Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History

Renju Thomas, Manoj Franklin
ECE Department
University of Maryland, College Park
{renju, manoj}@eng.umd.edu

Chris Wilkerson
Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com

Jared Stark
Microprocessor Research
Intel Labs
jared.w.stark@intel.com

Abstract

Deep pipelines and fast clock rates are necessitating the development of high accuracy, multi-stage branch predictors for future processors. Such a predictor uses a collection of predictors, each of which provides its predictions at a different stage of the pipeline front-end. A simple 1-cycle latency line predictor provides predictions in the first stage, followed in a couple of stages later by predictions from a more accurate global predictor. Finally, one or two stages later, a highly accurate corrector predictor selectively corrects the global predictor's prediction. As the corrector predictor has the final say, its accuracy must be very high. The focus of this paper is to propose and evaluate techniques to build high-accuracy corrector predictors.

Our techniques rely on using a long global history, and identifying correlated branches in this history by using run-time dataflow information. In particular, we identify for each dynamic branch a set of branches called "affectors", which control the computation that affect that branch's outcome. We propose efficient hardware structures to track dataflow and to identify the affector branches for each dynamic branch; the hardware overhead for identifying affectors for all dynamic branches from a 64 branch global history is only 312 bytes. We then propose two prediction schemes that put to use the affector branch information. Experimental studies show that adding an 8KB corrector predictor (that uses affector information) to a 16KB perceptron predictor (total size 24.2KB) reduces the average misprediction rate for 12 benchmarks from 6.3% to 5.7%, an improvement achieved only by a 64KB perceptron predictor.

1. Introduction

Processor pipelines have been growing deeper and issue widths wider over the years. If this trend continues, branch misprediction penalty will become very high. Branch misprediction is the single most significant performance limiter for improving processor performance using deeper pipelining [18]. Better branch predictors and misprediction recovery mechanisms are essential to cut down the number of cycles lost due to mispredictions.

Global branch predictors improve prediction accuracy by correlating a branch's outcome with the history of the preceding dynamic branches. For smaller predictors (less than around 16KB), whose accuracies are mainly limited by destructive interference, a small increase in predictor size delivers a large improvement in the prediction accuracy. These accuracies can be further improved by applying specific interference reduction techniques [1][5][12][13][19].

By contrast, for larger predictors (more than around 64KB), lack of enough information in the global history to generate an accurate prediction limits the prediction accuracy. For example, a 1MB 2bc-gskew predictor provides only nominal prediction accuracy improvements compared to a 64KB 2bc-gskew predictor [16]. This is because the existing global history predictors use the most recent branches for correlation, and linear increases in the global history length causes exponential increases in predictor size. Further, all of the additional branches included may not be correlated and they preclude the inclusion of any highly correlated branches from farther past in the global history.

Larger predictors increase the prediction delay. Hence, future branch predictors will likely employ multiple predictors with progressively increasing latencies and prediction accuracies in an overriding fashion [2][8][16]. A line predictor with a single cycle delay will be followed by a more accurate multi-cycle primary global predictor, which, in turn, will be followed in the pipeline by a corrector predictor that will attempt to selectively correct the primary predictor. The corrector predictor has the final say in the overall prediction and hence has to be very accurate.

In this paper, we propose to create corrector predictor histories by identifying correlated branches (called *affectors*) from a long global history using run-time dataflow in-

formation. A branch becomes an affector for a future branch if it can affect the outcome of the future branch by choosing whether or not certain instructions that *directly* affect the future branch's source operands are executed. Because affectors have a direct effect on a future branch's outcome, they have an unusually high correlation with the branch they affect.

The affectors for each dynamic branch are identified at run-time by tracking the dataflow of the program in the front-end of the pipeline. We propose efficient hardware structures to perform the affector identification at run-time. The hardware structure for identifying affector branches from a long global history with 64 branches requires only 312 bytes. We verify that the identified affector branches are indeed more correlated than the remaining branches in the long global history by comparing the weights of a perceptron predictor[9] that uses the same long global history.

We propose two prediction schemes that can use the affector branch information for prediction. In the first scheme (*zeroing*), the identified affector branches are retained at their respective positions in the long global history for creating the predictor history. In the second scheme (*packing*), the identified affector branches are collected together as an ordered set for creating the predictor history. These affector histories are then used in a corrector predictor that selectively corrects the predictions of a large primary global predictor. The corrector predictor can correct the mispredictions of a large global predictor because of its ability to use correlated branch information from a long global history. Our results show that adding an 8KB corrector predictor that uses affector history to a 16KB perceptron predictor (total size 24.2KB) improves the average misprediction rate for 12 benchmarks from 6.3% to 5.7%, an accuracy achieved only by scaling a perceptron predictor to 64KB.

The rest of this paper is organized as follows. Section 2 explains the motivation for our approach and explains the related work. Section 3 explains how affectors can be identified during run-time, and the hardware structures required for it. Section 4 explains two schemes for using the identified affector information to improve prediction. Section 5 explains the experimental framework and an experimental evaluation of the proposed predictor. Section 6 summarizes the paper.

2. Motivation and Related Work

Future transistor budgets permit larger area for branch predictors. They suggest that future branch predictors can have additional components that rely on complementary behavior, and can even be in a later stage of the pipeline front-end [4][16]. The corrector predictor that we propose uses this model.

2.1. Looking at Long Histories

One way of exploiting complementary behavior is by using a longer global history than a conventional predictor. For a branch under prediction, some of the correlated branches may have appeared at a large distance in the dynamic instruction stream. This can happen, for instance, if two correlated branches are separated by a call to a function containing many branches; by the time the fetch unit returns from the function, the recorded global history may only contain the outcome of the branches in that function. A longer history is likely to capture the outcome of the correlated branch that appeared in the dynamic instruction stream prior to the function call.

The questions that naturally arise are: *(i) how far in the past should we go? (ii) what is a good way to use the longer history?* The length of the global history used is assumed to be smaller than or equal to $\log_2$ of the number of entries of the branch prediction table in most of the academic studies. Seznec *et al.* [16] recently showed that for a 64KB 2bc-gskew predictor, a history length of 24 was optimal, still not a very long history. They also showed that scaling a 2bc-gskew predictor from 64KB to 1MB provides only limited additional returns. Prediction accuracies of the recently proposed *perceptron predictors* [9][10] indicate that looking at much longer histories (of the order of 64 branches) can provide useful information for prediction. The techniques proposed in this paper look at long histories, of the order of 64 branches.

2.2. Selecting Correlated Branches

The next important issue deals with the exact manner of using the long history. A brute force approach is to increase the second-level table size accordingly, leading to an exponential increase in the hardware size. Another way of utilizing longer history is to use a conventional size table, along with some form of hashing of the long global history to access the table. However, unless performed intelligently, this can cause too much aliasing.

The above situation calls for a more selective use of the longer history. Not all branches in the long history may be correlated to the branch under prediction. This is evidenced by the fact that different branches require different length histories to achieve high prediction accuracies [11][20][21][1]. Evers *et al.* [6] went one step further, and showed that for 16-bit histories, the prediction accuracy is not significantly affected when only a few key branches are ideally *selected* from the global history and predictions are made solely based on their history. The important point to note is that the selected branches need not be the most recent

[1]The use of uncorrelated information increases the amount of aliasing, especially in small global predictors.

branches. Evers *et al.* also identified two primary reasons for two branches to be correlated. The first reason is that the preceding branch's outcome affects computation that determines the outcome of the succeeding branch. In our terminology, the former branch is considered to be an *affector* of the later branch. The second reason for correlation between two branches is that the computations affecting their outcomes are (fully or partially) based on the same (or related) information. In this case, we call the former branch to be a *forerunner* of the later branch.

The recently proposed perceptron predictors attempt to selectively capture the correlation behavior by using a weight for each branch in the long history [9][10]. During the training process, the highly correlated history positions tend to obtain higher weight. This provides for a *fuzzy* selection of the correlated branches. The perceptron weights represent the aggregate correlation that a branch history position has for the branches that share the weights. However, the actual correlation may not be with a history bit position, but a static branch in the history; this is more true for a longer history. For customized processors, the influential branch history positions may be statically determined [17].

3. Identifying Affector Branches at Run-time

In this paper, we identify affector branches at run-time by employing dataflow. It is conceivable that static analysis could also help in affector identification, although its effectiveness may vary compared to dynamic identification.

Consider the control flow graph (CFG) given in Figure 1. Assume that control is going through the shaded path and that we are interested in predicting branch B8, the last branch in the CFG. In a conventional global branch predictor, its history will be a pattern that records the latest outcomes of branches {B0, B2, B3, B5, B7} (i.e., `TNNTN`, assuming the pattern to be 5 bits long).

The source operands of branch B8 are obtained from registers `R2` and `R3`. This `R2` value is produced in BB3, which, in turn, is fed with a value (`R1`) produced in BB2. The `R3` value used by B8 is produced in BB7. Thus, the *affector basic blocks* of B8 within this CFG are {BB2, BB3, BB7} (which are marked with darker shades in Figure 1). The branches that decided that control should go through these basic blocks are {B0, B2, B5}. These branches are marked with circles in Figure 1, and are the affectors of this instance of B8.

3.1. Identifying Affector Branches

Unlike global branch history, the affector information for a dynamic branch cannot be easily derived from the affector information of the preceding branch. Therefore, *in addition to keeping the conventional global history*, we track the run-time dataflow of the program, and determine the affector branches for the last updates of each *architectural register*[2]. When a prediction is to be made for a branch, the affector information of its source registers are used on the fly to determine that branch's affector branches.

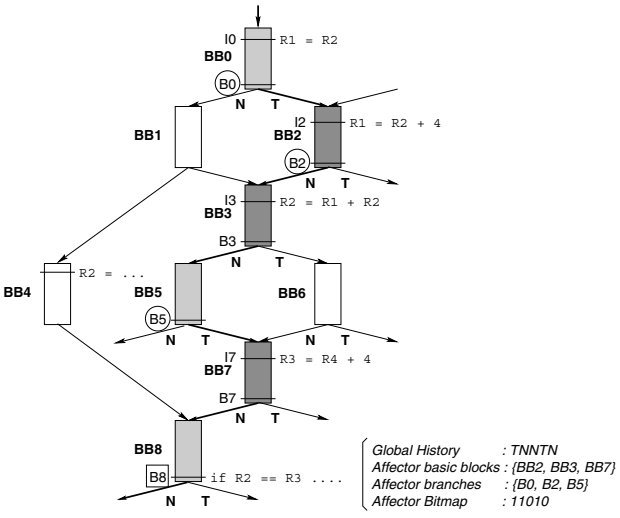

Figure 1. An Example Control Flow Graph (CFG) Consisting of Basic Blocks BB0 - BB8

In this section, we propose one way of systematically deriving affector information from the dynamic data flow graph (DDFG). The affector information is determined for each node in the data flow graph as a bitmap. The width of this bitmap is equal to the maximum number of recent branches over which affector branches are tracked. The bitmap is analogous to a global history except that the bits that are set to 1 in the bitmap indicate the affector branches, and the bits set to 0 indicate the non-affectors. In the ensuing discussion, the least significant bit (LSB) of the bitmap corresponds to the most recent dynamic branch. Thus, in Figure 1, the affector bitmap for B8 is `11010` (not including B8).

For calculating the affector branch information, we classify dynamic instructions into the following three non-overlapping classes:

- Conditional Branch Instructions
- Register-writing Instructions
- Non-register-writing Instructions: jumps, returns, nops, stores, system calls, etc.

[2]The affector instructions are those that form the backward slice of the branch in the dynamic data flow graph. The tracking of data flow is done at the granularity of dynamic instructions, but the affector information for each register is maintained only at the granularity of dynamic basic blocks.

Ideally, the dataflow communication through memory must also be tracked when identifying affectors. However, in this paper, we only track register-dataflow. The concepts presented in this paper can be extended to include memory-dataflow using techniques similar to memory renaming [3][14].

3.2. Hardware Structures: Affector Register File (ARF)

The affector information has to be dynamically generated and recorded in an efficient manner. We therefore need an efficient hardware structure to store the affector information of DDFG nodes. We keep a separate record of affector information corresponding to each architectural register as entries in an *Affector Register File (ARF)*. The number of entries in the ARF is same as the number of architectural registers, and each ARF register corresponds to an architectural register. Affectors are tracked through ARF in the front-end of the pipeline.

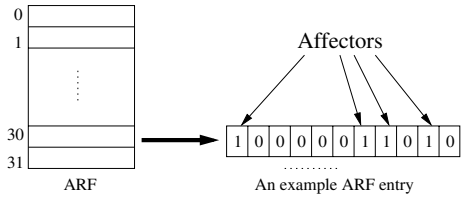

Figure 2. ARF Structure

An ARF structure and an example affector entry are shown in Figure 2. Each ARF entry records the affector information for latest computation that writes into the corresponding architectural register. For each entry, a 1 at bit position n from the LSB means that the n-th preceding branch in global history is an affector corresponding to that architectural register. Similarly, a 0 means a non-affector. Thus, for each entry in the ARF, the bit positions indicate whether the corresponding bit in the global history is an affector or not. The algorithm to mark the affectors is described below.

3.3. Affector Bitmap Generation Algorithm

When the processor encounters a conditional branch, all entries in the ARF are shifted left by 1 bit. The LSB for each ARF entry is made 0. This shifting makes the ARF entries move in lock step with shifts in the global branch history. The LSBs are set to 0 to indicate that the last branch in the global history has not yet affected the computation of any register.

Note that in energy-efficient implementations of ARF, the actual shifts may be replaced by a logical shift by using a single counter as the current column pointer. Instead of physically shifting, the ARF column pointed by the counter just needs to be cleared and the counter advanced to point to the next column. The ARF can further reduce energy internally and in the data path by taking advantage of the fact that it is more likely to store a 0 than a 1, by possibly keeping the internal state inverted.

When the processor encounters a register-writing instruction, the ARF entries corresponding to the source registers are read, OR'ed together and written to the ARF entry corresponding to the destination register along with a 1 in the LSB. The OR operation of the ARF entries of the source registers generates the affector information for the destination register as a union of the affector histories of the source registers. A 1 in the LSB marks the last branch in global history as an affector for the destination register. Thus, before encountering the next branch, all registers that get updated in that basic block will have the last branch marked as an affector in their corresponding ARF entries. Similarly, registers that do not get updated in the basic block will have the LSB of their corresponding ARF entries remain as 0.

Notice that for loads, this amounts to producing the affector information for the destination register using only the affector information of the address register, and not including the affector information of the corresponding store instruction.

The ARF entries, as we saw, keep track of the affector branches for the latest instructions that write into each architectural register. The affector branch information for a conditional branch instruction is inherited from its source operand producers. When the processor encounters a conditional branch instruction, the ARF entries corresponding to its source registers are read, and OR'ed. The resulting bitmap is called *Affector Branch Bitmap (ABB)*. Whether or not the ABB is latched depends on the timing design for the actual implementation. This bitmap is used along with global history for making predictions. The affector bitmap and global history of a branch can be combined in a variety of ways; Section 4 discusses two such schemes.

The formal algorithm for generating the affector branch information is given in Figure 3 for a specific instruction. The source registers are denoted by `rs1` and `rs2`, and the destination register is denoted by `rd`. And, ARF[x] denotes the affector register file entry corresponding to register `x`.

```
if (instruction is a conditional branch)
{
    ABB ← ARF[rs1] | ARF[rs2];
    Shift all ARF entries left by 1 bit
}
else if (it is a register-writing instruction)
    ARF[rd] ← ARF[rs1] | ARF[rs2] | 1;
```

Figure 3. An Algorithm for Generating Affector Branch Information at Run-Time

Table 1. Illustration of Affector Branch Determination for the Example given in Figure 1

Instr	Type	Update of ARF Entry or ABB						Left Shift ARF Entries
		Symbolic			Bit Patterns			
I0	Reg	ARF[1]	←	ARF[2]	xxxxx	←	xxxxx	No
B0	Br	ABB	←	....				Yes
I2	Reg	ARF[1]	←	ARF[2] | 1	xxxx1	←	xxxx0 | 1	No
B2	Br	ABB	←	....				Yes
I3	Reg	ARF[2]	←	ARF[1] | ARF[2] | 1	xxx11	←	xxx10 | xxx00 | 1	No
B3	Br	ABB	←	....				Yes
B5	Br	ABB	←	....				Yes
I7	Reg	ARF[3]	←	ARF[4] | 1	x0001	←	x0000 | 1	No
B7	Br	ABB	←	....				Yes
B8	Br	ABB	←	ARF[2] | ARF[3]	11010	←	11000 | 00010	Yes

Br: Branch instruction, Reg: Register-writing instruction, x: Don't know

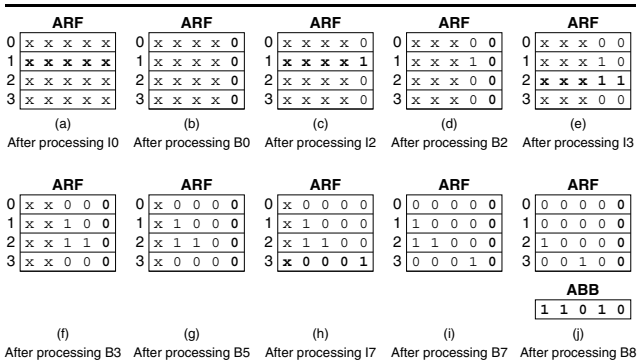

Figure 4. Contents of Affector Register File (ARF) and Affector Bitmap of Branch (ABB) after processing each instruction in the taken path in Figure 1. Five entries of the ARF are shown. Unknown bits are indicated by x. Changes that occurred in each step are shown in bold.

3.4. Example

Consider again the example CFG given in Figure 1. Table 1 and Figure 4 illustrate the working of our algorithm for this example code, showing what happens during the processing of each instruction in the taken path. For simplicity of presentation, we let each of the affector bitmaps to be only 5 bits long. Figure 4 shows the contents of the ARF and the ABB after executing each instruction in the taken path.

3.5. Misprediction Recovery

The updates done to the ARF by the affector identification algorithm are, in general, speculative, because of the speculative nature of the fetched instructions. When a branch misprediction is detected, speculative updates to the ARF have to be repaired. Whenever the processor performs a recovery operation, all of the ARF entries are shifted right by an amount equal to the number of branches squashed. For this purpose, the width of each ARF entry is extended by as many bits as the maximum number of branches that can be present in the pipeline. Every register-writing instruction in the pipeline carries along with it the ARF entry of the destination register before it was overwritten. The affector vectors carried along with the speculative instructions are rolled back from the youngest to the oldest of the squashed register-writing instructions.

Misprediction recovery will be simpler if the ARF tracks physical registers instead of architectural registers. In such a case, during recovery, the ARF entries corresponding to the recovered physical registers just have to be cleared and all other register entries shifted to the right by an amount equal to the number of branches squashed.

4. Building Predictors that Use Affector Information

In the previous section we presented a technique to generate a bitmap that identifies the affector branches in a long history. In this section, we propose two schemes that utilize the affector information to create affector histories for predicting branches. We also present an organization for a predictor that uses the affector history.

4.1. Turning Off Non-affector Bits and Hashing: Zeroing Scheme

In this scheme, all non-affector bits in the long global history are masked to become zeroes. This can be done, for example, by ANDing the branch's affector bitmap (ABB) with the long global history of the branch.

The resulting history register bits are hashed down to the required number of bits for the predictor index using a fold

and XOR hash technique. Figure 5(a) illustrates the zeroing algorithm for the case where the global history is stored as a pattern of outcomes.

In the zeroing scheme, the identified affectors are retained in their respective positions in the long global history before hashing and folding it. The non-affector branches in the long global history are represented as zeroes; thus forming "don't care" bubbles between affector branches. The affector branches actually represent the computation in the basic block they control and hence their representation can vary. For example, instead of representing each affector branch as a binary outcome, it can be represented as a hash of the PC values of the instructions in the basic block it represents, or other equivalent representations.

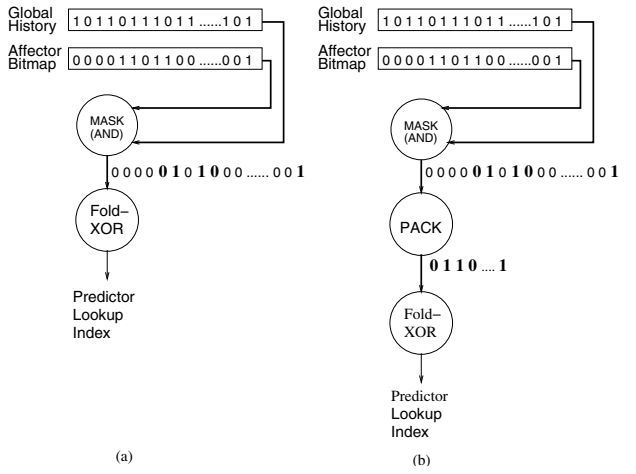

Figure 5. Illustration of Schemes for Using Affector Information in Branch Prediction (a) Zeroing; (b)Packing.

4.2. Packing Affector Bits: Packing Scheme

In the scheme we just saw, the non-affector bits in the global history are zeroed out. In the packing scheme, after the affectors in the global history are identified by the mask operation, we remove the non-affectors altogether. Thus, two neighboring affectors become bit neighbors after packing when each affector is represented by a bit. In this case also, the affector branch representation may use multiple bits as explained in the previous paragraph, to preserve the affector branch information while hashing. The resulting packed history is hashed down to the required number of bits for the predictor index. The packing scheme is illustrated in Figure 5(b) for the case where the global history is stored as a pattern of outcomes.

Notice that in the zeroing scheme, the affectors are retained in their respective positions in the long global history and the non-affectors are represented as zeroes. In the packing scheme, on the other hand, the non-affectors are removed completely from the long global history and the affectors are compacted together retaining their relative positions but losing their absolute positions in the long global history.

4.3. Proposed Predictor Organization

Figure 6 shows a pipeline timing diagram for the different predictors. When an instruction is being fetched, it is assumed that the logical register names can be made available early and hence the ARF entries for the source registers can be read. For RISC-like instruction sets with highly regular instruction encoding, the ARF read is possible along with the fetch [2][15]. It should be noted that for specific microarchitecture and timing design constraints, the implementations and optimizations will vary.

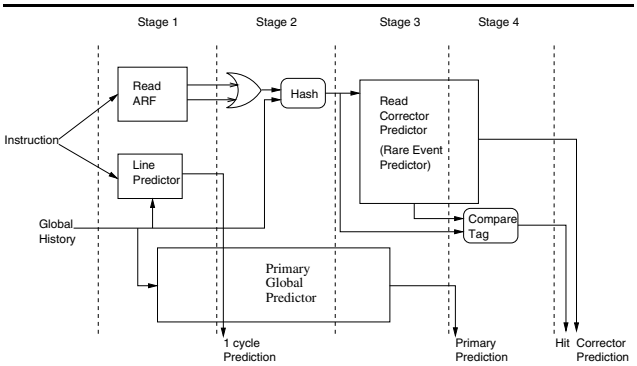

Figure 6. Approximate Pipeline Timing for the Corrector Predictor in a Deep Pipeline

The OR and hash logic for the corrector predictor index formation is estimated to take one cycle. An 8-way associative rare event predictor is used as the corrector predictor; each entry has 18 bits. The size of the corrector predictor is set to around half the size of the primary global predictor. Hence, a corrector predictor corresponding to a 16KB primary predictor will have only 512 sets. Hence, the additional delay in creating the affector history from the global history is compensated in part by a shorter table access time for the corrector predictor compared to the primary predictor. Both the primary and corrector predictors are assumed to be pipelined.

The speculation mechanism for the pipeline in Figure 6 is as follows. The line predictor provides a 1-cycle prediction in time for the next fetch. Later, the primary predictor provides its prediction and the fetch is redirected only if the primary predictor outcome contradicts the line predictor outcome. Afterwards, the corrector predictor provides

its prediction and the fetch is redirected only if the corrector predictor has a hit and the prediction outcome contradicts the primary predictor outcome. In any case, when the branch is actually executed, the processor recovers if it has been going down the wrong path.

The corrector predictor is a modified form of the rare event predictor (REP) design by Heil *et al* [7]. We use affector histories to train the REP whereas Heil *et al* used a history derived using data values. Each 18-bit entry in the REP comprises of a 2-bit saturating counter for training the prediction outcome, a 6-bit confidence mechanism called the replacement counter and a 10-bit tag. The entry with the lowest replacement counter value is replaced in a set. The replacement counter is incremented on a corrective prediction, and decremented on a misprediction or a redundant prediction compared to the primary predictor. Further, the REP does not provide a prediction even on a tag match if the replacement counter value is less than 2. The affector history in hashed form is split into index and tag for the REP. Thus the REP trains to provide a prediction only for affector histories that can consistently correct the mispredictions of the primary predictor. We update the primary predictor in all cases giving the primary predictor a chance to predict all branches, unlike the REP design by Heil *et al* where the conventional global predictor was selectively updated.

5. Experimental Evaluation

5.1. Experimental Framework

We use SimpleScalar v3.0 (sim-outorder) using Alpha ISA for the simulations in this paper. The measurements reported in this paper are taken for 12 benchmarks from the SPEC95 and SPEC2000 integer benchmark suites. The benchmarks used are `bzip, eon, gap, gcc, go, gzip, mcf, parser, perlbmk, twolf, vortex, and vpr`. The SPEC95 benchmark `go` is included because it is a difficult benchmark for branch prediction. For all benchmarks, the initial 2 billion instructions are skipped and then the measurements are taken for next 500 million instructions. We predict all conditional branches in the Alpha ISA. All graphs show the arithmetic mean for the result over the 12 benchmarks used, except where specially mentioned.

5.2. Experimental Evaluation

For the experimental evaluations of zeroing and packing algorithms, we represent an affector branch using a combination of the least significant 6 bits of the target of the affector branch and the address of the next conditional branch. Henceforth in this paper, we call this identifier for the basic block as BBID. This is a representation of the basic block that contains the affector instruction; other equivalent or perhaps better representations are also possible. We found that the number of bits used in BBID is not a highly sensitive factor affecting accuracy. The additional storage cost from using more bits for BBID is also not very significant.

Table 2 shows the sizes of the primary predictor, and corrector predictor, and the total size used in our experiments, with the corrector predictor size split into component sizes for (1) the rare event predictor and (2) the affector identification mechanism hardware. The affector identification overhead is calculated as the number of branches in the long global history used (given in Table 3) times the sum of the number of entries in the ARF and the number of BBID bits used. It can be seen that the affector identification hardware cost is comparatively very small.

Table 2. Sizes of the Corrector Predictor Used for Different Primary Predictor Sizes.

Primary Predictor Size (Bytes)	Corrector Predictor Size		Total Size (Primary + Corrector) (Bytes)
	REP Overhead (Bytes)	Affector Identification Overhead (Bytes)	
1K	0.56K	0.06K	1.62K
4K	2.25K	0.11K	6.36K
16K	8.02K	0.22K	24.24K
64K	36K	0.3K	100.3K
256K	144K	0.3K	400.3K

5.3. Misprediction Results

Figure 7 shows the misprediction rate when the zeroing and packing affector histories are used in a corrector predictor along with perceptron and YAGS primary predictors, respectively. The perceptron predictor used in Figure 7 is one of the most accurate branch predictors proposed so far. The total predictor size (primary+corrector) is shown on the X-axis. The three prediction schemes have comparable sizes for a given X-axis value, and hence provide a fair comparison.

It can be seen from Figures 7(i) and (ii) that corrector predictors that use zeroing and packing affector histories along with a primary predictor outperform a bigger primary predictor that has the same overall size. The benefit is more apparent for larger predictor sizes. One of the reasons is that the overhead for dataflow tracking is negligible compared to these predictor sizes. Another reason is that the absolute size of the corrector predictor REP is also increased for larger predictors (as can be seen in Table 2). Hence, the corrector predictor is able to hold more branches. The improvements of prediction accuracy for affector based predictors over a large primary predictor is a promising result that shows that this approach and future extensions along this

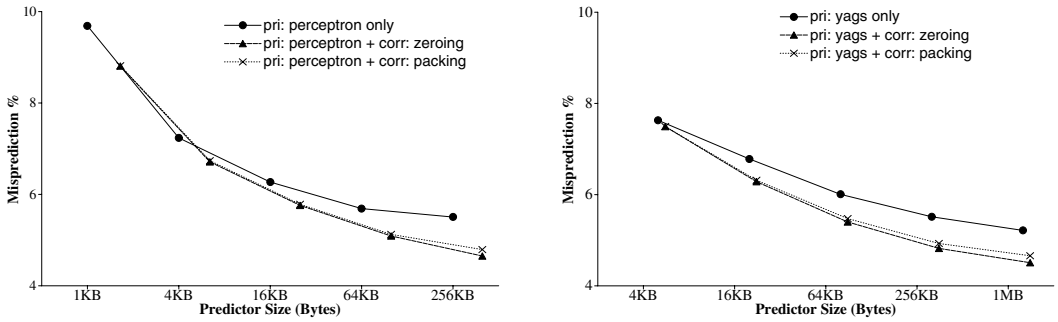

Figure 7. Misprediction Results for Zeroing and Packing techniques for our Corrector Predictor along with (i) Perceptron Primary Predictor; (ii) YAGS Primary Predictor

direction can be rewarding. For smaller predictors where interference is the major issue, interference reduction predictors like YAGS [5] already perform well. Zeroing is seen to slightly outperform packing for very large predictors, which have better capability to tolerate the extra patterns in the zeroing scheme.

Both zeroing and packing techniques show the ability to provide prediction accuracy improvements using a corrector predictor for both choices of primary predictors: perceptron and YAGS. We also simulated a Bi-mode predictor as the primary predictor, but it was seen inferior to YAGS. Seznec *et al.* show that there is no clear winner between YAGS and 2bc-gskew predictor [16]. A gshare predictor was also simulated as the primary predictor, but its accuracy was also inferior to the perceptron based predictor.

Next, we briefly present the results of analyzing the inherent correlation and pattern behavior of affector history using interference free simulations. By selecting, on the average, 4.39 branches out of 12-branch global history, the zeroing and packing affector histories achieve prediction accuracies of 93.75% and 93.7% respectively. In comparison, the use of all 12 branches in global history achieves a prediction accuracy of 94.5%. The affectors form only a (major) subset of the correlated branches. This accounts for the small accuracy loss compared to including all branches. In general, packing produced fewer patterns compared to zeroing, as expected, especially when identifying affectors from a large global history.

5.4. How Correlated are the Affectors?

Are the branches identified by the affector algorithms really the ones that matter? The magnitude of perceptron weights is indicative of the influence a prior branch has on a future branch [10]. Hence we verify that the average magnitude of the perceptron weights for the affector branches is indeed higher than the overall average of the perceptron weights for all branches in the global history considered by the perceptron. The third and fourth columns in Table 3 shows the above comparison. Note that the affector algorithms and the perceptron predictor compared look at the same size global history, as shown in Table 3.

Table 3 also shows the number of affectors identified for different history lengths. It can be seen that affectors are identified even from distant dynamic branches.

Table 3. Number of Affectors Identified, and Their Correlation

GH length	Avg. number of affector branches	PW for all branches	PW for affector branches
13	4.6	18.5	20.8
24	6.9	18.9	22.5
47	10.9	17.2	22.1
64	13.7	14.5	19.3

GH length: Global history length considered by perceptron (same used for affector identification), **PW**: Average magnitude of perceptron weights

The misprediction reductions obtained from using the affectors is not merely achievable by a *random* selection from a long history. We verified this by simulating a scheme where for each dynamic branch, we randomly selected the same number of branches as identified by our affector identification scheme. As expected, our results confirmed that a random selection provides very little complementary behavior. These results are not presented for lack of space.

The above experiments were primarily geared towards exploiting complementary information from the affectors in a long history. However, another simultaneous benefit comes from the reduction in patterns by using a subset of history instead of all branches from the global history. To demonstrate this benefit, we compare an affector history identified from 12 branches with a gshare history that uses all 12 branches. The affector history in this case is strictly a subset of the global history. Replacing the history for a 12-bit gshare with a 12-bit zeroing history in this manner increased the average prediction accuracy for the 12 bench-

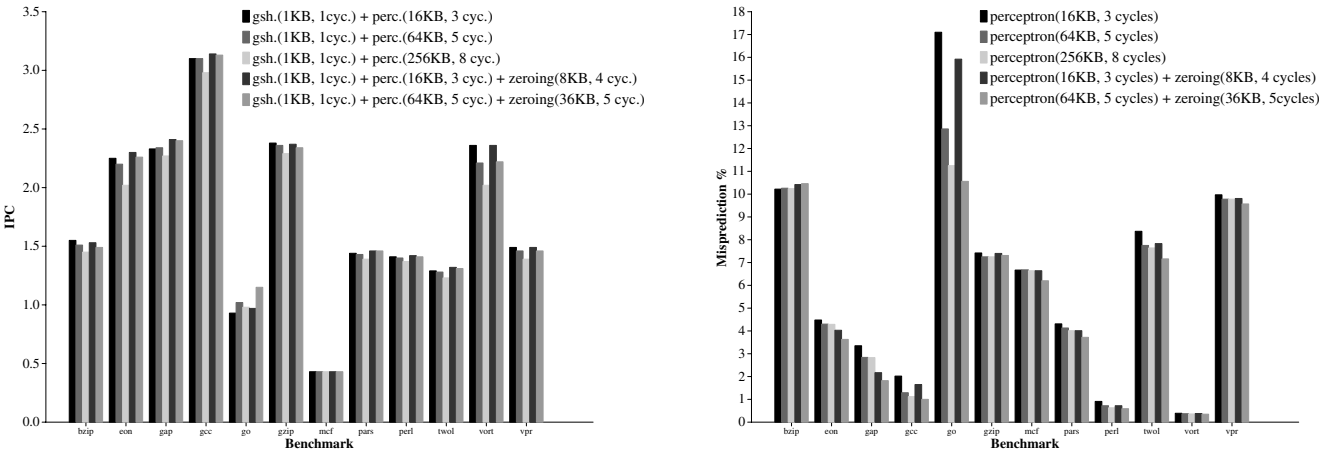

Figure 8. (i) Performance of a Modeled Superscalar for Various Branch Corrector Predictor Schemes. (ii) Per-benchmark Misprediction Rates for the Corresponding Corrector Predictors.

marks from 88.7% to 89.6%. Similarly, replacing the 12-bit gshare history with a packing history increased the prediction accuracy to 89.8% (detailed results are not presented due to space constraints). The gshare size is 1KB and the zeroing and packing predictors have 59 bytes overhead for identifying affectors. The benefits in this case is mainly due to reduced interference. Other interference reduction techniques may yield better results when restricting the history to 12 bits. For larger predictors, table interference is less of an issue, but warm-up effects may be significant.

Our preliminary results (not presented in this paper) indicate that tracking affectors through memory-dataflow can improve the prediction accuracy but implementations are typically expensive in hardware. The other source of correlation, the *forerunners* may also be identified. Discussing such techniques further is beyond the scope of this paper.

The number of branches selected and the particular branches selected from the long global history are not restricted while identifying affectors, and both can vary for each dynamic branch. Hence, to define a best selection, for each dynamic branch, we will have to search for the best number of branches to be selected and the best particular branch selections out of the long global history, an exponentially hard problem. Therefore, evaluating a limit for the best selection of branches from a long global history to compare our affector identification is not practical.

5.5. Performance Results

Next, we evaluate the performance of using our corrector predictors in a superscalar machine model with a 20-stage branch recovery pipeline, and with the configuration given in Table 4.

Figure 8(i) shows the IPC obtained using the different predictor schemes with conservative latencies, for the 12 benchmarks. The legend shows 5 different predictor combinations using different predictor types, sizes, and latencies. For example, the last scheme in the legend uses a 1KB gshare predictor with 1-cycle latency, along with a 64KB perceptron predictor with 5-cycle latency, and a 36KB zeroing REP predictor with 5-cycle latency as the overriding predictors. The speculation mechanism in the pipeline is as explained in Section 4.3. Figure 8(ii) shows the per-benchmark misprediction rates of the perceptron and zeroing REP predictors for the corresponding schemes used in Figure 8(i).

It can be seen from Figure 8(i) that the use of corrector predictors provides a net performance advantage, albeit small, for most of the benchmarks. The performance is slightly reduced for the compression benchmarks `bzip` and `gzip`. This can be attributed to a small loss in the prediction accuracy when using corrector predictors for these benchmarks, as can be seen in Figure 8(ii). The affector based corrector predictors provide a significant performance boost for the benchmark `go`, especially with a larger corrector predictor. In the future, enhancements to the affector based schemes, by tracking dataflow through memory, and identifying other types of correlated branches like forerunners, can significantly increase the prediction accuracy and boost performance.

Table 4. Modeled Superscalar Parameters

Fetch	8 instructions/cycle, 1 taken branch
Issue	8 instructions/cycle,
Inst. window	256 instructions, 128 loads and stores
L1 D-cache	16KB, assoc 4, 32 byte lines, 2 cycle lat.
L1 I-cache	32KB, assoc 4, 64 byte lines, 2 cycle lat.
L2 uni. cache	1MB, assoc 8, 64 byte lines, 12 cycle lat.
Memory	60 cycle lat.

6. Conclusion

A brute force approach to improving branch prediction accuracy using larger predictors provides only limited returns for large global predictors. Due to deeper pipelines and faster clock rates, the prediction latency of larger predictors will also be higher. Hence, a collection of predictors with progressively increasing delay and accuracy may be needed in an overriding fashion. The hard-to-predict branches of a primary global predictor is predicted by a very accurate corrector predictor with one or two cycles additional latency.

We proposed techniques by which a long global history can be used for this corrector predictor by identifying correlated branches in the history using run-time dataflow information. We tracked run-time dataflow using efficient hardware structures in the front-end of the pipeline. This mechanism identified for every dynamic branch, a set of branches from the global history called "affectors", which control the computation that affect the predicted branch's outcome. The affector identification for all dynamic branches from a global history with 64 branches takes only 312 bytes.

We proposed and evaluated two prediction schemes that used affector histories to lookup a corrector predictor tailored to correct only those branches that were consistently corrected by it. Our results showed that adding an 8KB affector history based corrector predictor to a 16KB perceptron primary predictor decreases the average misprediction rate for 12 benchmarks from 6.3% to 5.7%. In comparison, a perceptron predictor has to be scaled to 64KB, to achieve comparable accuracy. Corrector predictors based on affector histories can be improved further in future, and is a promising topic for future research.

Acknowledgements

We thank Prof. Donald Yeung and the anonymous reviewers for their many helpful comments. We also thank Intel Corporation and NSF (grant CCR 0073582) for funding this research.

References

[1] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 1996.

[2] L. Chen, S. Dropsho, and D. H. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Proc. 7th Int'l Symp. on High Performance Computer Architecture*, 2003.

[3] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. 25th Int'l Symp. on Computer Architecture*, pages 142–153, 1998.

[4] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proc. 31st Int'l Symp. on Microarchitecture*, 1998.

[5] A. Eden and T. Mudge. The yags branch prediction scheme. In *Proc. 31st Int'l Symp. on Microarchitecture*, 1998.

[6] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proc. 25th Int'l Symp. on Computer Architecture*, pages 52–61, 1998.

[7] T. H. Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. In *Proc. 32nd Int'l Symp. on Microarchitecture*, pages 28–37, 1999.

[8] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proc. 33rd Int'l Symp. on Microarchitecture*, 2000.

[9] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proc. 7th Int'l Symp. on High Performance Computer Architecture*, pages 197–206, 2001.

[10] D. A. Jimnez. Delay-sensitive branch predictors for future technologies. PhD thesis, Univ. of Texas, Austin, 2002.

[11] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proc. 25th Int'l Symp. on Computer Architecture*, pages 155–166, 1998.

[12] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proc. 30th Int'l Symp. on Microarchitecture*, pages 4–13, 1997.

[13] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 292–303, 1997.

[14] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proc. 30th Int'l Symp. on Microarchitecture*, pages 235–245, 1997.

[15] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternate approach. In *Proc. 26th Int'l Symp. on Microarchitecture*, 1993.

[16] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proc. 29th Int'l Symp. on Computer Architecture*, 2002.

[17] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *Proc. 28th Int'l Symp. on Computer Architecture*, pages 86–97, 2001.

[18] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proc. 29th Int'l Symp. on Computer Architecture*, 2002.

[19] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proc. 24th Int'l Symp. on Computer Architecture*, 1997.

[20] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Proc. 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 170–179, 1998.

[21] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao. Elastic history buffer: A low-cost method to improve branch prediction accuracy. In *Proc. Int'l Conf. on Computer Design*, pages 82–87, 1997.