Computer Architecture Project Report
Part A: SimpleScalar-based Study Project

Weichen Li
5120309662
Department of Computer Science and Engineering
School of Electronic Information
and Electrical Engineering
Shanghai Jiaotong University
Shanghai China 200240
Email: eizo.lee@sjtu.edu.cn

Abstract—**SimpleScalar is a tool set, which consists of compiler, assembler, linker, simulation, and visualization tools for the SimpleScalar architecture. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation.**

SimpleScalar provide simulators ranging from a fast functional simulator to a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. The tool set is partly derived from the GNU software development tools. It provides researchers with an easily extensible, portable, high-performance test bed for systems design.

I. Introduction

The objective of this project is get students hand-on experiences with some techniques (mainly on cache organization and optimizations) that they have learned from textbook using SimpleScalar.

Read the problem sheet and select three problems: two from P1-P5, and P8 is compulsory. Work on the three problems.

For problems 1 through 8, students can learn about the ”go” SPEC95 benchmark by looking at the web page *http://www.spec.org/osg/cpu95/news/099go.html*.

II. Problem 1

A. Basic setting and experiment description

1) Experiment description: This problem introduces the sim-cheetah cache simulator. In order to simulate several cache configurations at one time, the simulator utilizes an algorithm devised by Rabin Sugumar and Santosh Abraham while they were at the University of Michigan. You can look at the sim-cheetah summary by entering the command sim-cheetah by itself.

Use a single run of sim-cheetah to simulate the performance of the following cache configurations for the SPEC95 benchmark ”go”.

Configurations:

- least-recently-used (LRU) replacement policy
- 128 to 2048 sets
- 1-way to 4-way associativity
- 16-byte cache lines

2) Settings: The command line is:
./sim-cheetah -redir:sim SpecOutput/problem1/pro1.txt -R lru -C sa -a 7 -b 11 -l 4 -n 2 go.ss 2 8 go.in

where

- **-redir:sim** is the argument setting where the output goes.
- **SpecOutput/problem1/pro1.txt** stores the output.
- **-R lru** sets the replacement policy(LRU).
- **-C sa** sets the cache configuration(**sa** means set associative).
- **-a 7** sets the minimum number of sets (log base 2, $2^7 = 128$).
- **-b 11** sets the maximum number of sets (log base 2, $2^{11} = 2048$).
- **-l 4** sets the line size of the caches (log base 2, $2^4 = 16$).
- **-n 2** sets the maximum degree of associativity to analyze (log base 2, $2^2 = 4$).
- **go.ss** is the SimpleScalar binary for the ”go” benchmark.
- **2 8** are the play quality and the board size for the go simulation.
- **go.in** is a file that specifies the starting board position (in this case an empty file.)

B. Results

The two computer players as they present their moves. The game (and therefore the simulation) will end at move number 40.

In the output file **pro1.txt**, I get:

```
libcheetah: ** simulation parameters **
libcheetah: LRU Set-associative caches
            being simulated
libcheetah: number of sets from 128 to 2048
libcheetah: maximum associativity is 4
libcheetah: line size is 16 bytes

sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_ins   31394965 # total number of
```

```
                    instructions executed
sim_num_refs 8154766 # total number of
               loads and stores executed
sim_elapsed_time  1 # total simulation
                     time in seconds
sim_inst_rate 31394965.0000 # simulation
                 speed(in insts/sec)

libcheetah: ** end of simulation **
Addresses processed: 8155568
Line size: 16 bytes
```

And the **Miss Ratio Table** is :

TABLE I
MISS RATIO TABLE

No.	Asscociativity			
of sets	1	2	3	4
128	0.185862	0.094562	0.065089	0.051197
256	0.129740	0.062298	0.043251	0.034149
512	0.094917	0.043441	0.030266	0.024275
1024	0.057872	0.025081	0.017734	0.012934
2048	0.037176	0.014161	0.007808	0.004801

C. Discussion

• Using the output from sim-cheetah, for caches of equivalent size, verify using simple calculations whether increasing associativity or the number of sets in the cache gives the most benefit.

• Explain how you did your computations, give your results, and discuss the relative benefits of cache associativity verses the number of sets in the cache.

• You should include at least four comparisons. Is the trend always the same? Suggest reasons for the trends or the of lack of trends.

Answer.

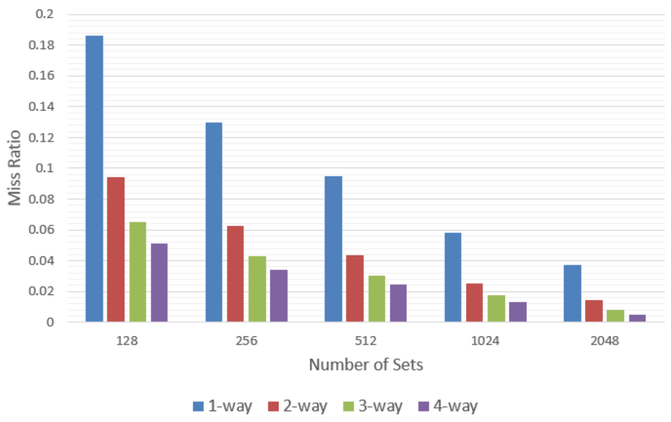

Fig. 1. Miss Ratio distribution

From Fig.1 and the **Miss Ratio Table** we can see that:

$$A_1 = \frac{\sum_{a=1}^{a=3} \frac{R_{128,a+1}-R_{128,a}}{R_{128,a}}}{3} = -33.88\%$$

$$A_2 = \frac{\sum_{s=128}^{s=256} \frac{R_{2s,1}-R_{s,1}}{R_{s,1}}}{2} = -28.52\%$$

Here $A_{s,a}$ is the growth rate of the Miss Rate $(R_{s,a})$ of the configuration which contains s sets and uses a-way associativity.

A_x is the average growth rate of the Miss Rate under two limited conditions: One changes the associativity, the other changes the number of sets. Both of these two average growth rate result in an increase in cache size of 4.

We can see that with the same beginning Miss Ratio, increasing a gives the cache more benefit here.

Similarly, we can get:

$$A_3 = \frac{\sum_{s=128}^{s=256} \frac{R_{2s,4}-R_{s,4}}{R_{s,4}}}{2} = -31.11\%$$

$$A_4 = \frac{\sum_{s=512}^{s=1024} \frac{R_{2s,1}-R_{s,1}}{R_{s,1}}}{2} = -37.40\%$$

We can see that Miss Ratio drops faster when a is smaller and s is larger, but since $R_{128,4} = 0.051197$ and $R_{512,1} = 0.094917$, $R_{512,4}$ is still smaller than $R_{2048,1}$, which means a still gives the cache more benefit here.

The third comparison is:

$$A_5 = \frac{\sum_{s=512}^{s=1024} \frac{R_{2s,4}-R_{s,4}}{R_{s,4}}}{2} = -54.80\%$$

$$A_6 = \frac{\sum_{a=1}^{a=3} \frac{R_{2048,a+1}-R_{2048,a}}{R_{2048,a}}}{3} = -48.43\%$$

Compare this result with the first comparison, We can see that with the same final Miss Ratio and larger a, increasing s gives the cache more benefit here.

The fourth comparison is:

$$A_4 = \frac{\sum_{s=512}^{s=1024} \frac{R_{2s,1}-R_{s,1}}{R_{s,1}}}{2} = -37.40\%$$

$$A_1 = \frac{\sum_{a=1}^{a=3} \frac{R_{512,a+1}-R_{512,a}}{R_{512,a}}}{3} = -34.79\%$$

Compare this result with the second comparison, We can see that with the same beginning Miss Ratio and smaller a, although increasing s will decrease the Miss Ratio faster, increasing a will give the cache more benefit here, since $R_{512,4} = 0.024275$ and $R_{2048,1} = 0.037176$.

D. Conclusion

From above, we can get a preliminary idea that increasing a when s is relatively small, or increasing s when a is relatively large, will gives the most benefit.

And the trend is not always the same, I think when a or s is large enough, the effect brought by the increase might be weaken.

III. Problem 3

A. Basic setting and experiment description

1) Experiment description: The sim-cheetah simulator implements the MIN set-associative cache replacement policy that was first suggested by Laszlo Belady. This unimplementable–but simulatable–policy uses oracle information to determine the block in a set that will be used the farthest in the future. This is different than the LRU replacement policy which "guesses" the block in a set that will be used farthest in the future by using information about when the blocks were used in the past.

The MIN algorithm knows which block will be used furthest in the future. Therefore, using the MIN replacement policy should result in better performance than using the LRU replacement policy.

Redo the simulation in Problem 1 using the MIN replacement policy (sim-cheetah calls the policy opt). If you have not done the simulation in Problem 1, do it first.

2) Settings: The command line is:

./sim-cheetah -redir:sim SpecOutput/problem3/pro3.txt -R opt -C sa -a 7 -b 11 -l 4 -n 2 go.ss 2 8 go.in

where

- **-redir:sim** is the argument setting where the output goes.
- **SpecOutput/problem3/pro3.txt** stores the output.
- **-R opt** sets the replacement policy(MIN).
- **-C sa** sets the cache configuration(**sa** means set associative).
- **-a 7** sets the minimum number of sets (log base 2, 2^7 = 128).
- **-b 11** sets the maximum number of sets (log base 2, 2^{11} = 2048).
- **-l 4** sets the line size of the caches (log base 2, 2^4 = 16).
- **-n 2** sets the maximum degree of associativity to analyze (log base 2, 2^2 = 4).
- **go.ss** is the SimpleScalar binary for the "go" benchmark.
- **2 8** are the play quality and the board size for the go simulation.
- **go.in** is a file that specifies the starting board position (in this case an empty file.)

B. Results

The two computer players as they present their moves. The game (and therefore the simulation) will end at move number 40.

In the output file **pro3.txt**, I get:

```
libcheetah: ** simulation parameters **
libcheetah: OPT Set-associative caches being
            simulated
libcheetah: number of sets from 128 to 2048
libcheetah: maximum associativity is 4
libcheetah: line size is 16 bytes

sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn    31394965 # total number of
                   instructions executed
sim_num_refs   8154766 # total number of
           loads and stores executed
sim_elapsed_time   2 # total simulation
                   time in seconds
sim_inst_rate  15697482.5000 # simulation
              speed (in insts/sec)

libcheetah: ** end of simulation **
Addresses processed: 8155568
Line size: 16 bytes
```

And the **Miss Ratio Table** is :

TABLE II
Miss Ratio Table

No.	Asscociativity			
of sets	1	2	3	4
128	0.149454	0.071916	0.048655	0.037641
256	0.106226	0.048559	0.032539	0.02493
512	0.079259	0.033936	0.021734	0.015905
1024	0.050421	0.019199	0.011578	0.007579
2048	0.03269	0.010186	0.005191	0.003463

C. Discussion

1) : For a direct-mapped cache (set size 1), would you expect the results for LRU and MIN replacement policies to be different? Why or why not?

Answer.

My suggestion is that the results will not be very different, because in a direct-mapped cache, a block has been allocated for some fixed blocks in main memory. Any optimization becomes useless under the circumstances. The conflict miss will be inevitable for blocks.

2) : Discuss the result for associativity 1.

Answer.

The results for associativity 1 are:

$$\begin{aligned} R_{128,1} =& \ 0.149454 \\ R_{256,1} =& \ 0.106226 \\ R_{512,1} =& \ 0.079259 \\ R_{1024,1} =& \ 0.050421 \\ R_{2048,1} =& \ 0.03269 \end{aligned}$$

Result: With the increasing of s, Miss Ratio decreases.

3) : Define the reduction R in Miss Rate as:

$$R = \frac{originalMissRate - newMissRate}{originalMissRate}$$

Compute the reduction in Miss Rate when changing from the LRU to the MIN replacement policy for all cache sizes and associativities 2 and 4.

Answer.

TABLE III
MISS RATIO REDUCTION TABLE

Cache	Associativity	
Size	2	4
4KB	0.23948309	-
8KB	0.220536775	0.26478114
16KB	0.218802514	0.269963981
32KB	0.234520155	0.344799176
64KB	0.280700516	0.41402505
128KB	-	0.278691939

4) : What do your results suggest about cache replacement policies? If MIN works better than LRU and MIN cannot be implemented (it uses information about the future), then what can be done to improve replacement policies?

Answer.

With associativities 2 and 4, MIN works better than LRU. We need some kind of prediction mechanism based on the register allocation record to improve replacement policies.

D. Conclusion

MIN works better than LRU (When associativity is greater than 1), however MIN cannot be implemented.

Inspired by MIN, we can develop some prediction mechanism to decrease the Miss Rate.

IV. PROBLEM 8

A. Basic setting and experiment description

1) Experiment description: Branch predictors.

2) Settings: This section consists of 3 parts.

Firstly, I run sim-profile to find out the distribution of instruction types for the SPEC95 benchmark ”go”. I use the command line:

./sim-profile -redir:sim SpecOutput/problem8/pro8.txt -iclass go.ss 2 8 go.in

where

- **-redir:sim** is the argument setting where the output goes.
- **SpecOutput/problem8/pro8.txt** stores the output.
- **-iclass** enable instruction class profiling.
- **go.ss** is the SimpleScalar binary for the ”go” benchmark.
- **2 8** are the play quality and the board size for the go simulation.
- **go.in** is a file that specifies the starting board position (in this case an empty file.)

Next, the sim-outorder simulator allows you to simulate 6 different types of branch predictors. You can see the list of them by looking at the -bpred parameter listed in the output generated when you enter sim-outorder by itself. For each of the 6 default predictors, run the same go simulation as you did above and note the branch-direction prediction rate and the IPC for each. Each command line will be similar to:

./sim-outorder -redir:sim SpecOutput/problem8/pro8_nottaken.txt -bpred nottaken go.ss 2 8 go.in

Finally, the sim-outorder simulator does not include a way to disable branch prediction. This is because out-of-order processors always benefit from predicting branches as taken or not taken, even if a more resource intensive predictor is not implemented.

Modify the sim-outorder simulator to include the parameter -bpred none. This parameter will cause the processor to simulate a processor with no branch prediction be treating every conditional branch as a mis-predicted branch. To do this, modify the file sim-outorder.c. Look at the code that handles the perfect branch prediction command-line parameter, and create new code for the parameter none.

B. Results

- In the output file **pro8.txt**, I get:

```
sim: ** starting functional simulation **

sim: ** simulation statistics **
sim_num_insn    31394965 # total number of
                  instructions executed
sim_num_refs    8154766 # total number of
              loads and stores executed
sim_elapsed_time    2 # total simulation
                       time in seconds
sim_inst_rate    15697482.5000 # simulation
```

```
                    speed (in insts/sec)
sim_inst_class_prof    # instruction
                     class profile
sim_inst_class_prof.array_size = 7
sim_inst_class_prof.bucket_size = 1
sim_inst_class_prof.count = 7
sim_inst_class_prof.total = 31394964
sim_inst_class_prof.imin = 0
sim_inst_class_prof.imax = 7
sim_inst_class_prof.average = 4484994.8571
sim_inst_class_prof.std_dev = 6467950.4911
sim_inst_class_prof.overflows = 0
# pdf == prob dist fn,
  cdf == cumulative dist fn
#   index              count    pdf
sim_inst_class_prof.start_dist
load                  6175411  19.67
store                 1979355   6.30
uncond branch          996935   3.18
cond branch           4001737  12.75
int computation      18241470  58.10
fp computation              0   0.00
trap                       56   0.00
sim_inst_class_prof.end_dist

ld_text_base     0x00400000 # program text
                      (code) segment base
ld_text_size         621600 # program text
                      (code) size in bytes
ld_data_base     0x10000000 # program
             initialized data segment base
ld_data_size         578004 # program
           init'ed '.data' and uninit'ed '
           .bss' size in bytes
ld_stack_base    0x7fffc000 # program
              stack segment base
            (highest address in stack)
ld_stack_size         16384 # program
                  initial stack size
ld_prog_entry    0x00400140 # program
                 entry point (initial PC)
ld_environ_base  0x7fff8000 # program
         environment base address address
ld_target_big_endian      0 # target
                executable endian-ness,
                non-zero if big endian
mem.page_count          282 # total
             number of pages allocated
mem.page_mem          1128k # total
           size of memory pages allocated
mem.ptab_misses         282 # total
           first level page table misses
mem.ptab_accesses 145809356 # total
                    page table accesses
mem.ptab_miss_rate   0.0000 # first
                level page table miss rate
```

- In the output files **pro8_nottaken.txt**, **pro8_taken.txt**, **pro8_perfect.txt**, **pro8_bimod.txt**, **pro8_2lev.txt** and **pro8_comb.txt** I get TABLE IV.

TABLE IV
PREDICTORS RESULT

-bpred	**branch direction-prediction rate**	**IPC**
nottaken	0.3214	0.6525
taken	0.3214	0.6599
perfect	1	1.1500
bimod	0.8434	0.9680
2lev	0.7514	0.9005
comb	0.8457	0.9737

C. Discussion

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

Answer.

From the **pro8.txt** we can know the percentage of the conditional branch instructions executed is 12.75%

From the equation below we can get that on average,

$$\begin{aligned} N_{average} &= \frac{(1 - P_{conditional\ branch\ ins}) \times N_{all}}{P_{conditional\ branch\ ins} \times N_{all}} \\ &= \frac{1 - P_{conditional\ branch\ ins}}{P_{conditional\ branch\ ins}} \\ &= \frac{1 - 12.75\%}{12.75\%} \\ &\approx 6.84 \end{aligned}$$

2) : For each of the 6 branch predictors, describe the predictor. Your description should include what information the predictor stores (if any), the amount of storage (in bits) that is required, how the prediction is made, and what the relative accuracy of the predictor is compared to the others (use the branchdirection measurements for this). Finally, describe how the prediction rate effects the processor IPC for the go benchmark. Do you believe that the results for go will generalize to all programs? If not, describe a program for which the prediction rates or the IPCs would be different.

Answer part a.

nottaken and **taken**: They both belong to static prediction which is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead it predicts the outcome of a branch based solely on the branch instruction. nottaken always predicts not taken and taken always predicts taken. And they have the same accuracy: 32.14%.

perfect: Its mechanism is like its name–no mispredictions. This predictor need to read all the branch instructions so that it can perform a perfet prediction. Its accuracy is 100%.

bimod: Bimodal Predictor is a state machine with four states: Strongly not taken, Weakly not taken, Weakly taken, Strongly taken. When a branch is evaluated, the corresponding state machine is updated. Branches evaluated as not taken decrement the state toward strongly not taken, and branches evaluated as taken increment the state toward strongly taken. It has 2048 entries. Each entry in the BTB is a 2-bit counter. So the storage needed is $2 \times 2048 = 4096$ bits. Its accuracy is 84.34%.

2lev: A two-level adaptive predictor remembers the history of the last n occurrences of the branch and uses one saturating counter for each of the possible 2^n history patterns.

Consider the example of n = 2. This means that the last two occurrences of the branch are stored in a 2-bit shift register. This branch history register can have 4 different binary values: 00, 01, 10, and 11; where 0 means "not taken" and 1 means "taken". Now, we make a pattern history table with four entries, one for each of the 2^n = 4 possible branch histories. Each entry in the pattern history table contains a 2-bit saturating counter of the same type as in figure 2. The branch history register is used for choosing which of the four saturating counters to use. If the history is 00 then the first counter is used. If the history is 11 then the last of the four counters is used.

It consists of 2 levels: Level 1 is a shift register with 8 bits recording branch history. Level 2 is the 2 bit predictor associated with the pattern in Level 1. There are 1024 entries in Level 2. So the total storage is $8 + 1024 \times 2 = 2056$ bits. Its accuracy is 75.14%.

comb: This predictor implements more than one prediction mechanism. The final prediction is based either on a meta-predictor that remembers which of the predictors has made the best predictions in the past, or a majority vote function based on an odd number of different predictors.

The number of entries in the combined predictor table is 1024. Its accuracy is 84.57%.

Answer part b.

IPC implies Instructions Per Cycle. When a predictor correctly predicts a branch, some prefetching activity may occur, which means more instructions can be executed simultaineously. So a more efficient predictor can lead to a greater IPC.

Answer part c.

I do not believe the results for go will generalize to all programs. Counter example: a program whose branch instructions hardly taken. Under the circumustance, nottaken will have a greater prediction rate, but taken is reverse.

D. Conclusion

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.