Software-based and Hardware-based Branch Prediction Strategies and Performance Evaluation

Gang Luo (gang@cs.widc.edu)
Hongfei Guo (guo@cs.wisc.edu)

Abstract

In a highly parallel computer system, performance losses due to conditional branch instructions can be minimized by branch prediction to fetch and issue subsequent instructions before the actual branch outcome is known. This paper discusses several software-based static and hardware-based dynamic branch prediction strategies and uses the modified release 3.0 of the SimpleScalar simulation tool set to evaluate their performance. According to our test result, the hardware-based dynamic branch prediction strategies always achieve high prediction accuracy than the software-based static branch prediction strategies.

1. Introduction

It is well known that in a highly parallel computer system, branch instructions can break the smooth flow of instruction fetching, decoding and execution. This results in delay, because the instruction issuing must often wait until the actual branch outcome is known. To make things worse, the deeper the pipelining is, the greater performance loss is.

To reduce delay, one can try to predict the direction that a branch instruction will take and begin fetching, decoding and issuing instructions before the branch decision is made. However, a wrong branch prediction may lead to more delay because the wrongly fetched instructions occupy the useful functional units, like the ALU, reservation stations, and memory bus. This leads to the need for highly accurate branch prediction strategies.

Branch prediction strategies can be divided into two basic categories: software-based static branch prediction strategy and hardware-based dynamic branch prediction strategy.

The most widely used software-based static branch prediction strategies are:

1. Taken: Predict that all branches will be taken.
2. Not-taken: Predict that all branches won't be taken.
3. Back-taken: Predict that all backward branches will be taken; predict that all forward branches won't be taken.
4. Predict that all branches with certain operation codes will be taken; Predict that the others won't be taken.

The most widely used hardware-based dynamic branch prediction strategies are:

1. One-bit: One-bit branch prediction buffer.
2. Two-bit: Two-bit branch prediction counter.
3. GAg.
4. PAg.
5. PAp.
6. Branch instruction table.

Since different data sets will let the programs have different dynamic branch behaviors, so usually hardware-based dynamic branch prediction strategies have better prediction accuracy compared to the software-based static ones.

In this paper, we discuss several representative software-based static and hardware-based dynamic branch prediction strategies. Due to the wide variation in branching behavior between different application programs, there exists no good analytical model for evaluating the performance of the branch prediction strategies. So we utilize the SimpleScalar Tool Set Version 3.0, which is a widely used simulation tool set developed at the computer sciences department of University of Wisconsin-Madison, and some benchmark programs to do performance evaluation.

Since the meaning of the software-based static branch prediction strategies are quite obvious, we only need to present the meanings of the different hardware-based branch prediction strategies in detail. Then, in section 4, we analyze the performance of the different branch prediction strategies.

2. Hardware-based Dynamic Branch Prediction Strategies

2.1. One-bit Branch Prediction Buffer

This is the simplest dynamic branch prediction schema. The branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. It contains a bit that says whether the branch was recently taken or not, which is the hint for the next branch instruction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This schema has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken.

To remedy this, the two-bit prediction schema is proposed.

2.2. Two-bit Branch Prediction Counter

The two-bit schema is a specialization of the n-bit schema. Since the two-bit predictors do almost as well as the n-bit predictors, we just rely on two-bit branch predictors rather than the more general n-bit ones. The two-bit predictor is essentially a two-bit counter which takes values between 0 and 3: when the counter is greater than or equal to one half of its maximum value 2, the branch is predicted as taken; otherwise, it is predicted not-taken. The counter is incremented on a taken branch and decremented on a not-taken branch. The finite-state machine for this schema is shown in figure 1.

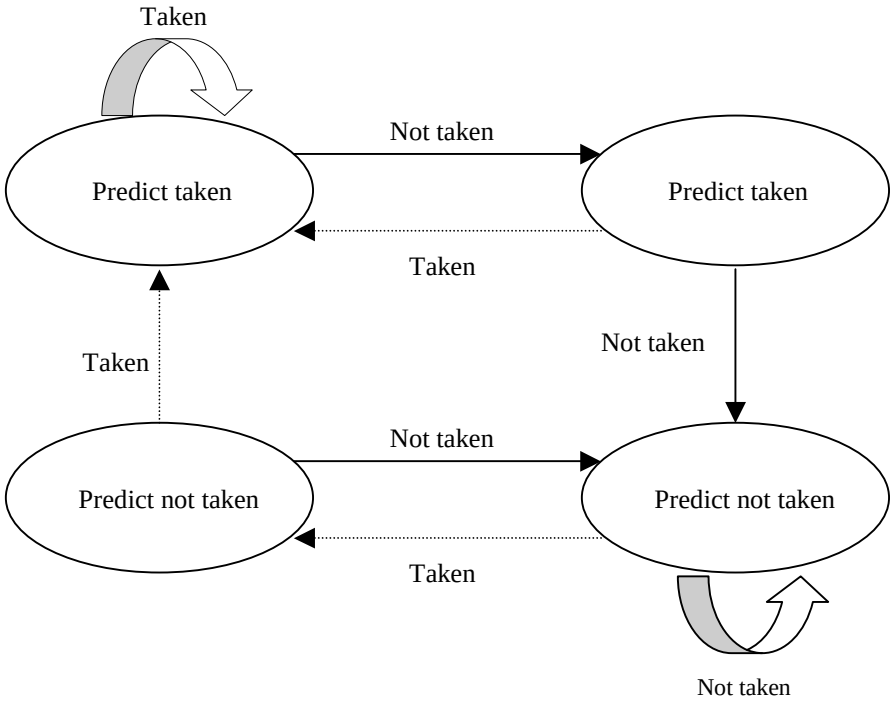

Figure 1. The states in a two-bit branch prediction counter.

The two-bit predictor schema uses only the recent behavior of a branch to predict the future behavior of that branch. Since sometimes in the program, the branch behavior of different instructions are related together, it is possible to improve the prediction accuracy if we also look at the recent behavior of other branches rather than just the branch we are trying to predict, which leads to GAg.

2.3. GAg

In the GAg schema, the global history of the most recent m branches is recorded in an m-bit shift register, named global branch history register, where each bit records whether the branch was taken or not taken. We have another global branch history pattern table, which has 2^m entries, each corresponding to the 2-bit counter for a global branch history. The prediction of a branch is based on the history pattern of the most

recent m branches. That is, the value of the global branch history register is used to query the global branch history pattern table to find the corresponding 2-bit counter. When the counter is greater than or equal to one half of its maximum value 2, the branch is predicted as taken; otherwise, it is predicted not-taken. After the conditional branch is resolved, the outcome is shifted left into the global branch history register, and the 2-bit counter is incremented on a taken branch and decremented on a not-taken branch.

2.4. PAg

In stead of using one global branch history register for all the branches as in the GAg, we can have a branch history register for each branch. That is, we have a branch history register table. The address of a conditional branch is used for hashing into this table. Each entry in this table is a branch history register with m bits, recording whether the most recent m branches corresponding to this entry are taken or not. We have another global branch history pattern table which is the same with PAg. The prediction of a branch is based on the history pattern of the last k outcomes of executing the branch. Whenever a conditional branch is encountered, we find the corresponding entry in the branch history register table and get the branch history register. The branch history register is used to address the global branch history pattern table to make the prediction. After the conditional branch is resolved, the outcome is shifted left into the branch history register in the least significant bit position and is also used to update the 2-bit counter in the global branch history pattern table entry.

2.5. PAp

In stead of having one global branch history pattern table for all the branches, we can have a branch history pattern table for each branch. That is, for each branch, there is a branch history register and a branch history pattern table. Whenever a conditional branch is encountered, we find the corresponding entry in the branch history register table, which is used to index the branch history pattern table for that branch to find the 2-bit

counter and make the prediction. After the conditional branch is resolved, the outcome is shifted left into the branch history register in the least significant bit position and is also used to update the 2-bit counter in the branch history pattern table entry.

2.6. Branch Instruction Table

Maintain a table of the most recently used conditional branch instructions that are not taken. If a conditional branch instruction is in the table, predict that it will not be taken; otherwise predict that it will be taken. Purge table entries if they are taken, and use LRU replacement to add new entries.

3. Implementation based on SimpleScalar Tool Set 3.0

3.1. SimpleScalar 3.0 branch prediction simulator

The SimpleScalar 3.0 branch prediction simulator implements a number of state-of-art branch prediction mechanisms. As regards to static branch prediction strategies, it supports branch always taken, and branch always not taken. For dynamic branch prediction strategies, it implements simple directly mapped bimodal predictor and two level adaptive branch predictor, which again includes GAg, GAp, PAg, and PAp.

SimpleScalar 3.0 branch predictor assigns a branch target buffer associated with each dynamic branch predictor. It records the latest taken branches and their target addresses. The branch prediction simulator works in the following way:

1. Create the branch predictor instance according to the command line parameters;
2. Fetch an instruction;
3. If the instruction is not a branch, go to 2;

4. Call the procedure bpred_lookup() to get predicted PC;
5. Call the procedure bpred_update() to update the branch predictor states, which include the BTB, the level 1 table, and the level 2 table, according to the comparison result between the predicted PC and the actual PC to be executed.
6. If (! the end of program is reached) goto 2;
7. Print out the branch predictor statistics.

3.2. Our work based on SimpleScalar 3.0 branch prediction simulator

In order to get a more comprehensive simulation result we add two branch prediction strategies into the simulator. One is a static branch predictor: Backward always taken, forward always not taken; the other is a one-bit branch prediction counter. Without changing the working flow of the simulator, we implemented these two strategies in the branch prediction module – bpred.c. Also, SimpleScalar 3.0 tries to support setting the branch predictor configuration with the command line parameters, but it fails to do so somehow. In our implementation, this bug is fixed. Accordingly, we modified the module options.c.

Following, we illustrate the working flow of each branch predictor one by one:

3.2.1 Static branch predictor

a. Always taken branch predictor

Algorithm: Always gives the branch target address as output.

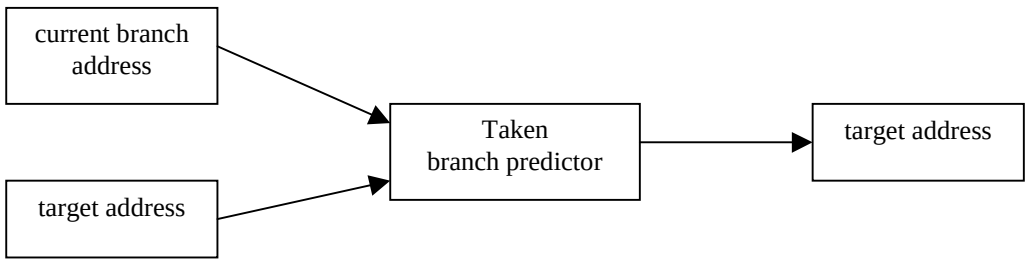

Figure 2. Always taken branch predictor

b. Always not taken branch predictor

Algorithm: Always give the next instruction address as output.

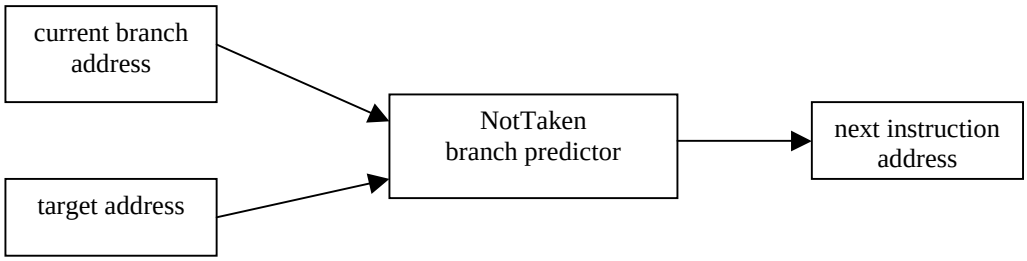

Figure 3. Always not taken branch predictor

c. Backward taken & forward not taken branch predictor

Algorithm:

```
if  (current branch address > target address)
    return  target address;
else
    return   next instruction address;
```

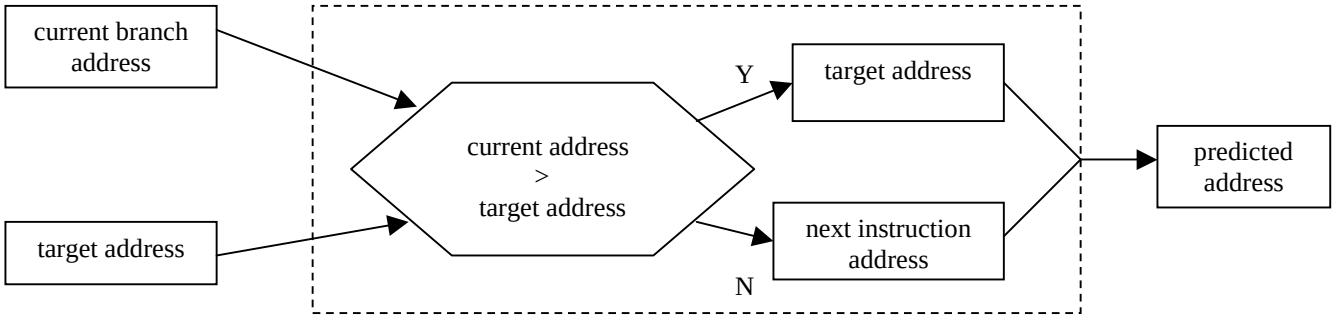

Figure 4. Backward taken & forward not taken branch predictor

3.2.2 Branch Prediction Counter

It includes the one-bit branch prediction counter and the two-bit branch prediction counter.

Algorithm:

1. Use the current branch address as the index to find the counter entry distributed in a hash table, then get the predicted direction;
2. Look up in the BTB to see if there is an entry for this branch;
3. if (predict taken) {

```
    if (find target address in BTB)
        return target address;
    else
        return 1;          //predict taken, but don't know target address yet
}
else
        return 0;          //predict not taken
```

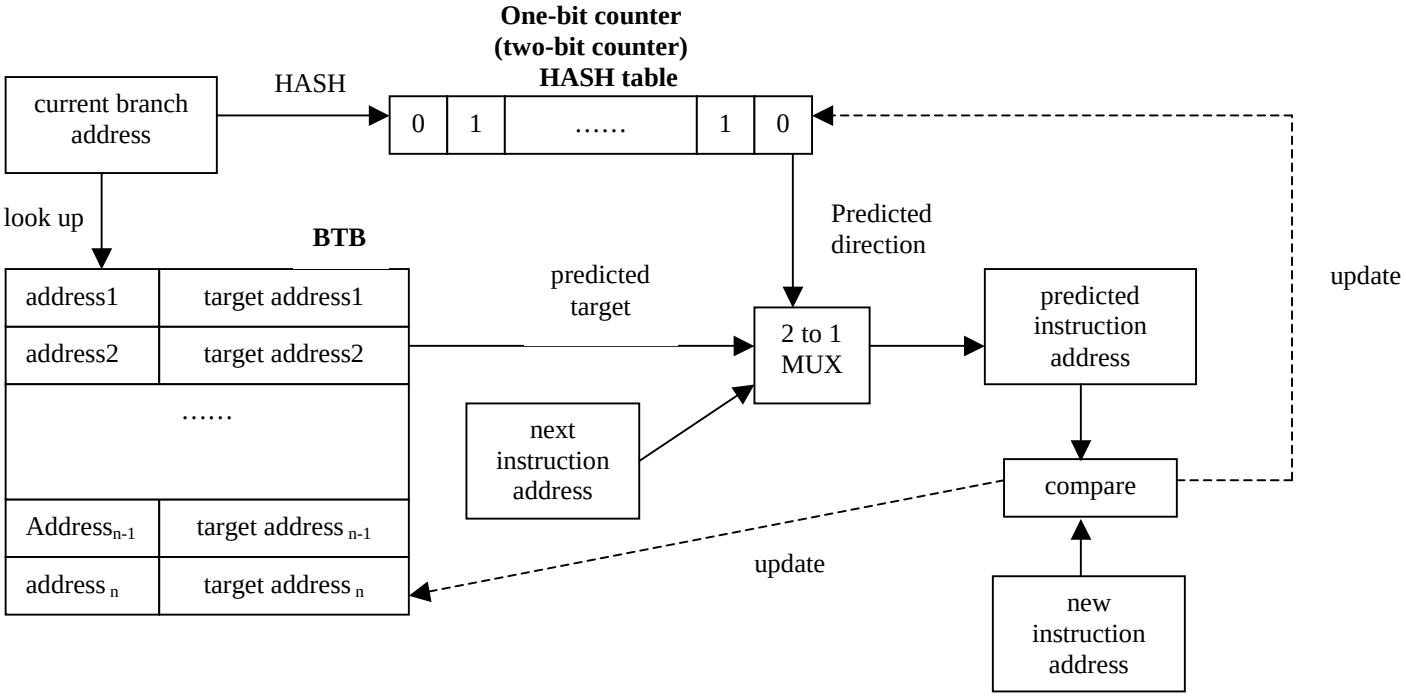

Figure 5. Branch prediction counter

3.2.3 Two-level Adaptive Branch predictor

It includes the GAg, GAp, PAg, and PAp branch predictors, whose type is decided by setting the command line configuration parameters.

Algorithm:

1. Use the current branch address as the index to find the entry in level 1 table;
2. Use the entry content we get as the index to find the level 2 table entry, then get the predicted direction;
3. Lookup in the BTB to see if there is an entry for this branch;
4. if (predict taken) {

```
    if (find target address in BTB)
        return target address;
    else
        return 1;          //predict taken, but don't know target address yet
}
else
        return 0;          //predict not taken
```

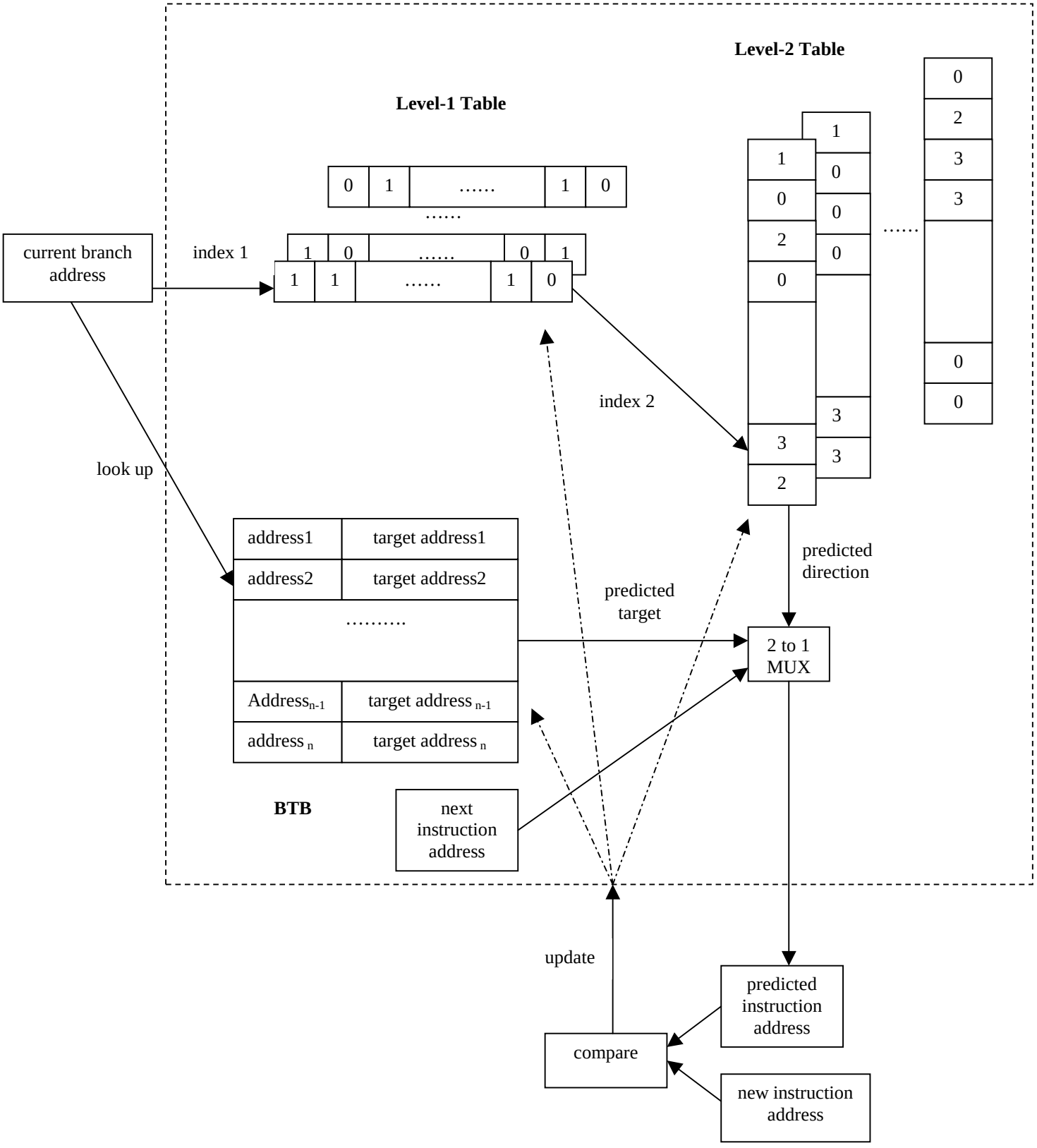

Figure 6. Two-level adaptive branch predictor

4. Simulation and Performance Evaluation

4.1. Methodology and Simulation Model

The simulation in our study is conducted using our modified version of the SimpleScalar tool set version 3.0, which is developed in the computer sciences department of University of Wisconsin-Madison. This simulation tool set offers both detailed and high-performance simulation of modern microprocessors.

Totally, three software-based static branch prediction strategies and five hardware-based dynamic branch prediction strategies are simulated. Data are collected on the branch prediction accuracy for each of the eight branch prediction strategies with five different Spec95 integer benchmark programs: Gcc, Compress, Li, M88k, and Perl. The integer benchmarks tend to have many conditional branches and irregular branch behavior. Therefore, the mettle of the branch predictor can be well tested using these integer benchmarks.

The numbers of dynamic instructions simulated for these benchmarks range from 3.6 million to 960 million. The distribution of the dynamic instructions and the dynamic conditional branches in these benchmarks are shown in Table 1.

The configuration and scheme of each simulation model in our study are listed in Table 1.

The branch target buffer has 512 entries, and uses 4-set association for each entry. In order to make the simulation results comparable, we assigned the same space cost to each dynamic branch predictor.

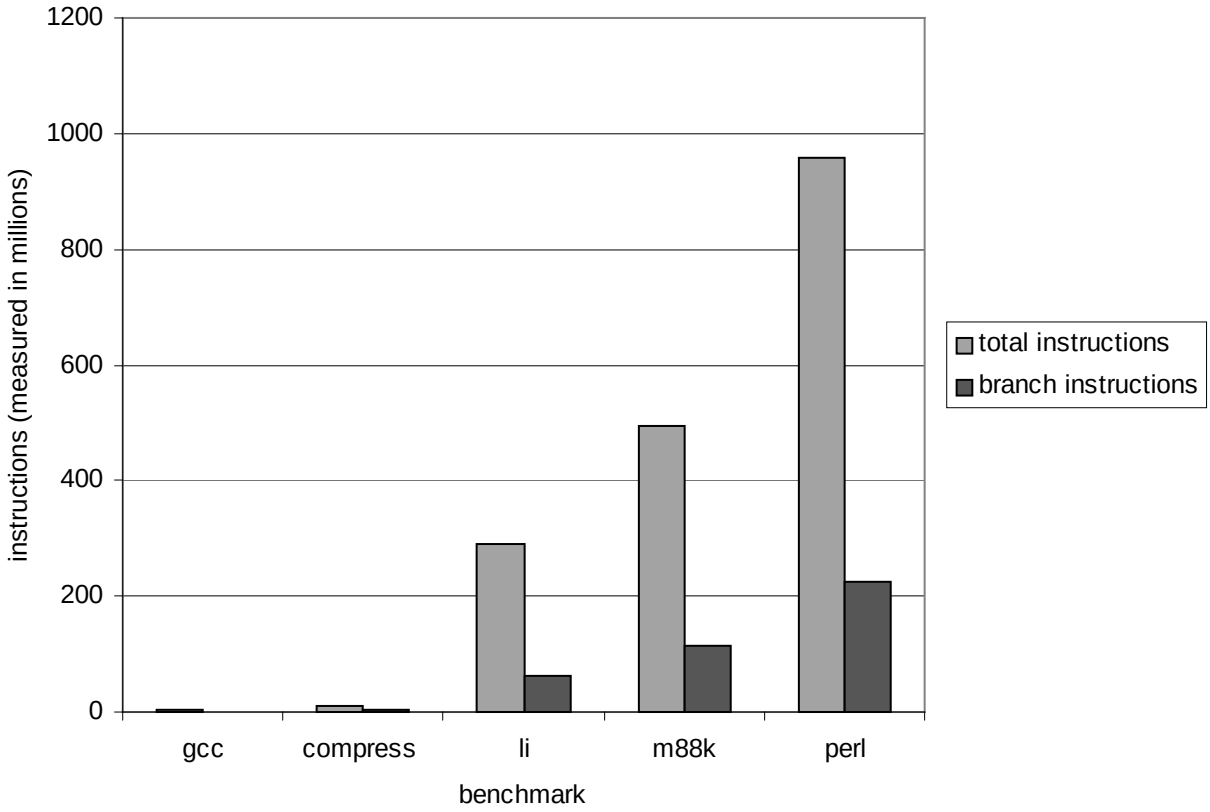

Figure 7. Distribution of dynamic instructions and branch instructions

Mode	Level 1 Table		Level 2 Table	
	# of Entries	Width of SR	# of Entries	Entry Content
One-bit	--	--	2048	1-bit counter
Bimode	--	--	2048	2-bit counter
GAg	1	11	2048	2-bit counter
PAg	4	11	2048	2-bit counter
PAp-1	4	7	2048	2-bit counter
PAp-2	8	3	2048	2-bit counter
BTB	512	Associ. 4	--	--

Table 1. Configurations of simulated branch predictors

4.2. Experimental Results and Analysis

We ran the five Spec95 benchmarks on our modified version of SimpleScalar. Table 2 lists the percentages of correct branch predictions from different predictors.

	Gcc	Compress	Li	M88k	Perl
Taken	**66.66**%	**82.22**%	64.77%	**82.52**%	62.96%
Not Taken	53.40%	35.43%	**69.71**%	36.00%	**71.64**%
Back Taken	47.02%	72.09%	51.22%	47.22%	57.59%
One bit	85.86%	95.04%	89.21%	93.01%	93.63%
Two bit	**89.98**%	**97.06**%	92.11%	**95.33**%	95.34%
GAg	89.05%	96.81%	**95.36**%	94.27%	**95.96**%
PAg	83.91%	95.59%	91.49%	90.24%	94.54%
PAp-1	87.25%	96.42%	93.23%	93.90%	95.30%
PAp-2	**90.17%**	**97.18%**	94.52%	**95.42%**	95.82%

Table 2. Percentages of Correct Branch Predictions from Different Predictors.

4.2.1. Performance Analysis of Software-based Static Branch Predictors

The performance of three software-based static branch predictors is shown in figure 8. On average, the Taken branch predictor gets the highest branch prediction accuracy, while the Not-taken and Back-taken branch predictors achieve equally worse performance. This can be explained in the following way. The Back-taken branch predictor is mainly pointed at the for loop statement and the while loop statement. It doesn't work well for other branch statements. Also, in the programs, the branch statements are more inclined to be taken rather than not to be taken. So, the Taken branch prediction strategy usually works better than the other two software-based static branch prediction strategies.

4.2.2. Performance Analysis of Hardware-based Dynamic Branch Predictors

The performance of the five hardware-based dynamic branch predictors is shown in figure 9. On average, all the five different hardware-based dynamic branch strategies achieve prediction accuracy better than 90%. Among all of them, the PAp-2 branch predictor and the Two-bit branch predictor are the best two. This can be explained in the following way. Since the One-bit branch predictor has an intrinsic shortcoming as described in section 2.1, the Two-bit branch predictor always works better than the One-bit branch predictor. For the PAp branch predictor, there is a two-bit counter for each global branch history pattern of each branch instruction, with our configuration each history pattern could get enough training to achieve the highest prediction accuracy. For the Two-bit branch predictor, there is a two-bit counter for each instruction. The number of the two-bit counters is large enough to ensure the high prediction accuracy.

As regards to the 2-level adaptive branch predictors, at the same space cost, PAp achieves the highest prediction accuracy with the shorter register; its prediction accuracy decreased when we made the shift register wide. We can explain this as follows. At the same space cost, with a longer shift register, since there are too many branch history patterns compared to the number of instructions executed, each branch history pattern can't get enough training to reach high performance. Therefore, PAp with a shorter shift register achieves higher prediction accuracy. While PAp with a longer shift register achieves even less prediction accuracy than GAg does.

For the PAg branch predictor, the branch histories of all the branch instructions share the same global two-bit counter tables, which leads to confusion. Therefore, it can't achieve as high performance as the GAg branch predictor.

4.2.3. Performance Analysis of Software-based Static Branch Predictors vs. Hardware-based Dynamic Branch Predictors

Comparing figure 8 and figure 9, we can see that the hardware-based dynamic branch predictors achieve much higher prediction accuracy that the software-based static branch predictors do. This can be explained in the following way. Since different data sets will let the programs have different dynamic branch behaviors, so the branch behaviors of the programs can't be simply predicted statically quite well. Even for the same data set, since the same branch instruction will be executed several times during the execution of the program and exhibits different dynamic branch patterns for the different times in the same running, its branch behaviors can't be simply predicted statically. That is, its branch behaviors are changed dynamically. So, the hardware-based dynamic branch predictors always work better than the software-based static branch predictors do.

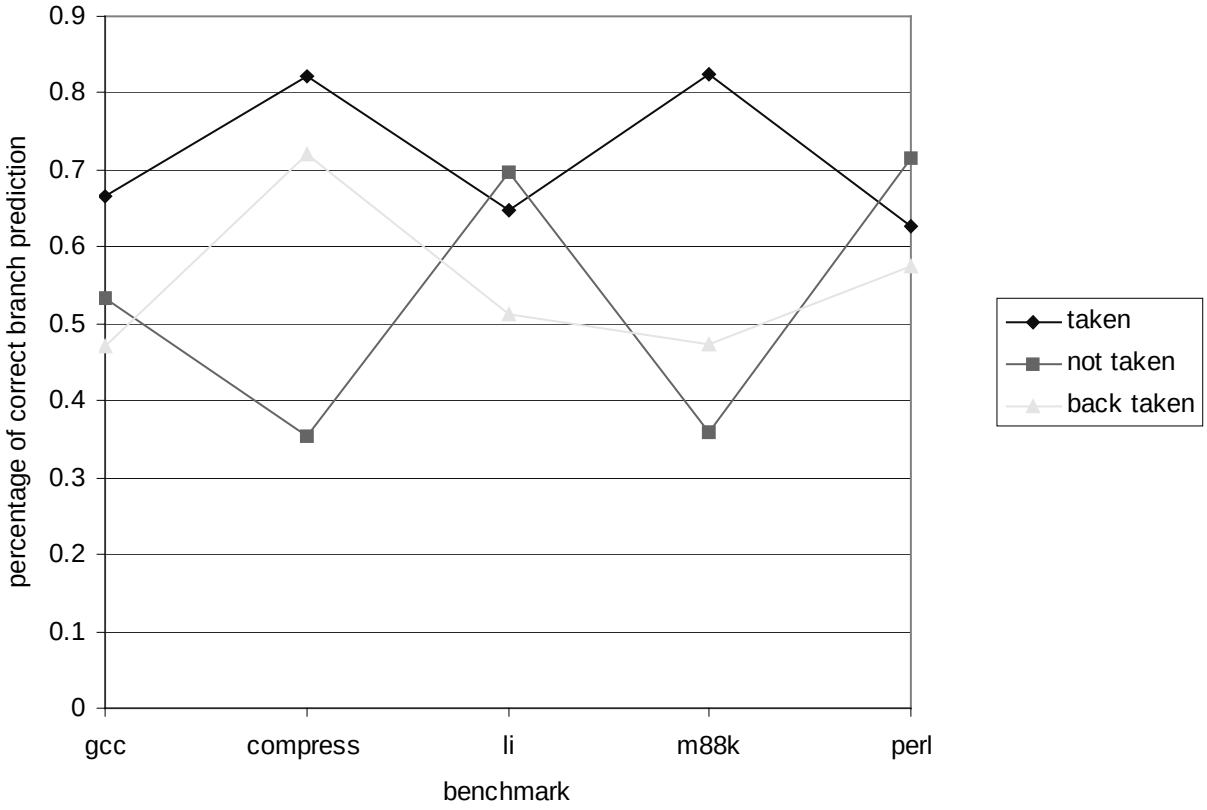

Figure 8. Software-based Static Branch Prediction Accuracy

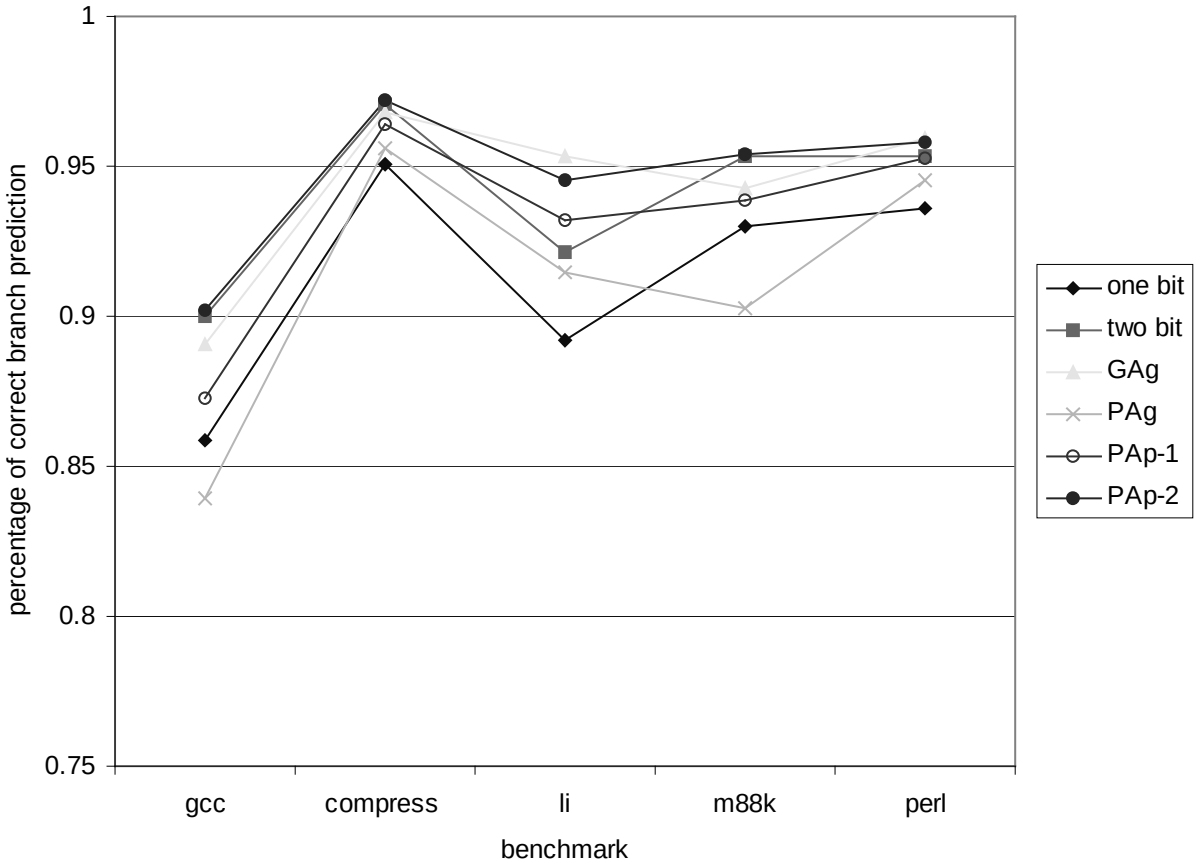

Figure 9. Hardware-based Dynamic Branch Prediction Accuracy.

5. Conclusion and Future Work

Branches in the programs are highly predictable, yet a wrong branch prediction may lead to more delay because the wrongly fetched instructions occupy the useful functional units. So we need some specific strategies to achieve high prediction accuracy. In this paper, we implement several representative software-based static and hardware-based dynamic branch prediction strategies and use the simulation tool set SimpleScalar Version 3.0 to evaluate their performance. According to our test result, the hardware-based dynamic branch prediction strategies always achieve high prediction accuracy than the software-based static branch prediction strategies.

At present, all the branch prediction strategies are quite simple. If some AI algorithms can be combined into them, then we can expect even higher prediction accuracy. That is, we can use the AI algorithms to adjust the parameters of the branch prediction strategies (such as the size of the branch history register table, the size of the branch history pattern table, and the length of the branch history register). Then we can ensure the most proper training time to achieve the highest performance.

References

1. David A. Patterson, and John L. Hennessy. Computer Architecture a Quantitative Approach, second Edition, Morgan Kaufmann Publishers Inc., 1996.
2. Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. Proceedings of the 24th Annual International Symposium on Microarchitecture, Nov. 1991.
3. J.E.Smith. A Study of Branch Prediction Strategies. Proceedings of the 8th International Symposium on Computer Architecture, May 1981.
4. K.C.Yeager. The MIPS R10000 Superscalar Microprocessor. IEEE Micro, April 1996.
5. Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin-Madison, WI, June 1997.