

SimpleScalar-based cache study

2015 fall computer architecture project on cache organization and optimization

Shen Jiyuan
CSE of SJTU
5130309194
Shanghai, China
sheryalyuan@126.com

Abstract—*SimpleScalar, which packs compiler, assembler, linker, simulation, and visualization tools, is used as a simulator here to simulate real programs on multiple system where cache design is specially focused on.*

Index Terms—*SimpleScalar, cache simulation, optimization, configuration.*

I. INTRODUCTION OF SIMPLESCALAR

This tool set 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. We 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.

In my experiments for the project here, version 2.0 is used, and three simulators are mainly executed, including sim-cheetah, sim-profile and sim-outorder.

Sim-cheetah: This program implements a functional simulator driver for Cheetah. Cheetah is a cache simulation package written by Rabin Sugumar and Santosh Abraham which can efficiently simulate multiple cache configurations in a single run of a program. Specifically, Cheetah can simulate ranges of single level set-associative and fully-associative caches. See the directory libcheetah/ for more details on Cheetah.

Sim-profile: This simulator implements a functional simulator with profiling support. Run with the '-h' flag to see profiling options available.

Sim-outorder: This simulator implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

II. INTRODUCTION OF GO-BENCHMARK

The CINT95 benchmark 099.go uses several techniques common in the field of artificial intelligence to play the ancient Asian game of Go. This program, written by David Fotland, is based upon a version of his commercially successful programs sold under the names 'Cosmos' and 'The Many Faces of Go'. Versions of this code have been leading contenders in international competition.

There are two folders named 'spec95-little' and 'spec95-big' in your installed simplescalar. You will choose one of them as the target benchmark according to your computer endian mode. Here I give a short C program for you to check your endian mode. You will make the C program on the linux environment to get the executable file, and you can execute the file in the terminal to check your endian mode.

```
#include <stdio.h>
int main(int argc, char **argv)
{
    union {
        short s;
        char c[sizeof(short)];
    } un;
    un.s = 0x0102;
    if (sizeof(short) == 2) {
        if (un.c[0] == 1 && un.c[1] == 2)
            printf("big-endian\n");
        else if (un.c[0] == 2 && un.c[1] == 1)
            printf("little-endian\n");
        else
            printf("unknown\n");
    } else
        printf("sizeof(short)=%d\n", sizeof(short));
    return 0;
}
```

Fig. 1. my C program for endian mode checking

In this report, little endian mode is taken.

III. ENVIRONMENT SETTING AND PREPARATIONS

The simplescalar tool is used in Ubuntu of version 14.04. You should first install it well in your virtual machine. Actually, the requirement in the project web is Ubuntu of version 9.04. But I use the tool with no problem in 14.04. And I keep on my experiment in Ubuntu of version 14.04.

A. Download Packages and Make

- 1) Download the package from the website
http://www.cs.sjtu.edu.cn/~yzhu/courses/comarc_15fall/Course_project/simplesim-3.0r.zip
- 2) Using standard Linux commands to unpack it.
Go to terminal, and firstly execute 'make config-pisa', secondly execute 'make'
- 3) Download the package from the website
http://pages.cs.wisc.edu/~mscalar/simplescalar.html
- 4) Using standard Linux commands to unpack it.
Go to terminal, and refer to the file 'INSTALL', just follow the tips and take the commands to install it.

B. Simple Test for Installation

You should first execute the endian-checking program mentioned above to ensure your endian mode. If you get little endian mode as mine, just use the 'go.ss' in the folder 'spec95-little', and vice versa. Use the command line './sim-fast go.ss 2 8 go.in'. You can see the simulation process if the former installation is successful.

IV. PRO1__CACHE SIMULATION AND ASSOCIATIVITY

This problem introduces the sim-cheetah cache simulator which is used to simulate several cache configurations at one time, and here we use it to investigate the set and associativity effects on miss rate.

A. Basic Setting and Experiment Description

[Problem Description]

- 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.
- The command line will be: sim-cheetah <arguments> go.ss 2 8 go.in where * <arguments> is the list of sim-cheetah parameters needed to produce results for the specified cache configurations. * Go.ss is the play quality and the board size for the go simulation. * "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).
- You can watch the two computer players as they present their moves. The game (and therefore the simulation) will end at move number 40.
- Using the output from sim-cheetah, for caches of equivalent size, verify using simple calculations whether increasing associativity or the number of sets 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.

[Basic Setting]

- Call: Executable file <sim-cheetah> in the folder <simplesim>;The integer benchmark <go> in the folder <spec95-little>.
- Command line:./sim-cheetah <parameters> ./pec95-little/go.ss 2 8 The 'go.in' is omitted as it is an empty file here in the experiment.
- Parameters:
There are a variety of parameters including cache configuration, set number, cache size intervals, etc.

And in this experiment, we focus on four parameters listed as follows:

TABLE I. Problem I Parameter Setting

	Default	Meaning
-a	7	minimum number of sets(log base 2)
-b	14	maximum number of sets(log base 2)
-l	4	line size of the cache(log base 2)
-n	1	maximum degree of associativity(log base 2)

My setting for parameters in this experiment:

LRU doesn't need to be set particularly, as sim-cheetah use LRU as the default replacement policy. For set_range[128,2048]: 128=log7, 2048=log11, both log base 2. Then setting parameters include: -a 7 -b 11. For way associativity range [1, 4]: max_associativity= 4 = log2, log base 2. Then the setting parameters include: -n 2. For line_size[16]: line_size= 16 = log4, log base 2. Then setting parameters include: -l 4.

- My command-line for this experiment:

Output the results to terminal:

```
./sim-cheetah -a 7 -b 11 -l 4 -n 2 ./spec95-little/go.ss 2 8
```

Output the results to an output file:

```
./sim-cheetah -a 7 -b 11 -l 4 -n 2 ./spec95-little/go.ss 2 8 >& ./cheetahlog1.txt
```

B. Results

TABLE II. Miss Ratio I [set vs. associativity]

set/ass	1	2	3	4
128	0.185877	0.094569	0.065089	0.051195
256	0.129746	0.062298	0.043249	0.034146
512	0.094919	0.043439	0.030262	0.024270
1024	0.057869	0.025078	0.017761	0.013018
2048	0.037194	0.014219	0.007864	0.004824

C. Discussion on Results

Using the output from sim-cheetah, for caches of equivalent size, discussions show as follows:

- Format explanation: 1. '(sets/associativity)' 2. '() verses ()' = set_inc verses associativity_inc. The two formats will be used to clarify my descriptions.
- ComputationEX1: an increase in cache size 4 (from 128/1 to 512/1) verses (from 128/1 to 128/4) benefit(128/1 \rightarrow 512/1)= 0.185877/0.094919= 1.958 benefit(128/1 \rightarrow 128/4)= 0.185877/0.051195= 3.631 Comparison: set < associativity
- ComputationEX2: an increase in cache size 4 (from 256/1 to 1024/1) verses (from 256/1 to 256/4) benefit(256/1 \rightarrow 1024/1)=0.129746/0.057869=2.242 benefit(256/1 \rightarrow 256/4)= 0.129746/0.034146= 3.800 Comparison: set < associativity
- ComputationEX3: an increase in cache size 4 (from 128/4 to 512/4) verses (from 512/1 to 2048/1) benefit(128/4 \rightarrow 512/4)= 0.051195/0.024270= 2.109 benefit(512/1 \rightarrow 2048/1)= 0.094919/0.037194= 2.552 Comparison: If the associativity decreases and the set increases the same magnification, the benefit from set will increase. If set decreases and associativity increases the the same magnification, the benefit from set will decrease.
- ComputationEX4: an increase in cache size 2 (from 128/4 to 512/4) verses (from 512/1 to 2048/1) benefit(256/4 \rightarrow 512/4)= 0.034146/0.024270= 1.407 benefit(512/1 \rightarrow 1024/1)= 0.094919/0.057869= 1.640 Comparison: If associativity decreases and the set increases the same magnification, the benefit from set will increase. If set decreases and associativity increases the the same magnification, the benefit from set will decrease.
- ComputationEX5: an increase in cache size 2 (from 512/2 to 512/4) verses (from 1024/1 to 1024/2) benefit(512/2 \rightarrow 512/4)=0.043439/0.024270=1.790 benefit(1024/1 \rightarrow 1024/2)=0.057869/0.025078=2.308 Comparison: If associativity decreases and the set increases the same magnification, the benefit from associativity will increase. If set decreases and associativity increases the the same magnification, the benefit from associativity will decrease.

D. Concise Conclusion

Compared to increasing the set, when we could choose optimization on set or associativity, increasing the associativity with the same magnification can introduce more benefits.

Benefits from set increase or associativity increase will be larger if the associativity decreases and the set increases the same magnification. Benefits from set increase or associativity may decrease if the associativity increases and the set decreases the same magnification.

V. PRO3__CACHE REPLACEMENT POLICIES

This problem further uses the sim-cheetah to discuss the cache upon the replacement policy. And we will focus on the comparison between the LRU and the MIN algorithm.

A. Basic Setting and Experiment Description

[Problem 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 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.
- 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?
- Redo the simulation in Problem 1 using the MIN replacement policy (sim-cheetah calls the policy opt). Discuss the results for associativity 1.
- Define the reduction in miss-rate as

$$(MissR^{\text{original}} - MissRate^{\text{new}}) / MissRate^{\text{original}} \quad (1)$$

- 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.
- 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?

[Basic Setting]

- Call: 1. Executable file <sim-cheetah> in the folder <simplesim> 2. The integer benchmark <go> in the folder <spec95-little>
- Command line:./sim-cheetah <parameters> ../pec95-little/go.ss 2 8 The 'go.in' is omitted as it is an empty file here in the experiment.
- Parameters:
As the same as Problem I, we only care about some parameters critical in this experiment, and they are listed in the following table:

TABLE III. Problem III Parameter Setting

	Default	Meaning
-a	7	minimum number of sets(log base 2)
-b	14	maximum number of sets(log base 2)
-l	4	line size of the cache(log base 2)
-n	1	maximum degree of associativity(log base 2)
-R	lru	replacement policy

My setting for parameters in this experiment:

Replacement policy should be set as opt, thus the setting parameter include: -R opt. For set_range [128, 2048]: 128=log7, 2048=log11, both log base 2. Then setting parameters include: -a 7 -b 11. For way associativity range [1, 4]: max_associativity= 4 = log2, log base 2. Then the setting parameters include: -n 2 For line_size[16]: line_size= 16 = log4, log base 2. Then setting parameters include: -l 4.

- My command-line for this experiment:
Output the results to terminal:
./sim-cheetah -a 7 -b 11 -l 4 -n 2 -R opt ./spec95-little/go.ss 2 8
Output the results to an output file:
./sim-cheetah -a 7 -b 11 -l 4 -n 2 -R opt ./spec95-little /go.ss 2 8 >& ./cheetahlog2.txt

B. Results

Table IV. Miss Ratio II [set vs. associativity]

set/ass	1	2	3	4
128	0.149464	0.071913	0.048650	0.037636
256	0.106230	0.048555	0.032534	0.024928
512	0.079260	0.033930	0.021733	0.015918
1024	0.050418	0.019220	0.011621	0.007627
2048	0.032724	0.010242	0.005236	0.003487

C. Discussion on Results

We firstly answer the required question:

For a direct-mapped cache (set size 1), I think there can be no difference between the results for LRU and MIN replacement policies. Because there is only one unique address for each block that I need to replace into the cache, which means no replacement policy is need for a choice.

Using the output from sim-cheetah, for caches of equivalent size, discussions show as follows:

- From the miss ratio results for associativity 1 ranging from set 128 to set 2048, we could first calculate the improvement scale on miss ratio for each change:
(set 128 to set 256)= 0.149464/0.106230=1.406985
(set 256 to set 512)= 0.106230/0.079260=1.340273
(set 512 to set 1024)= 0.079260/0.050418=1.572058
(set 1024 to set 2048)= 0.050418/0.032724=1.540704



- According to the reduction definition (1) mentioned above, we could calculate the reduction of miss ratio by using MIN as our replacement policy rather than LRU and taking the two miss ratio table results above.

Table V. Reduction on Miss Ratio [set vs. associativity]

set/ass	1	2	3	4
128	0.195898	0.239571	0.252562	0.264850
256	0.181246	0.220601	0.247751	0.269958
512	0.164972	0.218905	0.281839	0.344129
1024	0.128756	0.233591	0.345701	0.414119
2048	0.120181	0.279696	0.334181	0.277156

- From the results and computations discussed above, it is obvious that the MIN algorithm works better than the LRU algorithm. While the MIN algorithm cannot be implemented, then we could do the following to improve replacement policies based on LRU:

▲ LRU-2

The main idea of this modified algorithm is least-recent-used-twice compared to the original LRU whose main idea is least-recent- used-once.

While one more queue is needed for the implementation of the LRU-2 for access history recording. Then we name the two queues as ‘record queue’ and ‘cache queue’. They both uses the LRU algorithm. Only when the number of access history record of one data in the record queue reaches 2 can this record be changed into the cache queue.

This implementation will improve the hit rate. Meanwhile since it comes down to priority queue, the complexity and cost of memory, etc, will increase instead. Therefore, this algorithm is not suitable for large scale programs, where large memory is required, compared to the simple implementation of LRU.

▲ Two Queue

This implementation is similar to the LRU-2 above. The difference is in the first queue. We have known that ‘LRU-2’ uses a record queue as the first queue, while ‘Two Queue’ here uses a cache FIFO as the first queue instead.

Though it needs two queues, the implementation is still simple. However, the cost becomes a sum of that of FIFO and LRU.

▲ Multi Queue-2

MQ-2 algorithm divides the accessed data into 2 queues according to their access frequency. And data with highest frequency will be cached. The two queues are both managed by LRU. In order to prevent one entry from staying in one queue for too long time, a time should be determined as a measure, beyond which the priority of the data should decrease. We always choose the last data in the low priority queue as the replacement.

▲ A Little Analysis of above suggestions:

Table VI. Comparison on replacement policies

	Comparison
Hit rate	LRU-2 > MQ(2) > 2Q > LRU
Complexity	LRU-2 > MQ(2) > 2Q > LRU
Cost	LRU-2 > MQ(2) > 2Q > LRU

D. Concise Conclusion

Opt algorithm is the optimal replacement policy while it is unimplementable. We could have a variety of improvement on LRU algorithm to fit our results better close to the OPT one, including making use of more queues.

VI. PRO8__BRANCH PREDICTORS

This part the effects of branch predictors on the execution for the SPEC95 benchmark 'go' will be investigated by using the sim-outorder simulator.

A. Basic Setting and Experiment Description

[Problem Description]

- Run sim-profile to find out the distribution of instruction types for the SPEC95 benchmark 'go'. What percentage of the instructions 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).
- 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.

- 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 branch-direction 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.

[Basic Setting]

- Call: Executable file <sim-profile> in the folder <simplesim>; The integer benchmark <go> in the folder <spec95-little>; Executable file <sim-outorder> in the folder <simplesim>.

- Command line for 'go' benchmark instruction distribution checking: ./sim-profile -iclass ../spec95-little/go.ss 2 8 (The 'go.in' is omitted as it is an empty file here in the experiment.)
- Command line for branch predictors on 'go' (just take one predictor as an example): ./sim-outorder -bpred nottaken ../spec95-little/go.ss 2 8 (The 'go.in' is omitted as it is an empty file here in the experiment.)

B. Results

Table VII. Instruction Type Distribution

Instruction Type	Count	Pdf
Sim_inst_class_prof.start_dist		
Load	6164333	19.66
Store	1975290	6.30
Uncond branch	995621	3.17
Cond branch	3993773	12.74
Int computation	18229301	58.13
Fp computation	0	0.00
Trap	8	0.00
Sim_inst_class_prof.end_dist		

Table VIII. Six Branch Predictors

nottaken	taken	perfect
bimod	2lev	comb

The data shown above is got by checking the -bpred parameter list in the simulation.

Table IX. Data Recording on Branch Predictors

	Sim_IPC	Bpred_dir_rate
Nottaken	0.6539	0.3214
Taken	0.6612	0.3214
Perfect	0.1536	1.0000
Bimod	0.9708	0.8434
2lev	0.9027	0.7514
Comb	0.9766	0.8457

Sim_IPC: instruction per cycle.

Bpred_dir_rate: branch direction-prediction rate.

Bpred_dir_rate = all hits / updates

C. Discussion on Results

Discuss on instruction type distribution:

12.74% instructions executed are conditional branches. Since $(1 - 12.74\%) / 12.74\% = 6.849$, average instructions the processor executes between each pair of conditional branch instructions (do not include the conditional branch instruction) is about 6.849.

Description on six branch predictors:

The six branch predictors can be divided to two parts: static branch predictors (nottaken, taken, perfect) and dynamic branch predictors (bimod, 2lev, comb).

- Nottaken — always not taken(no more storage needed).
 Taken — always taken(no more storage needed).
 Perfect — initially it does not predict and it will directly fill the next instruction into npc. Indeed, it does not need to call the pred-lookup function. While sometimes it performs worse than bimod. And it is only implemented in outorder. This shows us the perfect situation in branch prediction. So the prediction rate should be 100%
 Bimod — the most common one. It uses 2-bit branch prediction table as the book said and takes branch address as reference for searching. In this way, there is only one parameter —the length of the prediction table.

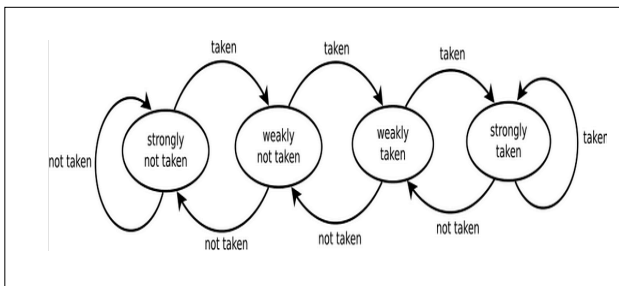


Fig. 2. Bimod table graph

- 2lev — two-level adaptive predictor. It uses two-level table. First level is a branch record table which maintains the branch history; Second level is a branch mode table which stores the 2-bit branch prediction mode for branch history. When predicting one instruction, using the its value in the first level table to index the second level table for corresponding prediction. There are four parameters in total.

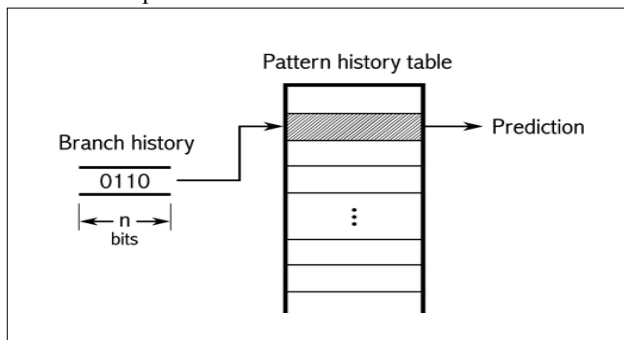


Fig. 3. 2lev table graph

- Comb — a combination of the last two methods and it introduces a meta table. Meta table is similar to the prediction table in the bimod, while it predicts whether using bimod or 2lev by 2-bit. The selected prediction method is called the first direction, and the other one is the

second. When updating, if the first direction differs from the second, then updating the meta table; else just updating their own table.

Discussion on relation between dir_rate and IPC:

IPC reflects the parallelism of the execution. From the results shown in the Table IX above, it seems that the comb simulator performs the best.

This may simply be the results for 'go' benchmark. It cannot generalize for all programs. For instance, when the instruction set includes three more if statements, the 2lev predictor will work better.

D. Extra Problem

[Problem Description]

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 by 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.

[Solution]

- 1) Use 'search key word 'perfect'' to find the following code in 'sim-outorder.c' file (Line 897-Line 902).

```

if (!mystricmp(pred_type, "perfect"))
{
    /* perfect predictor */
    pred = NULL;
    pred_perfect = TRUE;
}
  
```

- 2) Modify the 'sim-outorder.c' by adding -bpred none.

```

else
{
    if (!mystricmp(pred_type, "none"))
    {
        /* none predictor */
        pred = NULL;
        pred_perfect = FALSE;
    }
}
  
```

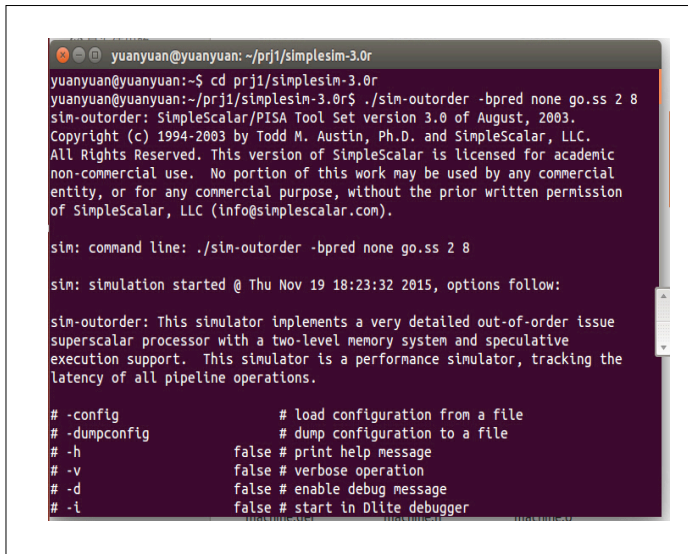
- 3) After modification, you should first delete the .o and the executable file of 'sim-outorder'. And use the command line 'make config-pisa' first in the terminal. Then use the command line 'make' in the terminal. (The process is just the same as the initial install steps

mentioned.) As a result, you will get a new executable file 'sim-outorder'.

4) use the command line

`./sim-outorder -bpred none go.ss 2 8`

The simulation is shown as follows:



```
yuanyuan@yuanyuan: ~/prj1/simplesim-3.0r
yuanyuan@yuanyuan:~/prj1/simplesim-3.0r$ ./sim-outorder -bpred none go.ss 2 8
sim-outorder: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use. No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

sim: command line: ./sim-outorder -bpred none go.ss 2 8

sim: simulation started @ Thu Nov 19 18:23:32 2015, options follow:

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support. This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config          # load configuration from a file
# -dumpconfig      # dump configuration to a file
# -h              false # print help message
# -v              false # verbose operation
# -d              false # enable debug message
# -i              false # start in Dlite debugger
```

Fig. 4. Parameter 'none' simulation

E. Concise Conclusion

Branch prediction is critically benefit to increase the extent of parallelism corresponding to instruction per cycle. Dynamic predictors perform better than static predictors at most cases. Among dynamic predictors, the performance queue depends on specific programs.