Journal of Instruction-Level Parallelism 2 (1999) 1-6 Submitted 3/99; published 12/99

Indirect Branch Prediction using Data Compression Techniques

John Kalamatianos KALAMAT@ECE.NEU.EDU
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA

David R. Kaeli KAELI@ECE.NEU.EDU
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA

Abstract

Branch prediction is a key mechanism used to achieve high performance on multiple issue, deeply pipelined processors. By predicting the branch outcome at the instruction fetch stage of a pipeline, superscalar processors become able to exploit Instruction Level Parallelism (ILP) by providing a larger window of instructions. However, when a branch is mispredicted, instructions from the mispredicted path must be discarded. Therefore, branch prediction accuracy is critical to achieve high performance. Existing branch prediction schemes can accurately predict the direction of conditional branches, but have difficulties predicting the correct targets of indirect branches. Indirect branches occur frequently in Object-Oriented Languages (OOL), as well as in Dynamically-Linked Libraries (DLLs), two programming environments rapidly increasing in popularity. In addition, certain language constructs such as multi-way control transfers (e.g., switches), and architectural features such as 64-bit address spaces, utilize indirect branching.

In this paper, we introduce and describe the use of a data compression technique as applied to the field of unconditional indirect branch prediction. We base our approach on techniques proven to work effectively in the field of data and text compression and more specifically, on the *Prediction by Partial Matching* (PPM) algorithm. We present realistic implementations of the PPM algorithm combined with dynamic per-branch selection of path correlation type, and compare its prediction accuracy against a variety of predictors. We also present results for a PPM predictor coupled with a filter. Finally, we perform cycle-based experiments in order to not only compare its performance to other predictors but to evaluate its cost effectiveness against conditional branch prediction in a dynamically scheduled, superscalar processor. Our findings are encouraging because the non-filtered PPM predictor can achieve a misprediction ratio of 10.55% (9.58% if we include a filter) as compared to 12.07% for the previously published most accurate indirect branch predictor, for approximately the same hardware budget. This improvement in prediction accuracy can translate into a significant reduction in execution time (by as much as 2% in the gcc SPEC'95 benchmark).

1. Introduction

Superscalar, deeply pipelined processors require a continuous stream of instructions to execute, in order to effectively utilize available hardware resources. Control-flow changes introduced by branches (conditional and unconditional, direct and indirect), disrupt the steady flow of instructions to the processor. Therefore, multiple issue execution is not able to exploit a high degree of ILP [1]. A branch predictor working at the instruction fetch pipeline stage can reduce the impact of control-flow breaks by predicting the branch outcome and fetching instructions from the predicted path. Speculatively executed instructions must be discarded when a branch is mispredicted. High accuracy of a branch predictor is essential to achieve high performance when using speculative execution [2].

Predicting a branch correctly involves two issues: (i) predicting the correct branch outcome (taken or not taken), and (ii) predicting the correct branch target. Branches can be classified using two characteristics: (i) type of transfer (conditional or unconditional), and (ii) type of target address generation (direct or indirect). Of the four possible combinations, conditional indirect branches are typically not implemented. Conditional direct branches transfer control flow to either the next sequential address (not taken) or to a backward/forward location within a range (taken). One major goal becomes to predict the direction of the branch. A plethora of branch predictors have been proposed, achieving very high prediction ratios. Unconditional direct branches are the easiest to predict since they always jump to a single target that is usually known at compile-time. Our work here is focused on the correct prediction of unconditional indirect branches, which will be referred to simply as indirect branches throughout the rest of this paper. Although their direction is known (always taken), they can visit multiple targets. They typically use the contents of a register, sometimes combined with a fixed offset, to compute the target address at run-time. Accurately predicting the next target of an indirect branch is a difficult task because it does not involve a binary decision as in the case of conditional branches. Past history of indirect branches is also difficult to capture since it requires storing an entire 32-bit or 64-bit address instead of just a direction bit. Currently available microarchitectures support indirect branch prediction using a Branch Target Buffer (BTB) which caches the most recent target of the branch. As a result, correct prediction rates are very low, even if a very large BTB [3] is implemented.

Indirect branch frequencies are not currently as high as conditional branches, but their associated misprediction overhead can be substantial, especially for architectures that exploit ILP. Recent studies have reported that indirect branches appear much more frequently in object-oriented languages (i.e., C++) than they do in procedural languages (i.e., C) [4]. The programming paradigm found in most OO languages such as C++, Java, Smalltalk and Self supports polymorphism based on run-time procedure binding [5]. Most implementations generate an indirect function call for every polymorphic call [6]. Indirect branches commonly appear in applications that use dynamically-linked libraries since they implement DLL calls [7].

The relative performance impact of indirect branches will be further increased as techniques such as *predicated execution* [8], *branch elimination* [9] and *coalescing of conditional branches into indirect jumps* [10] are increasingly applied. By reducing the number of conditional branches (sometimes at the expense of introducing extra indirect jumps [10]), these techniques will generate a higher percentage of indirect branches (relative to conditional branches), and thus, our ability to correctly predict indirect branches becomes increasingly important.

In this paper we present techniques, used in the field of data compression along with dynamic per-branch correlation selection to improve the accuracy of indirect branch prediction. We evaluate our approach in the context of a superscalar machine employing speculative execution although our technique can be applied to any machine exploiting ILP via speculative execution. This paper is organized as follows. In Section 2 we review related work in the field of indirect branch prediction. Section 3 introduces the work done in the field of data and text compression and its association with branch prediction while section 4 presents the PPM algorithm as it is applied to the field of indirect branch prediction. Section 5 includes experimental results. Finally Section 6 suggests directions for future work and concludes the paper.

2. Related Work on Indirect Branch Prediction

The Branch Target Buffer (BTB) is the first predictor that could be used to provide the next target for indirect branches [11, 12, 13]. In its simplest form, a BTB is a last-value predictor; it uses the most recently seen target to predict the next target for a branch. The indexing function is usually formed by taking the lower-order bits of the branch address. A n-bit saturating counter normally controls replacement in a direct-mapped BTB. Experiments in [14] showed that on the average a 2-bit counter achieved a lower misprediction ratio compared to a configuration where a target is replaced on a single misprediction. The 2-bit counter provides hysteresis allowing a target address to be replaced only after two consecutive mispredictions (we will refer to this BTB variation as *BTB2b*). More complex replacement strategies have been proposed for set-associative BTBs [13].

In [15], Kaeli and Emma describe a mechanism which accurately predicts the targets of subroutine returns. The mechanism is called a *Call/Return Stack* (RAS) and targets the prediction of subroutine returns by using the inherent correlation between procedure calls and returns to pair up the correct target address with the current return invocation. Several studies have concentrated on techniques of properly updating a RAS and/or recovering its correct state during speculative execution mode [16, 17]. In our work, we will not address issues related with the predictability of return instructions.

In [18], the *Case Block Table* (CBT) is proposed for predicting the next target of a switch statement. Depending on compiler heuristics, an indirect jump may be used to implement the control transfer for a switch statement. In that case, the CBT is essentially another indirect branch predictor. The CBT uses switch variable values to find the next target of a switch statement. It is able to resolve the switch statement targets since the switch variable value is an optimal predictor for a multi-way branch. The problem with the CBT is that the value of the switch variable is not always known at the time the code for the switch statement reaches the instruction fetch stage of a superscalar machine employing speculative execution [19].

In [3], Driesen and Holzle propose using two-level adaptive predictors for indirect branches. They examined a modified structure of the two-level adaptive branch predictor (originally proposed in [20]) where the first level consists of one global, or multiple (per branch) history registers that record partial previous targets of branches allowing path-based correlation to be exploited. Since every register records previous path history, it is called a *Path History Register* (PHR). The number of targets recorded in the PHR is called the path length. The second level consists of an array of 2-bit counters (Pattern History Table, PHT) that control the replacement of targets which are also held in the 2nd level. The array is indexed using various indexing functions. The most common is the lower-order bits of a branch's address. Another effective hash is the exclusive-OR between the branch's address and the contents of the selected PHR (gshare). In [3, 21], the authors proposed grouping and interleaving bits of the PHR and the branch's address.

Driesen and Holzle simulated a large number of ideal two-level predictor configurations. They obtained the best prediction accuracy using the GAp structure (Global PHR, Per-address PHT configuration) for their set of benchmarks. In more recent work [21], Driesen and Holzle explored realistic designs, varying the path length the size and associativity of the PHTs, and the hashing function. They also proposed a hybrid design, namely a dual-path predictor, where each component is a two-level predictor. The two components differ in their path length. A set of confidence counters, placed in the PHTs (and not in a separate array as originally proposed in [22]) was used to select which component would make a prediction. Their findings suggest that the best dual path predictor should have components with both a short and a long path length showing that most branches tend to be either short or long length correlated.

Chang et al. proposed using a family of two-level predictors [19], named Target Cache (TC), with a structure similar to the two-level adaptive scheme described in [3]. The most important feature of the *Target Cache* (TC) predictor is that the PHR records partial targets from a selected group of branches. In [19] the authors select branches based on their opcode, leading to the following types of path history : Per Branch path history (PB) from the targets of all branches, Per Conditional Branch (PCB) path history from the targets of only conditional branches, Per Indirect jump (PIJ) path history from the targets of indirect jumps and Per Procedure Call/Return (PPCR) path history from the targets of procedure call and return instructions. Each TC configuration utilizes PHR(s) from only one group of branches. The different contents recorded in the PHR allows variance in the type of path correlation across multiple TC configurations. The simulation results clearly show the dependence of indirect branch predictability on the type of path-based correlation.

Driesen and Holzle in [23] proposed using a filter along with a main component to construct a hybrid indirect branch predictor, namely the Cascaded predictor. The authors present a path-based predictor (hybrid dual-path) as the main component and a BTB-like structure as a filter. The latter is used to dynamically classify and predict easily predictable branches while the former predicts the remaining branches. By preventing insertion of the easy to predict branches into the main component, phenomena such as aliasing, cold-start and capacity mispredictions due to path history are significantly reduced.

The BTB structure of the filter permits the prediction of *monomorphic* [1] and *low entropy* branches [2]. Two different protocols were proposed to determine when and how prediction would be made by a filter: i) strict and ii) leaky. A strict filter prevents new entries from entering the main component when the branch is not found in the filter and allows the creation of an entry in the main component when a branch is mispredicted by the filter. In order to distinguish between a miss and a misprediction, the strict filter must maintain tags. A leaky filter allows entries to be created in both components on a branch miss. Only

1. A branch is monomorphic when it mostly accesses one target.
2. A branch has low entropy when its target changes infrequently.

correctly predicted branches (by the filter) avoid updating the main component. Since a branch miss and a misprediction are handled the same by a leaky filter, no tags are necessary in its implementation. The authors found that a leaky filter led to overall lower misprediction ratios than a strict filter [23].

In [24], Stark et al. proposed a branch prediction that exploits profile-based variable length path correlation for both conditional and indirect branches. In their implementation, a set of hashing functions combining a variable number of targets, is implemented in hardware. A PHR records the N most recently encountered branch targets and feeds them to hashing function hardware. The latter generates a set of indices which feed a multiplexer that selects one of them. The multiplexer is controlled by information bits (i.e., a hashing function index). These bits are found after profiling every branch. The compiler/linker is responsible for communicating the hashing function indices to hardware. The solution described in [24] uses the displacement field of the Alpha AXP indirect branches to store these indices. A table, indexed by the branch's address, is used to store these bits in hardware. The profiling step selects, for each branch, the hashing function that achieves minimum misprediction ratio. The index that goes through the multiplexer drives a 2nd-level table (similar to a PHT) that actually makes the prediction.

Recently, Roth et al. suggested using a dataflow engine to complement conventional path-based predictors in order to improve the predictability of virtual function calls [25]. Their solution detects the instruction sequence implementing a virtual function call [3], records the corresponding dependencies and launches instructions via a dataflow engine. These instructions are issued in the same order that object references are made, in an attempt to predict the next target before the indirect call is executed. The authors also propose the use of a lookahead scheme where stride-based address prediction can assist in early instruction launch, before their associated object references are detected in the dynamic instruction stream. Simulation results showed that their dependence-based prediction scheme can improve the misprediction ratio of a set-associative BTB and a two-level path-based predictor. Fig. 1 displays most of the predictor structures that have been described above.

Our work focuses on applying techniques taken from the field of data and text compression to the task of indirect branch prediction. More specifically, we examine the performance of the PPM algorithm. We show how this algorithm exploits variable-length path correlation. We propose coupling a PPM-based implementation with dynamic per-branch selection of path correlation type. We discuss implementation issues, describe one design implementation and present simulation results. We also compare the proposed design with previously described predictors, assuming the same hardware budget.

3. Data Compression Algorithms and Branch Prediction

In this section we present an overview of the compression techniques that have been proposed so far for the task of branch prediction. We discuss two major compression techniques: the first is the *Prediction by Partial Matching* (PPM) [26, 27] algorithm that has been proposed for predicting conditional branches in [28] and the second (and more recent) one is the

3. When virtual function tables (VFT) are used to support polymorphism, the instructions generated for single inheritance virtual function calls consist of a sequence of loads and an indirect procedure call [6].

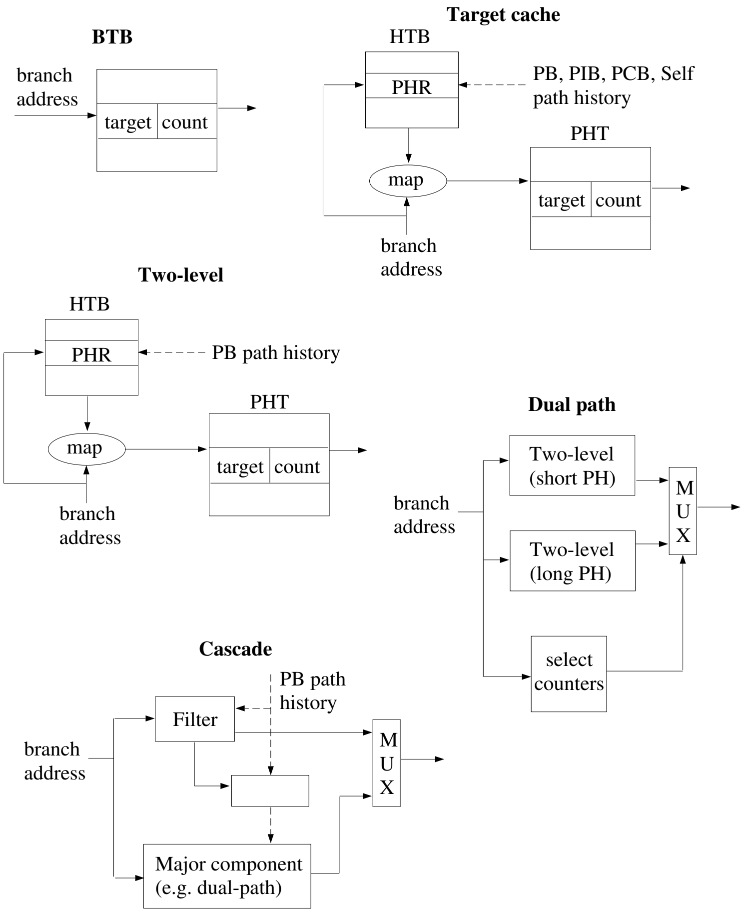

Figure 1: Indirect branch predictors.

Context Weight Tree (CWT) method [29] which has been presented in [30] as a another potential conditional branch predictor.

The primary goal of data compression is to represent an original data sequence with another that has fewer data elements. Modern data compression techniques try to use fewer bits to represent frequent symbols, thus reducing the overall size of data. The text/data compression algorithm usually defines a probabilistic model that is used to predict the next symbol with high accuracy. The ability of compression algorithms to be successfully applied in prediction stems from the close relation between compression and prediction: a sequence that is compressed well is easy to predict. On the other hand, a random sequence is both non-compressible and unpredictable [31, 32].

The CWT method builds a binary tree of length d, based on past branch outcomes. Every tree node is assigned a context, which is the unique bit sequence generated if we traverse the tree from the root to the node. The root has a null context. A node will be created only if its context has been seen in past branch history. Given a set of past branch outcomes S, the set of bits immediately succeeding the context of a node in S is called the subsequence of the node. The tree is called a context weight tree (CWT) because every node is assigned a weighted probability (P_w) which is derived from the conditional probability of finding the subsequence associated with the node (P_e) and the weighted probabilities of the node's children. The weighted probability of the root weights the probabilities induced by all possible complete subtrees. In order to predict the next branch outcome we assume that it is 1 (taken), append the new outcome in the branch history, recompute the weighted probabilities of all nodes in the path that are affected and find the ratio of the new over the old weighted probability of the CWT's root node. A randomized predictor is used to make the final decision as follows:

- if ratio $> 0.5 + \varepsilon \rightarrow$ prediction $= 1$ (branch is taken).
- if ratio $< 0.5 - \varepsilon \rightarrow$ prediction $= 0$ (branch is non-taken).
- else prediction $= 1$ with probability $\Phi(P)$

where $\varepsilon = \frac{1}{\sqrt{T}}$ and T is the branch history length.

The authors in [30] also discuss variations of the CWT algorithm, where they add bits from the branch address in the past branch history to further increase the depth of the context tree. In addition, the original CWT construction algorithm assumes that a complete context tree is built, where the total number of nodes is determined only by the branch history. This version of the algorithm is called static. By setting a limit on the number of nodes (and levels), Federovksy et al. create an adaptive version of the CWT algorithm. They limit the number of CWT nodes by deallocating nodes that no longer carry information within the branch history window. The number of CWT levels is limited by not allowing the extension of a particular context at the leaf node when a threshold is reached. The original CWT algorithm carried the following per node information:

- a count of 1s (denoted by b) and 0s (denoted by a) of the node's context,
- the probability estimate (P_e) of the node's subsequence, and
- the weighted probability (P_w) of the node.

The authors describe a static PPM algorithm based on a CWT, where every PPM node carries only counts of 1s and 0s. No weighted probabilities are computed or saved. Prediction is made by traversing the tree path (from leaf to root) determined by the current branch history until the number of 0s and 1s differ. The idea is to find the deepest context where sufficient prediction information exists. The conditional probability, given by $\frac{b+0.5}{a+b+1}$ makes a prediction using the same algorithm as in CWT. An adaptive PPM algorithm can also be derived if we consider generating and deallocating nodes as in the adaptive CWT algorithm. In [30], idealized versions of these algorithms are applied to conditional branch prediction with success. However, a realistic hardware implementation has yet to be shown.

Another version of the PPM algorithm is described in [28]. That version has been successfully applied to areas such as data compression [27], file prefetching [33] and cache replacement policies [34]. Again, the PPM predictor computes probabilities for symbols (as is the case with the CWT and the CWT-based PPM algorithm). Its corresponding version for conditional branches computes probabilities for branch outcomes, coded as 1 (taken) and

0 (non-taken). It provides the next outcome given the past sequence of branch outcomes that have been observed (branch history) and their relative dynamic frequency.

The foundation for the PPM algorithm of order m is a set of $m+1$ Markov predictors [27]. A Markov predictor of order m produces the next outcome using its m immediately preceding outcomes, acting like a Markov chain. It consists of 2^m states since every conditional branch has two outcomes. The frequency of branch outcomes guides the transitions among the different states. The predictor arrives at a certain state when it has seen the m-bit pattern associated with that state. It records the frequency counts by counting the number of times a 1 or 0 occurs in the $(m+1)$-th bit following the m-bit pattern. The predictor builds the transitional probabilities (determined by the frequency counts), and at the same time it predicts the branch outcomes. The prediction is made by selecting the next state that possessed the highest transitional probability (highest frequency count).

Figure 2 shows a 3rd order Markov predictor that has processed the input sequence (previous conditional branch outcomes) 01010110101. Since the order of the Markov predictor is 3, we expect to see a maximum of $2^3 = 8$ states. However, based on the input sequence already seen, the model has recorded transitions to 4 out of the possible 8 states.

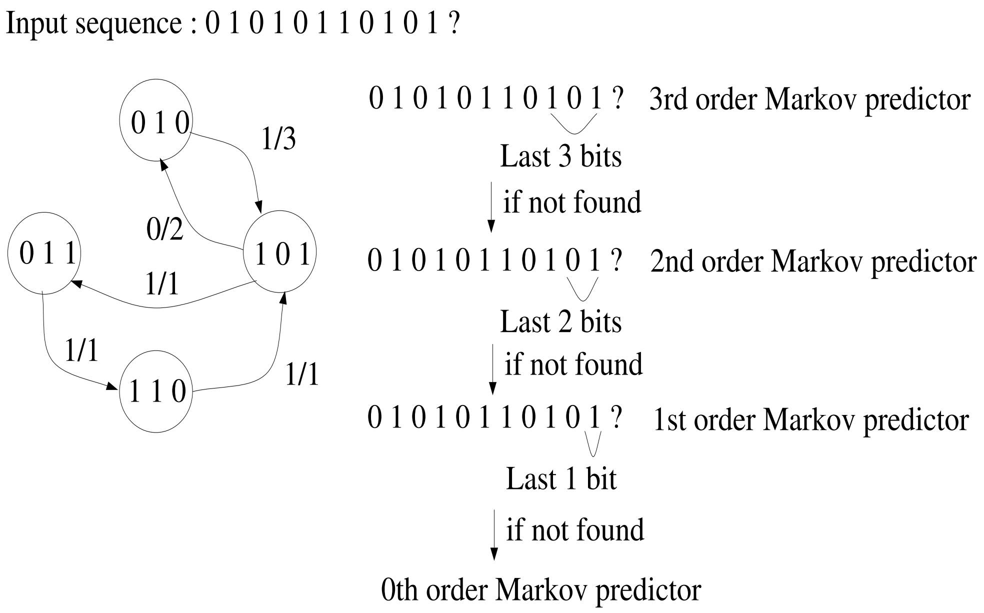

Figure 2: 3rd order Markov Predictor and its Prediction Algorithm.

Edges between states are established when one state follows the other. Every edge is labeled with a pair of numbers, the first number indicating the next predicted bit (that leads to the state at the end of the arc) and the other denoting the frequency count for that transition. For example, in Figure 2 state 110 has an outgoing edge to state 101, because the bit pattern 101 has been seen to follow the pattern 110 in the input sequence. Furthermore, since this transition has only been seen once, its frequency count is 1.

Given the input sequence shown at the top of Figure 2, the predictor has arrived at state 101 and needs to predict the value of the next bit. Pattern 010 has followed 101 twice, while pattern 011 has followed 101 only once. Therefore, the next state should be 010 and the predicted bit will be 0.

As mentioned above, a PPM predictor of order m consists of a set of $m+1$ Markov predictors. At step 1, the mth order Markov predictor is searched for a pattern formed by the m immediately preceding bits in the input sequence. If a match is found (meaning that the state associated with the pattern has at least one outgoing edge), a prediction is made. If the Markov predictor fails to find the pattern (meaning that the state associated with the pattern has no outgoing edges), it looks into the next lower order Markov predictor for a pattern formed by the $m-1$ immediately preceding bits. This process is continued until a prediction is made. Notice that a 0th order Markov predictor will always make a prediction based on the relative frequency of 0s and 1s in the input sequence. Returning to Figure 2, the PPM algorithm will first search for the 101 bit pattern in the 3rd order Markov predictor, then it will look for pattern 01 in the 2nd order Markov predictor, etc..

The update step of the frequency counts across the different Markov models in the PPM algorithm varies. Throughout this paper, we will use an *update exclusion policy* [28]. This protocol forces all predictors with a lower order (i.e., lower than the one that actually makes the prediction) not to be updated. Only the predictor that makes the decision, and the predictors with a higher order, are updated.

In [28], Chen et al. discussed how a PPM predictor of order m can be seen as a set of $m+1$ two-level adaptive predictors, thus providing us with a theoretical basis for supporting the observed predictability of the later. The key idea is that a Markov model, as shown at the top of Figure 2, is successfully emulated by a two-level adaptive predictor. The m-bit input sequence is recorded in an m-bit history register (HR). The same m-bit sequence can represent global or per branch history. The lookup of the pattern in the Markov model is simulated by the lookup of the PHT using the contents of the HR. The 2-bit up/down saturating counters, one provided per PHT entry, efficiently mimics the transitional probabilities. The states of the Markov model correspond to the entries in the PHT on a 1-to-1 basis.

The PPM algorithm of order m requires $m+1$ adaptive predictors; each one of the adaptive predictors uses an HR of different length. Therefore, the PPM algorithm for conditional branches makes a prediction using information from history lengths that range from m to 0. In other words, the PPM algorithm explores variable-length branch correlation. The 1st order Markov predictor is simply a 1-bit history predictor (i.e., it predicts the next branch outcome using its previous outcome). The protocol of the PPM algorithm we described above assigns priority to long-term correlation (assuming that m is large enough to reveal long-term branch correlation) since a prediction will always be made from the highest order Markov predictor that detects the pattern in the HR. Although this is necessary when performing data compression, it may not be optimal when applied to branch prediction. This issue is not addressed in this paper.

If we consider indirect branch prediction, a PPM predictor of order m will again consist of $m+1$ Markov components, each one being indexed by a PHR of different path length. The path length will range from m to 0, and therefore the PPM predictor will exploit variable-length path correlation. The 1st order Markov predictor will simply use the previous target to predict the next one (similar to a BTB entry). A PPM predictor for indirect branches will utilize a PHR to record past history and will exploit variable-length path correlation.

The only difference between the ideal PPM models for conditional and indirect branch prediction is that the total number of states for an mth-order Markov component is not 2^m but k^m where k is the maximum number of valid targets in the program. Driesen and Holzle proposed using two PHRs with different path lengths [21], while Stark et al. described a scheme where each branch is assigned one path length that is found via profiling. The path length per branch is fixed for the entire execution of the application. Our approach is fully dynamic. All path lengths in the range of 0 to m are used to provide a target for every branch.

4. PPM Predictors for Indirect Branches

In this Section we will discuss the problems of applying the PPM algorithm to indirect branch prediction, describe our implementation and discuss related design tradeoffs.

The major goal in indirect branch prediction is to correctly predict the target of the branch. Since indirect branch target address selection is not a binary decision, a direct approximation of the PPM algorithm is not straightforward. Accurately describing path history requires storing 32-bit or 64-bit addresses versus storing a single bit (i.e., the branch direction) in the case of conditional branches. Indexing a state in the Markov chain with the contents of past history is not straightforward. As discussed previously in Section 3, the number of possible states in each Markov component becomes k^m. A PPM predictor would require enormous hardware resources in order to represent all possible states of the Markov components.

Our solution is to use multiple BTB-like structures to implement the Markov chain states. These structures are equivalent to PHTs in two-level predictors and we will refer to them as PHTs. We use a PHR to record past history and a hashing function to form indices accessing the multiple PHTs. By limiting the size of the PHTs we are merging sets of the Markov chain states together. The final number of states after merging is equal to the number of PHT entries. The hashing function proposed in [19, 21] uses a gshare indexing scheme, where groups of bits from previous branches (conditional and/or indirect) are XOR'ed with the PC to form the index. In our case, we use a modified version of the *Select-Fold-Shift-XOR* (SFSX) hashing function described in [35]. Figure 3 shows an example of how our new mapping function called *Select-Fold-Shift-XOR-Select* (SFSXS) forms an index from the path history.

We assume a path length of 10 and a 10th order PPM predictor to illustrate how our indexing function works. The SFSXS mapping function selects the low-order 10 bits from target i [4], folds the selected value into 5 bits which are left shifted by i bits and XOR'ed together with the other values. The j high-order bits of the outcome form the index for the j-th order Markov predictor. An alternative solution would select the j low-order bits. From simulation results, we found little difference in the misprediction ratios when comparing these two schemes, so we decided to use the scheme shown in Figure 3.

Another major issue encountered when implementing the PPM algorithm for indirect branches is the selection of the next target. The original Markov model requires multiple outgoing arcs from each state, keeping frequency counts for each possible target. Obviously, this step does not involve a binary decision and can not be modeled by a 2-bit counter. It

4. Since instructions are word-aligned, we always ignore the last two bits of their addresses.

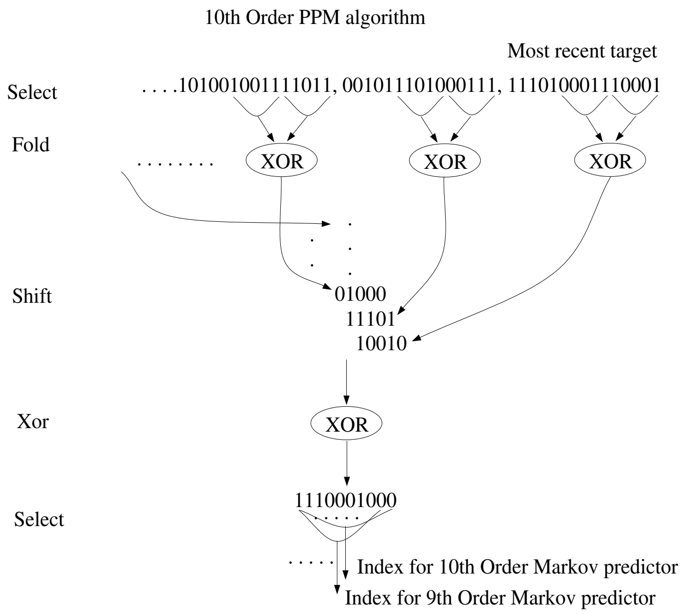

Figure 3: Select-Fold-Shift-XOR-Select Indexing Function.

requires storing multiple targets per PHT entry along with their frequency counts, and a majority voting mechanism to select the next target. Instead we store the most recently visited target and use it to make a prediction.

The complete design for a 3rd order PPM predictor is shown in Figure 4. We assume an implementation of the PPM predictor at the I-fetch stage of a processor employing speculative execution. The logical components forming the predictor are a Branch Identification Unit (BIU) that indicates that an indirect branch is being fetched. Clearly, we require one bit per branch to indicate whether it is conditional or indirect. We also need a PHR that records partial targets from a predetermined branch stream. To use the SFSXS mapping function, we will need to record 3 bits from each one of the last 3 targets. The branch stream considered will be global (one PHR for all branches) and will record the targets of all indirect branches (PIB path history). The boxes labeled as *Hashing function* implement the indexing scheme described in Figure 3. The Markov predictors are basically BTB-like structures, where every entry includes the most recently accessed target, a tag, a 2-bit up/down saturating counter for replacement and a valid bit. The 2-bit counter controls target replacement (the target currently found in an entry is replaced only on two consecutive misses). Finally, logic controls the target that will be selected from the Markov components according to the PPM protocol. Markov predictors can also be tagless. In the absence of tags, the only way to indicate a non-zero frequency count for a state of the Markov model is the valid bit. If we allow an update-exclusive update policy for the PPM predictor, most of the queries for a prediction will be directed to the highest order Markov component, leading to an underutilization of the available resources. For simulation results on a tagless PPM predictor and its relative performance with other tagless predictors, see [36]. In this paper we will focus on tagged implementations.

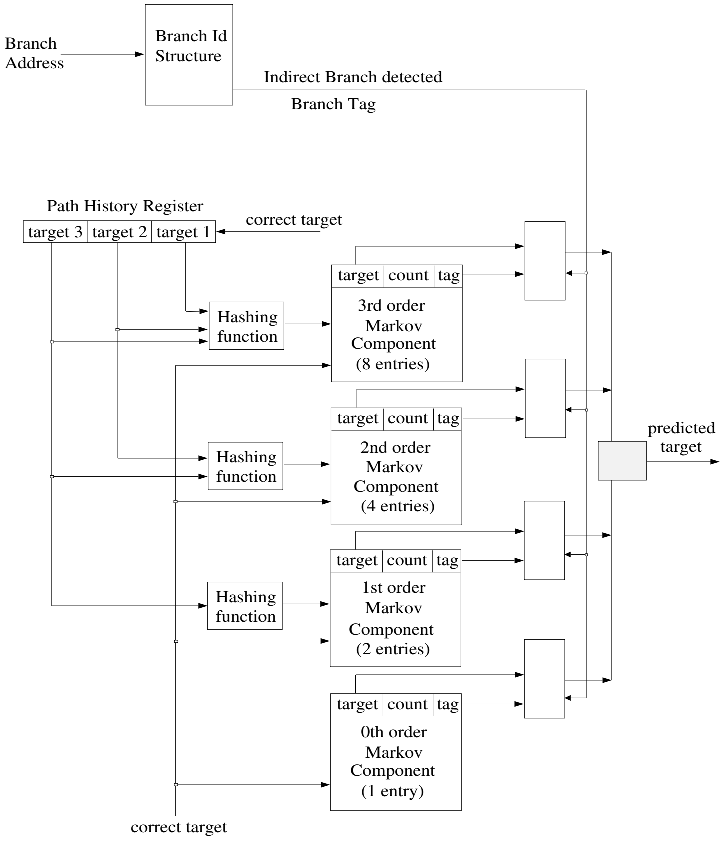

Figure 4: Implementation of a 3rd order PPM predictor.

The PPM predictor works as follows: the BIU unit is indexed with the branch address every time the processor fetches instructions and looks for indirect branches (as well for other branches). At the same time, the PHR is used to generate the different indices and accesses all the Markov predictors in parallel. If an indirect branch that is to be predicted [5], is detected in the instruction stream, a tag comparison is made to examine if the entry in the Markov component belongs to the branch under consideration and then the highest order Markov predictor with a tag match provides the next target. If there is no tag match no prediction is made (this event is classified as a misprediction in our experimental results). When updating the predictor, if a tag mismatch occurs, we only update the Markov component with the lowest 2-bit counter value. If there are multiple Markov components with the same value, we update the one of the highest order. This prediction step requires one level of table access, and thus, the predictor can be characterized as 1-level table.

The update step starts by shifting the actual target to the PHR, independent of whether we had a misprediction or not. By maintaining a bit string of m bits showing which component made a prediction, we can implement an update exclusion policy [28]. Each bit string can be associated with an indirect branch as it flows through the different stages of

5. Return instructions are ignored and indirect branches that are known statically to have a single target are not directed to the PPM predictor. See Section 5 for further discussion on the later issue.

the pipeline (one possible solution is to keep it as a separate entry in the reservation station of the branch unit, along with the indirect branch's information). For each component that needs to be updated, we set the valid bit (if not set) and update the target (either speculatively or with the actual outcome - timing issues related to updating are not discussed in this work). Finally we properly modify the state of the counter and the tag if the branch is being replaced in the PHT entry. All components can be updated simultaneously.

Previous studies have shown that the predictability of indirect branches varies, especially when it comes to inter-branch correlation behavior [19]. Our past experimental results revealed that most indirect branches were best correlated with either all previous branches or with previous indirect branches (labeled as Per Branch or PB-correlated, and Per Indirect Branch or PIB-correlated, respectively) [37]. Therefore we have combined the PPM predictor with a selection scheme that dynamically chooses between two types of path-based correlation of the same length : PB and PIB path-based correlation. Since we want to characterize the correlation of every individual indirect branch, we introduced a set of 2-bit up/down saturating counters in the BIU module. Based on the contents of the associated counter, the predictor selects between the two PHRs. The new design is shown in Figure 5.

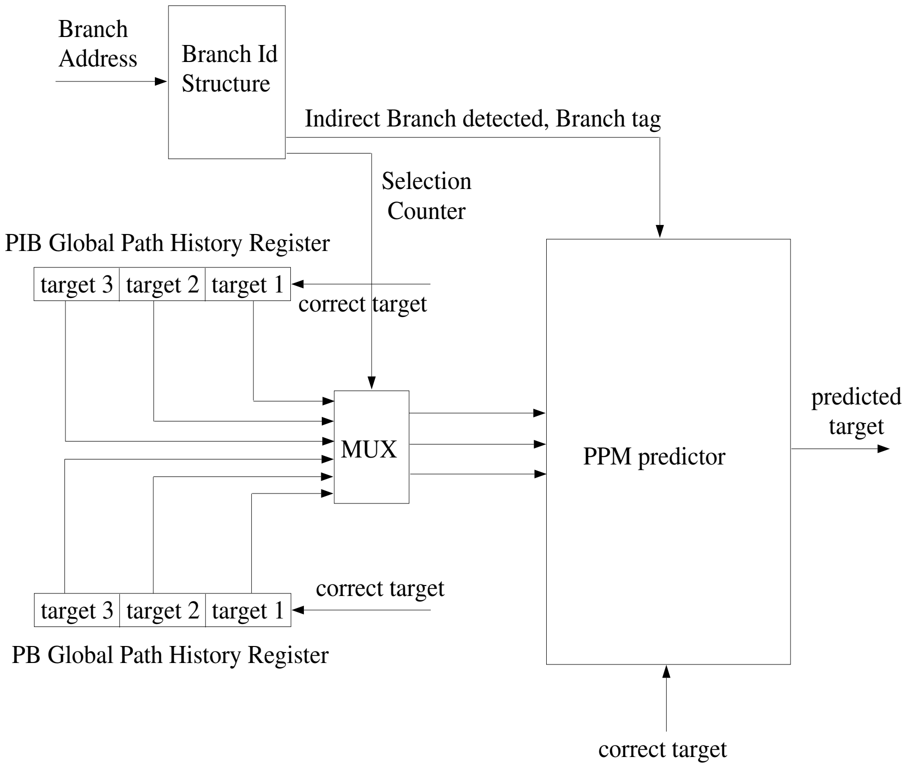

Figure 5: PPM predictor with Run-time Selection of Correlation Type.

Notice that this version of the PPM predictor requires two levels of table access (one for the BIU and one for the PPM predictor), and therefore it should be considered a *2-level predictor*. Depending on the physical implementation of the processor, the prediction may have to be pipelined into two phases, as discussed in [38].

This new PPM predictor implementation works as follows. Every time the BIU structure finds an indirect branch that must be predicted, it uses the contents of the associated 2-bit correlation counter to select one of the PHRs. The selected PHR provides the targets to the multiplexer and the selected targets provide indices to the Markov components of the main predictor body. The correlation selection counters are updated along with the Markov components. Every correlation selection counter follows the state machine shown in Figure 6.

Each state of the counter is labeled with the contents of the counter and the PHR selected for that state. The behavior of a branch can be characterized as *Strongly PB correlated* (state 00), *Weakly PB correlated* (state 01), *Weakly PIB correlated* (state 10) and *Strongly PIB correlated* (state 11). Dotted arcs represent a transition triggered by a misprediction, while solid lines denote a change in the state machine due to a correct prediction. The state machine in Figure 6 models a normal 2-bit counter, where the nature of correlation will change on two consecutive mispredictions. All counters are initialized to the *Strongly PIB-correlated* mode.

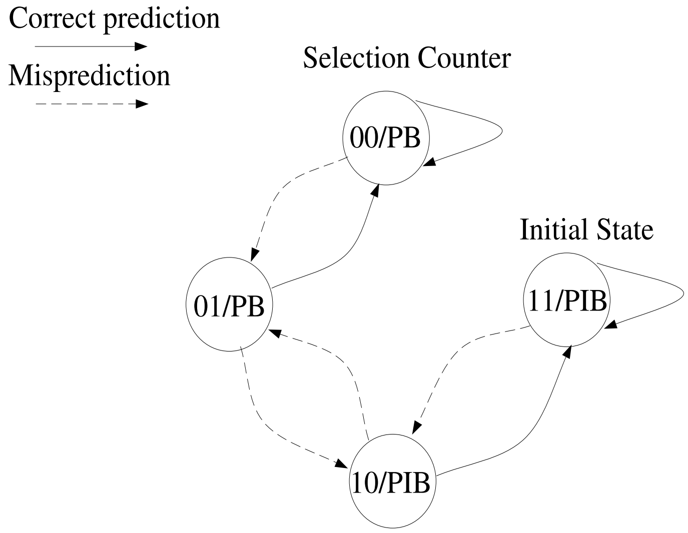

Figure 6: State machine of a 2-bit Up/Down Saturating Correlation Selection Counter.

Finally, we considered coupling a PPM predictor with a leaky filter [23]. This predictor works as follows: during prediction we use the PPM predictor if the branch is found in one of its Markov components. In case of a cold-start miss or a tag mismatch, we predict using the filter. When a filter misprediction occurs, we update both the filter and the PPM predictor (but not the correlation selection counters). When the filter correctly predicts the target, we do not update any of the Markov components or the correlation selection counters. In case of a PPM misprediction, we update the filter, the PPM predictor and the correlation selection counters. The leaky filter is implemented as a tagless BTB indexed by the lower order bits of the branch address. We did not include any 2-bit counters to control target replacement. The PPM predictor functionality remains the same.

5. Experimental Results

The results presented in this paper are obtained using trace and execution driven simulation. For all trace-driven experiments, we capture traces on Compaq Alpha-based workstations running Compaq Unix, with the ATOM tracing tool-set [39]. Execution-drive simulations are performed using the SimpleScalar v3.0 tool-set running the Alpha ISA [40]. The benchmark suite consists of C and C++ benchmarks exercising a large number of indirect branches. All benchmarks were compiled with the Compaq C/C++ compilers. and are described in Table 1. We use indirect branch prediction accuracy as our performance evaluation metric for trace driven simulations and total cycle count for execution driven simulations.

We focus on 2 indirect branches of the Alpha ISA (both of which are unconditional): *jsr, jmp*. The target address is found from the contents of their source register (no displacement is added). The ret instruction is not considered in our work since it is predicted accurately with a RAS [15]. We further partitioned the remaining jmp and jsr instructions into Single and Multiple Target (ST, MT). A MT indirect branch is one that has more than one possible target. Examples of MT branches found in our applications are: i) a jmp instruction implementing a switch statement, and ii) a jsr instruction implementing a pointer-based function call or a virtual function call On the other hand, an ST indirect branch has only one possible target. Examples of ST branches are DLL calls (using jsr instructions) and Global-Offset Table (GOT) calls (using either jsr or jmp instructions). Since all of our benchmarks are statically linked, they do not execute any DLL calls. A GOT-based call requires loading the function address from a GOT, based on the contents of the global pointer (GP). This mechanism is commonly used for resolving calls, where the distance between the caller and callee can not be encoded in the displacement field of the direct call instruction [41]. Our applications are initially compiled with a large number of GOT-based calls, but the vast majority of those can be replaced with direct procedure calls through link-time optimizations [42]. Those optimizations were enabled when compiling our benchmarks.

The compiler/linker can annotate indirect branches by setting one bit in their 16-bit displacement field to indicate whether they are ST or MT. The displacement field of Alpha indirect branches is not used during instruction execution and is usually filled (whenever possible) by the compiler with hints to allow faster I-cache access [43]. Run-time detection of MT indirect branches is achieved by recording this bit in each BIU entry and using it every time the same indirect branch is detected at the I-fetch stage.

The dynamic characteristics of the benchmarks taken from ATOM traces are shown in Table 1. They include the input file, the number of instructions executed in millions, the number of dynamic MT indirect branches, the number of static and activated MT indirect branches, the number of static MT indirect branches that contribute up to 90 and 95% of the total dynamic count, and the total number of targets visited by the activated MT indirect branches. Our benchmark suite includes perl and gcc from the SPEC95 benchmark suite. Also, we include edg (a C/C++ front end developed by EDG), gs (postscript interpreter), troff (text formatter from GNU groff v1.09), eqn (equation typesetter), eon (graphics renderer), ixx (IDL parser), porky (scalar optimizer of the SUIF v1.1.2 compiler) and lcom (HDL compiler). As we can see from Table 1, the relative percentage of MT indirect branches is small, but predicting indirect branches can have a significant impact on the performance of a wide-issue machine employing speculative execution (as shown in [19]). The number of MT branches responsible for 90 and 95% of all dynamic MT branches is given to clarify how many static branches generate most of the references. The number of visited targets is provided to define the indirect branch target working set of the applications.

Program	**Input**	**Instr.**	**dynamic MT ib**	**static MT ib**	**static MT ib**	**targets**
		Millions	Thousands	(total/active)	(90%/95%)	
perl(C)	scrabbl.in	28,795	171,340	619/51	3/3	221
perl(C)	jumble.in	3,023	16,548	619/52	3/4	217
perl(C)	primes.in	17,561	192,571	619/44	2/3	142
edg(C)	input.cc	269	1,072	1287/488	130/167	1342
edg(C)	pic.cc	395	1,709	1287/483	40/71	1334
gcc(C)	expr.i	1,309	5,651	647/231	41/59	1113
gcc(C)	c-decl-s.i	1,638	7,200	647/236	43/62	1164
gcc(C)	cp-decl.i	1,847	8,137	647/235	39/56	1128
gcc(C)	stmt-protoize.i	1,309	6,067	647/237	44/63	1178
gs(C)	tiger.ps	34	397	393/99	30/43	610
gs(C)	photon.ps	98	1,023	393/118	27/53	694
troff(C++)	gcc.1	136	929	839/203	23/36	397
troff(C++)	flex.1	135	925	839/189	21/35	344
troff(C++)	perlfunc.1	177	1,161	839/201	22/39	380
eqn(C++)	input.all	64	300	519/149	16/22	431
eon(C++)	chair.control	2,947	16,669	507/93	15/19	153
ixx(C++)	layouts.idl	48	196	1132/219	34/51	389
ixx(C++)	widget.idl	51	208	1132/220	33/51	390
porky(C++)	c-decl.c	921	7,154	3273/211	29/44	441
porky(C++)	combine.c	2,353	18,498	3273/214	27/42	470
porky(C++)	cse.c	1,574	12,153	3273/214	30/46	470
lcom(C++)	circuit3.l	206	1,730	908/374	8/17	837
lcom(C++)	circuitx.l	32	198	908/374	78/112	837

Table 1: Dynamic benchmark characteristics.

We simulated several indirect branch predictors and compare their misprediction ratios. The total number of entries for each predictor was set to 1K (1024) entries to provide some control over the degree of interference in the predictor tables and approximately equal hardware budget for every predictor. Notice that aliasing in a branch predictor depends on the number of entries available for storing targets and the actual number of targets the application will visit during its execution. Since the later number ranges from 142 to 1342 (column 7 of Table 1), we thought that having 1K available entries for storing targets would be adequate to test the ability of the predictors to handle interference. We simulated tagged predictors, since a Cascaded predictor requires tags and a tagged PPM predictor can activate predictions through all Markov components. Moreover, we do not provide results for BTB and the BTB2b, since their mispredictions ratios were found to be consistently higher than the rest of the simulated predictors. The list of the predictors follows:

- *GAp*: two 4-way set associative PHTs with 512 entries each, a 7-bit PHR recording the 2 lower-order bits from each target (path length = 4) using a gshare indexing scheme, and a 2-bit up/down saturating counter per PHT entry,
- *TC-PIB*: a 1K entry 4-way set associative Target Cache using a gshare indexing scheme, and an 8-bit PHR recording the 2 lower-order bits from previous targets (path length = 4) of indirect branches (PIB path history),
- *Dpath*: a Dual-Path hybrid predictor consisting of two GAp components. Each GAp predictor is formed by: a 512 entry 4-way set associative PHT, a 24-bit PHR, a 2-bit up/down saturating counter per PHT entry for updating entries and a 2-bit up/down selection counter per PHT entry for run-time component selection. A reverse interleaving indexing scheme is used to index the two components. One component predictor has a path length of 1 and the other has a path length of 3. All recorded bits are lower-order,
- *Cascade*: a Cascade hybrid predictor formed by a *Dpath* predictor with 1K entries and a leaky filter of 128 entries (tagless BTB). The *Dpath* component remains the same as before. Only the path lengths differ and are set to 6 and 4.
- *PPM*: a PPM predictor of order 9 with two 81-bit PHRs (9 targets, 9 low-order bits each) and 9 Markov components with a total of 1K entries simulated as direct mapped BTBs with a 2-bit up/down saturating counters per entry for target replacement on two consecutive misses. An SFSXS indexing scheme is used and a 2-bit up/down correlation selection counter is kept per branch,
- *PPM-filter*: a PPM predictor of order 9 with the exact same configuration as the *PPM* module. The only difference is that it is coupled with a leaky filter of 128 entries implemented as a tagless BTB.

The different pairs of path lengths between the *Dpath* and the *Cascade* predictors were taken from [21] and correspond to the optimal values for their benchmark suite. The misprediction ratios for the predictors generated by the ATOM experiments are shown in Figure 7, separated into two groups: i) C benchmarks, and ii) C++ benchmarks.

As we can see from Figure 7, the PPM predictor performs very well on average for all benchmarks. Overall, the non-filtered PPM predictor achieves an average misprediction ratio of 10.55%, the filtered PPM 9.58% and the Cascade predictor 12.07%. The PPM predictor eliminates mispredictions by exploiting variable-length path correlation, by dynamically selecting the type of path correlation and by better utilizing its available hardware budget. The later is due to the update-exclusive policy that distributes a target of an indirect branch to multiple components in the event of a misprediction.

The TC configuration achieves the lowest misprediction ratio for the perl experiment with jumble input (0.12%), since indirect branches show a strong PIB correlation. TC is the

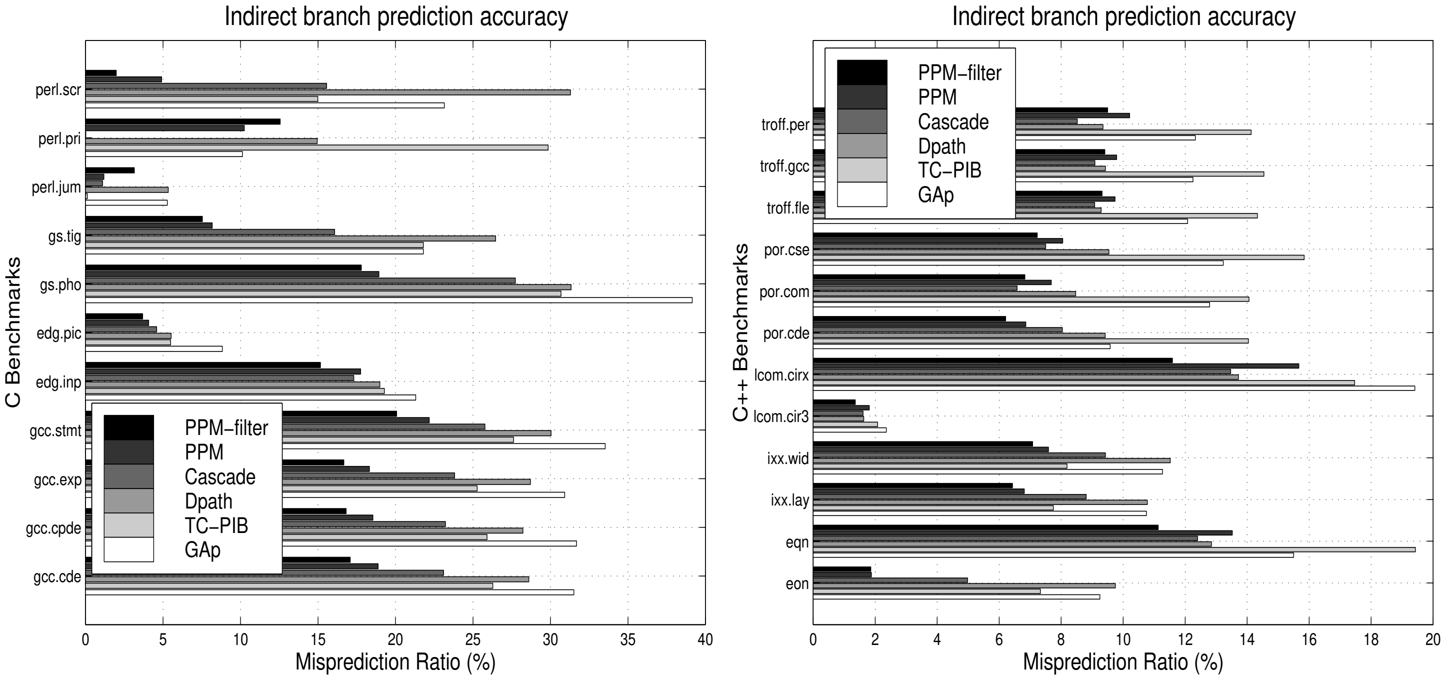

Figure 7: Misprediction ratios for indirect branch predictors utilizing 1K entries.

only path-based predictor among the ones we simulated that consistently utilizes PIB path history. The non-filtered version of the PPM algorithm was outperformed by the Cascade predictor in certain experiments such as eqn, troff, etc. Excessive aliasing, as in the eqn case, not only generates mispredictions but incorrectly updates the correlation selection counters and the 2-bit counters used to control the target update. Also, in case of a tag mismatch (which is considered as an aliasing misprediction), only one Markov component is updated. We do not apply the update-exclusive policy in that case. After applying filtering we can see that the difference in misprediction ratio diminished.

The filter had overall positive effects on the misprediction ratios. For all of the benchmarks (except two perl experiments), misprediction ratios were smaller after introducing a leaky filter. The filtering effects are more noticeable when the dynamic frequency of monomorphic indirect branches is high. Monomorphic branches occur in groups and so the probability of aliasing between them is high. The addition of a leaky filter will help to correctly predict them and prevent the PPM predictor from replacing other entries. In the case of perl, aliasing is not an issue since only 3 or 4 branches are responsible for the majority of dynamic branches (this will be shown in Table 2). However, misprediction ratios increased with the jumble and primes inputs when introducing filtering. For jumble, this occurred because of a shift towards predicting with the PB PHR (see Tables 2 and 3). More mispredictions were introduced since indirect branches were found to be highly PIB correlated for this experiment (the TC-PIB predictor achieved the lowest misprediction ratio for that experiment). For the primes input, the problem arose because filtering shifted accesses (and created mispredictions) to the 6th order Markov predictor (most of the accesses in the non-filtered version were done with Markov components of order 9, 8 and 7).

In order to further investigate the PPM predictor, we classified and measured the dynamic frequency of accesses and mispredictions based on either the selected Markov component or the selected PHR (PIB or PB in our experiments). Finally, we also measured the dynamic frequency of mispredictions due to aliasing (a branch accesses an entry that is currently occupied by another branch) to provide some insight on the effectiveness of the described hashing function. Table 2 presents the results for the original PPM predictor. The 2nd and 3rd columns list the percent of accesses and mispredictions, respectively, to the highest order (9th) Markov Component (M.C). The 4th column includes the percentage of mispredicts caused by aliasing. The 5th and 6th columns show the percentage of accesses and mispredictions, respectively, based on the indices generated by the PIB path history.

Program	**9th order M.C.**	**9th order M.C.**	**Aliasing**	**PIB PHR**	**PIB PHR**
	Accesses	Mispredictions	Mispredictions	Accesses	Mispredictions
perl.scr(C)	73.6	67.5	0.0	78.3	56.8
perl.jum(C)	78.1	22.0	0.0	18.3	79.0
perl.pri(C)	50.8	18.2	0.0	82.8	48.5
edg.inp(C)	85.7	85.8	26.9	48.6	34.1
edg.pic(C)	97.1	87.4	18.1	48.3	39.5
gcc.exp(C)	90.6	82.4	15.2	43.1	34.3
gcc.cde(C)	89.9	81.5	17.4	41.4	31.7
gcc.cpde(C)	90.1	81.7	16.1	44.3	34.2
gcc.stmt(C)	88.6	80.7	19.4	37.9	30.4
gs.tig(C)	88.4	81.2	7.5	57.7	42.7
gs.pho(C)	85.8	82.1	8.6	45.6	45.6
troff.gcc(C++)	95.6	96.3	6.7	59.2	46.2
troff.fle(C++)	94.7	96.4	5.8	57.8	46.7
troff.per(C++)	95.3	95.9	9.2	66.3	44.8
eqn(C++)	91.3	86.4	25.2	39.1	40.4
eon(C++)	90.4	82.8	0.0	45.8	54.5
ixx.lay(C++)	94.0	84.3	8.3	32.1	47.5
ixx.wid(C++)	94.2	89.7	8.3	33.0	46.7
porky.cde(C++)	97.0	95.8	6.1	60.1	49.3
porky.com(C++)	97.9	97.0	2.8	54.5	50.5
porky.cse(C++)	97.4	96.5	4.5	51.4	47.9
lcom.cir3(C++)	98.9	86.6	40.5	8.6	26.0
lcom.cirx(C++)	91.8	86.8	40.8	35.6	24.9

Table 2: Tagged PPM predictor statistics.

As we can observe from Table 2, the vast majority of decisions are made from the highest order Markov component. There is only one exception (perl with the primes input), where the highest order component serves half of the indirect branch occurrences. The highest order component is the primary vehicle for servicing requests because of the way the algorithm is constructed: it is updated on every dynamic indirect branch and always makes a decision whenever there is a match in the currently indexed entry. As expected, most of the mispredictions are derived from this single component (3rd column).

From column 4 of the same table, we can clearly distinguish the experiments where the hashing function causes a significant number of mispredictions due to aliasing (e.g., edg, eqn, gcc and lcom). Since a PPM-based predictor suffers from a higher misprediction ratio with respect to the Cascaded predictor for some of those experiments (e.g., edg, eqn and lcom), it may be beneficial to further improve upon the design of the hashing function.

If we examine the nature of path correlation type (revealed from the percentage of dynamic indirect branches serviced by PIB path history), we can identify a dependence upon the programming style and the type of the application. There exist cases where the vast majority of branches require PB path history (lcom.cir3) or PIB path history (perl.pri). Overall, there is a tendency towards PB path history, since conditional branches provide more information about the previous path leading to the same indirect branch.

We also generated similar results for the PPM predictor when coupled with a leaky filter. Table 3 includes these statistics. Columns 2, 3, 6 and 7 describe the exact same information as columns 2, 3, 5 and 6 in Table 2. Columns 4 and 5 show the accesses and mispredictions from the leaky filter. Notice that in case of a PPM module miss, the filter makes a prediction. Therefore, there is no aliasing misprediction column.

As we can see from Tables 2 and 3, filtering is reducing the number of accesses to the highest order Markov component. This is expected since most of the prediction traffic is directed to the highest order Markov component (Table 2). The number of predictions made by the filter ranges from 12 to 41%, with the only exception of perl where less than 7% of the predictions are directed to the filter. Based on the leaky filter protocol, a filter is used whenever the main component can not make a prediction (table miss). Since the PPM predictor effectively utilizes its entries by spreading the target of an indirect branch into many components (update exclusive policy), the probability that a branch will miss in all Markov components is limited. Filter accesses also depend on the dynamic frequency of monomorphic indirect branches and their distribution in the dynamic instruction stream. By looking at the last two columns we can see that filtering has no consistent effect on the selection mechanism for path correlation type.

In order to evaluate the cycle-based performance of the PPM predictor, we simulate a variety of indirect branch predictors in an out-of-order, dynamically scheduled machine described in Table 4.

Each IB predictor uses a total of 2K entries and all predictors are tagged. The list of predictors follows:

- *BTB*: a 2K entry BTB,
- *BTB2b*: a 2K entry BTB, with 2-bit up/down saturating counter per entry,
- *GAp*: 2 1K entry PHTs, a 10-bit PB path history register recording the 2 lower-order bits from each target (path length = 5) using a gshare indexing scheme, and a 2-bit up/down saturating counter per PHT entry,

Program	9th order M.C.	9th order M.C.	Filter	Filter	PIB PHR	PIB PHR
	Accesses	Mispredictions	Accesses	Mispredictions	Accesses	Mispredictions
perl.scr(C)	66.2	65.1	1.4	2.3	98.2	97.6
perl.jum(C)	60.8	1.7	6.6	0.0	16.2	0.0
perl.pri(C)	48.0	60.4	3.4	0.0	58.0	0.0
edg.inp(C)	50.9	63.9	41.0	20.0	43.7	44.5
edg.pic(C)	56.8	71.5	41.2	12.8	36.2	47.0
gcc.exp(C)	78.1	66.9	12.4	12.3	46.7	41.7
gcc.cde(C)	74.9	63.1	13.9	14.3	46.5	39.9
gcc.cpde(C)	77.0	65.4	13.0	12.9	48.1	42.1
gcc.stmt(C)	73.0	60.2	15.2	16.2	42.2	39.0
gs.tig(C)	69.2	65.0	17.9	5.0	56.1	50.2
gs.pho(C)	69.2	76.1	17.0	5.2	47.2	50.4
troff.gcc(C++)	64.8	92.0	32.2	3.8	44.4	48.3
troff.fle(C++)	66.1	92.2	30.0	2.6	47.9	49.8
troff.per(C++)	65.2	91.8	32.0	3.6	39.6	49.1
eqn(C++)	59.0	67.5	32.4	15.6	53.5	52.0
eon(C++)	51.2	75.3	37.5	0.0	53.5	55.5
ixx.lay(C++)	59.7	59.6	29.0	5.9	43.2	52.3
ixx.wid(C++)	54.5	60.1	27.9	5.6	42.8	51.0
porky.cde(C++)	71.9	92.5	23.6	2.2	48.7	53.8
porky.com(C++)	74.4	95.7	23.0	1.0	50.8	52.1
porky.cse(C++)	69.0	94.1	28.1	1.8	45.2	49.2
lcom.cir3(C++)	36.5	50.4	40.5	29.6	32.2	42.0
lcom.cirx(C++)	46.9	49.6	40.8	29.9	43.4	41.9

Table 3: Tagged PPM predictor with filtering statistics.

- *Dpath*: a Dual-Path hybrid predictor consisting of two GAp predictors and an 512-entry table of 2-bit up/down selection counters. Each GAp predictor is formed by: a 512 entry 2-way (LRU) PHT, a 24-bit PHR, a 2-bit up/down saturating counter per PHT entry for replacement, and a reverse interleaving indexing scheme. One component predictor has a path length of 2 and the other has a path length of 4. All recorded bits are lower-order,
- *Cascade*: a Cascade hybrid predictor formed by an identical *Dpath* predictor and a Leaky Filter (LF) of 128 entries,
- *PPM-hyb*: a PPM predictor of order 10, 2 100-bit PHRs (PIB and PB path history type, 10 targets, 10 low-order bits each), 10 Markov predictors with total 2K entries, an SFSXS indexing scheme, and a 1K selection counter table with 2-bit saturating counters per entry.

The predictors are assumed to assist the instruction fetch unit of a dynamically scheduled, out-of-order CPU by predicting the target of an indirect branch (excluding return

Unit	Description
L1 I-cache	16Kb, 32 byte line size, 2-way (LRU), write back, 1 cycle hit, 32 byte wide bus to L2
L1 D-cache	16Kb, 32 byte line size, 2-way (LRU), write through, 1 cycle hit, 32 byte wide bus to L2
Write buffer	12 entries, 32 byte entry size, retire-at-8, read-from-WB load hazard policy
L2 unified cache	256Kb, 64 byte line size, 2-way (LRU), write back, fetch-on-write on write miss, 7 cycles hit
Main Memory	85 cycles, 8 byte wide bus, 8KB page size, 2 memory ports
Instruction, Data TLB	32 entries, 4-way (LRU), 30 cycle miss latency
I-fetch unit	up to 8 insts/cycle from the same cache line, 8 Ins-queue, hybrid predictor(8Kb bimodal + 8Kb gshare, 12-bit HR + 8Kb 2-bit selector), 16 entry RAS, 4KB 2-way (LRU) tagged BTB, 2 cycles misfetch penalty, 6 cycles extra misprediction penalty, speculative update on ID stage
Register File	32 Integer & 32 FP registers
Decode unit	4 insts/cycle
Issue unit	4 wide out-of-order integer operation issue, 2 wide out-of-order FP operation issue, 64 instruction window size, 32 entry load/store queue
Execute unit	4 Integer ALUs, 1 Mul/Div Integer unit, 2 FP ALUs, 1 Mul/Div FP unit
Commit unit	4 inst/cycle

Table 4: Machine description.

instructions). All predictors are speculatively updated at the instruction decode stage of the pipeline. In order to examine the merit of indirect branch prediction we also simulate a base machine with no explicit indirect branch predictor. This configuration is labeled as *Dft* in all tables and plots. The base machine predicts the targets of all branches, including the indirect ones, using only one unified BTB structure. We assume that this BTB does not contain any prediction bits and is decoupled from the conditional branch outcome predictor component. The remaining machine configurations use the same BTB for conditional and direct unconditional branches but a different IB predictor for indirect branches except returns. Return instructions are predicted with a fully check-pointed RAS in all configurations [17].

In order to have a fair comparison in terms of hardware cost, the base machine has a number of BTB entries equal to the sum of the IB predictor table entries and the BTB entries for the machines with IB predictors. The size of the table where targets are kept usually dominates the hardware cost of the predictor. It is also the critical parameter that is most directly related to the problem we are trying to solve (correctly predicting the next target of an indirect branch). The base machine and the one deploying a BTB1b indirect branch predictor are very similar. The only difference in their fetch model is in the way they partition predictions and targets of indirect branches. While a single BTB is used to store the targets of all branches in the base machine, the BTB1b-based machine reduces aliasing by distributing predictions into two, basically identical, BTBs of half the size each.

Fig. 8 shows a performance evaluation for the indirect branch predictors using the relative cycle count percentage change between the base processor and a processor that employs an IB predictor. A bar with a positive percentage indicates a performance improvement.

Based on the results of Fig. 8, indirect branch prediction improves overall performance under certain conditions. The primary factor that determines the net effect is the relationship between the conditional branch working set and the number of available BTB entries to store them. As the number of BTB entries is halved to accommodate indirect branch prediction hardware resources, aliasing in the unified BTB increases. On the other hand, since traffic assigned to the same BTB drops (indirect branches are predicted via a separate IB predictor), so less aliasing should occur. Obviously, these two phenomena have opposing side effects on performance. The amount of aliasing for a specific input depends on two factors: the mapping employed by the predictor and the branches executed at run-time. Mapping can be static if the PC of the branch is solely used for accessing the BTB. Often, past history is used in conjunction with the branch PC to generate an index. In this case, the mapping is dynamic since the contents of the history register may be different every time the branch has to be predicted. For a given executable, the input scenario determines the branches to be executed. A conflict miss in the BTB accompanied with a correct prediction direction generates a misfetch, which in turn forces the fetch unit to either stall or bring the next sequential cache line/basic block, if there is enough cache bandwidth. Our current model adopts the first strategy. If the penalty paid by the extra misfetches offsets the savings in cycles from correctly predicting indirect branches and reducing aliasing in the main BTB, then we observe performance degradation (see edg experiments in Fig. 8). In all other cases, indirect branch prediction is beneficial, reducing the total number of cycles by 0.1% to 10%.

The PPM predictor outperforms all other configurations mainly via its superior misprediction ratios which are plotted in Fig. 9. As we can see, the PPM predictor consistently achieves the lowest misprediction ratios, outperforming all other predictors. In general, the highest misprediction ratios were achieved with the BTB configurations, leading to overall performance degradation in some cases (gcc, troff, eqn). Based on Fig. 8, a lower misprediction ratio has a positive impact on overall performance not only by reducing cycle penalties due to misfetches, but also by offsetting the negative aliasing effects in the main BTB. For example, in perl, address predictions via the main BTB are heavily influenced by the indirect branches executed for the selected simulation interval. The number of wrong

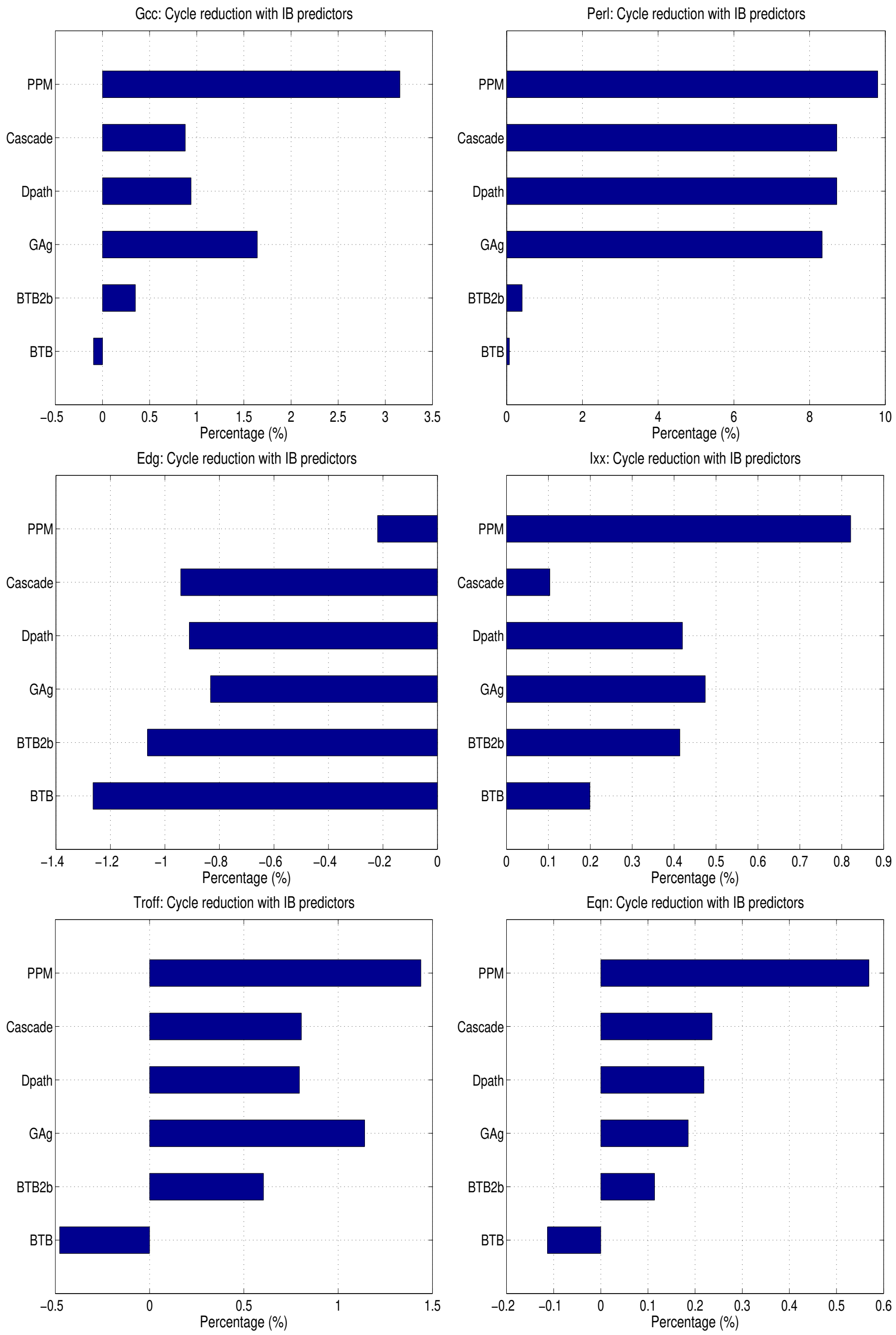

Figure 8: Cycle count reduction when using indirect branch predictors in a superscalar processor.

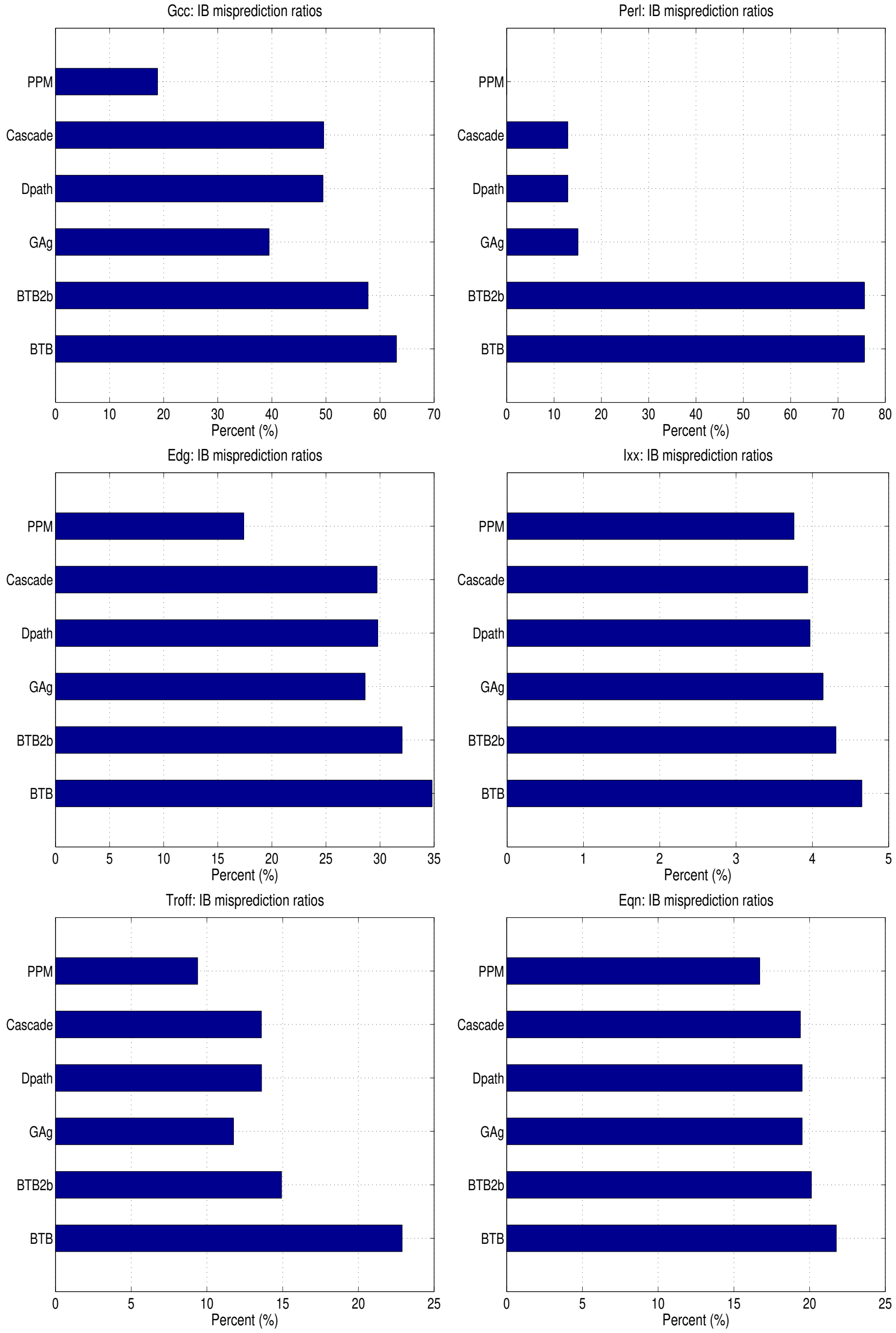

Figure 9: Misprediction ratios of indirect branch predictors in a superscalar processor.

address predictions in the unified BTB increases only by 1.1% (468,572 extra misfetches) when transitioning from the base machine to IB prediction with the PPM algorithm. In edg, the same statistic becomes 9.5% (4,527,411 extra misfetches). The extra overhead is partially offset by the IB predictor's superior performance, but overall performance degrades (0.6% for the PPM predictor configuration).

6. Conclusions

The accuracy of branch predictors is critical in today's superscalar processors. Recently, correctly predicting indirect branches has received greater attention due to the increased usage of those branches in popular programming environments.

In this paper we examine the performance of a new algorithm for indirect branch prediction. The algorithm is a approximation of a data compression algorithm (PPM) that has been proposed for conditional branch prediction as well. We discuss several design issues and present a viable implementation of the PPM algorithm, coupled with other techniques such as dynamic selection of path correlation type and filtering.

We demonstrated the performance potential of a PPM-based predictor using instruction traces void of speculative activity, as well as within the context of an out-of-order, dynamically scheduled machine. We compared the PPM predictor with a variety of other indirect branch predictors using their mispredictions ratios and analyzed its behavior via misprediction classification. We also provided cycle counts and misprediction ratios under a realistic machine model, where speculative update, wrong path instruction pollution and timing are accurately simulated. These experiments allowed not only to compare performance between IB predictors, but also to evaluate the cost-effectiveness of IB prediction against conditional branch prediction.

This study is by no means complete. There exist several features of the PPM predictor that warrant further study. First, new target selection and update policies should be explored (i.e., we do not have to start searching for a prediction from the highest order component). Confidence counters could be used to allow other components to be accessed first (and updated). More customized filtering for indirect branch prediction should be considered. Additional simulation should look at the sensitivity of the predictors to the path length and the size of the predictor table. Further study should look at the benefits from combining a PPM-based predictor with a precomputation engine for certain classes of indirect branches. Finally, PPM-like techniques should be applied to value prediction, especially on cases where last value or stride-based prediction does not perform well.

References

[1] A. Uht, V. Sindagi, and S. Somanathan, "Branch Effects Reduction Techniques," *IEEE Computer Magazine*, vol. 30, pp. 71–81, May 1997.

[2] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," in *Proceedings of the International Symposium on Computer Architecture*, pp. 257–266, May 1993.

[3] K. Driesen and U. Holzle, "Limits of Indirect Branch Prediction," Tech. Rep. TRCS97-10, University of California, Santa Barbara, June 1997.

[4] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences between C and C++ programs," *Journal of Programming Languages*, vol. 2, pp. 313–351, January 1994.

[5] B. Stroustrup, *The C++ Programming Language*. Addison Wesley, 2nd edition ed., 1991.

[6] H. Srinivasan and P. Sweeney, "Evaluating Virtual Dispatch Mechanisms for C++," Tech. Rep. RC 20330, IBM T.J. Watson Center, January 1996.

[7] D. Lee, P. Crowley, and J. B. et.al., "Execution Characteristics of Desktop Applications on Windows NT," in *Proceedings of the International Symposium on Computer Architecture*, pp. 27–38, June 1998.

[8] S. Mahlke and R. H. et.al., "Characterizing the Impact of Predicated Execution on Branch Prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 217–227, November 1994.

[9] R. Bodik, R. Gupta, and M. Soffa, "Interprocedural Conditional Branch Elimination," in *Proceedings of the International Conference on programming language and implementation*, pp. 146–158, June 1997.

[10] G. Uh, *Effectively Exploiting Indirect Jumps*. PhD thesis, Florida State University, 1997.

[11] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer Magazine*, vol. 17, pp. 6–22, January 1984.

[12] J. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the International Conference on Computer Architecture*, pp. 135–148, May 1981.

[13] C. Perleberg and A. Smith, "Branch Target Buffer Design and Optimization," *IEEE Transactions on Computers*, vol. 42, pp. 396–412, April 1993.

[14] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," in *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, pp. 397–408, January 1994.

[15] D. Kaeli and P. Emma, "Branch History Table Prediction of moving Target Branches due to Subroutine Returns," in *Proceedings of the International Symposium on Computer Architecture*, pp. 34–42, May 1991.

[16] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark, "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms," in *Proceedings of the International Symposium on Microarchitecture*, pp. 259–271, December 1998.

[17] S. Jourdan, T. Hsing, J. Stark, and Y. Patt, "The Effects of Mispredicted Path Execution on Branch Prediction Structures," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 58–67, October 1996.

[18] D. Kaeli and P. Emma, "Improving the Accuracy of History-based Branch Prediction," *IEEE Transactions on Computers*, vol. 46, pp. 469–472, April 1997.

[19] P. Chang, E. Hao, and Y. Patt, "Target Prediction for Indirect Jumps," in *Proceedings of the International Symposium on Computer Architecture*, pp. 274–283, June 1997.

[20] T. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[21] K. Driesen and U. Holzle, "Accurate Indirect Branch Prediction," in *Proceedings of the International Symposium on Computer Architecture*, pp. 167–178, June 1998.

[22] S. McFarling, "Combining Branch Predictors," Tech. Rep. TN-36, DEC WRL, June 1993.

[23] K. Driesen and U. Holzle, "The Cascaded Predictor: Economic and Adaptive Branch Target Prediction," in *Proceedings of the 31st International Symposium on Microarchitecture*, November 1998.

[24] J. Stark, M. Evers, and Y. Patt, "Variable Length Path Branch Prediction," in *Proceedings of the International Conference on Programming Languages and Operating Systems*, pp. 170–179, October 1998.

[25] A. Roth, A. Moshovos, and G. Sohi, "Improving Virtual Function Call Target Prediction via Dependence-based Pre-Computation," in *Proceedings of the International Conference on Supercomputing*, June 1999.

[26] A. Moffat, "Implementing the PPM Data Compression Scheme," *IEEE Transactions on Communications*, vol. 38, pp. 1917–1921, November 1990.

[27] J. Cleary and I. Witten, "Data Compression using Adaptive Coding and Partial String Machines," *IEEE Transactions on Communications*, vol. 32, pp. 396–402, April 1984.

[28] I. Chen, J. Coffey, and T. Mudge, "Analysis of Branch Prediction via Data Compression," in *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pp. 128–137, October 1996.

[29] F. Willems, Y. Shtarkov, and T. Tjalkens, "The Context Tree Weighting Method: Basic Properties," *IEEE Transactions on Information Theory*, vol. 41, May 1995.

[30] E. Federovksy, M. Feder, and S. Weiss, "Branch Prediction based on Universal Data Compression Algorithms," in *Proceedings of the International Symposium on Architecture*, pp. 62–72, June 1998.

[31] T. Bell, J. Cleary, and I. Witten, *Text Compression.* Prentice Hall, 1990.

[32] M. Feder, N. Merhav, and M. Gutman, "Universal Prediction of Individual Sequences," *IEEE Transactions on Information Theory*, pp. 1258–1270, July 1992.

[33] T. Kroeger and D. Long, "Predicting File System Actions from Prior Events," in *Proceedings of the USENIX Winter Technical Conference*, January 1996.

[34] V. Phalke and B. Gopinath, "Compression-based Program Characterization for Improving Cache Memory Performance," *IEEE Transactions on Computers*, vol. 46, pp. 1187–1201, November 1997.

[35] Y. Sazeides and J. Smith, "Implementations of Context-based Value Predictors," Tech. Rep. TR-ECE-97-8, University of Winsconsin, Madison, December 1997.

[36] J. Kalamatianos and D. Kaeli, "Predicting Indirect Branches via Data Compression," in *Proceedings of the International Symposium on Microarchitecture*, pp. 272–281, December 1998.

[37] Y. Kalamatianos, *Microarchitectural and Compile-Time Optimizations for Performance Improvement of Procedural and Object-Oriented Languages.* PhD thesis, Northeastern University, in preparation 1999.

[38] T. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," in *Proceedings of the International Symposium on Computer Architecture*, pp. 124–134, May 1992.

[39] A. Srivastava and A. Eustace, "ATOM : A System for Building Customized Program Analysis tools," in *Proceedings of the International Conference on Programming Language, Design and Implementation*, pp. 196–205, June 1994.

[40] D. Burger and T. Austin, "The SimpleScalar Tool Set, v2.0," Tech. Rep. UW-CS-97-1342, University of Winsconsin, Madison, June 1997.

[41] Digital Equipment Corporation, Maynard, Massachusetts, *DEC OSF/1 Programmer's Guide*, 1993.

[42] A. Srivastava and D. Wall, "Link-Time Optimization of Address Calculation on a 64-bit Architecture," Tech. Rep. RR 94/1, DEC WRL, February 1994.

[43] R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual.* Digital Press, 1995.