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. In this paper we will discuss a few of the branch prediction schemes that exist and are used in today's processors.

I. 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. 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.

Many papers and research reports have been issued on different strategies of branch prediction as a solution regarding the need for better performance in complex systems. One example is "A Study of Branch Prediction Strategies" [1], presenting 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.

Another notable work is "A Comparative Analysis of Schemes for Correlated Branch Prediction" [2]. 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 is presented. The framework is used to compare branch prediction schemes, and to analyze why they work, as well as showing how a static correlated branch prediction scheme increases branch bias and thus improves overall branch prediction accuracy. It also identifies the fundamental differences between static and dynamic correlated branch prediction schemes.

Data compression techniques are applied to establish a theoretical basis for branch prediction, and to illustrate alternatives for further improvement in “Analysis of Branch Prediction via Data Compression” [3]. To establish a theoretical basis, a conceptual model is introduced to characterize each component in a branch prediction process. Further, the paper explains how current "two-level" or correlation-based predictors are, in fact, simplifications of an optimal predictor in data compression, Prediction by Partial Matching (PPM).

II. 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). 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. Figure 1 shows a few schemes that are or have been used in branch prediction.

Prediction scheme	Source modification or profiling	Information processor		predictor
		selector	dispatcher	
forward not-taken, backward taken	no	(address - target)	many-to-one	constant
2-bit counter	no	outcome, classified by address	one-to-one, mapped with address	2-bit counters
path correlation	no	target, in execution order	one-to-many, mapped with address	several Markov predictors
gshare	no	address, outcome, XOR together	one-to-one	a Markov predictor
GAg	no	outcome, in execution order	one-to-one	a Markov predictor
GAs	no	outcome, in execution order	one-to-many, mapped with address	several Markov predictors
PAg	no	outcome, classified by address	many-to-one, multiplexed	a Markov predictor
PAs	no	outcome, classified by address	many-to-many, mapped with address	several Markov predictors
PSg	no	outcome, classified by address	many-to-one, multiplexed	constant
branch correlation	yes	statistics from previous runs, hint bit	one-to-one, mapped with address	constant
hybrid predictor	yes	combinations of above	combinations of above	combinations of above

Figure 1. Branch prediction schemes [3]

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.

An interesting general conceptual model that Young, Gloy and Smith introduced in their work [2] consists of three major components: a source, an information processor, and a predictor. Although some components are often combined in a hardware implementation, this three-part model is useful in describing the principles behind different prediction schemes.

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.

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.

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.

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.

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.

III. 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.

IV. 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%. A few strategies are discussed further.

Strategy 1

Predict that all branches will be taken. One factor that must be considered when evaluating prediction strategies is 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 (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 models 9 and 137 attempts [1] 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.

V. DYNAMIC PREDICTION STRATEGIES

It has been observed however [1], [3], 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 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. Unfortunately, strategy 2 is not physically realizable, because theoretically, there is no bound on 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). 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, then one is about as good as can reasonably be expected. Strategy 1 is apparently more program sensitive than strategy 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.

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.

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 most of them were decided. 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.

VI. DATA COMPRESSION AND PREDICTION

To continue the idea of dynamic branch prediction, we will further discuss another interesting approach, the work of Chen, Coffey and Mudge [3]. 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. To perform this compression effectively, a compression algorithm must predict future data accurately to build a good probabilistic model for the next symbol. Then, 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.

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. 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.

Markov predictors

The basis of the PPM algorithm of order m is a set of $(m + l)$ Markov predictors. A Markov predictor of order j predicts the next bit based upon the j immediately preceding bits; it is a simple Markov chain. 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 $(j + 1)$ bit that follows the j bit pattern. The chain is built while it is used for prediction and thus parts of the chain are often incomplete. To predict a branch outcome the predictor simply uses the j 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.

VII. TWO-LEVEL BRANCH PREDICTION AS AN APPROXIMATION OF PPM

Among the various branch prediction schemes, two-level or correlation-based predictors are one of the best. In addition, these predictors all share very similar hardware components. 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. 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.

From the above discussion on two-level adaptive branch predictors and the one on Markov predictors, 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.

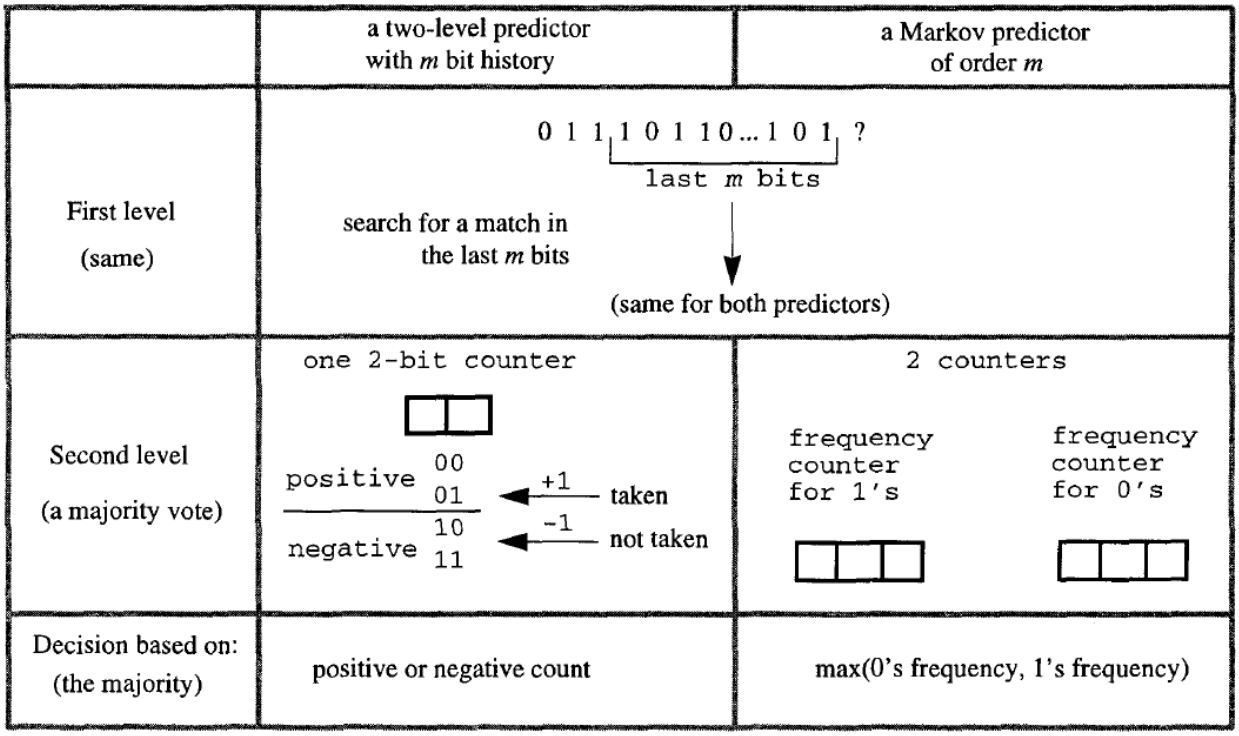

Figure 2. Two-level predictor (left), Markov predictor (right) [3]

Figure 2 shows the similarity between a two-level predictor and a Markov predictor. Both predictors behave 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 2-bit counter.

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 approximates 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 per-address 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.

VIII. CONCLUSIONS

This paper described the accuracy of the existing branch prediction strategies as well as establish the connection between data compression and branch prediction. Based upon this theoretical basis rather than just simulation results, we can now have a good idea on different prediction strategies, such as static or dynamic, and a reasonable degree of confidence in the performance of two-level predictors. Although two-level predictors are close to optimal if unlimited resources are available, PPM can still outperform two-level predictors when branch-target buffers are small. This is because PPM has better mechanisms for handling misses.

IX. REFERENCES

[1] J. E. Smith, "A study of branch prediction strategies," *Proceedings of the 8th annual symposium on Computer Architecture,* pp. 135-148, 1981.

[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] 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.