14.97	54.14	0.32
compress95.alpha	4842	15.49	11.49	4.83	14.21	53.67	0.00
cc1.alpha	51259	16.37	4.81	3.73	13.76	61.21	0.06

TABLE II

Part 1 Setup 2 Yi Wu

We run the same command on two different computers. For anagram.alpha, go.alpha and compress95.alpha, we got two nearly the same results, but for cc1.alpha we got two different results. We think it's because of the different setup.

B. Part1: Answers for questions

1) Is the benchmark memory intensive or computation intensive?

Nearly half of the instruction is for computation. Load/Store only takes up to 40%. They are computation intensive.

2) Is the benchmark mainly using integer or floating point computations?

Since anagram.alpha has 43.78% integer computation comparing to 0.20% floating point computation and go.alpha has 54.14% integer computation comparing to 0.32% floating point computation, both of them mainly use integer computation. The same situation happens to the other ones.

3) What % of the instructions executed are conditional branches? Given this %, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions)

For anagram.alpha: 11.57% are conditional branches. Since (1-11.57%)/11.57%=7.64,average execution between each pair of conditional branch instructions (do not include the conditional branch instructions) is about 7.64.

For go.alpha: 14.97% are conditional branches. Since (1-14.97%)/14.97%=5.68,average execution between each pair of conditional branch instructions (do not include the conditional branch instructions) is about 5.68. For compress95.alpha: 14.22% are conditional branches. Since (1-14.22%)/14.22%=6.03,average execution between each pair of conditional branch instructions (do not include the conditional branch instructions) is about 6.03.

For anagram.alpha: 11.18% are conditional branches. Since (1-11.18%)/11.18%=7.94,average execution between each pair of conditional branch instructions (do not include the conditional branch instructions) is about 7.94.

C. Part 2: Results and descriptions

In this part, we will compare the PISA and Alpha ISA by executing the same benchmarks on the two configurations. The commands are nearly the same for the four benchmarks.In the Alpha configurations:
test-math: ./sim-profile -iclass tests-alpha/bin/test-math
test-fmath: ./sim-profile -iclass tests-alpha/bin/test-fmath
test-llong: ./sim-profile -iclass tests-alpha/bin/test-llong
test-printf: ./sim-profile -iclass tests-alpha/bin/test-printf
When switching to PISA configurations, we just need to change the 'tests-alpha' to 'tests-pisa', and change 'bin' to 'bin.little'.
Here are the results.

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
test-math	49400	17.17	10.42	3.94	11.07	55.36	1.88
test-fmath	19489	17.71	12.53	4.70	11.27	53.20	0.43
test-llong	10617	17.79	14.61	5.43	12.39	49.49	0.10
test-printf	983463	17.99	10.73	4.82	11.39	54.84	0.09

TABLE III

SETUP 1 SONG YANG-ALPHA

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
test-math	49400	17.17	10.42	3.94	11.07	55.36	1.88
test-fmath	19489	17.71	12.53	4.70	11.27	53.20	0.43
test-llong	10617	17.79	14.61	5.43	12.39	49.49	0.10
test-printf	983463	17.99	10.73	4.82	11.39	54.84	0.09

TABLE IV

SETUP 2 YI WU-ALPHA

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
test-math	213703	15.96	10.67	4.22	13.84	54.42	0.88
test-fmath	53459	16.14	14.43	4.24	15.09	49.95	0.11
test-llong	29642	16.34	18.02	4.36	15.42	45.81	0.10
test-printf	1813891	19.22	9.28	5.13	17.01	49.33	0.01

TABLE V

SETUP 1 SONG YANG-PISA

Benchmark	Total # of Instructions	Load%	Store%	Uncond Branch%	Cond Branch%	Integer Computation%	Floating pt Computation%
test-math	213703	15.96	10.67	4.22	13.84	54.42	0.88
test-fmath	53459	16.14	14.43	4.24	15.09	49.95	0.11
test-llong	29642	16.34	18.02	4.36	15.42	45.81	0.10
test-printf	1813891	19.22	9.28	5.13	17.01	49.33	0.01

TABLE VI

SETUP 2 YI WU-PISA

We got the exactly-same result under two different setups.

D. Answers for questions

Draw histograms. What can you conclude about the two ISAs from the Histogram?
Here are the histograms.

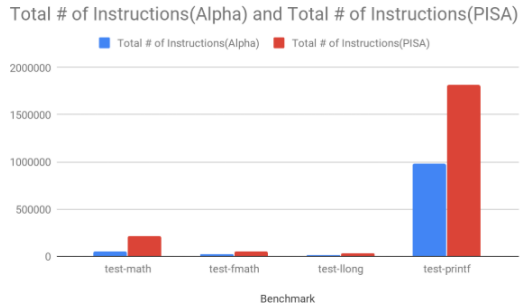


Fig. 1. Number of Instruction

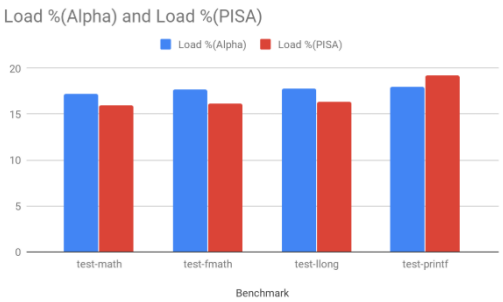


Fig. 2. Load

Fig. 3. Store

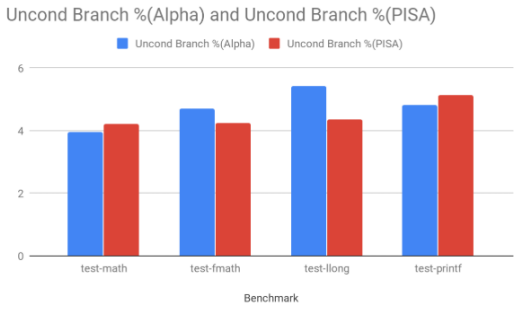


Fig. 4. Uncond Branch

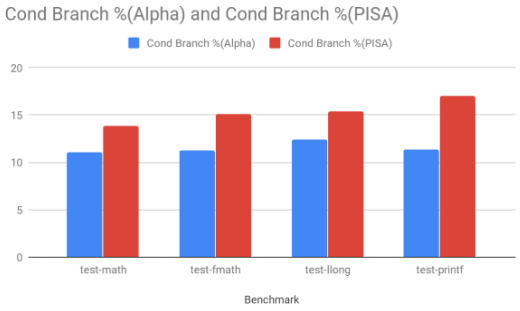


Fig. 5. Cond Branch

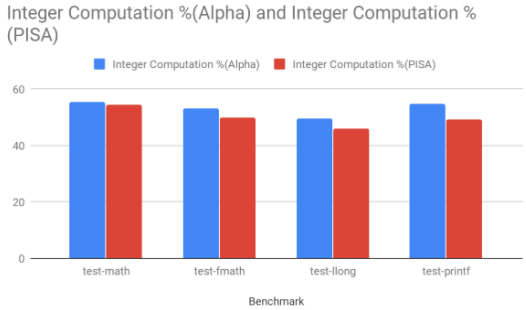


Fig. 6. Integer Computation

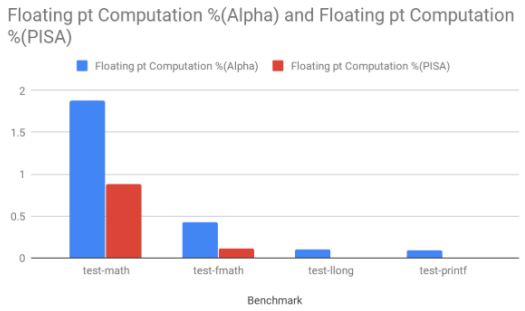


Fig. 7. Floating pt Computation

PISA has 3 times or more instructions for the same program. Generally, they have a similar percentage of different instructions.

E. Challenges

Since this guide is detailed, we did not meet challenges in this part.

III. Lab Experiment 6 (Caches I)

A. Results and discriptions

First we change to PISA ISA. For this simulation, we need to use the sim-cache simulator. According to the configuration files online, we can type the command below to do the simulation:
./sim-cache -cache:il1 il1:32:16:8:l tests-pisa/bin.little/test-math
'il1' in the command means instruction cache. If we want to get the result of data cache, then we change 'il1' to 'dl1'. '32' means the number of sets, and '1' represents '1-way'. So according to the table below, we just need to change these parameters to get the results we want.

Miss Ratio (I-Cache)	1-way	2-way	4-way	8-way
32 sets	0.4269	0.3734	0.2879	0.1983
64 sets	0.3764	0.2920	0.2255	0.1613
128 sets	0.3030	0.2268	0.1618	0.1028
256 sets	0.2503	0.1720	0.0994	0.0176
512 sets	0.1833	0.1010	0.0348	0.0142

TABLE VII
Miss Ratio I-Cache

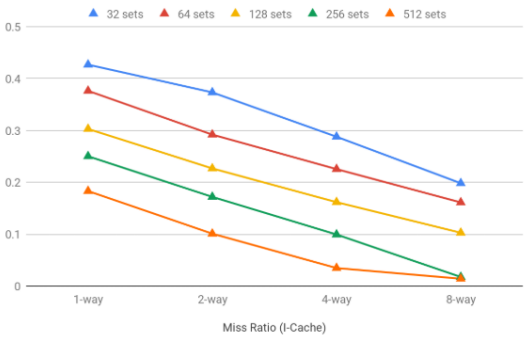


Fig. 8. Miss Ratio (I-Cache)

Miss Ratio (D-Cache)	1-way	2-way	4-way	8-way
32 sets	0.1372	0.0503	0.0236	0.0185
64 sets	0.0654	0.0280	0.0187	0.0184
128 sets	0.0408	0.2233	0.0184	0.0177
256 sets	0.0273	0.0194	0.0177	0.0176
512 sets	0.0209	0.0179	0.0176	0.0176

TABLE VIII
Miss Ratio D-Cache

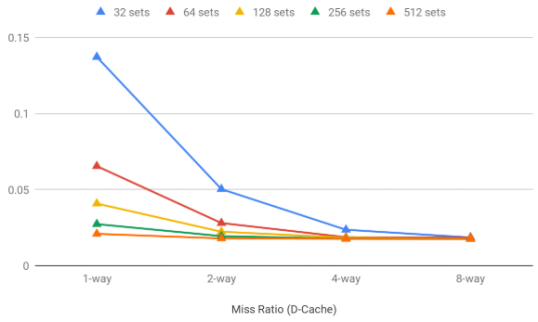


Fig. 9. Miss Ratio (D-Cache)

B. Answers for questions

1) For a given number of sets, what effect does increasing associativity have on the miss ratio?

For both I-Cache and D-Cache, increasing associativity decreases the miss ratio.

2) For a given associativity, what is the effect of increasing the number of sets?

For both I-Cache and D-Cache, increasing the number of sets decreases the miss ratio.

3) For a given cache size, how does the miss ratio change when going from an associativity of one to two to four? Explain.

I Cache Size(KB)	1-way	2-way	4-way
1	0.3764	0.3734	-
2	0.303	0.292	0.2879
4	0.2503	0.2268	0.2255
8	0.1833	0.172	0.1618

TABLE IX
Miss Ratio I-Cache

D Cache Size(KB)	1-way	2-way	4-way
1	0.0654	0.0503	-
2	0.0408	0.028	0.0236
4	0.0273	0.0223	0.0187
8	0.0209	0.0194	0.0184

TABLE X
Miss Ratio D-Cache

Based on the blank above (Table IX Table X), for both I-Cache and D-Cache, increasing associativity decrease the miss ratio in a little range (10% lower).

4) If you were to design an Instruction cache, limited to a total cache size of 4 kbytes, which cache organization would you choose, based solely on performance?

For a 4KB instruction cache, if theres a limitation of associativity which up to 4-way (according to question 3), then a 4-way cache with 64 sets has the best performance with the lowest miss ratio at 22.55%. But if theres no limitation, then an 8-way cache with 32 sets has the best performance which reduces the miss ratio to 19.83%.

5) If you were to design a data cache, limited to a total cache size of 4 Kbytes, which cache organization would you choose, based solely on performance?

For a 4KB data cache, a 4-way cache with 64 sets has the best performance (the lowest miss ratio).

For a 4KB data cache, if theres a limitation of associativity which up to 4-way (according to question 3), then a 4-way cache with 64 sets has the best performance with the lowest miss ratio at 1.87%. But if theres no limitation, then an 8-way cache with 32 sets has the best performance which reduces the miss ratio to 1.85%.

C. Bonus Part

Repeat part 1) for only Instruction cache except that change the replacement scheme from LRU to Random. Plot the results as shown in the above figure. You don't need to answer the questions in part 1).

We just need to change the 'l' in the command we discussed above to 'r', cause 'l' means LRU, and 'r' means Random.

./sim-cache -cache:il1 il1:32:16:8:r tests-pisa/bin.little/test-math

Miss Ratio (I-Cache)	1-way	2-way	4-way	8-way
32 sets	0.4269	0.3609	0.2848	0.2189
64 sets	0.3765	0.2937	0.2301	0.1661
128 sets	0.3031	0.2318	0.1669	0.0848
256 sets	0.2504	0.1733	0.0894	0.0287
512 sets	0.1834	0.0995	0.0375	0.0183

TABLE XI
MISS RATIO I-CACHE RABDIN SCHEME

The tutorial is really detailed, we just met problems calculating the cache sizes and finally solved.

IV. Lab Experiment 5 (Branch Prediction)

A. Results and description

First we change to Alpha ISA. For this simulation, we need to use the sim-bpred simulator. The commands are:

./sim-bpred -bpred taken Benchmarks/cc1.alpha -0 1stmt.i

This is an example for cc1.alpha. For other benchmarks, we can change the command using the way discussed in Lab Experiment 1. In order to change the way of prediction, we can replace 'taken' in the command by 'nottaken' or 'bimod'.

Benchmark	Taken	Not Taken	Bimod
anagram.alpha	0.7568	0.5968	0.5822
go.alpha	0.6682	0.4822	0.8977
compress95.alpha	0.8353	0.4182	0.7568
cc1.alpha	0.6476	0.6501	0.9325

TABLE XII
BENCHMARK

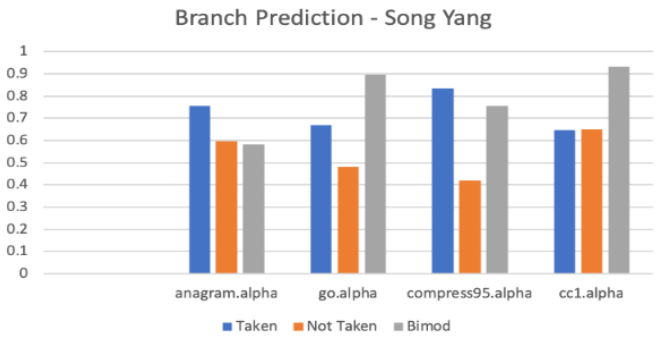
The command using sim-outorder to calculate CPI is simple enough-just change'sim-bpred' to 'sim-outorder'. Then from a whole bunch of results we can find the CPI.

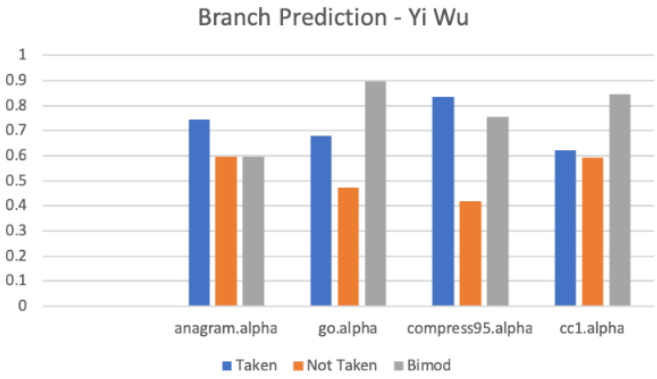
CPI Benchmark	Taken	Not Taken	Bimod
anagram.alpha	2.7798	2.8020	2.7206
compress96.alpha	2.5548	2.5740	2.0941

TABLE XIII
CPI BENCHMARK

B. Answers for questions

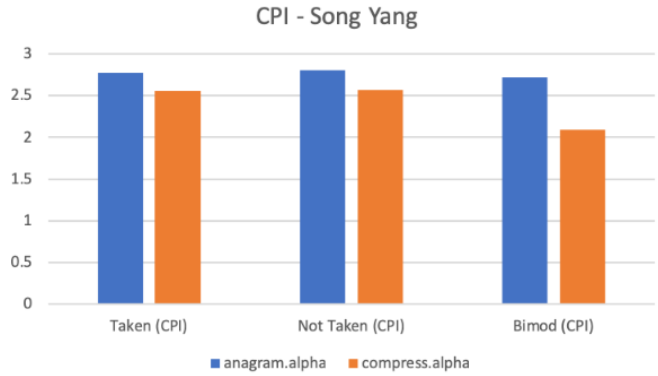
Plot a histogram for all the four benchmarks and draw conclusion. Does your conclusion agree with your intuition?

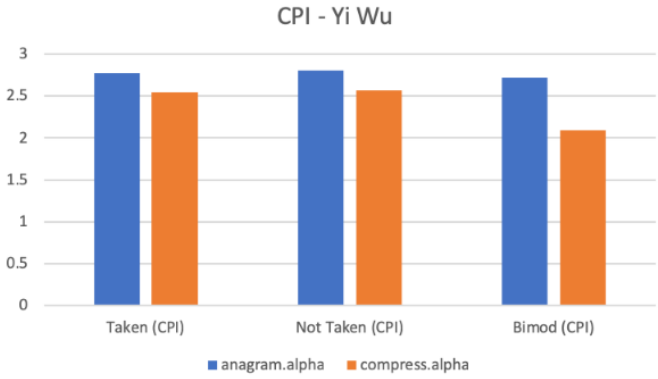




In the branch prediction, we still tested benchmarks on 2 different hardware environments and got similar results on both. For anagram and compress benchmarks, Taken Prediction has the best performance over these 3 prediction types. While for go and cc1 benchmark, Bimod is the king. Not Taken has the worst prediction performance on these 4 benchmarks.

Plot a histogram for the two benchmarks for the three schemes.





CPI Benchmark: Results are similar on both environments. Compared with compress benchmark, anagram always has better CPI performances over these 3 types of prediction.

C. Bonus Part

Write, compile and run a C program targeted towards PISA ISA which contains a for loop that iterates 10 million times. Please see bonus lab1 to know how to install gcc for PISA ISA. Calculate Branch address pred-Rate for the three case Taken, Not Taken and bimod.

For this bonus question, a c program with 10million loop is created and compiled with the gcc tools which is included in simplescalar 2.0. The compile command is :
../bin/sslittle-na-sstrix-gcc -o a.out bonus.c
The compiled program file is named a.out. So the command inserted in the bash is:
./sim-bpred -bpred taken a.out
And the result is listed below.

Branch address pred-Rate	Taken	Not Taken	Bimod
10 million loop	1	0.5	1

TABLE XIV
BRANCH ADDRESS PRED-RATE

D. Challenges

The guide for this problem is pretty detailed.

As for the bonus question , in Simplescalar, since the lack of offical guide for version 3.0, it's very difficult to setup the gcc cross compiler. There is a lot of errors and exceptions showed at first. I searched online, looking for tutorials and official guide and finally successfully finished this problem.