*Schemes of Branch Prediction*

Nicolae-Andrei Vasile

*Faculty of Computer Science and Automatic Control*

*POLITEHNICA University of Bucharest*

Bucharest, Romania

*Abstract – In high-performance computer systems, performance losses due to conditional branch instructions can be minimized by predicting a branch outcome and fetching, decoding, and/or issuing subsequent instructions before the actual outcome is known.*

1. **INTRODUCTION**

As the design trends of modern superscalar microprocessors move toward wider issue and deeper super-pipelines, effective branch prediction becomes essential to exploring the full performance of microprocessors. A good branch prediction scheme can increase the performance of a microprocessor by eliminating the instruction fetch stalls in the pipelines [1]. As a result, numerous branch prediction schemes have been proposed and implemented on new microprocessors.

Branch instructions can break the smooth flow of instruction fetching and execution. This results in delay, because a branch that is taken changes the location of instruction fetches and because the issuing of instructions must often wait until conditional branch decisions are made.

To reduce delay, one can attempt to predict the direction that a branch instruction will take and begin fetching, decoding, or even issuing instructions before the branch decision is made. Unfortunately, a wrong prediction may lead to more delay if, for example, instructions on the correct branch path need to be fetched or partially executed instructions on the wrong path need to be purged. The disparity between the delay for a correctly predicted branch and an incorrectly predicted branch points to the need for accurate branch prediction strategies.

[2] We present a framework that categorizes branch prediction schemes by the way in which they partition dynamic branches and by the kind of predictor that they use. The framework allows us to compare branch prediction schemes, and to analyze why they work. We use the framework to show how a static correlated branch prediction scheme increases branch bias and thus improves overall branch prediction accuracy. We also use the framework to identify the fundamental differences between static and dynamic correlated branch prediction schemes. This study shows that there is room to improve the prediction accuracy of existing branch prediction schemes.

[3] Our study begins in the next section, with two branch prediction strategies that are often suggested. These strategies indicate the success that can reasonably be expected. They also introduce concepts and terminology used in this paper. Strategies are divided into two basic categories, depending on whether history was used for making a prediction or not. In subsequent sections, strategies belonging to each of the categories are discussed, and further refinements intended to reduce cost and increase accuracy are presented. Levels of confidence are attached to branch predictions to minimize delay when there are varying degrees to which branch outcomes can be anticipated (for example, prefetching instructions is one degree, pre-issuing them is another).

[1] In this paper, we apply techniques from data compression to establish a theoretical basis for branch prediction, and to illustrate alternatives for further improvement. To establish a theoretical basis, we first introduce a conceptual model to characterize each component in a branch prediction process. Then we show that current "two-level" or correlation-based predictors are, in fact, simplifications of an optimal predictor in data compression, Prediction by Partial Matching (PPM).

[4] In this paper, we explore various types of artificial neurons and propose a two-level scheme that uses perceptrons instead of two-bit counters. A key advantage of our approach is its ability to utilize long branch history lengths. In our predictor, each static branch is ideally allocated its own perceptron to predict the branch outcome, and the space required by our scheme scales linearly with the history length.

1. **BASIC DEFINITIONS**

Given a conditional branch in a program, the goal of a branch prediction scheme is to predict accurately the outcome of that conditional branch (i.e. that the branch will take or that the branch will fall through).1 The most accurate branch prediction schemes predict the next action of a branch based on some function of the past actions of that branch and possibly other branches in the program.

To understand the capabilities of these branch prediction schemes and to compare competing schemes in a meaningful manner, we must be able to identify and quantify the important properties of branch prediction schemes. To achieve this goal, this section defines a set of mathematical tools that allow us to analyze program and branch behavior in an abstract manner.

Let a branch execution be a pair consisting of an identifier and a direction variable. Intuitively, the identifier uniquely specifies a static branch in a program, and the direction variable indicates the direction that the branch went. We define an execution stream or just stream as a sequence of branch executions. Intuitively, this corresponds to a branch trace of one invocation of a program, identifying in trace order the conditional branches executed and the directions that they went. A stream can also be formed by concatenating the streams of multiple invocations of a program (possibly with different inputs). We refer to the original stream of all executions in a run of the program as the program execution stream. A substream of a stream is a subsequence of. A predictor is a simple mechanism that predicts the next direction of a stream. A predictor may consider program characteristics (e.g. the opcode of the next branch to predict) in addition to any part of the past program execution stream.2 The accuracy of a predictor is the number of correct predictions divided by the total number of predictions; accuracy measures how closely the predicted stream matches the actual stream. A prediction scheme is a comprehensive mechanism that takes a program execution stream, divides it into substreams, and directs each substream to a unique predictor. Figure 1 illustrates this concept. The objective in dividing the execution stream into substreams is that each substream should be more accurately predictable by its predictor. The accuracy of the prediction scheme is the total number of correct predictions divided by the total number of predictions.

The general conceptual model we introduce for branch prediction consists of three major components: a source, an information processor, and a predictor, as illustrated in Figure 1. Although some components are often combined in a hardware implementation, this three-part model is useful in explaining the principles behind different prediction schemes.

2.1.1 Source

The source is simply the machine code of the programs we are running. The source contains program semantics and algorithmic information. To aid branch prediction, this information can be explored and extracted during the compile-time. It can be stored and passed on to be used during execution. A hint bit in branch instructions is one means of passing this information. In addition, the source can be modified to produce more predictable branches using statistics from previous test-runs. This is how code restructuring and code profiling work.

2.1.2 Information processor

In a hardware implementation, the information processor is often combined with predictors and, hence, overlooked. However, the information processor plays a key role in the prediction process and thus deserves a close study. Conceptually, it can be subdivided into two components: selector and dispatcher.

2.1.2.1 Selector

The selector selects which run-time information should be used for branch prediction and encode it. This information can be branch address, operation code, branch outcome, target address, hint bits, or statistics from test-runs. Prediction accuracy depends heavily on the mix of run-time information that is employed. Once the information is determined, the selector decides what formats to represent the information. For example, suppose branch outcomes and branch addresses are selected as information, the selector can combine the outcomes with addresses into one single stream or keep outcomes as individual streams classified by branch addresses. Good encoding can extract the essence of information, producing a concise and efficient representation to help prediction.

2.1.2.2 Dispatcher

The dispatcher determines how the information is mapped (fed) to the various predictors, since multiple information streams and predictors may exist in a prediction scheme. The mapping can be one-to-one, many-to-one, one-to-many, dedicated, or multiplexed (time-shared). Different mappings often have great influence on the final prediction accuracy.

2.1.3 Predictor

A predictor is simply a finite-state machine that takes input and produces a prediction. It does not need to know the meaning of the input. Common examples are a constant or static predictor, a 1-bit counter, a 2-bit up-down saturating counter [SmithS1], and a Markov predictor. A Markov predictor forms the basis of recent two-level prediction schemes and is discussed in detail in Section 3. For the moment, a Markov predictor is simply a finite state machine that generates predictions based on a finite number of previous inputs.

1. **BRANCH PREDICTION STRATEGIES**

Branch instructions test a condition specified by the

instruction. If the condition is true, the branch is taken: instruction execution begins at the target address specified by the instruction. If the condition is false, the branch is not taken, and instruction execution continues with the instruction sequentially following the branch instruction. An unconditional branch has a condition that is always true (the usual case) or is always false (effectively, a pass). Because unconditional branches typically are special cases of conditional branches and use the same operation codes, we did not distinguish them when gathering statistics, and hence, unconditional branches were included.

1. **STATIC PREDICTION STRATEGIES**

A straightforward method for branch prediction is to

predict that branches are either always taken or always not taken. Because most unconditional branches are always taken, and loops are terminated with branches that are taken to the top of the loop, predicting that all branches are taken results typically in a success rate of over 50%.

Strategy 1

- Predict that all branches will be taken.

Figure 1 summarizes the results of using strategy 1 on the six FORTRAN benchmarks. From Figure 1, it is evident that the majority of branches are taken, although the success rates vary widely from program to program. This points to one factor that must be considered when evaluating prediction strategies:

program sensitivity. The algorithm being programmed, as well as the programmer and the compiler, can influence the structure of the program and, consequently, the percentage of branches that are taken. High program sensitivity can lead to widely different prediction accuracies. This, in turn, can result in significant differences in program performance that may be difficult for the programmer of a high—level language to anticipate.

Strategy 1 always makes the same prediction every time a branch instruction is encountered. Because of this, strategy 1 is called static. It has been observed and documented,5 however, that the likelihood of a

conditional branch instruction at a particular location

being taken is highly dependent on the way the same

branch was decided previously. This leads to dynamic prediction strategies in which the prediction varies, based on branch history. Strategy 1 (always predict that a branch is taken) and its converse (always predict that a branch is not taken) are two examples of static prediction strategies. A further refinement of strategy 1‘is to make a prediction based

on the type of branch, determined, for example, by

examining the operation code. This is the strategy used in some of the IBM System 360/370 models9 and 137 attempts to exploit program sensitivities by observing, for example, that certain branch types are used to terminate loops, while others are used in

IF-THEN-ELSE—type constructs.

Strategy 1a

- Predict that all branches with certain operation codes will be taken; predict that the others will not be

taken. The six CYBER 170 FORTRAN programs were examined, and it was found that "branch if negative". "branch if equal", and "branch if greater than or equal" are usually taken, so they are always predicted to be taken. Other operation codes are always predicted to be not taken. This strategy is somewhat tuned to the six benchmarks, because only the benchmarks were analyzed to determine which opcodes should be predicted to be taken. For this reason, the results for strategy 1a may be slightly optimistic. Figure 3 shows the results for strategy 1a when it was applied to the CY170 programs. Generally, greater accuracy was achieved with strategy 1a than with strategy 1. The largest increase was in the GIBSON program in which the prediction accuracy was improved from 65.4% to 98.5%. The only program showing a decrease in accuracy was the SINCOS program in which there was a drop from 80.2% to 65.7%. The changes in both the GIBSON and SINCOS programs can be attributed to predicting that "branch if plus" was not taken. If it had been predicted as taken, the accuracy of the GIBSON program would have dropped nearly to its original value, and the accuracy of the SINCOS program

would have risen nearly to its original value. Other static strategies are possible. For example, predictions based on the direction of the potential

branch or on the distance to the branch target can be

made. Following is a detailed description of one of

these strategies.

Strategy 3

- Predict that all backward branches (toward lower

addresses) will be taken; predict that all fonNard

branches will not be taken. The thought behind strategy 3 is that loops are terminated with backward branches, and if all loop branches are correctly predicted, the overall accuracy will be high.

Figure 4 indicates that strategy 3 often worked well,

sometimes exceeding strategy 2 (probably because of

the anomalous decision case). There is, however, one

program in which its performance was poor: in the

SINCOS program, the accuracy for strategy 3 was about 35%. This indicates that program sensitivity is

significant and that performance can suffer considerably for some programs. A disadvantage of strategy 3, and of other strategies using the target address, is that the target address may need to be computed or compared with the program counter before a prediction can be make. This tends to

make the prediction process slower than for other

strategies.

1. **DYNAMIC PREDICTION STRATEGIES**

Strategy 2

- Predict that a branch will be decided the same way

as it was on its last execution. If it has not been previously executed, predict that it will be taken.

The results (Figure 2) of using strategy 2, indicate that strategy 2 generally provides better accuracy than strategy 1. Unfortunately, strategy 2 is not physically realizable, because theoretically, there is no bound on 136 the number on individual branch instructions that a program may contain. (In practice, however, it may be possible to record the history of a limited number of past branches; such strategies are discussed in a subsequent section.)

Strategies 1 and 2 provide standards for judging other

branch prediction strategies. Strategy 1 is simple and

inexpensive to implement, and any strategy that is

seriously being considered for use should perform at

least at the same level as strategy 1. Strategy 2 is

widely recognized as being accurate, and if a feasible

strategy comes close to (or exceeds) the accuracy of

strategy 2, the strategy is about as good as can

reasonably be expected.

Strategy 1 is apparently more program sensitive than

strategy 2. Evidence of this is the wide variation in

accuracy for strategy 1 and the much narrower variation for strategy 2 (Figures 1 and 2). Strategy 2 has a kind of second-order program sensitivity, however, in that a branch that has not previously been executed is predicted to be taken. Lower program sensitivity for dynamic prediction strategies is typical, as results throughout this paper show.

Finally, it is interesting that one aspect of branch

behavior leads occasionally to better accuracy with

strategy 1 than strategy 2. Often, a particular branch

instruction is predominately decided one way (for

example, a conditional branch that terminates a loop is most often taken). Sometimes, however, it is decided the other way (when ’'falling out of the loop"). These anomalous decisions are treated differently by strategies 1 and 2. Strategy 1, if it is being used on a branch that is most often taken, leads to one incorrect prediction for each anomalous not taken decision. Strategy 2 leads to two incorrect predictions; one for the anomalous decision and one for the subsequent branch decision.

The handling of anomalous decisions explains those

instances in which strategy 1 outperforms strategy 2

and indicates that there may exist some strategies that

consistently exceed the success rate of strategy 2.

Some strategies base predictions on past branch history.

Strategy 2 is an idealized strategy of this type, because it assumes knowledge of the history of all branch instructions. The strategies discussed in this section are actually realizable, because they use bounded tables to record a limited amount of past branch history.

Branch history can be used in several ways to make a

branch prediction. One possibility is to use the outcome of the most recent execution of the branch instruction; this is done by strategy 2. Another possibility is to use more than one of the more recent executions to predict according to the way a majority of them were decided; this is done by strategy 7. A third possibility is to use only the first execution of the branch instruction as a guide; a strategy of this type, although accurate, has been found to be slightly less accurate than other dynamic strategies. First, strategies are considered that base their predictions on the most recent branch execution (strategy 2). The most straight forward strategy is to use an associative memory that contains the addresses of the n most-recent branch instructions and a bit indicating whether the branch was taken or not taken.

The memory is accessed with the address of the branch instruction to be predicted, and the taken or not taken bit is used to make the prediction.

If a branch instruction is not found in the table, two

issues must be considered: (1) the prediction that is to

be made, and (2) the table entry that should be replaced to make room for the new branch instruction. First, if a branch instruction is not in the table, some static strategy must be reverted to for a default prediction. A good choice is to predict that the branch is taken as in strategy 2.

A more complex default strategy could be used (strategy 1a, for example), but using the simpler always predict taken strategy has a positive side effect. In particular, only branch instructions that are not taken need to be put into the table; then, the existence of a branch in the table implies it was previously not taken. Branches that were recently taken are given the proper prediction by default. One bit of memory is saved, but more importantly, histories of more branch instructions are effectively remembered. For example, if two out of the eight most-recent branch instructions executed are not

taken, then all eight consume only two table entries,

although all are predicted to have the same outcome as on their previous executions. A dual strategy is to use a default prediction of branch not taken and to maintain a table of branches most recently taken. Because most branch instructions are taken, however, this strategy is generally less accurate.

As far as replacement strategies, first-in first—out (FlFO) and |east—recently used (LRU) seem to be two reasonable alternatives. For the application here, in which the sequence of branch instructions tends to be periodic because of the iterative structure of most

programs, there is actually little difference between the FIFO and LRU strategies as far as prediction accuracy. The LRU strategy does appear to be more compatible with the scheme mentioned previously in which only branches that were not taken are recorded. Then, if a branch in the table is taken, it is purged from the table, and that table location is recorded as being least recently used. A branch that is taken subsequently fills the vacancy in the table rather than replacing a good table entry. Such a scheme for filling vacancies in the table fits naturally with the LRU replacement strategy.

1. **DATA COMPRESION AND PREDICTION**

Like branch prediction, data compression relies on prediction. In data compression, the goal is to represent the original data with fewer bits. The basic principle of data compression is to use fewer bits to represent frequent symbols, while using more bits to

represent infrequent symbols. Thus, the net effect is to reduce the overall number of bits needed to represent the original data. In order to perform this compression effectively, a compression algorithm

has to predict future data accurately to build a good probabilistic model for the next symbol [Bell90]. Then, as shown in Figure 2, the algorithm encodes the next symbol with a coder tuned to the probability distribution. Current coders can encode data so effectively that the number of bits used is very close to optimal and, consequently, the design of good compression relies on an accurate predictor. The problem of designing efficient and general universal compressors/predictors has been extensively examined. In our experiments we draw on these techniques, adapting them to the new context of branch prediction.

3.1 Prediction by Partial Matching

Prediction by partial matching (PPM) is a universal compression/ prediction algorithm that has been theoretically proven optimal and has been applied in data compression and prefetching [Cleary84, Kdshnan94, Kroeger96, Moffatg0, Vitter91 ]. Indeed, it usually outperforms the Lempel-Ziv algorithm (found in Unix compress) due to implementation considerations and a faster convergence rate [Curewitz93, Bell90, Witten94]. As described

above, the PPM algorithm for text compression consists of a predictor to estimate probabilities for characters and an arithmetic coder. We only make use of the predictor. We encode the outcomes of a branch, taken or not taken, as 1 or 0 respectively. Then the PPM predictor is used to predict the value of the next bit given the prior sequence of bits that have already been observed.

3.1.1 Markov predictors

The basis of the PPM algorithm of order m is a set of

(m + l) Markov predictors. A Markov predictor of orderj predicts the next bit based upon the j immediately preceding bits--it is a simple Markov chain [Ross85]. The states are the 2 j possible patterns ofj bits. The transition probabilities are proportional to the observed frequencies of a 1 or a 0 that occur given that the predictor is in a particular state (has seen the bit pattern associated with

that state). The predictor builds the transition frequency by recording the number of times a 1 or a 0 occurs in the (j + 1)-th bit that follows the j-bit pattern. The chain is built at the same time that it

is used for prediction and thus parts of the chain are often incomplete. To predict a branch outcome the predictor simply uses thej immediately preceding bits (outcomes of branches) to index a state and predicts the next bit to correspond to the most frequent transition out of that state.

**Two-level Branch Prediction as an**

**Approximation of PPM**

In this section, we show that recently proposed two-level or correlation based predictors are approximations of, PPM, an optimal prediction algorithm.

4.1 Description of two-level predictor

Among the various branch prediction schemes, two-level or correlation based predictors are among the best. In addition, these predictors all share very similar hardware components. As Figure 5 shows, they have one or more shift-registers (branch history

registers) to store history information in the first level and have one or more tables of 2-bit counters (pattern history tables) in their second level [Yeh91 ]. The contents of the first level shift-registers are typically used to select a 2-bit counter in one of the second-

level tables. Predictions are made based on the value of the 2- bit counter selected.

4.2 Two-level branch predictors as Markov predictors

From the above discussion on two-level adaptive branch predictors and the one on Markov predictors in Section 3.1.1, there are strong similarities. Though different schemes of two-level branch predictors exist, they differ only in what information is used for history and what subsets of branch outcomes are used to index and update the counters. As a result, there exists a corresponding Markov predictor for each scheme.

Figure 6 shows the similarity between a two-level predictor and a Markov predictor. Both predictors behave exactly the same in the first level. They both use the last m bits of branch outcome to search the corresponding data structure. Note that an m-bit shift

register serves two functions: first, it limits the information used for prediction to m previous outcomes and, second, it uniquely defines a finite-state machine in which each state has exactly two

predefined next states. In the second level, the Markov predictor uses a frequency counter for each outcome, while the two-level predictor uses a Saturating up-down 2-bit counter [Smith81].

Whenever a branch is taken/not taken, the 2-bit counter increments/ decrements. The decision for a two-level predictor depends on whether the value of the counter falls in the positive half or the negative half. Similarly, a Markov predictor simply predicts the next branch to be the most frequent outcome based on two frequency counters. Both predictors are utilizing a majority vote via different implementations. The saturating counter is an approximation to this that can be realized in hardware efficiently. An interesting illustration is to see how a two-level predictor, the per-address branch history register with global pattern history table (PAg), corresponds to a Markov predictor. This peraddress

scheme uses one table of 2-bit counters and multiple shift registers where each register records only outcomes of a particular branch. Although multiple shift registers exist, all shift registers operate the same and correspond to the same transition rule for a

finite-state machine (state diagram). In addition, all shift registers share the same global table of 2-bit counters and, hence, share the same value (counter) in each state. Therefore, this per-address scheme uses one Markov predictor that is time-shared and updated

among various branches.

# **REFERENCES**

|  |  |
| --- | --- |
| [1] | I.-C. K. Chen, J. T. Coffey and T. N. Mudge, "Analysis of branch prediction via data compression," *ACM SIGPLAN Notices,* pp. 128-137, 1996. |
| [2] | C. Young, N. Gloy and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," *Proceedings 22nd Annual International Symposium on Computer Architecture,* pp. 276-286, 1995. |
| [3] | J. E. Smith, "A study of branch prediction strategies," *Proceedings of the 8th annual symposium on Computer Architecture,* pp. 135-148, 1981. |
| [4] | D. A. Jimenez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Transactions on Computer Systems,* pp. 369-397, 2002. |