Advanced Engineering Forum *Online: 2011-09-09*
ISSN: 2234-991X, Vol. 1, pp 241-246
doi:10.4028/www.scientific.net/AEF.1.241

Researches on Design and Implementations of two 2-bit Predictors

Zhang Long , Fang Tao , Xiang Jinfeng

Lenovo Mobile Communication Technology Limited Haidian District, BeijingP.R. China
School of Adult Education Henan University of Economics and Law Zhengzhou, Henan P.R. China
School of Computer Engineering & Science Shanghai University, Shanghai P.R. China
zhanglong@lenovomobile.com, zzytf@sina.com.cn, xiangjinfeng@shu.edu.cn

Keywords: Branch Prediction; Two-bit Predictor; Instruction Level Parallelism;

Abstract: High-performance superscalar processors, which can be used in building base-stations of cell phones and cloud-based web servers, rely heavily on aggressive branch predictors to avoid stalls no matter the branch is taken or not. Dynamic branch predictor which is based on the historical records of the previously executed branches always outputs good performance. Two-bit predictor discussed in this paper is one of the most popular ones that always practically employed. This paper demonstrates two implementations of two-bit predictors. Using BTB to process solo conditional branches is the first one, which is normally used. Other categories of branch instructions may results in several bubbles as the penalty. While the other implementation employs BTB dealing with all kinds of branch instructions, generating target addresses without any delay cycle. Simulation results show that the second implementation has much better performance than the former one. It decreases the mis-prediction rate from 12.26% to 11.48%, and also has much higher prediction accuracy on indirect jumps. With these results, we have our predictor re-designed accordingly and implemented successfully in superscalar processors.

I. Introduction

Nowadays, base-stations of mobile system and their cloud-based web servers rely heavily on high-performance processors. In order to increase instruction level parallelism (ILP), both superscalar and deep pipeline techniques are employed, controlling and preventing hazards from taking advantages of all the available ILP. Within superscalar and deep pipeline techniques, branch prediction is proved to be critical to performance, and it's one of the most useful mechanisms to overcome control hazards.

The performance of branch prediction depends on the prediction accuracy and the cost of mis-predicton. Prediction accuracy can be improved by inventing better branch predictors. Branch prediction strategies [5] can be divided into two basic categories, depending on whether or not past history is used for making a prediction. Static branch prediction can not make use of the past history. It comprises machine-fixed prediction and complier-driven prediction. While the dynamic branch prediction can dynamically alter the branch prediction depends on the past history. It always has a superior performance than static branch prediction.

Dynamic branch predictors are dominantly employed now. Most of the popular strategies contain two-bit Prediction, Two-Level adaptive branch prediction, hybrid predictor and neural predictor. Two-bit Prediction [5], which is one of the most traditional methods, has been widely used in practice. Several multiprocessors such as PowerPC 604, MIPS R10000, Cyrix 6x86, take use of the two-bit predictors and achieve high-performance.

In this paper, we study two implementations of two-bit predictors. The first one has a branch history table (BHT) to get a direction prediction and a branch target buffer (BTB) comes up with a target address prediction. The BTB only covers conditional branches. While the second one conserves a BTB to perform direction target address prediction together. BTB covers conditional branches, unconditional branches as well as indirect jumps. Experiments results show the second implementation has a higher performance. Compared with the first implementation, it cuts off the mis-prediction rate from 12.46% to 11.48%.

The rest of this paper is organized as follows: section 2 provides the details of two-bit predictor we will discuss in this paper. Section 3 describes the alpha ISA, instruction pipeline and the details of the implementations about two-bit predictor. In Section 4, simulation results and analysis are illustrated. Also the pros and cons of the two implementations are discussed in detail. In Section 5 we draw conclusions.

II. Implementations of two-bit Predictors

Two-bit prediction is one of the most traditional dynamic predictions, which can be implemented in two methods. One is by assigning two bits to each entry of the Branch Target Buffer (BTB). Another is use a separate Branch History Table (BHT) to conserve the counter values. The two-bit prediction scheme can be extended to n-bit scheme, but Hennessy and Patterson[1] noted that 3 or more bits do not make much significant than two-bit counter does. Two-bit predictor's advantages are easy to implement and has a low hardware delay, but it always holds lower prediction accuracy than two-level predictor and hybrid predictor.

In this section, we expose the implementations details of the two different two-bit predictors, including the elements and prediction algorithm. The two-bit predictors' implementations are based on alpha ISA.

A. *Control Instructions of Alpha ISA*

Alpha architecture is a 64-bit load and store RISC architecture designed with particular emphasis on multiple instruction issue. In this paper, alpha ISA is selected for the simulator which is used to evaluate the performance of the predictors.

Branch prediction is act on the control instructions. Alpha's control instructions can be categorized into three classifications: conditional branch, unconditional branch and indirect jumps[6]. Table 1 gives a list of these three kinds of control instructions.

TABLE I. LIST OF THE THREE CLASSIFICATIONS OF CONTROL INSTRUCTIONS

Classification	Instructions
Conditional Branch	BEQ, BGE, BGT, BLBC, BLBS, BLE, BLT, BNE, FBEQ, FBGE, FBGT, FBLE, FBLT, FBNE
Unconditional Branch	BR, BSR
Indirect Jump	JMP, JSR, RET, JSR_COROUTINE

Conditional branch is taken when the specified relationship is true, and the new program counter (PC) is loaded with the target address which is calculated by the displacement and current PC; otherwise, execution the next sequential instruction continuately.

Unconditional branch is always taken. The displacement is added to the updated PC to form the target address.

Indirect Jump is always taken too, but the new PC is supplied from a register. Indirect jump has hint information which is filled by compiler. Correct setting of these hint bits can improve prediction performance.

Note that BSR, JSR and JSR_COROUTINE need to push the PC into the return address stack (RAS), RET and JSR_COROUTINE require to pop a new PC form the RAS.

B. *Pipeline of the Fetch Engine*

The simulator used here has a instruction pipeline comprised of branch prediction, itag access, icache access, instruction decode, registers map, issue, registers read, executed and retire. This section focuses on the fetch engine which is closely-related with the branch prediction.

The stages of the fetch engine including Branch Prediction, ITag Access, ICache Access and Instruction Decode, they are several front stages of the pipeline. Fig. 1 shows these stages.

Fetch engine is in charge of instruction fetch and generation of the predicted branch target address. Following are the function description of every stage:

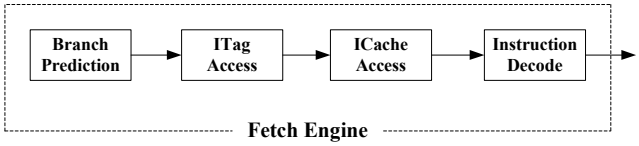


Figure 1. Stages of fetch engine

Branch Prediction (BP): BP stage is used to predict a branch instruction target address. It reads the BHT and BTB, and gets a target address with the delay is one cycle. Up to four aligned instructions can be fetched from ICache per cycle.

ITag Access (ITA): ITA stage accesses the ITag which will be used to read the data from ICache. The delay of this stage is one cycle when hit the right ITag.

ICache Access (ICA): ICA stage accesses the data from ICache according to the ITag provided by ITA. The delay of this stage is one cycle.

Instruction Decode (ID): ID can decode two instructions per cycle. The instruction's opcode and operand will be obtained. The delay of this stage is one cycle.

C. First Implementation of Two-bit Predictor

The first implementation of two-bit predictor (see Fig. 2) has a 32-entry, fully-associative BTB, and a separate BHT which is composed of 2K entry table of two-bit saturating counters.

BHT performs direction prediction and BTB does target address prediction. Only the conditional branch's target address is conserved in BTB. In BP stage, the predictor gets a target address when the BHT's predicted direction is taken and a BTB entry is hit. In ID stage, the ICU repairs the target address if needed, the RAS operation are processed here too.

The details of the first implementation of the two-bit predictor are described below:

- BP Stage

In BP stage, bits [14:4] of PC are used as the index to read BHT. If the counter's value is bigger than 1, then the prediction is taken, otherwise is not taken.

Access an entry in BTB by tag when the prediction is taken. Get target address if hit the right entry and deliver it to the following stages. Otherwise deliver the missing information to following stages too.

A sequential address is predicted when the prediction is not taken. Deliver the address to the following stages.

- ID Stage

In this stage, instruction's opcode and operand can be known. So the instruction control unit (ICU) can calculate the branch address by displacement or hint.

Depending on the information delivered from BP stage, for different kinds of branches, ID stage has variant processes:

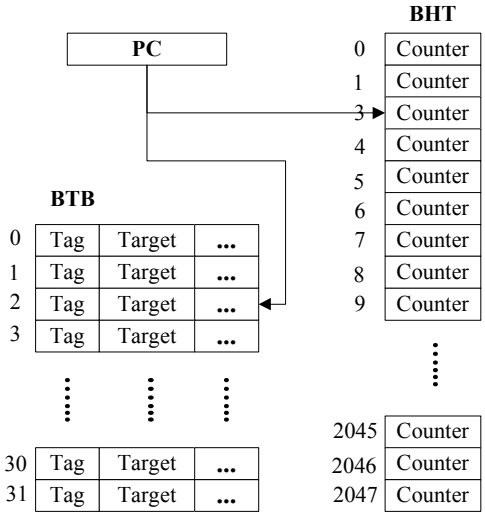


Figure 2. The structure of the first implementation of two-bit predictor

1). for conditional branch, ICU calculates the real target address by instruction's displacement when the BHT prediction is taken. Compare this real target address with the predicted target address. Repair the predicted target address when they are not equal.

2). for unconditional branch, don't care the prediction of BP stage, repair the target address by the address calculated by ICU directly. Push the sequential PC into the RAS when the current instruction is BSR.

3). for JMP and JSR, ICU repairs the target address by the address calculated from hint bits and the current PC. Do a push operation on RAS when met a JSR.

4). for RET and JSR_COROUTINE, the branch address must pop out from return address stack (RAS).

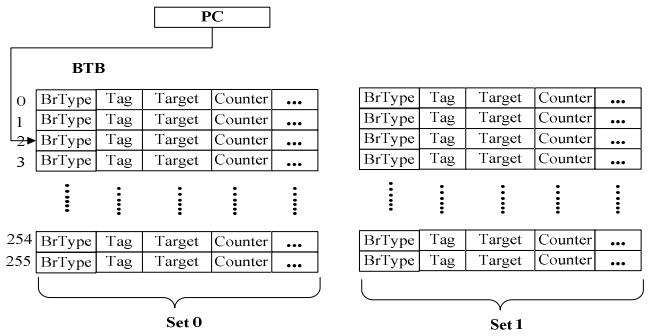
The limitations of this predictor are that it has a lot of bubbles and ICU is complex because of repair. Use repair penalty to measure the bubbles of this implementation. The repair penalty as given is measured from the cycle after the ID stage of the instruction which triggers the repair, to the ID stage of the new target, and ignoring any instruction pipeline stalls or queuing delay that the triggering instruction might experience. The repair penalty of this implementation is four cycles. It means for all unconditional branch and indirect jump instructions, and for some conditional branch instructions, four cycles' bubbles have to been induced.

D. Second Implementation of Two-bit Predictor

The second implementation of two-bit predictor (see Fig. 3) has a 512-entry, 2-way set-associative BTB with every entry has a two-bit counter.

BTB performs both direction prediction and target address prediction. For conditional branch, BrType is 0. For unconditional branch, JMP and JSR, BrType is 1. In BP stage, the predictor gets a target address when an entry is matched. In ID stage, the ICU repairs the target address if needed, the RAS operation are processed here too.

The details of the second implementation of the two-bit predictor are described below:



The structure of the second implementation of two-bit predictor

- BP Stage

In BP stage, bits [11:4] of PC are used as the index to read BTB, bits [17:12] are used the tag to pattern an entry. If the tag matched, read the hit entry out, otherwise a sequential address is predicted.

When BrType is 0 and the counter's value is bigger than 1, compose a branch target address by the "Target" read from BTB. If the counter's value is smaller than 2, a sequential address is predicted.

When BrType is 1, compose a branch target address by the "Target" read from BTB.

- ID Stage

In this stage just hold two kinds of process about the prediction. One case is when the instruction is JSR, BSR, RET or JSR_COROUTINE, it needs to do a push or pop operations on RAS. Another case is when the instruction is really a non-branch, but the predictor predicted it as a branch instruction with BrType is 1 or 0, in this situation ICU will repair the branch address to a sequential one.

Compare to the first implementation, this implementation has few bubbles and ICU's repair operation is reduced. The repair penalty of this implementation is four cycles too, but repair operation will be happened less than the first implementation.

III. Simulation Results

In this section we quantify the performance of first implementation and second implementation. We build a simulator which is based on alpha ISA to evaluate the effect of different branch prediction mechanisms. The simulator has a pipeline structure introduced in section3.2, and realized the branch prediction strategies described in section3.3 and section3.4. The simulator's fetch width is 2, and issue width is 3. It can issue 2 integer instructions and 1 float instruction per cycle. ICache has a size of 32KB and DCache is 512KB. Our simulations use a set of ten integer programs of SPEC2000[14] in a test size.

A. Performance Analysis

Fig. 4 shows the misprediction rate of the two implementations. 175.vpr(1) and 175.vpr(2) are the same SPEC2000 benchmark with different inputs. The average misprediction rate of first implementation is 12.26%, and the second implementation is 11.48%. The total execution cycle (see Fig. 5) of the first implementation is 4.7E+09, and the second implementation is 4.37E+09.

Compared with the first implementation, the second implementation degrades the misprediction rate by 6.37%, and decreases the total execution cycles by 7.02%. It shows the second implementation has higher performance than the first one.

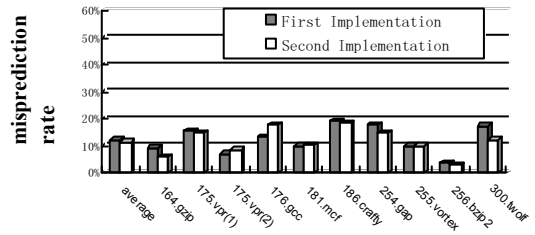


Figure 3. Misprediction rate of the two implementations

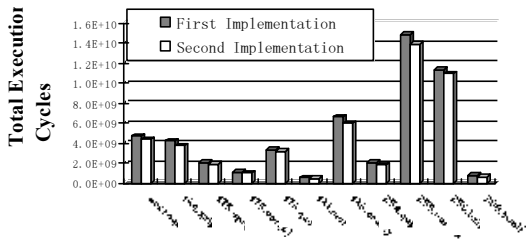


Figure 4. Total execution cycles of the two implementations

We capture some statistics about the implementations when running the benchmarks. Make note of the misprediction times of all control instructions and get the correct prediction rate of them (see Fig. 6).

Following is performance analysis of the control instructions:

- Conditional Branch

Conditional branch occurs very common, about 73.20% of the control instructions are conditional branch in SPEC2000 integer benchmarks used in this paper. The conditional branch prediction is uppermost for a predictor. For conditional branch, the first implementation has a correct prediction rate of 89.31%, and the second implementation is 87.67%. The first implementation has a better misprediction rate performance. This is dues to the ICU repair in ID stage of the first implementation. The repair will make sure the predicted target address is right whenever the BHT's direction prediction is correct. But repair also generates bubbles which could affect the final performance. The BTB of first implementation is a 32-entry fully- associative structure, which has a smaller physical dimension than the second implementation that holds a 512-entry, 2-way set-associative BTB.

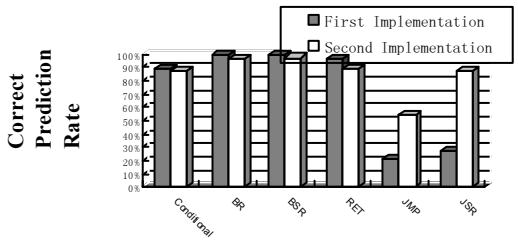


Figure 5. Total execution cycles of the two implementations

- Unconditional Branch: BR & BSR

Because of the fixed target address and direction, unconditional always has a good prediction performance. For BR and BSR, the first implementation has a 100% correct prediction rate, but has a penalty of four cycles for every unconditional branches. The second implementation's correct prediction rate is about 97%, but the instructions can execute without bubbles.

- Return Instruction: RET & JSR_COROUTINE

A return instruction is paired with a function call, this feature makes it possible to use RAS maintains the prediction. The RAS prediction mechanism can approximately achieve a correct prediction rate of 90%. For RET (No JSR_COROUTINE occurred in our benchmarks) the first implementation has a correct prediction rate of 95.77%, and the second implementation is 88.26%.

- Indirect Jump: JMP & JSR

Indirect jumps are used to implement constructs such as virtual function calls, switch-case statements, jump tables and interface calls. Indirect branches are more common in object-oriented languages such as Java, C# and C++. Because of the changeable of target address, current processors are not good at indirect branch predicting. Indirect branches can incur a significant fraction of branch misprediction overhead even though they occupy a little rate of the total instructions. Some new methods [10, 11, 12, 13] had been researched to reach a better correct prediction rate for indirect jump. For JMP and JSR, the correct prediction rates are 21.05% and 28.22% of the first implementation. The second implementation's correct prediction rates are 55.09% and 87.35% which have improved for a big range.

B. Comparison Results

From the experiments, the final comparison result of the two implementations is that the second one has a much better performance than the first one. Its mis-prediction rate is 11.48%, and total execution cycle is 4.37E+09, which are decreased by 6.37% and 7.02% compared to the first implementation.

The first implementation has a much smaller physical dimension, because it measures a simple and small BTB which is a 32-entry, fully-associative cache structure. However the first implementation's BTB only has 32 entries, but because of the implementation complexity of fully-associative cache structure, the advantage of physical dimension could not be very distinct. It also has a better performance for the prediction of conditional branches, which shows fully-associative with small size could achieve a well performance too. The defects are that ICU needs to repair frequently and not good at indirect jump predicting. Repair frequently can induce a lot of bubbles which affects the final performance seriously. The indirect jump's correct prediction rates is very low which shows use the hint bits set by complier for prediction is not a good policy.

The second implementation has a much larger BTB which is a 512-entry, 2-way set-associative structure. It does not hold a separate BHT, the counters used for direction prediction are combined with BTB together. The conditional branch, unconditional branch and indirect jump have records in the BTB, so all of these instructions can achieve a well prediction performance without induce much bubble. The second implementation's conditional branch correct prediction rate is lower than the first implementation, its dues to the collision of different kinds of branches and the mechanism of set-associative which different instructions may map to the same entry. Enlarge the size of BTB or improve the degree of set-associative may reach a better performance.

IV. Conclusions

We have explored two different implementation schemes of two-bit predictor. The first one separates the BHT and BTB, and can not avoid penalty bubbles in case of mis-prediction occurs. The other one uses its BTB to record several kinds of control instructions, and can generate correct predictions without any delay.

We also measured the prediction accuracy of these two implementations. Due to the improved accuracies on indirect jumps, conditional and unconditional branches, the second implementation has a much better performance than first one. The total execution cycles of these two implementations are examined. Results show that the second implementation scheme has less execution cycles than the first one.

With these results mentioned above, we successfully implemented the 2-bit predictor in our self-designed high-performance processors, and achieved high scores when performing SPEC2000 tests.

References

[1] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kaufmann Publishers, Inc., 1996.

[2] T-Y Yeh and Y.N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. Proceedings of the 19th International Symposium on Computer Architecture, May, 1992, pp: 124~134.

[3] S.McFarling. Combining branch predictors. Technical note TN-36, DEC-WRL, 1993.

[4] Daniel A. Jimenez, Calvin Lin. Neural Methods for Dynamic Branch Prediction. In Proceedings of the Seventh International Symposium on High Performance Computer Architecture, 2001, pp197–206.

[5] James E. Smith. A Study of Branch Prediction Strategies. ISCA-8, 1981, pp135~148.

[6] Alpha Architecture Handbook (Version 4). Compaq Computer Corporation, October 1998, pp74~79.

[7] ShienTai. Pan, Kimming. So and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. ASPLOS-5, 1992.

[8] Johnny K. F. Lee, Alan Jay Smith. Analysis of Branch Prediction Strategies and Branch Target Buffer Design. IEEE Computer 21(7). 1984

[9] John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach (Third Edition). USA: Elsevier Science, 2003

[10] Jose A. Joao, Onur Mutlu, Hyesoon Kim and Yale N. Patt. Dynamic Prediction of Indirect Jumps. IEEE Computer Architecture Letters, 2007, vol(6),

[11] Yul Chu, M. R. Ito. An Efficient Indirect Branch Predictor. 2001

[12] Karel Driesen, Urs Holzle. Accurate Indirect Branch Prediction. Technical Report TRCS97-19. March 15, 1998.

[13] Oliverio J. Santanna, Ayose Falcon, Enrique Fernandez, Pedro Medina, Alex Ramirez and Mateo Valero. A Comprehensive Analysis of Indirect Branch Prediction. Springer-Verlag Berlin Heidelberg 2002, ISHPC 2002, LNCS 2327, pp. 133–145.

[14] Standard Performance Evaluation Corporation. http://www.spec.org.