Data Compression Techniques for Branch Prediction

Jeremy S. De Bonet

Artificial Intelligence Laboratory
Learning & Vision Group
545 Technology Square Massachusetts Institute of Technology
Cambridge, MA 02139

jsd@ai.mit.edu
http://www.ai.mit.edu/people/jsd

June 9, 1999

Abstract

Without special handling branch instructions would disrupt the smooth flow of instructions into the microprocessor pipeline. To eliminate this disruption, many modern systems attempt to predict the outcome of branch instructions, and use this prediction to fetch, decode and even evaluate future instructions. Recently, researchers have realized that the task of branch prediction for processor optimization is similar to the task of symbol prediction for data compression. Substantial progress has been made in developing approximations to asymptotically optimal compression methods, while respecting the limited resources available within the instruction prefetching phase of the processor pipeline. Not only does the infusion of data compression ideas result in a theoretical fortification of branch prediction, it results in real and significant empirical improvement in performance, as well. We present an overview of branch prediction, beginning with early techniques through more recent data compression inspired schemes. A new approach is described which uses a non-parametric probability density estimator similar to the LZ77 compression scheme [23]. Results are presented comparing the branch prediction accuracy of several schemes with those achieved by our new approach.

1 Introduction

The accurate prediction of the outcome of branch instructions is a critical component for achieving high performance for many of today's modern processors. Branch instructions can conditionally change the path a program will take, thereby disrupting the smooth flow of instructions which is necessary for finding instruction level parallelism, and fully exploiting the acceleration afforded by pipelining. When a conditional branch is encountered there is a delay before the direction of the branch is determined, this causes a disruption in the flow of instructions into the pipeline. To eliminate this disruption, many modern systems attempt to predict the outcome of branch instructions. This prediction is used to begin the process of fetching future instructions and memory. In some cases instructions can even be executed and their results cached before the outcome of the earlier branch has been determined (commonly known as *speculative execution*). When the outcomes of branches are predicted correctly computation can continue seamlessly, as future instructions are fed into the pipeline. When predictions are incorrect however, a large performance hit is incurred in order to flush both the pipeline and cached speculative results. Furthermore, it is generally true that the more computation done in advance of determination of the real branch outcome, the larger the cost of purging partially executed instructions and reinitiating computation along the correct path [19]. Increasing prediction accuracy is necessary to acquire a greater computational speed from longer and increasingly intricate pipelines, instruction reordering schemes, and speculative execution.

Initial work on branch prediction focused on the prediction of *direct conditional branches*. These branch instructions are either *'taken'*, resulting

in execution beginning at the target address, or *'not taken'*, in which case, execution continues at the next instruction. This early work was empirically driven and used computational mechanisms of limited complexity to mimic simple intuitions about program flow. We review some of this work here.

More recently, a connection between branch prediction and data compression was realized [1]. Predicting a branch outcome given knowledge of a program's past behavior is essentially the same problem as predicting a symbol given the preceding sequence of symbols. Accurate symbol prediction requires the ability to estimate the probability distribution over future symbols, which is the fundamental problem in data compression. Substantial progress has been made in developing branch prediction schemes that approximate asymptotically optimal compression methods, while respecting the limited resources available within the instruction-prefetching phase of the processor pipeline. Not only did the infusion of data compression ideas result in a theoretical fortification of branch prediction, there was a real and significant empirical improvement in performance, as well.

Here we will discuss data compression inspired methods that achieve very high levels of prediction accuracy. While these methods do not yet achieve optimal performance, they are certainly closing in on this goal. It is important to note that *optimal* performance is not *perfect* performance. This is because while a system may make optimal use of all information available to it at prediction time, there are cases in which the outcome cannot be determined beforehand because the program is operating on unknown data. This deviation between optimal prediction and perfect prediction, while extremely difficult to quantify, can be thought of as the *entropy* of a program and its data.

More recently branch prediction research has shifted from direct conditional branches, which have two outcomes (one if 'taken', one if 'not taken'), to indirect branches which can have multiple outcomes. For indirect branches the branch target (the address at which execution will continue after the branch) is determined by the content of a register and/or memory location[1]. Predicting the outcome of a branch when it has multiple possible targets is more difficult than when there are only two possible outcomes. This is clearly true in light of the observation that multiple target branches are a generalization of the simpler two-outcome case. In the domain of indirect branches, solutions based on data compression techniques have also proven successful over more ad hoc methods. We will discuss the generalization to multiple symbol alphabets data compression techniques, and potential hardware implementations which approximate these algorithms while observing real world constraints.

2 Early Work on Branch Prediction: Smith 1981

In his seminal 1981 paper on branch prediction [19], James Smith describes a series of branch prediction schemes that increase in complexity and performance. One of the major contributions of this work is that it enumerates each scheme – many of which are simple, and perhaps obvious – and evaluates and compares their relative performances.

In this work a distinction is made between static and dynamic branch prediction schemes. Static schemes are those which make predictions solely based upon the structure of the program code. Dynamic strategies adjust their predictions based upon flow of the program preceding the branch. In later works this distinction is irrelevant because static schemes are vastly outperformed by dynamic ones, and are therefore no longer the main focus[2]. Nevertheless it is important to evaluate such schemes at least once to measure the performance that they can achieve. This provides a baseline level of performance against which other schemes can be compared. Additionally, performance differences between various static techniques can identify predictors (contextual clues which help make a prediction) which can be combined with dynamic techniques.

Most early work only considered direct conditional branches, which are either 'taken', resulting in processing beginning at new address, or 'not taken', in which case processing continues at the next instruction.

2.1 Scheme: Predict all branches taken (or not taken)

The simplest prediction scheme is one in which all branches are predicted 'taken'; of course, an equivalently simple scheme is to assume 'not taken' for

[1] In some cases the indirection is an offset from an address specified by the branch argument, in others the indirection specifies an absolute address.

[2] However, static schemes are still used in combination with dynamic schemes to "seed" in early stages before sufficient data has been collected.

all branches. The prediction accuracy of these techniques do two things. First they provide a baseline performance measure; any newly proposed scheme had better outperform them. Second they illustrate the variability of the distributions of branch outcomes across different programs. Prediction accuracy varied from 57.4% to 99.4% for the programs Smith tested. While Smith does not discuss these results in an information theoretic framework, the accuracy of (the better of) these two techniques is a direct measure of the zero-th order entropy of a programs branch outcomes[3] This scheme is *static* as it does not vary during the execution of the program.

2.2 Scheme: Predict branches with certain opcodes taken

A second static scheme predicts that branches with certain operation codes (opcodes) will be taken, and that branches with other opcodes will not be taken. This branch prediction technique has been used in some IBM 360/370 models. For most of the tested programs performance improved, in some cases by as much as 33%; however, in two cases performance decreased by as much as 14%. This scheme can be though of as a measure of the first order entropy of branches measured with respect to their associated opcode.

2.3 Scheme: Predict backward branches taken

Another static strategy is to predict that all branches whose target address is lower (i.e. backward branches) will be taken; forward branches are assumed to be 'not taken'. This method is designed to capture the intuition that 'while' loops are terminated with backward branches and make up a large fraction of the branches encountered. While it may be true that 'while' loops have a skewed distribution, it is unclear why non-loop branches preferentially not be taken. Only in this case should this approach be better than the "predict all branches taken" scheme. For all but one tested program this seems to be true, however, the author offers no explanation as to why this might be the case. Performance with this technique is a measure of the first order entropy of branches measured with respect to their target direction. This scheme is used as the fall-back prediction method (used when more sophisticated methods are inapplicable) in the Intel Pentium, Pentium II and Pentium III line of processors [7].

[3]In actuality, for all programs tested, predicting that all branches 'not taken' was better than predicting 'taken'. This is probably due to looping constructs in which a branch continually falls through until loop termination. However, this bias may not be the case for all programs.

2.4 Scheme: Predict branch not taken when in table

This scheme represents the simplest dynamic prediction strategy. The idea is to maintain a table of the most recently evaluated branches which were 'not taken'. If a branch is in the table when it is encountered a prediction is made that the branch is not taken. If the prediction is wrong, the branch is removed from the table. If the encountered branch is not in the table, branch is predicted to be 'taken'. If wrong, the branch is added to the table by replacing the least recently used entry (thus maintaining a constant table size). This scheme is termed 'dynamic' because the branching history of the running program is used to change the prediction of future branches. Performance of this method increases with increasing table size, as a history bit can be maintained for successively more branches. A scheme like this is used by the DEC Alpha 21064 [9].

In the limit of an infinite table, there is a one-bit history for every branch, and this technique can be thought of as a measure of the entropy of a branch given its last outcome. While Smith does not discuss his models in terms of data compression we can begin to draw similarities to probability density modeling methods used in data compression and other areas. This model can be thought of as a two-state Markov chain with a probability of only 0 or 1 at each node.

With even a modest table size of four addresses performance of this technique is substantially better (as much as 30%) for many of the tested programs, with only very small (less than 5%) performance losses for others.

Smith discusses several potential physical instantiations of this algorithm: using associative memory; using a fixed table that is kept for only those instructions in the instruction cache; and using a hash table. There are only slight performance tradeoffs between each instantiation, they are distinctive principally in the hardware complexity required to implement them.

2.5 Scheme: Predict branch using a table of 2-bit counters

The final prediction scheme described by Smith is computationally only a small step from the preceding algorithm; however statistically, this scheme can be viewed as a true probability estimator. The notion in this scheme is to maintain a table containing the m most recently encountered branches. For each branch entry in the table, keep a two bit *saturating counter*. Each time a branch in the table is visited, predict the branch using the value of the counter. A counter value of 00, or 01 results in a prediction of 'not taken', and 10 or 11 yields a prediction of branch 'taken'[4] Regardless of the validity of the prediction, the table is updated in the same way: if the branch was not taken, the counter associated with the branch is decremented, if it was taken, the counter is incremented. The counter is not decremented below 00 or incremented above 11, and hence is termed 'saturating'. Several processors use this scheme: Intel Pentium processor [7], MIPS R10000 [13], Sun UltraSPARC [20] and the DEC Alpha 21164 [9].

While Smith does not discuss these algorithms in terms of statistical representations, each counter can be viewed as a 2-bit estimator of the probability that a branch will be 'taken'. The prediction in this case corresponds to choosing the maximum likelihood (ML) estimate from this distribution. This prediction scheme results in the highest prediction accuracy of all the schemes and for all of the programs that Smith examined. The substantial and consistent improvement in prediction accuracy can be explained by the improvement in the estimation of the "true" branching probability.

The obvious question then becomes, "what about using counters larger than 2-bits?" Smith showed that with additional counter bits performance in some cases actually diminishes. Smith provides a fairly weak argument in which he attributes this decline in performance to "the 'inertia' that can be build up with a larger counter in which history in the too-distant past is used." When viewed within a probabilistic framework, this effect can be explained in terms of the lack of stationarity of the branching distribution. A 2-bit estimator which is completely determined by the last four branches, while only able to provide rough approximation, is better able to adjust to the changes in the branching probability distribution. This observation suggests some improvements over this scheme, which will be discussed later.

[4]Smith actually views the counter as a one bit signed number using a two's complement implementation. In this representation 00 and 01 are positive, and predict branch 'taken', and 10 and 11 predict 'not taken'. We change the enumeration here to emphasize the similarity to a probability estimator.

3 Predictor Combination: McFarling 1993

In any field where multiple competing approaches each meet with reasonable success, there is the potential to achieve better results by using the approaches in combination. The technique of combining multiple methods is common to many fields, for example it is known as "boosting" in machine learning [17, 8] and "mixture modeling" in statistical modeling [5].

A 1993 technical report by McFarling [12] attempted to achieve a boost in branch prediction accuracy by combining three of the best known approaches. The first technique, termed "bimodal" by McFarling, is essentially the same as the method of predicting branches using a table of 2-bit counters described in [19]. The other two techniques, both due to Yeh and Patt [22], make use of several past branch outcomes to predict future outcomes. The two schemes, while similar, can be classified into *local* and *global*, and are outlined below.

3.1 Scheme: Predict using the recent history of each branch

This scheme is designed to use the last n outcomes of a particular branch as a predictor for future outcomes. Each branch is predicted using only its own branch outcome history, which is maintained separately. This method is therefore termed *local*. As in the previous scheme, information is accumulated for the most recently executed branches. For each branch an n-bit list is kept containing a history of that branch's last n outcomes. In addition to the branch histories, a table is maintained. The table is indexed by the 2^n possible branch patterns which can occur within a recent history of n branches. For each of the 2^n entries in this table, a 2-bit counter is maintained. Prediction is straightforward. When a branch is reached, its current recent history is used to index into the table, and a 2-bit counter is obtained. This counter is used to make a prediction in the same way as in the previous scheme. Each counter is then updated with the actual outcome of the branch using the same method described for the scheme above.

Performance of this scheme is significantly better than that of the previous scheme when the number of recent branches maintained is sufficiently large[5]. Histories of between 2 and 12 outcomes were maintained for each branch, with little improvement in performance beyond maintaining a history of 6 outcomes.

The intuition which served as the motivation for creating these techniques was that "many branches execute repetitive patterns." In a more information theoretic sense, we can think of this scheme as a measure of the entropy of the branches with respect to their recent history. In fact this scheme, which was constructed in an unprincipled fashion turns out to be a coarse approximation of the method of prediction by partial matching (PPM) method described in 1984 in [2] and developed later in [14]; however, this observation was not made until 1996 [1], which will be discussed later.

3.2 Scheme: Predict using the recent history of all branches

The *global* version of the above scheme is essentially the same, except that a single global history, and a single history-indexed table is maintained. This global history is then used to index into the table of 2^n counters associated with each of the possible histories of length n.

Performance of this scheme is significantly worse than the local scheme; however, with sufficiently large table sizes (1,024 bytes) it does outperform the "bimodal" approach introduced by [19]. The success of this approach can be attributed to its ability to leverage correlations in outcomes between various branches. Additionally, as McFarling points out, with sufficiently large histories, outcome patterns emerge which can uniquely identify various branches, and can therefore begin to leverage some of the individual history prediction of the local scheme.

3.3 Scheme: Predict using global history and local address

McFarling is able to combine some of the benefits of the global and local schemes by a method of combining global history and branch address to create an index into a pattern table. This synthesis was attempted in two ways. The first approach uses simple concatenation of the global history and the least significant bits of the branch address. The second approach builds a hash index from the global history and the branch address by XORing them together. The boost in performance observed in [12] is relatively minor.

[5] McFarling does not indicate how many branches were maintained however, when the total storage size approaches 128 bytes this scheme begins to outperform the previous one. With the largest tested configuration this scheme resulted in half as may mispredictions.

4 Data Compression Techniques Applied to Branch Prediction: Chen *et al.* 1996

We have been drawing connections between branch prediction and data compression techniques throughout this paper, but this connection was only first formally realized in 1996 by Chen *et al.* in [1]. Chen *et al.* observed that many of the best known branch prediction schemes – schemes which were conceived through heuristic methods and refined using empirical measurements – could be viewed as approximations of the prediction by partial matching algorithm.

4.1 Scheme: Prediction in data compression

PPM is an algorithm designed for data compression. Like most compression algorithms, it operates by modeling the probability distribution over the next symbol in a sequence conditioned on information obtained from previously received symbols. Typically this conditional distribution is lower in entropy than the zero-th order entropy of the symbols in the sequence. Because of this reduction in entropy, the number of bits required to code the next symbol conditioned on this distribution is (on average) lower than the cost of coding this symbol unconditionally. This is the fundamental observation, due to Shannon [18], upon which all source coding (data compression) is based[6]

4.2 Prediction by Partial Matching

The fundamental innovation in the PPM method is the particular probability density estimator used. The name of this technique, "Prediction by Partial Matching", gives some indication of how it works. In PPM a match of all or part of the symbol history is used to compute the expected distribution over

[6] For a complete introduction to source coding see [4]

the next (unknown) symbol. The expected distribution is computed by examining the entire symbol history and locating all sequences of symbols that match the sequence preceding the unknown symbol. Each of these matching sequences is succeeded by a symbol. The distribution of these succeeding symbols form an estimate of the distribution for the unknown symbol. In cases where there are no, or very few, matches of the length n history, the partial match required is then reduced to $n-1$. Smaller and smaller partial matches are used, until a sufficient number of matches are found in the symbol history to obtain a reasonable estimate of the distribution of the next symbol.

4.2.1 PPM: An example.

Consider the sequence of symbols from a 4 letter alphabet: 'a b c b d a b c a b a b ?' where '?' is an unknown symbol we want to predict or transmit. Consider the operation of PPM of order 4 on this sequence.

First, the length 4 history of the unknown symbol is considered; however this sequence, 'a b a b', does not occur anywhere else in the symbol history. Next, a history of length 3 is considered, but this sequence, 'b a b' also does not occur elsewhere in the data. The partial match required is then reduced to two symbols: 'a b'. This sequence occurs three times earlier in the data, and is followed by the symbols: 'c', 'c' and 'a'. From this distribution of symbols which have been observed to follow the pattern 'a b' an estimate for the distribution over the next symbol can be estimated. The estimate of this is distribution is not as simple as 2/3 'c' and 1/3 'a' however. If we were to use such an estimate, we would be attributing 0 probability mass to symbols other than 'c' or 'a', and as a result there would be no way to code the next symbol if it were one of the other symbols (e.g. 'b' or 'd'). To correct for this, some probability mass must be reserved as *escape probability*, indicating that none of the observed matches in the past accurately predict the next symbol. Suppose we chose an escape probability of ϵ. Our estimated distribution for the next symbol would be $P(\text{'a'}) = 1/(3+\epsilon)$, $P(\text{'c'}) = 2/(3+\epsilon)$ and $P(\text{escape}) = \epsilon/(3+\epsilon)$. This distribution can then be used to generate a code for the next symbol. Typically this is done with arithmetic encoding, though other approaches can be used.

Suppose the next symbol were not 'a' or 'c'. In such a case, the escape probability is then coded. When an escape occurs a new distribution for the next symbol is estimated using a one symbol shorter partial history match. In addition to matching this shorter pattern it must also *not* match the longer pattern. Continuing with our example, a distribution is generated from the symbols following partial matches of the pattern 'b': 'c', 'd', 'c' and 'a'. However, only 'd' follows the pattern 'b' but not the pattern 'a b'. Giving us an estimated distribution of: $P(\text{'d'}) = 1/(1+\epsilon)$ and $P(\text{escape}) = \epsilon/(1+\epsilon)$. The next symbol is then coded with this distribution. If an escape occurs again, (i.e. if the symbol 'b' were to come next) the next symbol is coded using the relative frequencies of each symbol. If an escape occurs yet again, which would happen if some never-before-seen symbol (i.e. 'e') should occur, the symbols are then coded using a uniform distribution.

Suppose the next symbol were 'd'. This would be coded by coding an 'escape' followed by the code for the symbol 'd' given the probability $P(\text{'d'}) = 1/(1+\epsilon)$. The decoding process would be as follows: First the decoder would realize that there are no length 4 matches (i.e. the sequence 'a b a b', has not been observed before) at which point a length 3 match would be tested. Again the coder would realize that no length 3 match occurs (i.e. 'b a b' has never occurred elsewhere) and testing move to a match of length '2'. The distribution is then estimated, exactly as was done above and the message 'escape is decoded. The effect of this escape is to drop down to a length 1 partial match. Again a distribution is estimated, and then the symbol 'd' is decoded and inserted into the sequence.

Of course in actual implementations, it is not necessary to continually revisit the transmitted data to search for partial matches. Instead the distribution of symbols which follow all partial matches of any length can be maintained in a set of tables. Using a principal of *update exclusion* only the frequency counters which are considered for a given pattern are updated. Thus, shorter patterns only reflect the distribution of symbols that would not be explained by longer pattern matches.

When described within the maintained table framework, the similarity between the PPM algorithm and the schemes proposed for branch prediction becomes clear. The only three differences between PPM and the 'Predict using the recent branch history' schemes presented above are:

- PPM is a scheme that generalizes to alphabets of arbitrary size, while the branch prediction schemes above are only binary predictors ('taken' or 'not taken').

- PPM uses full counters to maintain probability estimates, while the branch prediction schemes use 2-bit saturating counters in order to observe realistic hardware limitations and

- PPM drops down to successively shorter contexts until a reasonable probability distribution can be estimated, while the branch prediction schemes initialing their distributions with simple a priori assumptions[7].

The connection was only first described by Chen *et al.* [1] in 1996. Not only did they illustrate this connection, but in addition they showed that a full PPM predictor outperforms the less principled history based schemes described above. Performance gains were modest but consistent, achieving a roughly 25% decrease in misprediction rate.

In addition to demonstrating the performance boost of full PPM over the schemes which were devised in an ad hoc fashion and only approximate PPM, Chen *et al.* also enumerate several predictor and table configuration variations worth mentioning. Unfortunately performance measures are not given for these variations, but they nevertheless serve to illustrate the modularity between the prediction method and the features predicted.

In addition to the schemes described by McFarling, in which histories were computed locally (i.e. for each branch independently) or globally (i.e. all branch outcomes in combination) and used to index into a single branch pattern table, Chen *et al.* describe two schemes, one local and one global, which maintain a pattern table individually for each branch. Each of these four combinations (local or global histories used to index into local or global pattern tables) can be used with the simpler single saturating counter method predictor, or can be used with a PPM predictor. The strategy used in the Intel Pentium II and Pentium III processors consists of local 4 outcome histories indexing into local pattern tables containing 2-bit saturating counters [7].

[7] McFarling mentions that initialization with a bias toward 'taken' is used, however, one could imagine using one of the static strategies described in [19] to initialize the counters/distribution. In fact, in the Intel Pentium, and Pentium II and Pentium III processors, such static fall-back schemes are used [7].

5 CTW, Adaptive CTW and Adaptive PPM applied to Branch Prediction: Federovsky *et al.* 1998

Inspired by the observation of the similarity between the prediction of branch outcomes and the predictors used in data compression, Federovsky *et al.* [6] extended the study begun by Chen *et al.* [1]. In addition to examining the performance of adaptive versions of the PPM algorithm, they also consider static and adaptive versions of the context tree weighting (CTW) algorithm introduced by Willems *et al.* [21].

5.1 Scheme: Prediction using CTW algorithm

The CTW algorithm is conceptually very similar to the PPM algorithm, except that CTW uses an unbiased probability density estimate. Given a history match which is followed by a zeros and b ones, PPM estimates the probability that the next symbol x_i conditioned on the past n symbols $x_{i-1} \ldots x_{i-n}$ with:

$$P_{\text{PPM}}(x_i | x_{i-1} \ldots x_{i-n}) = \frac{b}{a+b} \qquad (1)$$

Given a history which is followed by a zeros and b ones, CTW estimates the probability of the next symbol x with the Krichevsky-Trofimov (KT) estimate [21]:

$$P_e(a,b) = \int_0^1 \frac{1}{\sqrt{(1-\theta)\theta}} (1-\theta)^a \theta^b d\theta \qquad (2)$$

Consider an n depth binary tree in which each path from the root to a leaf represents one possible branch history. The path from each node in the tree to the root identifies a partial history of length less than or equal to n. In the case of PPM, we can think of each of these nodes as the lower length contexts which are visited when a full length n match cannot be found. Using such a tree for PPM would require storing the count of branches 'not taken', a, and 'taken', b, for those branches whose partial history, s, matches the path from the tree root to that node. In CTW, instead of simply storing the outcomes of the branches a weighted probability is stored. This weighted probability is defined recursively in the tree, where each weight is a function of its probability and the weights of its children:

$$P_w^s = \frac{1}{2} P_e(a_s, b_s) + \frac{1}{2} P_w^{0s} P_w^P 1s \qquad (3)$$

At the leaves of the tree:

$$P_w^s = P_e(a_s, b_s) \tag{4}$$

The KT-estimator can be computed sequentially using the following relation:

$$P_e(a+1, b) = \frac{a + \frac{1}{2}}{a + b + 1} P_e(a, b+1) \tag{5}$$

This property allows for rapid updating of the weights in the context tree.

The key difference between CTW and PPM is that in CTW, predictions from all partial history matches of all lengths (less than n) are combined to form a single estimate while in PPM only predictions from the longest satisfactory match are used.

The predictions throughout the context tree are combined to form a single probability distribution over the next symbol. The weighted probability of all possible complete subtrees is given by $P_w^\lambda(x_1 \ldots x_i)$ which is the weight in the root node of the context tree. Thus to determine the probability that the next symbol is a 1, i.e. that the next branch is 'taken', a new root weight $P_w^{'\lambda}(\cdot)$ is computed under the hypothesis that the next branch is 'taken'. A prediction is made in the following way:

$$\frac{P_w^\lambda}{P_w^{'\lambda}} \begin{cases} > .5 + 1/\sqrt{n} & \rightarrow & \text{taken} \\ < .5 - 1/\sqrt{n} & \rightarrow & \text{not taken} \\ \text{otherwise} & \rightarrow & \text{taken with probability } \Phi(P) \end{cases} \tag{6}$$

where:

$$\Phi(P) = \frac{P_w^\lambda}{P_w^{'\lambda} - .5 - 1/\sqrt{(}n)} 2/\sqrt{(}n) \tag{7}$$

It is not clear from [6] why a randomized predictor is used. Furthermore, it seems that a better way to determine the most likely branch outcome is to compare $P_w^{'\lambda}$ to $P_w^{''\lambda}$ where $P_w^{''\lambda}$ is the weight of the root node after assuming that the next branch was 'not taken'. This solution may not be used however, because of the additional cost required to update the tree for both the 'taken' and the 'not taken' hypotheses.

The prediction accuracy of the PPM and CTW algorithms were essentially identical. This suggests that the additional conceptual and computational complexity of the CTW algorithm is unwarranted. However, the fact that a compression algorithm which was conceived of completely independently of the branch prediction problem can be applied to this domain and achieve state-of-the-art levels of performance is encouraging. Perhaps other compression techniques can also be applied with as much or more success. In section 7.1 we suggest a technique which is similar to the LZ77 compression algorithm.

5.2 Scheme: Adaptive PPM and CTW

Federovsky *et al.* propose adaptive versions of the PPM and CTW algorithm. They are adaptive in that they build their internal probability models use only data from a limited window of branch history. While they do not mention why this may be a good idea in the body of the paper, in their abstract they suggest that adaptive algorithms may better match "the non-stationary nature of the branch sequence" [6]. This represents a large conceptual shift from the ideas of Smith, who said, when explaining why longer counters are not as good as 2-bit counters, "this [performance decrease] is partially attributed to the 'inertia' that can be built up with a larger counter in which history in the too-distant past is used". Federovsky *et al.* have really made the connection between a *branch outcome sequence* and a series of samples from drawn from a *branch distribution.* When doing prediction, the important issue is the accurate characterization of the *distribution.*

To make implementing adaptive versions of PPM and CTW feasible, Federovsky *et al.* suggest a clever technique. The idea is to maintain a second "shadow tree" which is of the same form as the main tree, but is constructed with branch history data which has been delayed by a fixed number of branches. The values of the shadow tree can then be used to remove the influence of older branches as their age becomes larger than the delay. The length of the delay between the main tree and shadow tree is the effective length of the adaptation window.

Using adaptation windows of 25,000 branches, [6] found that the adaptive methods yielded a 7% performance improvement over their non-adaptive counterparts. Curiously the data also seem to indicate that performance continued to improve through adaptation windows of 30,000 branches; however, they did not present data for larger windows. Trees are maintained for 2^8 hashed addresses using histories of up to 14 branches. This indicates that the maintained trees are only about 0.6% full[8], suggesting that while these techniques are reaching new levels of performance, they may be dedicating

[8]There are 2^8 trees which contain 2^{14} nodes, and $25000/(2^8 * 2^{14}) \approx .00596$

vast resources which are primarily unused. There may be ways of achieving these same levels of performance with far smaller resource requirements, or perhaps even better performance with a redistribution of the same resources.

Nevertheless, techniques based on data compression consistently yield accuracy of 90% or more for predicting the outcome of conditional branches. While there may still be room for some improvement, the inherent branch entropy of the program fundamentally limits prediction accuracy[9] There is only limited room for additional improvement for predicting the outcome of *conditional branches* which have only two possible outcomes; however, predicting the behavior of *indirect branches* which can have multiple outcomes is significantly more difficult.

6 Using Data Compression Methods to Predict Indirect Branches: Kalamatianos & Kaeli 1998

Initial work in predicting indirect branches followed the same course as did early work on the prediction of conditional branches: the ad hoc development of pattern matching counter based methods. In [11], however, a principled approach was taken, based upon the PPM data compression scheme.

Our above description of the PPM presented the algorithm in its general form of multiple size alphabets. When applied to the prediction of two-way conditional branches, a simplified binary version of PPM was used. When applied to prediction of multiple outcome indirect branches the full generality of PPM must be exploited.

Because the potential alphabet of branch targets is huge, either 2^{32} (32 bit addresses) or 2^{64} (64 bit addresses). A direct implementation of PPM on this alphabet would require an enormous probability table. Instead, Kalamatianos and Kaeli suggest an approach in which they apply a hashing function to the target addresses to effectively reduce the alphabet size. Hashing in this way has the effect of merging sets of history patterns together. While there is no theoretical justification for why such a shortcut might be valid, empirical results seem to indicate that hashing collisions do not significantly diminish performance for sufficiently large tables. However, this suggests that there might be a better way to deal with the full alphabet of possible target addresses.

The hashing technique used in [11] is described as *select-fold-shift-XOR-select* (SFSXS). Essentially the branch target addresses are reduced to a fixed number of bits, k, by:

- The number of bits in each selected address is reduced by folding which consists of XORing subsequences of length k or smaller bits together.
- Multiple folded addresses are combined by shifting each by a different amount and XORing them together.
- A PPM context (i.e. lookup index) is computed by selecting a set of k bits from the combined addresses.

The simplifications used in [11] are primarily motivated by realistic hardware restrictions; however, there are several simplifications which seem both unprincipled and unjustified.

The purpose of the SFSXS scheme is to hash the recent target addresses into a smaller domain, but SFSXS is only one of an enormous number of possible methods for doing this. No justification, theoretical or empirical, is given for why this method might be better than any other.

The final selection in SFSXS scheme, which chooses k of the shifted and XOR'd folded address bits, mediates the influence of each address on the outcome history. In [11] they suggest that selecting the first or last k bits is reasonable; however, selecting bits in this way causes either the last or first history target address to effect only one bit in the final hashing key; while the addresses on the other extreme (first or last) effect all k bits. Perhaps a more reasonable scheme is to select k bit so that each address has equal influence, or alternatively select the bits so that there is an influence fall off with increasing age. However this issue is not addressed.

In addition to reducing the effective alphabet size, the technique in [11] also uses a single bit probability estimator, i.e. only the last outcome is remembered and used as a prediction. Thus the scheme can be described as follows: A target address history is maintained for each branch. The addresses in the history are hashed using the SFSXS scheme. The hashed addressed are used to index into a local table containing of the target address taken the last time the branch was reached with a history hashing to the same value.

[9] While branch entropy is immeasurable in practice, any the branches within any program operating on non-static data will have an entropy greater than zero.

While this scheme is an exceedingly simple version of PPM, they were nevertheless able to achieve prediction accuracy of about 91%. The apparent arbitrariness of many of the decisions within this method suggest that a more systematic study may be able to still yield significant improvements. What [11] does illustrate, however, is that even in the presence of significant simplifications, data compression based techniques can yield highly accurate branch predictions.

7 Non-Parametric Compression Techniques Applied to Branch Prediction: De Bonet 1999

History based indexing techniques – including the ad hoc methods and both PPM and CTW – maintain tables which grow exponentially large with the length of the history. This exponential growth causes two problems. First, there is the obvious problem of memory constraints. As history length or complexity increases, the resources available within the branch prediction circuitry are exhausted quickly. The second problem, known as "data scarcity", is more subtle. While an exponential number of probabilities must be maintained, data (branch outcomes) are only acquired linearly. Sufficient data can be collected to "fill in the tables" in the case of the small tables which result from the above restrictions; however, if the history lengths were much longer, or the history elements more complex, most table entries would never be encountered. This is known as the "data scarcity" problem, and while it does not necessarily reduce the effectiveness of a technique (PPM and CTW explicitly handle data scarcity by shifting to simpler contexts) it results in an exponentially increasing fraction of the resources being unused. As a result, the techniques described here severely limit the length and complexity of branch history.

Here we present a non-parametric approach to branch prediction which does not attempt to maintain a table of probability estimates as do the earlier approaches. Instead, earlier data is used directly as a density estimator. While the form of the density estimation is reminiscent of the PPM algorithm, the non-parametric nature of the pattern matching system is similar to LZ77 compression algorithm due to Ziv and Lempel [23].

7.1 Scheme: Predict using branch order matching (BOM)

Because we are not using a tabular method, we are free to consider prediction contexts which consist of strings of symbols from very large alphabets. In this scheme, we consider the history of branch *addresses* – not branch *outcomes* (i.e. 'taken', or 'not taken') – which occurred before the current branch. In the current scheme a list is maintained consisting of the addresses of each encountered branch. As each branch is encountered, its address is prepended to the list. To respect memory limitations, the list is kept to a fixed maximum length by removing the oldest address on the list. It is important to note that this list retains the addresses in the order that they are visited. If visited multiple times, a branch address can appear in the list in several places. For each branch in the list, the outcome of the branch – 'taken' or 'not taken' – is stored as well. A prediction is made by examination of the list and no other supplementary information is needed. The list can be considered a string of symbols, s from the alphabet of possible addresses $\mathcal{A}$. Let x_i be a substring which occurs in s at position i; i.e. $ax_ib = s$ for some a, and b, where $|a| = i$. Note that $a \sqsubset s$ and $b \sqsupset s$ and $|a| + |x_i| + |b| = |s|$.

Each substring of x_i that matches the head of the string contributes to the prediction of the outcome. Let c be the length of a common prefix of x_i and s. The contribution of that substring, k_i^c is proportional to the square of the length divided by the square of distance of the substring from the head of the string. i.e.

$$k_i^c = \frac{c^2}{i^2} \tag{8}$$

At each substring of addresses x_i a branch outcome is also recorded in the history list. If the total contribution from matching 'taken' substrings is greater than the contribution from matching 'not taken' substrings, the branch is predicted 'taken'. If 'not taken' substrings have a largest total contribution, a prediction of 'not taken' is made.

There are several characteristics of this scheme worth highlighting. First, substring x_i will yield no contribution unless x_i begins with the same branch as does s, which is the branch for which a prediction is needed.

Another important aspect, is that if a the longest string match between x_i and s is has length given by

$$c_{max} = \left| \arg\max_y \{y \sqsupset x_i \wedge y \sqsupset s\} \right| \tag{9}$$

then substrings of all shorter lengths (i.e. x_i^1 through $x_i^{c_{max}}$) also contribute to the prediction. In this way information from all partial matches is integrated into the probability estimation, this is in distinct contrast to PPM which uses only the longest match.

Finally, because the list of encountered branches is *ordered chronologically* the age of a partial match can be taken into account when determining its influence. This allows the scheme to deal with non-stationary branching distributions.

7.2 Experiments

To measure the performance of this scheme relative to some of the others presented here, we implemented each scheme within a software microprocessor simulator. We modified the DLX simulator, dlxsim, distributed with Hennessey and Patterson's "Computer Architecture: A Quantitative Approach" [10].

While the distributed version of dlxsim does not contain branch prediction, it can be added without significantly modifying the DLX pipeline simulation. Using the C language compiler dlxcc, which is a modified GNU gcc, and which is distributed with dlxsim, we were able to simulate branch prediction performance on several C programs. Because the dlxcc compiler was not distributed with standard libraries, we were limited in the complexity of the programs we could simulate.

Six programs were tested:

- **SIEVE(150)** - the Sieve of Eratosthenes method for determining primes up to 150
- **TOGGLE(30)** - a simple program which toggles between printing *'s and +'s, 30 times
- **HPSORT(500)** - a program that performs a heap sort of 500 random numbers, modified from [16]
- **SHELL(500)** - a program that performs a shell sort of 500 random numbers, modified from [16]
- **PIKSRT(200)** - a program that performs an insertion (pick) sort of 200 random numbers, modified from [16]
- **FIB(20)** - recursive computation of Fibonacci's number of 20

Four branch prediction schemes were compared:

- **2Bit**: Predict branch using a table of 2-bit counters
- **Hist**: Local history indexing into local pattern table of 2-bit counters
- **PPM**: PPM using local history and local pattern table
- **BOM**: Prediction using Branch Order Matching (BOM)

Each scheme was equated for total memory used. Seven memory configurations were tested, corresponding to PPM with a history of 1, 2, 3, 4, 5, 6, or 7 history bits, maintained for 50 branches. Results are plotted in 1.

Each graph contains 4 curves, one for each prediction scheme. Prediction error percentage is plotted as a function of memory size. In current processors, branch predictors have access to memory comparable to a memory size of 512 bits to 4096 bits. However, future processor technology may be able to dedicate larger resources to branch prediction, potentially making the data larger memory profiles more relevant.

For he first five programs, the current technique BOM, performs the best or among the best. The last program, **FIB(20)**, illustrates one of the failures of this prediction method. Because the recursive computation of Fibonacci numbers uses only one conditional branch, the branch address history used by BOM is not able to capture any structure. This suggests that perhaps a predictor which combines branch outcome with branch address might be better.

While this data is encouraging it is too limited to make any substantial claims. To get a more accurate measure of its performance tests of larger systems, including large commercial software packages, will be needed. In recent papers (e.g. [11, 1]) proposed branch prediction schemes are measured against enormous programs with millions of instructions. While such a study does not produce any theoretically justifiable measure of a scheme's proficiency, it does yield significant empirical evidence *on exactly the types of programs which are expected to be encountered.* Without a good model for the probabilistic distribution over the types of programmatic constructions a processor is likely to encounter, testing on likely samples drawn from the true distribution is the best that can be done.

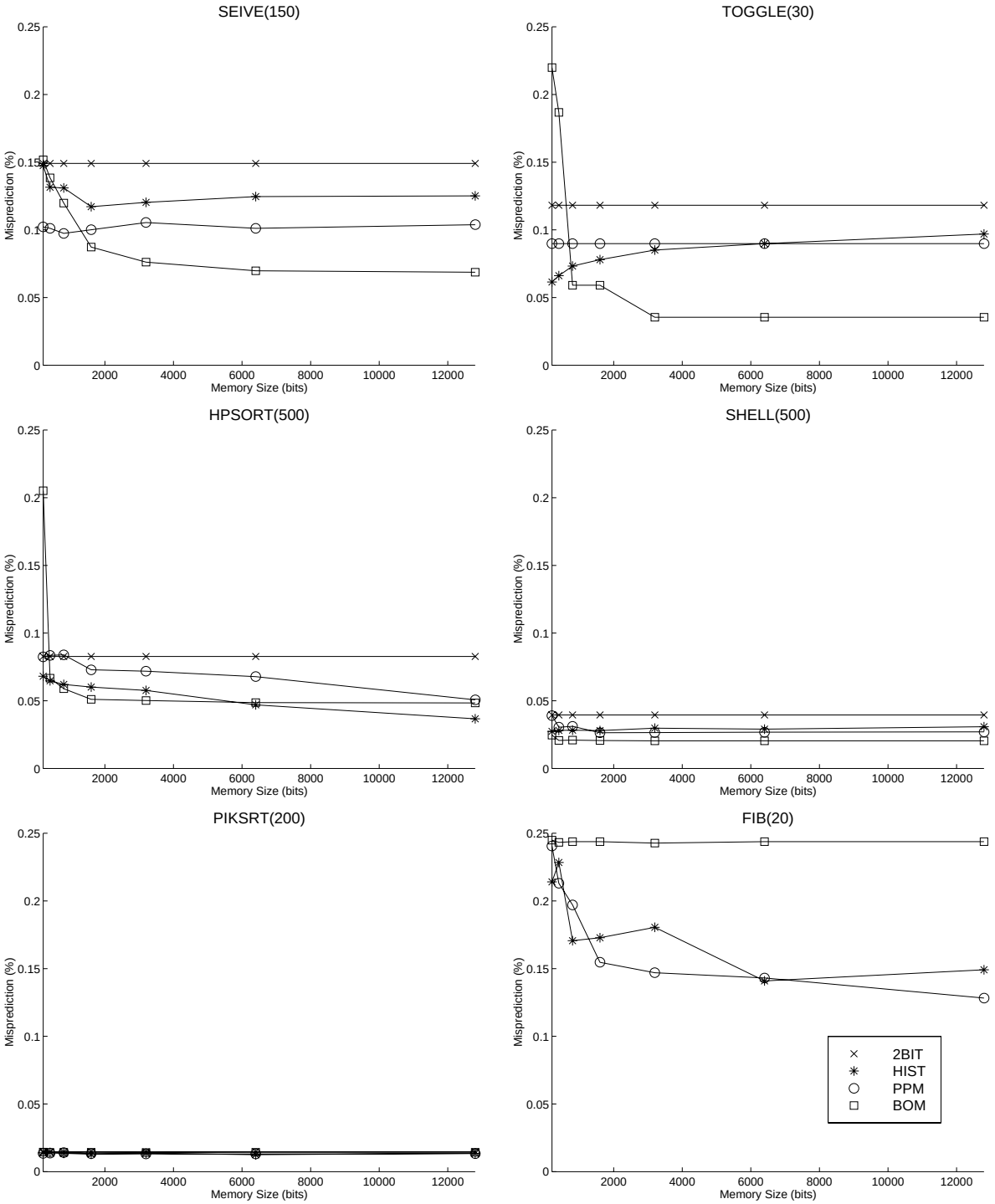

Figure 1: Branch misprediction percentages for 4 techniques, applied to six programs.

7.3 Feasibility

The branch order matching method seems to outperform other methods when equated for memory size; however, the computational complexity of the string matching operation could be prohibitive within the short time (1 cycle) allotted to branch prediction. The computational complexity of this approach, when performed in the naive way, is $O(M^2)$ where M is the total number of addresses stored in the address list. Using the Knith-Morris-Pratt algorithm developed for the general string matching problem, it can be done in $O(M)$ [3]. However, the approach presented is inherently completely *parallelizable* – each string match can be tested independently and in parallel; therefore, using special purpose hardware, this technique could possibly made to work in $O(1)$, and produce a prediction in a single cycle. Since the entire branch prediction system is special purpose hardware that is integrated into the general purpose pipeline, it is feasible that an approach such as the one presented here, could be used in practice.

7.4 Other predictors

Using the same general principles, this approach can modified to use different sorts of predictors. Instead of using the order of branches visited, there are many other possible predictors that could yield better results. Possibilities include: branch target address or register values, or some combination of these predictors.

Other approaches, including PPM and CTW restrict both the length and alphabet size of their contexts. The major contribution of this method is its framework, which can be used with any context without length or alphabet size restrictions.

8 Conclusions

We have reviewed many of the techniques which have been developed for branch prediction. More recent approaches, based on ideas borrowed from data compression yield significant improvements. Perhaps these approaches will be integrated into the development of future processors[10].

Additionally we have presented a new non-parametric branch prediction technique. This technique is similar to the string matching within the LZ77 data compression algorithm. While the computational complexity of this approach provides some barrier to its usage within the narrow time window afforded during branch prediction, it is inherently parallel and could perhaps be done efficiently using special circuitry. This new technique achieves significant performance improvements over other approaches for most of the tested examples; however, additional testing, including tests of much larger systems will be required to get a more accurate measure of its performance. More importantly however, this technique suggests a whole new class of potential prediction algorithms.

[10] However, the newest Intel processor, code named Merced, does not use branch prediction. It instead uses a technique called *predication* [15]

References

[1] I-C. K. Chen, J. T. Coffey, and T. N. Mudge. Ananysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 128–137, Cambridge, MA, October 1996.

[2] J. Cleary and I. Witten. Data compression using adaptive coding and partial string machines. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[3] T. H. Cormen, C. R. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, first edition, 1990.

[4] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[5] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[6] E. Federovsky, M. Feder, and S. Weiss. Branch prediction based on universal data compression algorithms. In *Proceedings 25th Annual Symposium on Computer Architecture*, pages 62–72, June 1998.

[7] A Fog. Branch prediction in the pentium family. http:// www.x86.org / articles / branch / BranchPrediction.html.

[8] Y. Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

[9] Edmondson J. H., P. I. Rubinfeld, and etal. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1990.

[11] J. Kalamatianos and D. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, November 1998.

[12] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corp., June 1993.

[13] R10000 technical briefing. http:// www.sgi.com / processors / r10k / tech_info / Tech_Brief.html.

[14] A. Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

[15] J. C. Park and M. S. Schlansker. On predicated execution. Technical report, Hewlett Packard Laboratories, 1991.

[16] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipies In C: The Art and Science of Computing*. Cambridge University Press, New York, 1988.

[17] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[18] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, October 1948.

[19] J. E. Smith. A study of branch predition strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[20] The ultrasparc processor – technology white paper. http:// sunsite.ics.forth.gr / sunsite / mirror1 / sun_microelectronics / UltraSparc / ultra_arch_architecture.html.

[21] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context tree weighting method : basic properties. *IEEE Transactions on Information Theory*, pages 653–664, May 1995.

[22] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive training branch predition. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 51–61, November 1991.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, 1977.