Predicting Indirect Branches via Data Compression

John Kalamatianos & David R. Kaeli
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA
E-mail : kalamat,kaeli@ece.neu.edu

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 the 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 describe a new algorithm for predicting unconditional indirect branches called Prediction by Partial Matching *(PPM). We base our approach on techniques proven to work optimally in the field of data compression. We combine a viable implementation of the PPM algorithm with dynamic per-branch selection of path-based correlation and compare its prediction accuracy against a variety of predictors. Our results show that, for approximately the same hardware budget, the combined predictor can achieve a misprediction ratio of 9.47%, as compared to 11.48% for the previously published most accurate indirect branch predictor.*

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 [24]. A branch predictor working at instruction fetch time can minimize 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. Therefore the accuracy of a branch predictor is essential to achieve high performance when using speculative execution [27].

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 referenced simply as indirect branches throughout the rest of the paper. Although their direction is known (always taken), they can transfer control flow to more than one target. They typically use the contents of a register, sometimes combined with a fixed offset, to compute the target address at run-time. This class of branches is more difficult to predict accurately 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 microarchitec-

tures 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 we use an very large BTB [7].

Indirect branch frequencies are not currently as high as conditional branches, but their associated misprediction overhead can be substantial, especially for superscalar architectures. 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) [3]. The programming paradigm found in most OO languages such as C++, Java, Smalltalk and Self supports polymorphism based on run-time procedure binding [22]. Most implementations generate an indirect function call for every polymorphic call [19]. Indirect branches commonly appear in applications that use dynamically-linked libraries since they implement DLL calls [14].

Moreover, the relative performance impact of indirect branches will be further increased as techniques such as *predicated execution* [16], *branch elimination* [1] and *coalescing of conditional branches into indirect jumps* [23] become increasingly popular. By reducing the number of conditional branches (sometimes at the expense of extra indirect jumps [23]), 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. This paper is organized as follows. In Section 2 we review related work in the field of indirect branch prediction. Section 3 introduces the PPM algorithm as it is applied to the field of conditional branch prediction, simply to illustrate the concept. Section 4 presents one possible implementation of a PPM-based predictor that dynamically exploits different types of path-based correlation to predict indirect branches and Section 5 presents simulation results. Finally Section 6 suggests directions for future work and concludes the paper.

2. Related Work

In [15], Lee and Smith describe several BTB variations. The simplest BTB keeps the most recent target for each branch. In the case of a BTB target mispredict, the predicted target address is replaced. An improvement upon that basic BTB configuration was proposed by Calder and Grunwald [2], where a two-bit counter is used to limit the update of the target address only after two consecutive mispredictions have occurred (we will refer to this strategy as *BTB2b*). The BTB2b can produce a better branch prediction ratio for C++ applications by taking advantage of the locality exhibited by targets of virtual function calls (C++ polymorphic calls).

In [10] and [11] Kaeli and Emma described two mechanisms which accurately predict the targets of two special classes of indirect branches: (i) subroutine returns, and (ii) indirect jumps generated by switch statements. A *Call/Return Stack* (RAS) was described which uses the inherent correlation between procedure calls and returns to pair up the correct target address with the current return invocation.

The *Case Block Table* (CBT) utilizes the switch variable values to find the next target of a switch statement. The CBT is able to resolve the switch statement targets since the switch variable value is an optimal predictor for this 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 [4].

In [7], Driesen and Holzle examined a modified structure of the two-level adaptive branch predictor (originally proposed in [25]), to predict the targets of indirect branches. Their design allows path-based correlation to be exploited by recording partial previous targets in the history register (instead of branch directions). Driesen and Holzle performed an exhaustive search on a large number of two-level predictor configurations. They obtained the best prediction accuracy using the GAp structure (Global history register, Per-address Pattern History Table (PHT) configuration). However, the application of their results is limited since they recorded full target addresses in the history register and assumed infinite PHTs. In more recent work [8], Driesen and Holzle explore realistic designs, varying how and when targets are recorded in the history register, the size and associativity of the PHTs, and the size of the history register. They also proposed using dual-path hybrid predictors with confidence counters to improve prediction accuracy under a fixed hardware budget. Each component was a two-level predictor with a different path length [1]. Their findings suggest that the best dual path predictor had components with a short and a long path length.

Chang et al. proposed using a family of two-level predictors [4] with a structure similar to the two-level adaptive scheme described in [7]. One important feature of the *Target Cache* (TC) predictor was that the history register recorded partial targets from a selected group of branches (the group may include all branches, indirect branches, conditional branches or calls/returns). Therefore, they allowed different types of path-based correlation to be exploited. Their simulation results clearly showed the de-

[1] Path length determines the number of branch targets recorded in the history register.

pendence of indirect branch predictability on the type of path-based correlation.

Recently, Driesen and Holzle [9] performed another study where they examined the predictability of indirect branches using filtering. Their filtering scheme improved the prediction ratio by isolating *monomorphic* [2] and *low entropy* branches [3] from a main body predictor and reducing the collision factor in the later. Their Cascade predictor included a GAp/Dual-path hybrid predictor as their major component and a strict/leaky filter, implemented as a BTB-like structure.

In this work we apply the PPM algorithm to the task of indirect branch prediction. We show how this algorithm allows variable length path correlation over a certain range to be exploited. We propose coupling a PPM-based implementation with dynamic per-branch selection of the type of path-based correlation. We discuss implementation problems, describe one possible complete solution and present simulation results. We also implement all other indirect branch predictors proposed to date and compare their misprediction ratios assuming the same hardware budget restriction.

3. PPM Prediction Algorithm

Next we discuss the *Prediction by Partial Matching* (PPM) algorithm as it was applied in the field of conditional branch prediction [6, 17]. In section 4 we will describe an implementation of the algorithm tailored to indirect branch target prediction.

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. In order to accomplish this task, a data compression algorithm usually consists of a predictor and a coder. The predictor must predict the next symbol with high accuracy. The approach taken is to build a probabilistic model to predict the next symbol. This is done by selecting the symbol with the highest frequency count. The probabilistic model is then used by the coder to produce the encoded compressed data sequence. Since current coders perform near optimal code compression, the performance of a data compression technique relies heavily on the predictor accuracy.

The PPM algorithm is a universal compression/prediction algorithm that has been proven to be optimal and has been applied to areas such as data compression [6], file prefetching [13] and conditional branch prediction [5]. The PPM predictor for text compression computes probabilities for symbols. 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 its relative frequency.

The foundation for the PPM algorithm of order m is a set of $m+1$ Markov predictors [6]. 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 and 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 transition probabilities (determined by the frequency counts) at the same time it predicts the branch outcomes. The prediction is made by selecting as the next state the one with the highest transitional probability (highest frequency count).

Figure 1 illustrates the concept by presenting a 3rd order Markov predictor that has processed the input sequence 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.

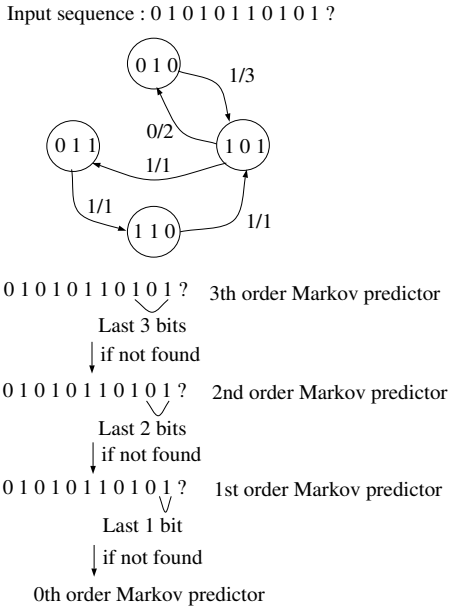

Figure 1. 3rd order Markov Predictor and its Prediction Algorithm.

Edges between states are established when one state

[2] A branch is monomorphic when it mostly accesses one target.

[3] A branch has low entropy when its target changes infrequently.

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 1 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 at the top of Figure 1, 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 is a set of $m+1$ Markov predictors. At step 1, the mth order Markov predictor is searching 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 in the input sequence. Returning to Figure 1, 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* [5]. 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 [5], 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 1, 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 in each 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 a History Register (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 path correlation. The 1st order Markov predictor is simply a 1-bit history predictor (predicts the next branch outcome using its previous outcome). The protocol of the PPM algorithm 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.

4. A PPM Predictor 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. An HR which captures complete path history would require an enormous PHT (even if it records only a few targets) in order to represent all states of the Markov model.

The solution is to use a hashing function applied to targets stored in the HR in order to form an index accessing a small PHT. In a sense, we are merging sets of states of the Markov model together. The final number of merged states is equal to the number of PHT entries. The hashing function proposed in [4, 8] 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 [18]. Figure 2 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 an HR that records 10 targets and a 10th order PPM predictor. The SFSXS mapping function selects the low-order 10 bits from target i, folds the selected value into 5 bits which are left shifted by i bits and XOR'ed to-

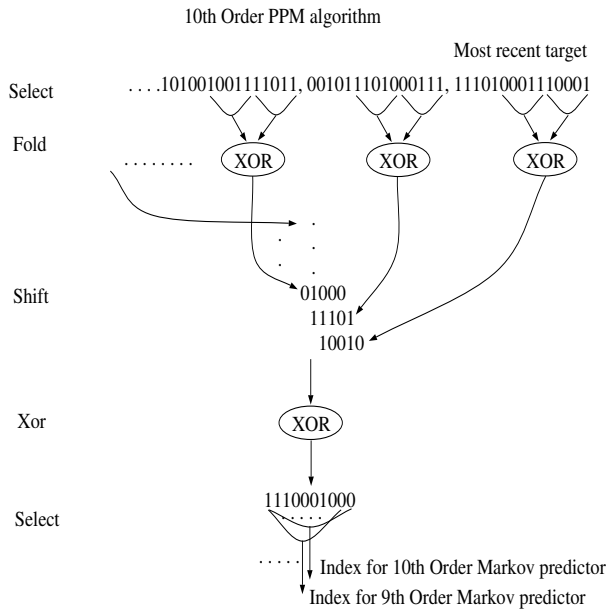

Figure 2. Select-Fold-Shift-XOR-Select Indexing Function.

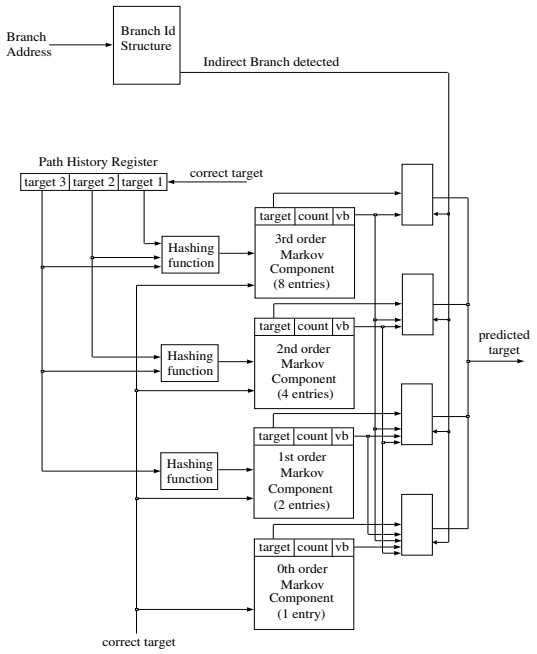

Figure 3. Implementation of a 3rd order PPM predictor.

gether with the other values. The j high order bits 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 2.

Another major problem 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 requires storing multiple targets per PHT entry along with their frequency counts, and uses a majority voting mechanism to select the next target. Instead we store the most recently visited target and use it to make a prediction (similar to [4, 8]).

The complete design of the PPM algorithm is shown in Figure 3 for a 3rd order predictor. 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 have a *Path History Register* (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. The boxes labeled as Hashing Functions implement the indexing scheme described in Figure 2. The Markov predictors are basically BTB-like structures, where every entry includes the most recently accessed target, a 2-bit up/down saturating counter and a valid bit. Since every entry in a Markov predictor ideally represents one of its states, the valid bit indicates a non-zero frequency count for that state. Also, the counter is used to control the update step of the target; the target is updated on two consecutive misses. Finally, each one of the Markov predictors drives a buffer that takes as input the selected target (if there is one).

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 is detected in the instruction stream, the selected targets from each Markov predictor are sent to the buffers and the highest order Markov predictor having a valid entry (valid bit = 1) provides the next target. Therefore the valid bits of the higher order predictors should control the buffers of the lower order predictors. The prediction process requires one level of table access, and thus, the predictor can be characterized as 1-level.

The update step starts by shifting the actual target to the

PHR, independent of whether we had a misprediction or not. By maintaining the old state of valid bits separately from the selected entries of each of the Markov predictors, we can also update the contents of the predictors using an *update exclusion* policy [5]. For each predictor that needs to be updated, we set the valid bit (if not set), update the target based on the contents of the counter, and modify the state of the counter.

Previous studies have shown that indirect branches have different characteristics, especially when it comes to inter-branch correlation [4]. Experimental results revealed that most indirect branches were best correlated with either all previous branches or with previous indirect branches (labeled as Per Branch, PB-correlated, and Per Indirect Branch, PIB-correlated, respectively) [12]. 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 wanted to characterize the correlation of every individual indirect branch, we introduced 2-bit up/down saturating counters in the BIU module. Based on the contents of the counter, the predictor selects between the two PHRs. The new design is shown in Figure 4.

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 [26].

Our PPM predictor implementation works as follows. Every time the BIU structure finds a multiple target indirect branch, it uses the contents of the associated 2-bit counter to: 1) select one of the PHRs, 2) access the Markov predictors, and 3) select one of the targets. The update protocol is exactly the same as far as the PPM predictor is concerned. The PHRs and the correlation selection counters are always updated. Every correlation selection counter follows the state machine shown in Figure 5.

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.

Although the upper state machine in Figure 5 models a normal 2-bit counter (where the nature of correlation will change on two consecutive mispredictions), the lower state machine is slightly different. It is labeled as *PIB-biased* since it favors the PIB PHR in the following way. On a single misprediction, a branch's state changes from *Strongly PB correlated* to *Weakly PIB correlated* or from *Weakly PB correlated* to *Strongly PIB correlated*. The reason for implementing such a selection scheme is that several benchmarks had branches that were strongly PIB correlated, but they bounced between weak PIB and weak PB correlation due to aliasing between different branch instances in the Markov predictors. This caused a number of mispredictions that were eliminated with the introduction of this biased state machine. All counters are initialized to the *Strongly PIB correlated* mode for both state machines.

5. Experimental Results

The results presented in this paper were obtained using trace-driven simulation. Traces were captured on DEC Alpha-based workstations running Digital Unix, using the ATOM tracing tool-set [20]. The benchmark suite consists of C and C++ benchmarks exercising a large number of indirect branches. All benchmarks were compiled with the DEC C/C++ compilers. The benchmarks are described in Table 1. We present indirect branch prediction accuracy in this study.

The Alpha AXP ISA provides 4 indirect branches (all of which are unconditional): *jsr, jmp, ret* and *jsr_coroutine*. Each one of them computes the target address using the contents of their source register (no displacement is added). The last instruction class was not found in our traces. The ret instruction is not considered in our work since it is predicted accurately with a RAS [10]. 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. A jmp instruction implementing a switch statement and a jsr instruction implementing a pointer-based function call are two examples of MT branches found in our applications. 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. However, they do execute a lot of GOT-based calls, but those can be replaced with direct procedure calls through link-time optimizations [21] [4]. Hence, they are not considered in this study. 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. Since the displacement field of indirect branches is not used during instruction execution, we believe that this modification will not modify the ISA, reduce the address range of branches or generate incompatible binaries. Run-time detection of MT indirect

[4] Notice that DLL calls can not be statically resolved and dynamically linked applications will execute indirect branches calling DLLs that the H/W will have to predict.

branches is achieved by recording this bit in each BIU entry.

The dynamic characteristics of the benchmarks are shown in Table 1. They include the input file, the total number of instructions executed in millions and the total number of MT jsr and jmp indirect branches executed. Perl and gcc are taken from the SPEC95 benchmark suite, edg is a C/C++ front end developed by EDG, gs is a postscript interpreter, troff is part of GNU groff v1.09, eqn is an equation typesetter, eon is a graphics renderer, photon is a diagram generator and ixx is an IDL parser. As we can see from Table 1 the percentage 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 [4]).

Program	Input	Instr.	MT jsr	MT jmp
perl(C)	jumble.in	3,023	224	12,973,836
edg(C)	input.cc	268	493,196	567,180
edg(C)	pic.cc	394	607,839	1,042,418
gcc(C)	expr.i	1,202	267,794	4,951,751
gcc(C)	c-decl-s.i	1,422	332,970	5,880,604
gs(C)	tiger.ps	34	88,893	149,198
gs(C)	photon.ps	98	260,325	590,831
troff(C++)	gcc.1	136	864,997	64,114
troff(C++)	flex.1	135	862,525	62,368
troff(C++)	perlfunc.1	177	1,065,095	96,292
eqn(C++)	input.all	64	210,858	89,793
eon(C++)	chair.control	2,947	16,668,673	30
ixx(C++)	layouts.idl	48	73,758	123,001
ixx(C++)	widget.idl	52	73,757	134,432
photon(C++)	Not required	86	47,092	144,028

Table 1. Dynamic benchmark characteristics. The number of total instructions (3rd column) is expressed in millions.

We simulated several indirect branch predictors and compared their misprediction ratios. The total number of entries for each predictor was kept constant (set to 2K entries). We simulated tagless predictors (with the exception of the Cascade predictor which needs tags) in order to explore the design space behind tagless implementations which are less costly in hardware area than tagged ones. The list of the predictors follows:

- *BTB*: a tagless 2K entry BTB,
- *BTB2b*: a tagless 2K entry BTB, with 2-bit up/down saturating counter per entry,
- *GAp*: 2 tagless 1K entry PHTs, a 10-bit 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,
- *TC-PIB*: a tagless 2K entry Target Cache using a gshare indexing scheme, and an 11-bit PIB path history register recording the 2 lower-order bits from previous targets of indirect branches,
- *Dpath*: a Dual-Path hybrid predictor consisting of two GAp predictors and a 1K table of 2-bit up/down selection counters. Each GAp predictor is formed by: a tagless 1K entry PHT, a 24-bit path history register, a 2-bit up/down saturating counter per PHT entry for updating entries, and uses a reverse interleaving indexing scheme. 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 and a Leaky Filter (LF) of 128 entries. In the *Dpath* predictor, PHTs are 4-way set-associative using a true LRU replacement policy and the path lengths were 6 and 4 for the two GAp components, and
- *PPM-hyb*: a PPM predictor of order 10, 2 100-bit PHRs (10 targets, 10 low-order bits each), 10 Markov predictors with total 2K entries, an SFSXS indexing scheme, and a 2-bit up/down correlation selection counter per branch.

The reason for having different pairs of path lengths between the *Dpath* and the *Cascade* predictors is that the first scheme implements tagless PHTs, while the Cascade predictor requires tags. The values of the path lengths were taken from [8] and correspond to the optimal values for their benchmark suite. The misprediction ratios for the predictors are shown in Figure 6.

As we can see from Figure 6, the PPM predictor performs very well on average for all benchmarks. Overall, the PPM predictor achieves an average misprediction ratio of 9.47%, the second best was the Cascade predictor with 11.48% while third was the TC-PIB with 13.0%. The TC configuration achieves the lowest misprediction ratio for photon (0.95%) and is the only scheme outperforming the PPM predictor (1.35%) for that benchmark. This is because the benchmark's branches are easy to predict (according to previous experiments, we found that an oracle predictor recording complete PIB path history was able to achieve 99.1% accuracy when using a path length of 8). The PPM predictor experiences a few extra mispredictions due to the collisions between branches in the Markov predictors. The PPM algorithm has not managed to achieve the same low misprediction ratio as the Cascade, TC-PIB and the Dpath predictors for perl. The extra mispredictions are due to the aliasing between 3 individual branches that are executed frequently. Collisions in the PPM predictor not only cause extra mispredictions, but also incorrectly update the correlation selection counters and the 2-bit counters used to control the target update. We will address this issue when we describe a modification to the PPM predictor that alleviates some of the negative effects of false counter updates. Finally, the Cascade predictor achieves a lower misprediction ratio in eqn and in one of the edg experiments due to *filtering effects*. After analyzing the mispredictions on the PPM predictor we found that the

difference in mispredictions is due to branches that could be adequately predicted by a BTB-like structure (such as the Leaky Filter in Cascade). Those branches, when fed to the Markov predictors, displaced other branches that were strongly correlated.

If we look at the relative misprediction ratios of the predictors we can see that the BTB and the BTB2b structures are inefficient in predicting indirect branches. We can verify that the BTB2b improves the misprediction ratio in certain cases. Generally speaking, the accuracy of BTBs depends on the target entropy and the monomorphic nature of branches. The GAp and the Dpath predictors always improve predictability over the BTBs, with the Dpath outperforming the GAp for the majority of the experiments due to its ability to adapt to two different path lengths (short and long-term path correlation). The TC configuration was also very effective (in fact, it outperformed the Dpath predictor in most cases). The PIB path history recorded by a PHR was better at identifying branch instances, compared to using the history of MT jsr and jmp instructions recorded within the Dpath predictor. The Cascade predictor was able to improve prediction accuracy over the Dpath predictor for all benchmarks due to its unique ability to isolate monomorphic and low entropy branches, thereby eliminating their aliasing effects from the main predictor unit (Dpath component). As a result, the Cascade predictor suffered less mispredictions than the TC predictor in most benchmarks.

In order to evaluate the features that contribute to the accuracy of the PPM predictor we measured the distribution of accesses and misses to each individual Markov component. We found that for all the benchmarks at least 98% of the accesses (and misses) occur in the highest order Markov component. This is due to the selection mechanism and the update policy of the next order component in the PPM predictor. A lower order Markov table will be addressed for a prediction only when the selected location of its immediately higher order component is empty (valid bit is equal to 0). We use the valid bit because the simulated predictor is tagless and there is no way of identifying a branch in each Markov component. In addition, the update exclusion policy will always update the highest order component and all following references will be addressed by that component. Therefore, the exploitation of variable length path correlation is limited and the predictability improvement comes mainly due to the dynamic selection of correlation type and the indexing of the mapping function.

We also simulated two other versions of the PPM predictor, one having PIB-only path history (which requires one level of table access) labeled *PPM-PIB*, and a scheme similar to the PPM-hyb (with the exception of using the PIB-biased protocol for correlation selection) labeled *PPM-hyb-biased*. Simulation results are shown in Figure 7.

As we can see, the PPM-PIB predictor, which requires a single table access, improved prediction accuracy only when branches could be predicted efficiently with PIB history (e.g., eon, perl and both runs of ixx). For those experiments, the PPM-hyb predictor achieved slightly higher misprediction ratio due to unavoidable branch collisions in the Markov predictors. These collisions incorrectly modified selection counters, allowing branches to momentarily switch between the two weakly correlated states, causing some extra mispredictions. The PPM-hyb-biased predictor attempts to decrease the negative effects of this phenomenon. As we can see, for all four experiments mentioned above (eon, perl and both runs of ixx), the PPM-hyb-biased predictor eliminates the effects of aliasing and improves accuracy even more. For the rest of the benchmarks though, it was consistently outperformed by the PPM-hyb predictor (the gs benchmark run with the photon input being the only exception). This is because not all branches are strongly PIB-correlated.

This study is by no means complete. There exist several features of the PPM predictor that warrant further study. First, we assumed that the BIU module was of infinite size. Although such a unit is used by all predictors, limiting its size may have a larger impact on the PPM-hyb predictor due to its dependence on the selection counters. In addition, we need to consider tagged versions of all the predictors and analyze their prediction accuracy. The tagged version of the PPM predictor will allow for better exploitation of variable length path correlation. Moreover, it will lead to a fairer comparison with the Cascade predictor since the latter requires tags. We also did not consider the effects of varying table sizes. Finally, the sensitivity of the TC, GAp, Dpath and Cascade predictors on the path length was not addressed.

6. Conclusions

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

In this paper we present a new algorithm for indirect branch prediction. The algorithm is a modification of a data compression algorithm (PPM) and has been proposed for conditional branch prediction as well [5]. We discuss several design issues and present a viable implementation of the PPM algorithm coupled with dynamic selection of path-based correlation type on a branch basis.

We demonstrated that a Tagless PPM predictor achieved excellent prediction accuracy even against the best known indirect branch predictor (Cascade) which uses tagged ta-

bles. In the future, we plan to explore the design space in several ways: incorporate a filter for monomorphic and low entropy branches such as the one used in the Cascade predictor, simulate a tagged version of the PPM predictor, assign confidence on the prediction of different Markov components, and modify the update protocol.

7. Acknowledgments

We would like to thank the Micro'31 referees for their thoughtful comments. This research was supported by Microsoft Research and the National Science Foundation and by generous contributions from IBM and EDG.

References

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

[2] 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*, pages 397–408, January 1994.

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

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

[5] 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*, pages 128–137, October 1996.

[6] J. Cleary and I. Witten. Data Compression using Adaptive Coding and Partial String Machines. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[7] K. Driesen and U. Holzle. Limits of Indirect Branch Prediction. Technical Report TRCS97-10, University of California, Santa Barbara, June 1997.

[8] K. Driesen and U. Holzle. Accurate Indirect Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 167–178, June 1998.

[9] 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.

[10] 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*, pages 34–42, May 1991.

[11] D. Kaeli and P. Emma. Improving the Accuracy of History-based Branch Prediction. *IEEE Transactions on Computers*, 46(4):469–472, April 1997.

[12] J. Kalamatianos and D. Kaeli. On the Predictability and Correlation of Indirect Branches. Technical Report ECE-CEG-98-020, Northeastern University, June 1998.

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

[14] 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*, June 1998.

[15] J. Lee and A. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer Magazine*, 17(1):6–22, January 1984.

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

[17] A. Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

[18] Y. Sazeides and J. Smith. Implementations of context-based value predictors. Technical Report TR-ECE-97-8, University of Winsconsin, Madison, December 1997.

[19] H. Srinivasan and P. Sweeney. Evaluating Virtual Dispatch Mechanisms for C++. Technical Report RC 20330, IBM T.J. Watson Center, January 1996.

[20] A. Srivastava and A. Eustace. ATOM : A System for Building Customized Program Analysis tools. In *Proceedings of the SIGPLAN Conference on Programming Language, Design and Implementation*, pages 196–205, June 1994.

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

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

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

[24] A. Uht, V. Sindagi, and S. Somanathan. Branch Effects Reduction Techniques. *IEEE Computer Magazine*, 30(5):71–81, May 1997.

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

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

[27] 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*, pages 257–266, May 1993.

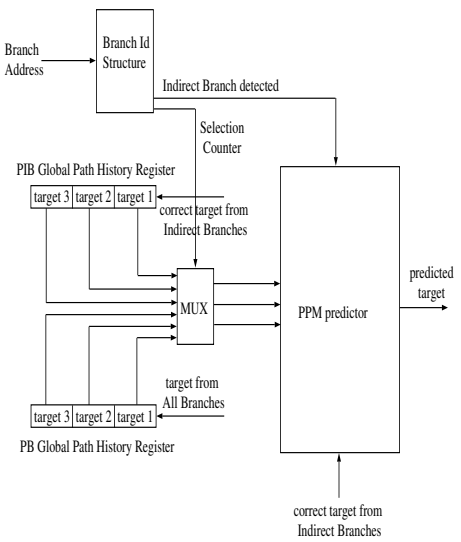

Figure 4. PPM predictor with Dynamic Correlation Selection.

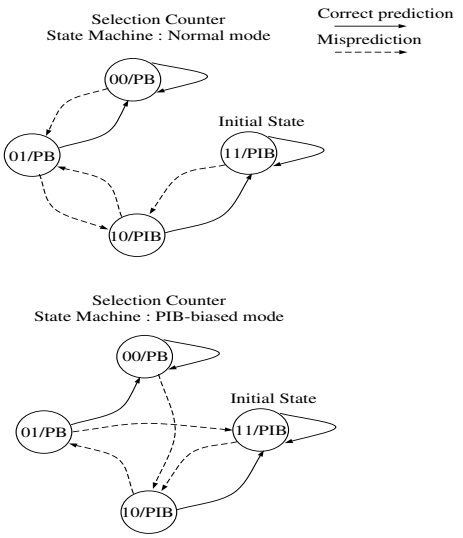

Figure 5. State machine of a 2-bit Up/Down Saturating Correlation Selection Counter.

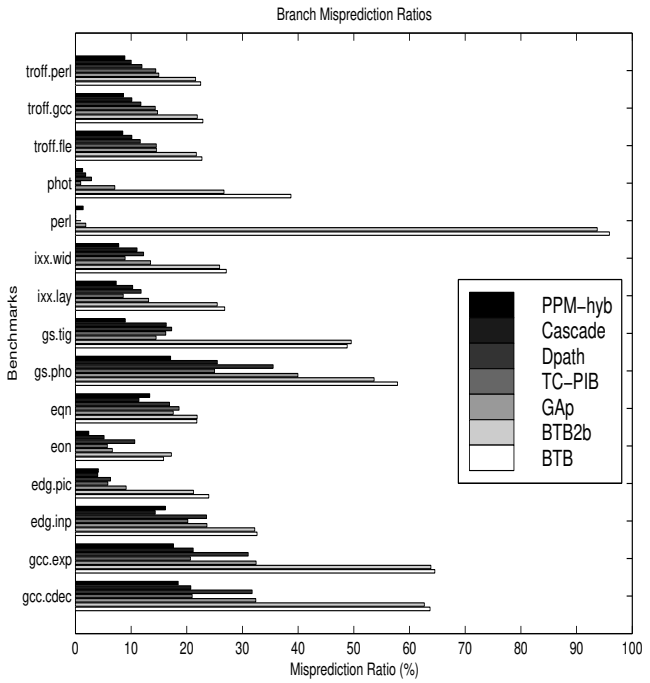

Figure 6. Misprediction ratios for indirect branch predictors utilizing 2K entries.

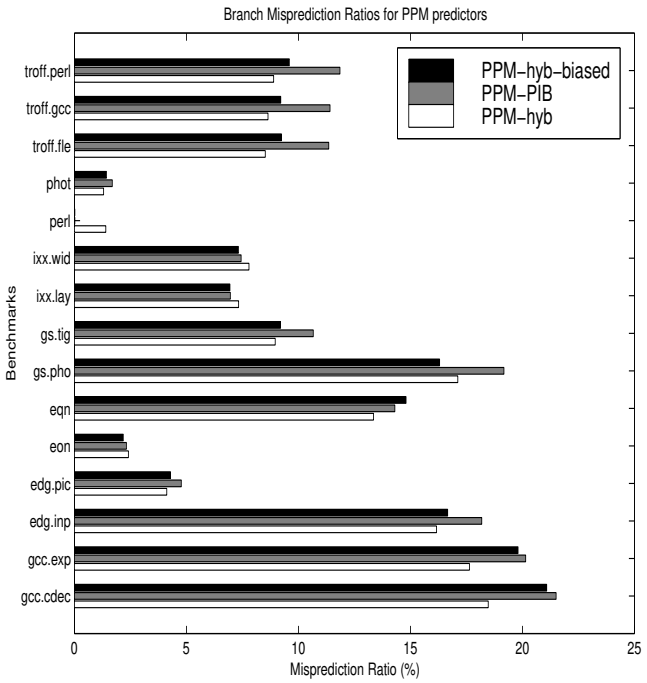

Figure 7. Misprediction ratios for 3 variations of the PPM predictor.