Predictive Branching Methods and Architectures - A survey

Chhayadevi M bhamare, Meghanand Bhamare
Department of Electrical and Electronics
BITS-Pilani Goa Campus
Goa, India

cmb@bits-goa.ac.in, mab@bits-goa.ac.in

Abstract

In high-performance computer systems, performance losses due to conditional branch instruction can be minimized by predicting a branch outcome and fetching, decoding and/or issuing subsequent instructions before the actual outcome is known. This paper discusses various branch prediction strategies and architectures with the goal of maximizing prediction accuracy.

Keywords—Branch, Branch Prediction, Branch Transfer Buffer, Pipelining, Two Level Branching, Branch Penalty

1. Introduction

This paper attempts to cover various branching strategies which have been used by various microprocessor designs. Various methods of predicting the direction of conditional branches using branch history have been developed. Different predictor mechanisms take advantage of different observed patterns in branch behaviour. Pipelining is the major technique for improving the performance of machines. Branches cause performance losses in pipelines by making the pipeline stall till the branch instruction is evaluated. Branches comprise about 15-30% of the total instructions. Stalling for evaluation of the branch instruction increases with an increase in pipelining and causes performance losses. Additionally as processors get wider with deeper pipelines branch prediction becomes critical since the relative cost of incorrect prediction increases. With increasing pipeline depth, the number of instructions which have to be flushed out of the pipeline increases and this increases the incorrect prediction cost. An accurate branch predictor can substantially improve the performance of a machine by scheduling correct instructions in the pipeline after a conditional branch. The strategies are divided into two basic categories; depending on whether or not past history was used for making a prediction. Branch prediction strategies are thus divided in to static and dynamic prediction strategies. A static prediction strategy (Strategy 1) makes the same prediction every time a branch instruction is encountered. A dynamic prediction strategy takes in to account previous instruction execution history before predicting a branch. In this paper various branch prediction schemes will be discussed.

2. Branch prediction schemes

Strategy 1: Predict all branches as taken. Program sensitivity plays an important role in determining the effectiveness of this strategy. The algorithm used to encode a program, the programmer and the compiler can influence the program structure and thus influence the number of branches taken. This can result in significant differences in the program performance which may not be immediately visible to a programmer working on a high level language.

Strategy 1a: Predict that only certain branch operation codes will be taken; predict that others won’t be taken. This is a variation of strategy 1. It shows slightly better performance than strategy 1, but the improvement varies significantly with changes in program and programming styles. Strategy 1 and 1a are both static prediction strategies.

Strategy 2: Predict that only backward branches (towards lower addresses) will be taken; predict that all forward branches will not be taken. Loops are always terminated with backward branches. The strategy tries to utilise this concept in order to improve the accuracy.

Strategy 3: Always predict that a branch will be decided as on its last execution. If it has not been previously executed predict that it will be taken. Strategy 3 is a dynamic prediction strategy. This strategy is provides better accuracy than strategy 1. However this strategy cannot be physically realised since there is no limit on the number of individual branch instructions that a program may contain. Hence it is considered as a benchmark for the accuracy of predicting branch executions. It is also less program sensitive than strategy 1.

Strategy 4: Maintain a table of the m most recent branches that are not taken. Predict that only branches found in the table will be not taken. Purge table of entries if they are taken and use LRU
replacement for adding new entries. The accuracy for this strategy is close to that of strategy 1 for a small table size and increases to grow closer to strategy 2 the table size grows.

Strategy 5: Maintain a bit for each instruction in the cache.
If an instruction is a branch instruction, the bit is used to record if it was taken on its last execution. Branches are predicted to be decided on their last execution; if a branch has not been executed it is predicted to be taken. This strategy gives results close to strategy 2, as the instruction cache hit ratio is usually at least 90%. A similar strategy uses a static prediction along with the bit maintained in the cache. The static prediction is EX-ORed with the cache prediction. Whenever a wrong prediction is made, the cache prediction bit is complemented. This gives an output similar to that of strategy 5, but prediction memory only need to be updated when there is a wrong prediction, thus saving the time penalty for updating the memory.

Strategy 6: Bimodal Branch Prediction
Behaviour of a branch is not random. The branch is either usually executed or usually skipped. The bimodal branch prediction technique takes advantage of this property to predict whether a branch will be taken. This strategy works well when branch is strongly biased in one direction. This strategy uses 2-bit counters for each branch execution. The counters are indexed by the low order address in the PC. For each branch which is taken, the corresponding counter is incremented. For each branch instruction ignored, the appropriate counter is decremented. If the counter is not incremented more than 3(11) or decremented below 0(00). The most significant bit determines the prediction. By using a 2-bit counter, a branch going in the unusual way can be tolerated, and the usual prediction be made for the next instance of the same branch instruction. For small tables, multiple branches may share the same counter set, thus leading to a decrease in the efficiency of the prediction accuracy. In order to rectify this, it is possible to store a tag with each counter and use a set associative table to match the counter with the appropriate counter.

Strategy 7: Local Branch Prediction
The local branch prediction method tries to take advantage of this repetitive pattern in branch instructions by using two tables. The first table uses history of the recent branch instructions. This table will store a bit which will specify whether the branch was taken or not. The second table is an array of 2-bit counter similar to the one used in the bimodal branch prediction. However the array is indexed by the branch history stored in the first table. Thus this table uses the bits stored in the history table to decide the execution of the branch instruction. If the number of history is less than the number of iterations or repetitions in the pattern, the local predictor can suffer from two kinds of contention. First the history table may reflect a mix of histories of all the branches that map to each history entry. Secondly due to the common counter array, there may be conflict between two patterns. This can be avoided if adequately sized history table is used. For small sized predictors, the local prediction scheme performs worse than the bimodal scheme. However once the size of predictor rises above 128 bytes, the accuracy reached by this method can go up to 97.1%

Strategy 8: Global Branch Prediction
This scheme uses a single shift register called the Global Register to record the direction taken by the most recent conditional branch instruction. It takes advantage of two types of patterns. First, the direction taken by the current branch may depend strongly on other recent branches. Secondly, the branch instruction may replicate the behaviour of a local branch instruction. This can occur when global history includes all the local history needed to make a prediction. This strategy hence tries to optimise the behaviour of the global branch so that the global branch prediction resembles the behaviour of the local branch prediction. It is expected that many local branches will occur for every global branch executed. Due to this it is advantageous for the execution to closely resemble the local branch prediction behaviour. Global branch prediction does better than the bimodal scheme when the predictor size increases above 1 KB.

Strategy 9: Global Predictor with Index Selection
As seen in the previous scheme the global history information is less efficient at identifying the current branch than simply using the branch address. This suggests that it may be possible to make more efficient predictions by using both the branch address and the global history. In this scheme the counter table is indexed with a concatenation of global history and branch address bits. The efficiency of the architecture thus depends on the number of global history and branch address bits used in the concatenation. For comparison purposes, the optimal combination of global history and branch address bits are used and the strategy called gselect-best. The gselect-best outperforms both the bimodal as well as the global prediction schemes. For small sizes, the gselect-best matches the performance of bimodal prediction. However for larger number of global history bits, the performance is substantially better than the bimodal scheme.

Strategy 10: Global history with Index sharing
It is seen that global history information weakly identifies the current branch. This suggests that redundancy is present in the counter index used by
gselect. If many address bits are used in the counter index to identify the branch, it follows that the number of repetitions in the global history combinations are less. This effect can be used advantageously by hashing the branch address and the global history together. The EX-OR operation can be used with the global history and branch address as the operands to have more information that either component alone. Also since more address bits and global history bits are used, the predictions will be better than gselect. For predictor sizes above 256 bytes, the algorithm explained above (gshare-best) outperforms gselect-best by a small margin. However for smaller predictors, gshare underperforms gselect because there is too much contention for counters between different branches and adding global information makes it worse.

**Strategy 11: Branch Target Buffer (BTB):** The BTB can be used to reduce the performance penalty of branches by predicting the path of the branch and caching information about the branch. The BTB can cache up to four types of information: a tag identifying the branch the information corresponds to (usually branch address), prediction information for predicting the path of the branch, the branch target address and instruction bytes at the branch target address. As each instruction is fetched from memory, the instruction address is used to index into the BTB. If a valid BTB address is found for that address, then the instruction is a branch. The branch path is then predicted by the branch’s prediction information. If the branch is predicted as taken, the instructions cached in the BTB are supplied to the processor while if the branch is predicted as not taken, sequential execution continues. As the processor finishes executing the branch, it checks to see if the BTB correctly predicted the branch. The branch prediction information and branch target address are updated after every branch. An extension to the BTB concept is the idea of a special call/return stack. An ordinary BTB is of little use when the return instruction may return to a different address each time. By using a BTB with a call/return stack it is possible to capture these branches as well. However all the BTB’s considered in this paper predict taken/not-taken based on the previous history for that branch.

**Strategy 12: Modified Branch Transfer Buffer**
This technique uses a different hardware model to reduce branch penalties. Following is the improved model of BTB: A cache memory called Modified Branch Target Buffer (MBTB) with 4 fields. • Address field: storing the branch address • Instruction field: storing the first instruction’s address • Target field: when branch taken storing the branch target address when not taken storing the second instruction’s PC at taken directory. • History Field: Storing the predicted information. MBTB stores the address of both the cases i.e. if the branch is taken and if branch is not taken. Hence no delay occurs no matter whether the prediction is correct or not because MBTB keeps two direction of information, so if it knows that the prediction is incorrect then it puts the correct instruction and corrects the PC to pipeline slot. Also once MBTB is established we need not maintain it because there is no delay or branch penalty whether the prediction is correct or incorrect. Hence MBTB performance is better than plain BTB in same hit ratio scenario although it will be same just when predict correction is hundred per cent.

**Strategy 13: Multi-Level Branch Target Buffer**
A multilevel BTB, each level possibly containing different amounts/types of information per entry, is able to maximise performance by achieving a better balance of number of entries and quantity of information per entry. The first level, the highest performance level contains a branch tag, prediction bits, target address and target instruction bytes for each entry. The second level contains a branch tag, prediction bits and a target address for each entry. The third level is a hash table of prediction bits. Multi-level BTBs are strongly dependent on the implementation. RISC processors with delayed branches and very small branch penalties will not find this scheme very helpful, while CISC processors with deep pipelining and larger branch penalties will see significant improvement in performance.

**Strategy 14: Hybrid Branch Predictors**
As noted before, there is a large variance in the performance of two-level schemes across the different benchmarks. Hence a single predictor may not be best for all types of source codes and branches. A hybrid branch predictor consists of two or more component predictors and a selection mechanism to choose which of the predictors to use for each branch. The selection mechanism uses a table of saturating two-bit counters to keep track of which predictor is currently more accurate for each branch. Each branch is mapped to a counter using the fetch address LSBs. If the counter value is less than or equal to 2, the first predictor is used else the second one is used. If the first predictor was correct, the counter is incremented while if second predictor is correct, it is decremented. If both predictors are correct, the counter does not change.

**Strategy 15: Delayed Branches**
In a delayed branch instruction, one instruction following the branch is always executed, whether or not the branch is taken. This aims to reduce the cost of pipeline stalls caused by successful branch instructions. The success of this strategy depends on the ability of compilers to find a suitable instruction which will execute whether or not the branch succeeds.
Strategy 16: Prophetic Branching

In this technique instead of filling the pipeline with delay slots the Prophetic technique makes a prediction and fills the pipeline with the code pertaining to the “Likely Branch”. There is no difference in case of Unlikely Branch case between the two mechanisms. The implementation of this mechanism requires the addition of a register –the FAS (Fall-through Address Store- and a minimal change in the control logic for annulling the instructions after the condition evaluation. The address of the Fall-through Instruction is stored in Fall-through Register (FAS). In case the prediction is incorrect the address of the next instruction is obtained from the Fall-through Register and annulls the instruction from the target. The limited decrease in performance comes from the NOPs scheduled in the code. A second loss in performance arises when the compiler mispredicts.

Strategy 17: Branch Folding

This technique uses a decoded instruction cache with each non branch instruction transformed into a line of microcode in a cache of decoded instructions. Each entry includes an address field which points to the next line of microcode to be executed. Conditional branches make use of a second address field in the microcode and static branch prediction. A bit in the branch instruction predicts whether the branch will be taken or not and the predicted next address field is used to find the next line to be executed. If the wrong path is predicted, the incorrect results are discarded and the correct path is executed. Both the possible outcomes are encoded in the microcode thus resulting in zero cycle execution time for a correctly predicted branch.

Strategy 18: Profiler branches

The performance of squashing branches is limited by the assumption that all branches are usually taken. Performance can obviously be increased if the compiler knows which branches are usually not taken. Such information is supplied by an execution profiler. This is a software based branch predictor and thus saves hardware costs and chip area. The paper shows that such a software based profiler predicts slightly better than a 128 entry counter method. The BHR is used to index into the PHT to select which two-bit saturating counters. The BHR is used to index into the PHT to select which two-bit counter to use. For multiple PHTs, the PHT chosen is based on the least significant bits of the fetch address. Once the two-bit counter is selected, prediction follows the two-bit counter method.

3. Conclusion

This paper deals with different Branch prediction strategies used to reduce branch penalty. The performance of the strategies discussed above has been compared by different authors in different studies. Since the benchmarks used for comparison in each study are different, it is not possible to conclude with absolute certainty which strategy is better. In general dynamic strategies perform better than static prediction strategies. While choosing a strategy various parameters like the available memory, cost, architecture complexity, speed and required accuracy are taken into consideration. This paper brings together all of the strategies discussed in the reference papers and can thus be used as an introductory article for branch prediction mechanisms.

4. References