Evaluation of Dynamic Branch Prediction Schemes in a MIPS Pipeline

ELE 475 Final Project Report

Debajit Bhattacharya & Ali JavadiAbhari

Abstract

Increasing exploitation of Instruction Level Parallelism has been the key to high-performance computing in the past few decades. One of the most critical roadblocks to this parallelism is the presence of branches. Branches (both conditional and un-conditional) redirect the stream of instructions resulting in dead cycles in the front-end of the pipeline. In super-pipelined and superscalar architectures, branch costs increase with the number of stages before branch resolution and the width of the pipeline respectively. In this project, we explore some well-known branch prediction schemes proposed in the literature that can alleviate this problem. We write benchmarks to observe the superiority of individual branch predictors in different situations. We find different patterns in a code that might take advantage of each of these predictors. We find that the bimodal (1-level) 2-bit predictor performs best for most of the benchmarks. We also explore some of the hardware costs associated with these techniques.

Introduction

In search for ever increasing computing performance, computer architects have made use of increasing degrees of instruction level parallelism (ILP). Both super-pipelining and superscalar designs have become very common in the last few decades to exploit the ILP. But, for both these techniques, branch prediction has become increasingly important in determining the machine performance since the branch costs are much worse than the basic one-way, short pipelines.

A naïve solution, stalling the pipeline till the branch is resolved, is not anymore acceptable for higher demands in performance. Moreover, some of the static compiler assisted techniques for minimizing branch costs in the early RISC designs are also becoming less appropriate. In particular, branch delay slots are not much popular, as the number of delay slots to fill increases but not enough instructions are available to fill those slots. All these trends led the designers to design hardware methods to overcome these branching problems.

A branch prediction problem consists of two major components. First, a prediction of the direction of conditional branches is needed (BNE, BEQ, BLTZ, etc.). Unconditional branches are always taken (J, JR, JAL and JALR), hence, they need not go through this step. Second, for taken branches, the branch target must be made available in the Fetch stage. One way to implement this quickly is to use a branch target buffer (BTB), which is a special cache (can be direct-mapped or set-associative) tagged by the branch PCs and store the targets of the taken branches [1].

In this project, we have worked on a few hardware-based dynamic branch prediction strategies. The most popular technique, also known as bimodal branch prediction, makes a prediction depending on the direction of the branch in the last few occurrences. Another method takes the history of each
branch individually and makes use of their repetitive patterns. This is called two-level local branch predictor. The other technique uses the combined history of all recently taken branches to make the prediction of a current branch. This technique is referred to as the two-level global branch prediction in our project. The advantages of each of these predictors in different situations are evaluated in this report.

**Simulation Platform**

The five-stage fully bypassed MIPS processor (pv2byp from the lab) without any branch prediction scheme was taken as the reference processor. Figure 1 depicts the five stages of the pipe where the unconditional branches (or jumps) are resolved at the Decode (D) stage and the directions of the conditional branches (or branches) are resolved at the Execute (X) stage. Hence, the jumps and the branches have one and two cycle(s) misprediction penalties respectively for this particular pipe.

![Figure 1. 5 stage pipeline of pv2byp processor](image)

**Dynamic Branch Prediction**

1. **Bimodal Prediction**

Most branches in typical instruction sequences are either usually taken or usually not taken. Bimodal branch prediction takes advantage of this branch behavior and attempts to distinguish usually taken from usually not-taken branches. The simplest approach to implement this is shown in Figure 2. The figure shows a table of counters indexed by the low order address bits in the program counter. The counters provide the prediction bits and the branch target is found using the BTB.

![Figure 2. One-level branch prediction](image)
a. One-bit Saturating Counter

This is perhaps the simplest implementation of the bimodal prediction scheme. The table of counters indexed by the PC has single bit entries. Each bit represents the outcome of a finite state machine and the states are – Taken (1) and Not Taken (0). If a particular branch was taken the last time, the branch history table (BHT) stores a 1 in a particular entry for that branch. Next time that branch PC is encountered, a table lookup is performed against that PC and the branch is predicted Taken this time. Figure 3 shows the state diagram of the FSM.

![State diagram for one bit saturating counter](image)

This counter exploits the temporal correlation between the two states of the branches. The limitation of the counter is its intolerance towards any changes in direction. For example, there will be always two mispredictions for a backward inner branch loop using this implementation.

Table 1. Code snippet for nested conditional branches

<table>
<thead>
<tr>
<th>Code snippet</th>
<th>Nested conditional branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:</td>
<td>ADD $1, $2, $3;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>L2:</td>
<td>BNE $1, $2, L2;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>BNE $2, $3, L1;</td>
<td></td>
</tr>
</tbody>
</table>

In the above code sequence, the inner BNE always gets mispredicted twice at the beginning and at the end of the loop because of the single bit history. As we will see later, this problem can be overcome using a two-bit history.

b. Two-bit Saturating Counter

This is an advanced implementation of the previous scheme, storing the history of multiple instances of the same branch. Each counter, indexed by the lower significant bits of the PC, is two bits long. For each taken branch, the appropriate counter is incremented. Likewise for each Not-Taken branch, the appropriate counter is decremented. Since the counter is saturating, it is not decremented past zero, nor is it incremented past three. The most significant bit determines the prediction. The state diagram for the FSM is shown in Figure 4.

Here, repeatedly taken branches will be predicted Taken, and repeatedly Not-Taken branches will be predicted to be Not-Taken. But unlike a one-bit counter, this predictor can tolerate a branch going an unusual direction one time and keep predicting the usual branch direction. In the above
code sequence, the beginning of the inner loop will mispredict only at the very first and last iterations of the outer loop. In all the other cases, the branch is predicted correctly because of the extra “weak” state.

![Figure 4. State diagram for two bit saturating counter](image)

2. **Two level local branch prediction**

One way to improve on bimodal prediction is to recognize that many branches execute repetitive patterns. Perhaps the most common example of this behavior is loop control branches. For example,

Table 2. Code snippet for FOR loop

```c
For (i=1; i<5; i++) {}  
```

If the loop test is done at the end of the code sequence, the corresponding branch will execute the pattern \((11110)^n\), where \(n\) is the number of times the loop is executed. Clearly, if we knew the direction in which this branch had gone on the previous four executions, then we could always be able to predict the next branch direction.

![Figure 5. Two-level branch prediction scheme](image)

A branch prediction method based on the one developed by Yeh and Patt [2] that can take advantage of this type of pattern is shown in Figure 5. The figure shows a branch predictor that uses two tables. The first table which records the history of recent branches is also known as the branch history register (BHR). We index the BHR by the low-order bits of the branch PC.

Each BHR entry records the direction taken by the most recent \(n\) branches whose addresses map to this entry, where \(n\) is the length of the entry in bits. The second table is an array of counters (both one and two bit) identical to those used for bimodal branch prediction. However, here they are indexed by
the BHR entry stored in the first table. In this project, this approach is referred to as local branch prediction because the history used is local to the current branch. In Yeh and Patt’s nomenclature this method is referred to as a per-address (PAd) scheme.

If there are plenty of branches in the program, a local predictor can suffer from two kinds of contention. First, the branch history may reflect a mix of histories of all the branches that map to each history entry. Second, since there is only one counter array (global counter) for all branches, there may be conflict between patterns. For example, if there is another branch that typically executes the pattern (0110 instead of) 1110 there will be contention when the branch history is 110 (common to both). However, with 4 bits of history and $2^4$ counters, this contention can be avoided.

3. **Two level global branch prediction**

In the local branch prediction scheme, the only patterns considered are those of the current branch. Another scheme proposed by Yeh and Patt [2] is able to take advantage of other recent branches to make a prediction. A single shift register BHR, records the direction taken by the most recent n conditional branches (Fig. 6). Since the branch history is global to all branches, this strategy is called global branch prediction. We can derive global branch predictor from the earlier local branch predictor by making the BHR single-entry.

Global branch prediction is able to take advantage of interdependent branch history. Consider the example below:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &gt; 1)</td>
<td>If (x &gt; 1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(x &lt; 1)</td>
<td>If (x &lt; 1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If x is indeed greater than 1, we know that the second “if” will not be taken. If however x ≤ 1, then we don’t know conclusively which way this branch will be taken, but the probability may well be skewed one direction or the other. If so, using global history, we should be able to make a better prediction than if we had no information about the value of x.

Both the two level schemes discussed here (local and global) require some initial settling time to set the history registers to proper values. The length of the BHR is a trade-off. Longer length can capture more history but requires more time to settle. Hence, long BHRs should only be used in situations where there is sufficient branch history to capture and also the initial settling misses are negligible compared to the other predicted hits.

4. **GSelect**

We implemented a modified version of the basic global branch prediction scheme, suggested by Yeh and Patt [2]. It is observed that global history information is less efficient at identifying the current branch than simply using the branch address. Pan, So, and Rahmeh [3] suggested that a more efficient prediction might be made using both the branch address and the global history. Their approach is shown in Figure 6 and also popularly known as Gselect. Here the PHT is indexed with a concatenation of global history and branch address bits.

The performance of global prediction with selected address bits (Gselect) is shown in Figure 6. With the bit selection approach, there is a tradeoff between using more history bits or more address bits. Rather than exploring all these possibilities, we used equal number of bits from the BHR and the PC.
Design with Verilog

1. In the PC stage of our “pv2byp” pipeline, we added a new multiplexor to select between the PC+4_Phl, PC+4_Xhl and the BTB target address (Figure 7). If the PC in the branch stage corresponds to a previously faced branch instruction (i.e. undergoes tag match in BTB and is valid), then we fetch the predicted target address stored in the BTB.

2. All the hardware structures (BTB, BHT, BHR and PHT) are read at the Fetch (F) stage of the pipe. They are written at the D stage (jumps) and at the X stage (branches). If both the X and D stage are trying to write the tables simultaneously, priority is given to the X stage, since the branch instruction is ahead in the pipeline. Here, the writing of the BTB for the jump instruction is ignored which may result in minor performance degradation.

3. The tables were implemented as register files and their sizes were varied by the corresponding parameterized Index_Size. The reads were combinational but the write was sequential and controlled by a Write-enable signal. The write_enable signal has a few components –
   a. As jumps are always taken, a misprediction in jump implies the jump-target was wrong. J_miss_Dhl signal handles this case and the solution is to jump to the right address.
   b. A branch can be mispredicted not taken, but actually it is taken. Br_miss_nt_Xhl signal handles this case and the solution is to load the branch target address to the PC.
   c. A branch can be mispredicted taken, but actually it is not taken. Br_miss_t_Xhl signal handles this case and the solution is to load the PC+4_Xhl to the PC in the next cycle.

4. For two-bit counters, even if the branches were predicted correctly, we may want to write the tables if the counter is in a “weak” state.

5. We faced challenges in initializing the abovementioned signals. The comparison of the PCs of the D stage and the branch target of the X stage was invalid, as they were don’t cares at the very beginning of execution. We tackled the problem by loading 32’b0 in the comparison PCs at the beginning i.e. triggering the reset signal.
Testing and Evaluation

We used several small benchmarks to test the performance of each of our implementations. These tests were written either in assembly or C, and were mainly designed to give us insight about different aspects of code execution in which each implementation performs well. These aspects of the predictors have been discussed above. Below, each of the toy tests is shortly explained along with the significance of each:

**50loop:** This is a strongly biased loop, where for 50 consecutive iterations the branch is taken and it’s not taken on the last iteration. It performs well on those predictors that can train quickly.

**nestedloop:** Two loops inside each other, where the inner loop is repeated 10 times and the outer loop 50 times. It exploits the tolerance in two-bit counters.

**ifelse:** Contains 5 different BNE instructions inside an outer BNE instruction, where all the inner branches have a pattern of 11110, and the outer loop executes for 20 times. This tries to exploit the global pattern detector in Gselect.

However, we also wanted to test our designs with some real applications. There were several problems with this, such as lack of I/O in the processors, and the slow simulation time. This made running the whole standard program for many of the benchmarks virtually impossible.

To realize our goal, we turned our attention to the StreamIt software package available from MIT [6]. StreamIt is a programming language and compilation infrastructure designed for modern streaming systems, which allows for their easy programming. The package comes with a set of ready benchmarks written in C, such as Fast Fourier Transform (FFT), Audio beamformer, Discrete Cosine Transform (DCT), DES encryption, etc. We chose DCT as our base benchmark, and tried to manipulate it to be compatible with our custom processors. This meant taking out the functions in the stdio.h library, and shortening some of the computationally intensive tasks.

A discrete cosine transform expresses a sequence of finite data points in terms of a sum of cosine functions oscillating at different frequencies. Here we use a 2-D DCT, which is the cornerstone of many image processing applications such as lossy image compression (e.g. MPEG, JPEG, …). The code is the realization of the IEEE standard for this algorithm, and acts on blocks of 8x8 data, converting them to another 8x8 block, in the frequency domain. This is calculated according to the formula below:

\[
X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \left( \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} \cos \left( \frac{\pi}{N_2} \left( n_2 + \frac{1}{2} \right) k_2 \right) \right) \cos \left( \frac{\pi}{N_1} \left( n_1 + \frac{1}{2} \right) k_1 \right)
\]

To make it compatible with our processor, we gave it an arbitrary block of input data instead of reading it from file. The string of inputs and outputs (64 integers) is shown below:

**INPUT:**

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 9 8 7 6 5 4 3 2 1

**OUTPUT:**

190 -9 0 -1 0 0 0 0 38 -25 0 -3 0 -1 0 0 -107 0 0 0 0 0 0 -46 9 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24 -6 0 -1 0 0 0 0 -8 0 0 0 0 0 0 -23 5 0 1 0 0 0 0

Since most of the information is stored in the lower frequencies of an image, we see that less significant information are encountered as we progress along the matrix. This makes the DCT very useful for sparse representation and storage of data.
The full DCT algorithm has 1,317,739 instructions, of which 136,448 are branch instructions. It is a comprehensive benchmark for our purposes as it includes a large amount of branches in different styles (heavily biased, nested, ...) and in different places in the code, so it can exploit the BTB, BHT and PHT structures fully. It takes around 10 minutes to execute on the simulators, and the generated .vcd file is around 9.5GB.

Figure 8 shows the results of running each of these benchmarks on our custom processors. Here, we decide to use a measure of “Prediction Accuracy” instead of IPC to report the performance, since it reflects the success of each predictor better. This is a change from the presentation we gave in class. A counter was employed to count the total number and the number of successfully predicted branches.

Some interesting observations obtained from this graph:

1. We see that in all cases employing even a simple prediction scheme causes significant improvement over the base fall-thru prediction case.
2. In most cases, the 1-level 2-bit predictor is the best performing. This is due to the robustness of this design and the tolerance it provides, as well as the fact that it keeps individual histories for the branches so they do not conflict.
3. Codes containing strongly biased loops such as 50loop and vector addition are very poor executing on the fall-thru predictor because it predicts incorrectly almost every time. On the other hand, they are very suitable for one level predictors, since they train quickly (one-bit or 2-bit does not really matter). Their performance falls when we have two level predictors that take time to train.
4. Nested loops perform well on two-bit predictors, because the first iteration of the inner loop which misses every time in the one-bit counter now only misses on the very first outer loop iteration, due to increased tolerance. The binary search benchmark’s behavior can be explained by this trait, since it contains nester for loops.
5. The two-level one-bit predictor has a poor performance for nested loops and if-else constructs. This might seem unexpected at first, but it is really due to the fact that we have taken the length of our BHR to be larger than what is required from these small tests, such as
the number of iterations of the inner loop. As a result, the tests are not able to train the predictor well enough to reach a stable prediction scheme, and the predictor oscillates a lot before reaching a correct pattern.

6. The if-else test performs very well for the Gselect predictor, since several branches show a similar history and thus the global predictor can recognize that. However, a surprising result is that a 2-level 2-bit predictor does slightly better on this test. The reason for that is that the 1110 pattern is very favorable for the 4bit PHT of this processor.

7. The DCT is a very computationally intensive test, and we see that the 1-level predictor and the Gselect perform best on it. We can thus loosely infer that the DCT contains nested loops, shares a global history between its branches and that fast training time helps its performance. The masked filter test shows a very similar behavior.

8. To employ a 2-level predictor, it seems that keeping a global history and predicting locally as in the case of Gselect offers more gain than the opposite scheme. This is exemplified when we use a large test as the DCT where we can actually detect global branch patterns.

To see the effects of the size of the added hardware on the accuracy of our predictors, we vary two of the parameters inside our code: BHT Index Size and PHT Index Size. The results of this trade-off are depicted in Figures 2 and 3 for the one-level-two-bit and two-level-two-bit predictors, respectively.

![Figure 9. Effect of BHT index size on prediction accuracy for a one-level two-bit predictor](image)

In Figure 9, we see that for very simple tests such as loops, the performance does not change at all. This is due to the fact that these codes simply do not have enough branches to cause conflict in the table. However, for all other tests, we see that due to a high number of conflicts in branches aliasing to the same table entry, a two-bit (4-entry) BHT causes very low prediction accuracies. It is worth noting that going from an 8-bit to 16-bit wide index size does not cause performance increase in most cases, because they don’t have more than 256 branch instruction PCs (although the actual number of branches can be much higher than this due to loops). In the DCT algorithm, because of the huge number of instructions, we actually do see a slight increase in performance for the 16-bit case. However this is less than 1%, whereas the hardware cost is 256x.
Figure 10 shows that most of the tests suffer when we increase the PHT index size. This is because these are small tests and the warm-up time to train the PHT dominates. They simply do not have enough branches to make use of the longer pattern history. However, the DCT test with lots of branches can actually take advantage of the longer PHT and increase its prediction accuracy, although the gain is not too significant and is probably not worth the extra hardware cost.

**Conclusion**

In this project, we have evaluated different dynamic branch prediction strategies using different benchmarks that highlight the prediction accuracies of the strategies in different situations. The bimodal 2-bit predictor emerged as the best scheme for most of the benchmarks. This suggests that most of the branches in the test codes were strongly biased in one direction. The if-else test was specifically written to establish the superiority of Gselect over others but all the micro-benchmarks and the DCT benchmark showed that one-level saturating counters did a better job than the local and global history based two-level counters. The DCT benchmark also revealed that there is enough global history to exploit in the code. Gselect performed reasonably well for this benchmark and produced very comparable prediction accuracy with those of the bimodal predictors. We observed noticeable improvement in the branch prediction accuracy for all the benchmarks even with a simple prediction scheme, which we did not observe in case of IPC. This is because the IPC involves all the instructions, and if the number of branches is negligible compared to the total number of instructions, IPC is not affected considerably with unless there is a very significant improvement in the prediction accuracy. We learned that keeping longer BHTs and shorter separate BTBs is a good trade-off because it reduces hardware cost and reuses a branch history even if the branch is evicted from the BTB. We observed that smaller BHTs lead to aliasing problems for multiple branches. Hence, the index size of the BHT should be chosen in such a way so that each branch maps to a unique entry in the BHT. Keeping a longer BHT than this is a waste since prediction accuracy saturates after this point. Similarly for two level predictors, longer BHR entry should be provided only when the benchmark has enough branch history to keep track of. Longer branch history leads to mispredictions in the warm-up stage, which must be compensated by the extra information in the BHR for predicting future branches. Only the DCT benchmark, having more than 100k branches, took advantage of this.
References


