Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing

Harish Patil and Joel Emer
Alpha Development Group, Compaq Computer Corporation
Harish.Patil@compaq.com Joel.Emer@compaq.com

Abstract

Dynamic branch predictor accuracy is known to be degraded by the problem of aliasing that occurs when two branches with different run-time behavior share an entry in the dynamic predictor and that sharing results in mispredictions for the branches. In this paper, we analyze the use of static prediction of certain branches to relieve the aliasing problem in dynamic predictors. We report on our experience with using profile-directed feedback to select branches that can profitably be predicted statically in combination with some well known dynamic branch predictors. We found prediction rate improvements of up to 75% for a simple branch predictor (ghist) and up to 14% for a very aggressive hybrid predictor (2bcgskew) for certain programs.

1. Introduction

Conditional branch prediction is a vital component of modern microprocessors. Conditional branches produce control flow changes that in general can only be known after the branches are executed. This can result in pipeline stalls. To minimize these pipeline stalls, processors rely on branch prediction techniques to decide which instructions to fetch following a conditional branch before the branch is executed. Correct branch predictions avoid pipeline stalls, but an incorrect prediction degrades performance because the processor has wasted time and resources evaluating wrong path instructions. As processor pipelines get increasingly deeper this performance degradation is becoming increasingly significant. Hence predicting conditional branches correctly is very crucial for performance.

Branch prediction techniques are divided into two main groups: static and dynamic. Static branch predictors use the results of pre-run-time analysis whereas dynamic branch predictors use run-time behavior of recent branches for making predictions.

Static prediction uses the knowledge of program structure or profiles from previous runs of a program to accurately predict the run-time outcome of some branches. Note that once a branch is predicted statically the prediction is typically fixed during a run of the program. On the other hand, dynamic branch prediction adapts to changing branch behavior of a program while it is run. Unfortunately, most dynamic predictors suffer from the 'aliasing problem' when two branches share some location in the dynamic predictor. If the two branches have different behavior this sharing can cause the dynamic predictor to repeatedly mispredict the branches. This effect is called destructive aliasing. The converse situation where the sharing improves prediction accuracy is called constructive aliasing.

In theory, the destructive aliasing problem can be avoided by making sure that each branch gets its own data location in the dynamic predictor. But that would make dynamic predictors too large to be practical, particularly given the sophisticated indexing schemes used by most modern branch predictors, and would also eliminate the possibility of constructive aliasing.

We analyze a scheme wherein the compiler (or an executable optimizer) alleviates the destructive aliasing problem in the dynamic branch predictor by predicting certain branches statically. The basic scheme was first proposed in [19]. We found that the key to the effectiveness of this scheme is choosing the right set of branches to predict statically. The static prediction scheme should complement the dynamic prediction scheme such that statically predicting certain branches either reduces the conflicts in the dynamic predictor or captures branches hard for the dynamic predictor to predict.

In this paper, we describe our experience with combining static and dynamic branch prediction for the Alpha architecture. We use profile-based analysis for selecting branches for static prediction. Such analysis can be easily implemented in an executable optimizer such as Spike[1]. We combined our static prediction scheme with well-known dynamic prediction schemes. We measured the number of conditional branch mispredictions per thousand instructions executed (MISPs/KI) with and without static prediction. We believe MISPs/KI is a better metric than prediction accuracy for measuring branch predictor effectiveness as

the latter can be deceptive if the test programs have too few or unevenly distributed branches. Fisher et al. [16] make a similar argument against using prediction accuracy as a performance metric.

We found static prediction to be very effective in conjunction with simple predictors such as 'ghist' and 'gshare' where static prediction can achieve the effect of doubling predictor size. In fact, we see an improvement of up to 75% in the MISPs/KI for 'ghist' (4Kbytes: m88ksim). Hybrid predictors of smaller size (2-4Kbyte) benefit substantially from static prediction. For example, the mispredictions per thousand instructions for a '2bcgskew'[2] dynamic predictor improved by up to 14% (2Kbytes:gcc) when combined with a static prediction scheme. Also, programs with a higher branch density, such as gcc, benefit with static prediction for all the dynamic predictor configurations we simulated.

Static prediction is prone to worst-case behavior when the branches do not behave the same way as anticipated by the compiler/optimizer. We did observe this phenomenon with our profile-based static prediction scheme when certain branches changed behavior going from the training to the final run. Fortunately, our planned analysis platform, Spike, can help iron out differences in branch behaviors in different runs of a program as it merges profiles from those runs. Further, as suggested by [5], static prediction hints can be re-written by a binary rewriting tool to tailor to a particular input.

In this study, we explored the use of static branch prediction to improve the prediction accuracy of dynamic branch prediction. The original idea was proposed in [19]. We experimented with static prediction schemes that are easy to implement and yet effective when used with very aggressive dynamic predictors. We also studied the effect of static prediction on collisions in various dynamic predictors.

The contributions of this work are as follows:

1] We discovered that although predicting certain branches statically reduces destructive aliasing in dynamic predictors the outcomes of those branches are sometimes crucial for capturing the 'branch correlation' principle (described later) that some dynamic predictors are based on. Thus, it is sometimes helpful to keep shifting outcomes of statically predicted branches in the 'ghist' register (again described later).

2] We found that the combination of a static and a dynamic branch prediction scheme is most effective when the two predictors work on different principles, i.e., when the two prediction schemes complement each other. For example, combining a bimodal predictor with a static predictor predicting highly 'biased' (i.e., mostly taken or not taken) branches shows very little improvement as both of them target highly biased branches. On the other hand, combining a ghist predictor with static prediction of highly biased branches shows excellent improvement. This is because the 'ghist' predictor does well on the set of branches showing correlation in behavior with other branches and the correlating branches are not necessarily highly biased.

In addition, we observed some interesting phenomena during our study:

- Conditional branches that are highly biased are highly predictable. There was a strong correlation of prediction accuracies of various dynamic predictors we simulated with the percentages of highly biased branches in our test programs. We observed this phenomenon even though each of the dynamic predictors simulated works on a different principle.
- Using the right training input for profile-based static branch prediction is very important. In our experience with SPEC95 programs we found that behavior of certain branches can dramatically change when the input to the programs were changed from 'training' to 'reference'. We propose a practicable solution for that problem.

2. Background

Most dynamic branch predictors predict the future behavior of branches using their past behavior. Simple branch prediction schemes use either the past behavior of the branch being predicted (as in 'bimodal' [17] scheme described later) or the behavior of neighboring branches (as in 'ghist', called GAg in [18], scheme) or a combination of the two (as in 'gshare' [7]).

At the heart of most simple branch predictors is a table of counters. Various branch prediction schemes differ in the way this table is indexed. On encountering a conditional branch in program flow, the table of counters is indexed for the given branch. The most significant bit of the counter at the indexed entry is used as the prediction for the branch. The counter is updated ('trained') once the outcome of the branch is known. Multi-level branch predictors have multiple tables where the final prediction is determined after a series of lookups with each lookup using the outcome of the previous lookup as index. Hybrid branch predictors combine two or more simple branch predictors. A "meta-predictor" or "chooser" is used to select among the predictions from the component predictors. The training of a hybrid predictor may involve updating all the component predictors ('total update' policy) or only a subset of the component predictors (a 'partial update' policy). Further the training may depend on whether the prediction was correct ('good') or incorrect ('bad').

Depending on the indexing scheme and the size of the table of counters in a simple branch predictor multiple branches in a program may share the same counter. This phenomenon is known as 'aliasing' and various branches are said to 'collide' with one another. If two colliding branches behave the same way the collision may in fact be 'constructive' as the two branches drive the shared counter value in the same direction resulting in correct predictions. On the other hand, if the two colliding branches behave differently, they would try to push the shared counter in different directions causing a lot of mispredictions. Young et al. [3] have shown that collisions in dynamic branch predictors are more likely to be destructive than constructive.

Many researchers have studied aliasing in dynamic predictors. Sprangle et al. [4] observe that there are three basic techniques to deal with the aliasing problem:

1. Increasing predictor size, possibly causing conflicting branches to map to different locations.
2. Selecting an indexing scheme that best distributes the available counters among different combinations of branch address and history.
3. Separating different classes of branches so that they do not use the same prediction scheme, and thus cannot possibly interfere.

We believe that the third technique can be very useful if one of the prediction schemes used is static prediction because static prediction can be tuned for a specific application. Also, the hardware requirement for static prediction is lower than dynamic prediction schemes. We assume the presence of two bits of static prediction hint similar to those available in Intel's upcoming IA-64 processor[5]. One bit in the conditional branch instruction is used to convey the static prediction to the hardware. The other bit conveys whether to use the static prediction or not. Thus, as described in [19], one of the bits is a static sub-component and the other bit is a static meta-predictor.

We simulated five different dynamic branch prediction schemes. Three classic simple branch prediction schemes 'bimodal', 'ghist' and 'gshare' and two of the best hybrid prediction schemes 'bi-mode' and '2bcgskew'. We now briefly describe these dynamic predictors.

The 'bimodal'[17] branch predictor gets its name from the 'bimodal' distribution in statistics. It works on the principle that branches in programs are either mostly taken or mostly not taken. In bimodal branch prediction scheme a table of saturating up-down counters (typically 2-bit) is maintained in hardware. This table is indexed with some bits from the address of the conditional branch being predicted. The most significant bit of the counter read from the hardware table is used as the prediction for the branch. The counter is incremented if the branch is actually taken or decremented if the branch is actually not taken. The counter values saturate to the maximum (minimum) value for mostly taken (not taken) branches. Studies [6] have indicated that there is hardly any benefit in increasing the size of the hardware table beyond 8K entries because a table of that size generally allows each conditional branch in a typical program to have its own counter. Thus there is very little aliasing present in a bimodal table of size larger than 2Kbytes (8K 2-bit counters).

The 'ghist' (*GAg* in [18]) branch predictor works on the principle of 'branch correlation'. It assumes that the outcome of a branch is dependent on the outcome of other branches, i.e., branching behavior is correlated. The table of saturating up-down counters in a ghist predictor is indexed using a 'ghist' register. The 'ghist' register maintains the 'global branch history'. It simply is a record of the outcomes of past few branches in the running program. On encountering a conditional branch the current value of the 'ghist' register is used to index the table of counters to obtain a prediction. When the outcome of the branch is known, it is shifted in the 'ghist' register. Unlike the bimodal branch predictor, a 'ghist' predictor does benefit from increased table size. This is because the index used in 'ghist' scheme is based on the outcome of a few recent branches and these outcomes keep changing at different program points. So the prediction for the same branch may use multiple counters in the hardware table depending on the 'ghist' values at the time of prediction. Also, multiple branches may use the same counter. Thus, there is a lot of 'aliasing' in the 'ghist' scheme.

The 'gshare'[7] branch prediction scheme tries to capture the best of the 'bimodal' and the 'ghist' prediction schemes. The index for accessing the hardware table of counters is computed using both the address of the branch being predicted and the value of the 'ghist' register. Just like the 'ghist' predictor aliasing is a problem for gshare scheme. Further, studies [8] have shown that the length of the global history is an important parameter for 'gshare' (and also 'ghist') schemes and that the 'best' value of history length varies with hardware table sizes and with programs.

The 'bi-mode' predictor [9] is a hybrid predictor with two gshare components. The choice predictor is a classic bimodal predictor whose output is used to choose between the predictions of the two gshare predictions. The bi-mode predictor uses a partial update policy whereby only the selected gshare component is updated with the branch outcome. The choice predictor is always

updated with the branch outcome except that when the choice is opposite to the branch outcome and the selected gshare makes a correct final prediction. As mentioned above, the best history length to use for gshare varies with table size and test program. Although in the version of the bi-mode predictor we simulated we always chose as many bits of global history as required by the gshare table.

The '2bcgskew' predictor [2] is another hybrid predictor with two component predictors. One of the component predictors is a bimodal predictor. The other component, called 'e-gskew', is itself another hybrid predictor with a bimodal and two gshare components. The same bimodal predictor is actually used both as a component of the final predictor ('2bcgskew') and a sub-component of the other component predictor ('e-gskew'). There is no choice predictor for the component hybrid predictor. Instead, a majority vote is taken to choose among the three outcomes from the sub-component predictors. The meta-predictor for the overall predictor is a gshare predictor that chooses between the outcome of the bimodal and the majority vote. The '2bcgskew' predictor uses a partial update policy:

- On a bad prediction all three banks of the e-gskew component are updated.
- On a correct prediction only the banks participating to the correct prediction are updated.
- The meta-predictor is updated only when the two component predictors disagree; the updating either re-enforces the current choice (on a 'good' overall prediction) or tries to push the meta-predictor to choose a different component (on a 'bad' overall prediction).

In our simulations of 2bcgskew the indexing functions for the gshare sub-components (2 predictors in 'e-gskew' and the meta predictor) were chosen carefully to avoid/minimize destructive aliasing. Unlike in the case of 'bi-mode' predictor mentioned above, we did select the best history lengths for various gshare sub-components for our simulation of '2bcgskew'.

In our simulations we selected between the dynamic prediction from one of the predictors described above and the static prediction predetermined by one of our selection schemes described in Section 4 later.

3. Related Work

Chang et al. [10] propose a *branch classification* mechanism. Branches are put into different categories depending on their run-time behavior. Branches in different categories are predicted by different predictors at run-time. They also propose using a static meta-predictor to choose between static and dynamic predictions. One of our schemes for static prediction (Static_95) is based on this work. We identify mostly taken/not-taken (highly biased) branches as 'easy to predict' branches and predict them statically to free up space in the dynamic predictor.

Grunwald, Lindsay, and Zorn [11] describe a way to combine static and dynamic prediction. Static prediction is used as a "meta-predictor" in the scheme they propose. They argue that in a hybrid dynamic predictor the final outcome is better selected statically. Statically generated hints are used to select among the outcomes of various dynamic predictors. Lindsay, in his thesis [19], investigated the use of a static sub-component in a hybrid predictor. He obtained excellent results with a variety of static hybrids he simulated. Our experiments corroborate his findings. In Lindsay's work the selection of branches to be predicted statically was with an iterative process involving profiling and simulations. One of the static selection schemes we studied (Static_Acc) is a simpler, single iteration, version of Lindsay's scheme.

Young and Smith[12] use a profile-based code transformation that exploits branch correlation to improve the accuracy of static branch prediction schemes. They analyze the program with the help of the profile data to figure out whether a given branch behaves differently if reached via different paths in the program. If it does then they duplicate code to provide different static predictions for the branch on different paths. This use of profile data to exploit correlation principle is orthogonal to our use of the profile data to exploit easy-to-predict and hard-to-predict branches. We can imagine using correlation to further improve the benefits seen by our static prediction method.

Sprangle et al. [4] describe an 'agree mechanism' for reducing destructive collisions. They propose using a table accessed by branch addresses to store a 'bias bit' for each branch. The 'bias bit' for a branch indicates the direction the branch is likely to take. They use a classic simple predictor with 2-bit counters as well. Although, instead of using the most significant bit of the outcome of the simple predictor as the branch prediction they use it to decide whether to use the 'bias bit' as the prediction for the branch being predicted. The idea is that if the biasing bit is well chosen, two branches using the same entry in the simple predictor are more likely to update the counter in the same direction.

The '2bcgskew' [2] scheme described above uses special indexing functions for the sub-component 'gshare' predictors. The idea is that if two branches collide in a given sub-component they will not collide in others. Further, the 'majority vote' makes sure that even if there is a misprediction from a sub-component predictor due to destructive aliasing it may be

compensated by a possibly good prediction from the other sub-component.

The 'bi-mode' [9] predictor scheme tries to minimize destructive collisions by channeling branches with the similar behavior to the same component 'gshare'. The meta-predictor in the bi-mode predictor is a classic bimodal predictor. The outcome of this bimodal predictor is used to choose between the predictions from the two gshare component predictors. Thus, for mostly taken branches one of the gshare predictors will be selected and for mostly not taken branches the other will be selected.

The mechanism we examined for reducing conflicts is orthogonal to the dynamic schemes proposed by other researchers. The dynamic predictor used in our scheme can be any modern dynamic predictor using sophisticated alias-reducing mechanism. In fact, we have studied combining static prediction with 'bi-mode' and '2bcgskew' predictors described above to see if the aliasing in them can be reduced further.

4. Methodology

We performed all our experiments with the Atom[13] binary instrumentation tool. We chose test programs from the SPEC95[14] benchmarks. All the programs chosen are integer (SPECINT95) benchmarks because we found that the branches in the floating point programs (SPECFP95) are highly predictable and hence uninteresting for branch prediction studies. SPEC-standard 'train' and 'ref' input sets were used for the test programs except when the 'ref' set consisted of multiple inputs. The multi-input cases, with the selected 'ref' input in parentheses, were gcc (2cp-decl.i), perl (scrabble.in), go (9stone21.in), and ijpeg (vigo.ppm). The test programs were compiled using Compaq's TRU-64 Unix compilers with the highest level of optimizations turned on.

Table 1 shows some characteristics of our test programs. The columns marked 'CBRs/KI' show the number of dynamic conditional branches per thousand instructions executed. These columns show for the test programs typically every 7^{th} or 8^{th} instruction executed is a conditional branch. The values of CBRs/KI also give the worst case (upper bound) values for our performance metric MISPs/KI (mispredicts per thousand instructions).

We instrumented conditional branches in our test programs using Atom. On each conditional branch we call a procedure that performs branch prediction using a pre-selected scheme and then updates misprediction statistics using the prediction and the actual outcome of the branch.

We ran our experiments in two phases. The first phase was the selection phase where we decided which branches from our test programs will be predicted statically and what their static predictions should be. We recorded the decision of this selection phase in a database. The second phase was the actual simulation of a dynamic predictor that used static hints from the previously generated database.

We targeted two types of branches for static prediction:

1. branches that are easy to predict for the dynamic predictor and
2. branches that are hard to predict for the dynamic predictor.

The motivation for statically predicting certain branches is to free up resources in dynamic predictor tables. Our hypothesis was that the branches in category (1) are presumably easy to predict hence using up dynamic resources for them would be a waste. Whereas branches in category (2) are better predicted statically because the dynamic predictor is not predicting them correctly.

Table 1. Benchmark Characteristics

Program	#Instructions (static)	#Cond.itional Branches (static) CBRs	Input:Train #Dynamic Instructions	Input:Train CBRs/KI (dynamic)	Input:Ref #Dynamic Instructions	Input:Ref CBRs/KI (dynamic)
Go	76 K	7777	0.52 Bil.	113	31.7 Bil	117
Gcc	314 K	38852	1.4 Bil.	155	1.6 Bil	156
Perl	95 K	9569	0.01 Bil.	112	25.3 Bil	122
M88ksim	57 K	5365	0.09 Bil	108	62.9 Bil	115
Compress	20 K	2238	0.03 Bil.	108	41.4 Bil	123
Ijpeg	62 K	5290	1.3 Bil.	69	27.6 Bil.	61

Table 2. Percentage of highly biased branches and branch prediction accuracy

Program	Highly biased branches (dynamic): bias > 95	Prediction accuracy: Bimodal (size 16Kbytes)	Prediction accuracy: Ghist (size 16Kbytes)	Prediction accuracy: Gshare (size 16Kbytes)	Prediction accuracy: Bimode (size 16Kbytes)	Prediction accuracy: 2bcgskew (size 16Kbytes)
Go	15.9%	75.7%	81.6%	82.0%	80.5%	83.1%
Compress	49.1%	84.5%	93.2%	92.5%	93.1%	93.3%.
Ijpeg	51.2%	89.6%	91.0%	91.2%	91.3%	91.5%.
Gcc	53.9%	89.2%	89.1%	93.7%	93.7%	94.8%
Perl	71.4%	93.7%	94.8%	98.0%	97.7%	98.2%.
M88ksim	85.5%	96.6%	96.4%	98.6%	98.4%	98.9%

We define the *taken bias* of a branch as the ratio between the number of times a branch is taken to the number of times the branch is executed. The *not-taken bias* is defined in a similar way. Note that since a branch is either taken or not taken *taken-bias* equals (1 – *not-taken-bias*). For measurements we define the *bias* of a branch as ***Max***(*taken-bias, not-taken-bias*). The bias of a branch can also be expressed as a percentage, e.g., a branch with a bias of 95% goes in one direction 95% of the times it is executed.

For selecting easy to predict branches, we simply looked at the bias of various branches. Any branch with a bias higher than a pre-selected cut-off bias was selected for static prediction. The actual static prediction for the branch was set to the direction of the bias (taken / not-taken). It turns out that highly biased branches are easy to predict for *any* dynamic predictor. Hence the phase for selecting easy to predict branches is independent of the dynamic predictor simulated in the second phase.

On the other hand, the set of hard to predict branches in a program changes with the dynamic predictor used to predict them. The same branch may be easy to predict for one dynamic predictor but hard for another. Thus, for selecting hard to predict branches, we actually simulated the dynamic predictor in the first phase. We then looked at the prediction accuracy of the simulated dynamic predictor for each branch. We selected those branches for static prediction for which the biases of the branches were higher than their prediction accuracies. The motivation being that by using the dominant biases of those branches as static prediction hints final prediction accuracies for those branches will never be worse. Notice that our methodology for selecting hard to predict branches is more complicated than simple profiling. We use per-branch prediction accuracy of the dynamic predictor. This data can be obtained by binary instrumentation or by on-line performance tools such as ProfileMe[15]. Grunwald et al. [11] used a similar technique in their study.

We assume that static prediction can be conveyed to the hardware using two hint bits as described in [5] – one of the bits describes the static prediction and the processor chooses between the static and dynamic prediction depending on the other hint bits. In our experiments we found that if the dynamic predictor maintains a global history (ghist) register then in some cases shifting outcomes of the statically predicted branches in the ghist register improves performance of the dynamic predictor substantially. Controlling the shifting of ghist for statically predicted branches may be done on a per application basis using an architectural flag or on a per branch basis using one extra hint bit in the conditional branch instruction.

5. Results

In this section, we report the results of our experiments to measure the benefit of combining static and dynamic prediction. For our basic set of experiments, profiling and measurements for static prediction were done with the same input. Such experiments with "self-trained" profiling depict the upper bound on improvements using static prediction. We discuss results with "cross-trained" profiling (where profiling and measurements are done with different inputs) at the end of this section. Also, unless otherwise noted, we did not shift outcomes of statically predicted branches in the global history (ghist) register.

Table 2 shows the prediction accuracies of various branch prediction schemes for our test programs. Also shown is the dynamic percentage of highly biased branches (taken/not taken bias > 95%). There are two things to note here. First, except for go, many of the branches in the test programs are highly biased—more than half the branches executed dynamically have a bias greater than 95% (Chang et al. [11] had reported similar findings). Second, there is a correlation between the percentage of dynamic highly biased branches and prediction accuracy of any of the branch predictor schemes simulated. Specifically, the more the

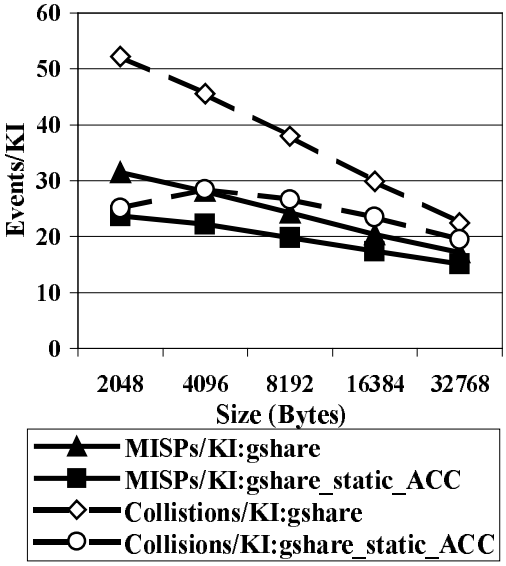

Figure 1. Go (train): gshare + static prediction: effect of increasing branch predictor size

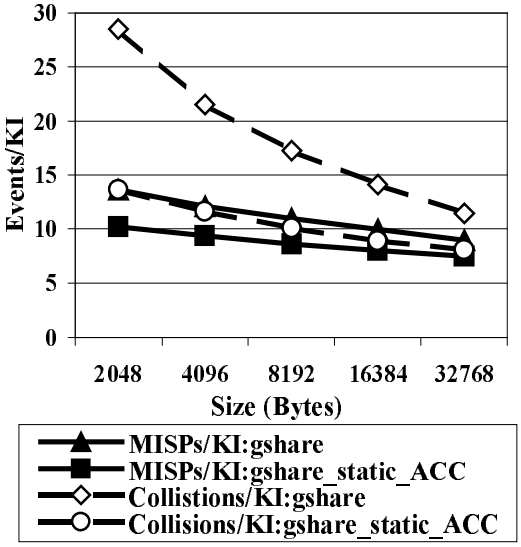

Figure 2. Gcc (train): gshare + static prediction: effect of increasing branch predictor size

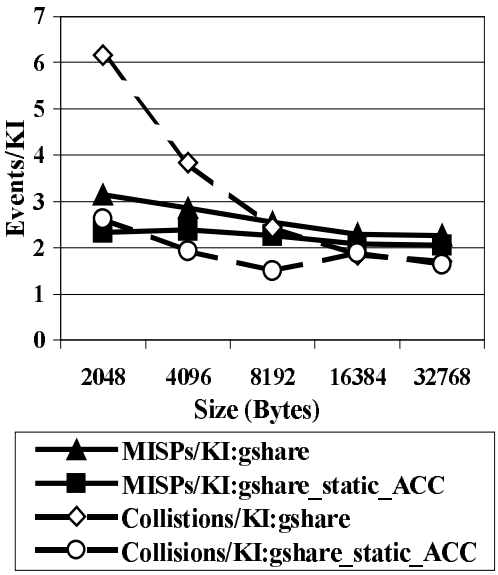

Figure 3. Perl (train): gshare + static prediction: effect of increasing branch predictor size

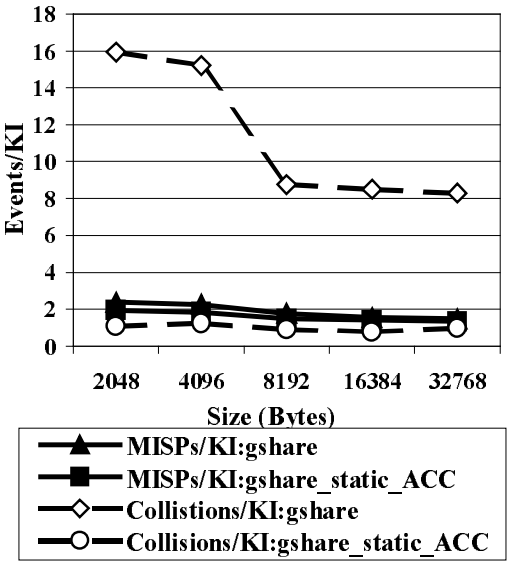

Figure 4. M88ksim (train): gshare + static prediction: effect of increasing branch predictor size

percentage of highly biased branches in a program, the higher the prediction accuracy of *any* dynamic predictor for that program. This correlation holds despite the fact that the simulated branch predictors work on different principles. Thus, it appears that highly biased branches are highly predictable. Data (except for compress) in Table 2 supports this hypothesis.

Figures 1-6 show the effect of increasing branch predictor size on MISPs/KI with and without static prediction. The base branch predictor is a gshare. The static prediction scheme chosen (*static_Acc*) selects branches each of which has a bias greater than the prediction accuracy of gshare for that branch. Also plotted in the figures are the total numbers of *collisions* observed for various data points. The collisions were counted by maintaining a tag for each counter in the

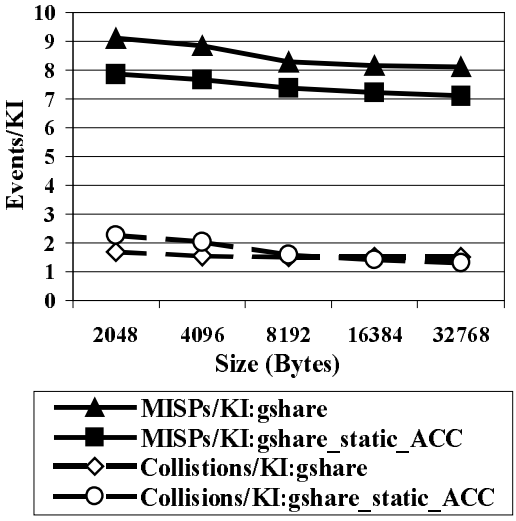

Figure 5. Compress (train): gshare + static prediction: effect of increasing branch predictor size

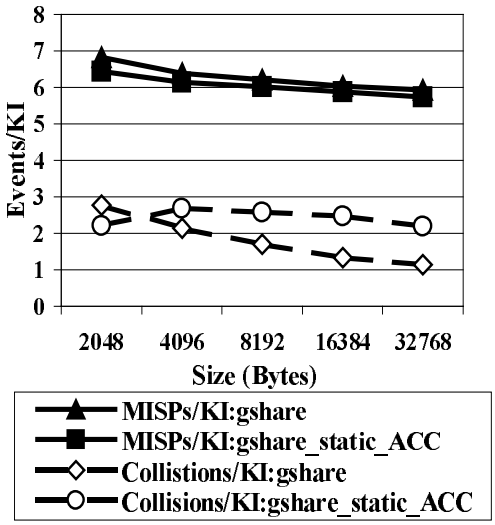

Figure 6. Ijpeg (train): gshare + static prediction: effect of increasing branch predictor size

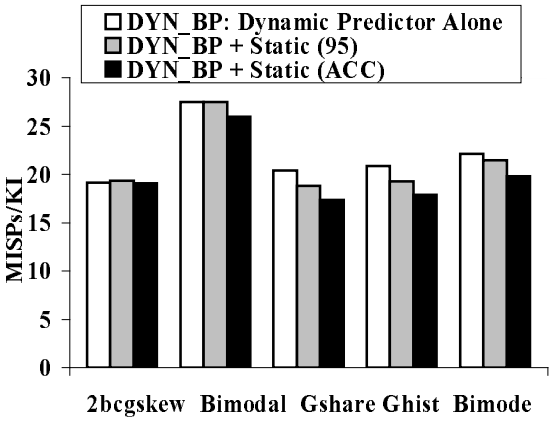

Figure 7. Go (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes]

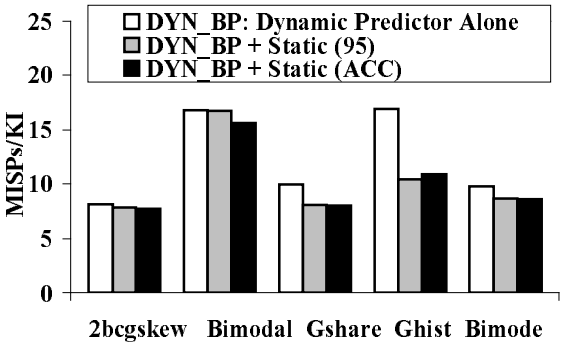

Figure 8. Gcc (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes].

dynamic predictor. The tag for a counter was used to store the address of the last branch using that counter. When we looked up the table of counters in the dynamic predictor for a particular branch, if the address of the branch did not match the tag then we counted the event as a collision. The number of collisions is expected to change with static prediction because 1) statically predicted branches do not index dynamic predictor tables and 2) if the outcomes of statically predicted branches are not shifted in 'ghist', then the indexing for dynamically predicted branches changes. When we found a collision, if the overall prediction was correct we considered the collision as constructive otherwise we considered it destructive. Thus, constructive and destructive collisions as we define them are different, but simpler to compute, than those defined by Young et al. [3].

There are many interesting points to note in Figures 1-6. First, static prediction always improves (i.e., reduces) MISPs/KI for Gshare for all the test programs at all the predictor sizes tested. The improvement due to static prediction is more at smaller predictor size though. The reason is that there are more collisions for smaller predictors resulting in more opportunity for static prediction. Total number of collisions almost always drop with static prediction except for a few cases, notably in case of ijpeg. We looked at the nature (constructive / destructive) of collisions for all our data points. It turns out that for ijpeg most of the increased collisions with static prediction are constructive. Hence there is no negative impact on MISPs/KI. As mentioned earlier, collisions depend upon multiple factors (predictor size, indexing scheme etc.). We do not yet

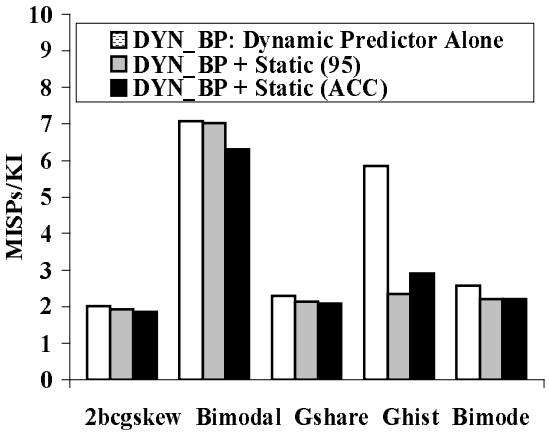

Figure 9. Perl (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes].

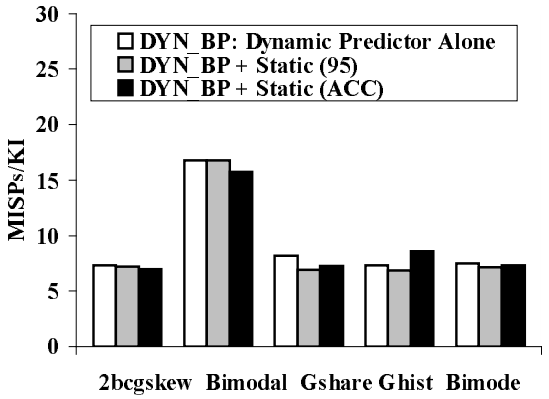

Figure 11. Compress (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes].

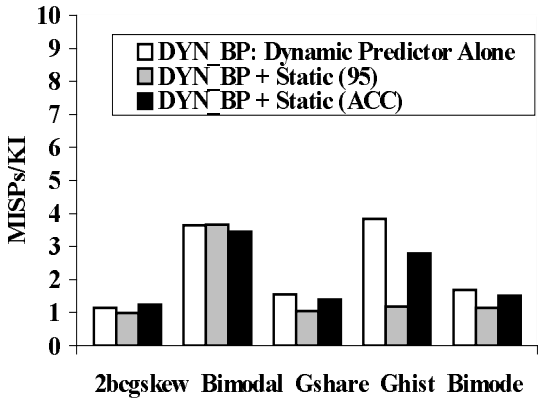

Figure 10. M88ksim (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes].

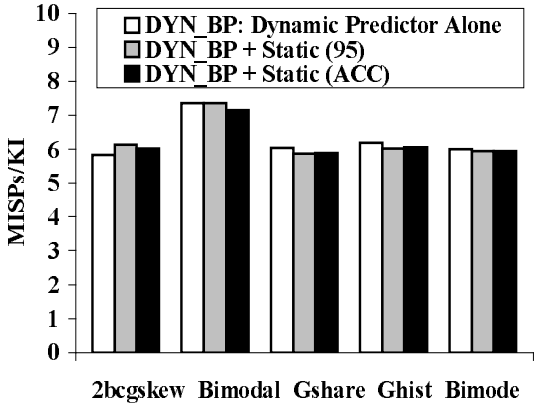

Figure 12. Ijpeg (train): Effect of static prediction algorithms [branch predictor size : 16Kbytes].

have a good explanation for the fact that with static prediction constructive collisions increase for ijpeg. This does, however, suggest another way of selecting branches for static prediction: we want to predict only those branches statically that will boost constructive collisions and reduce destructive collisions. We plan to explore this idea in the future.

Figures 7-12 summarize the effect of two different static prediction schemes on MISPs/KI for our test programs. There are 5 sets of bars for 5 different dynamic prediction schemes. Each set of bars depicts MISPs/KI for three different static prediction schemes: 1) No static prediction, 2) *Static_95*: static prediction of branches with bias greater than 95% (highly biased branches), and 3) *Static_Acc*: static prediction scheme of branches with bias greater than the accuracy of the corresponding dynamic predictor (difficult to predict branches). Note that the scale on the Y-axis is different for each figure in 7-12.

We see that the bimodal predictor does not benefit at all with a static prediction scheme of selecting highly biased branches (*static_95*). That is because the bimodal predictor itself targets and does well for highly biased branches, as highly biased branches drive bimodal counters to saturation quickly. In addition, a bimodal dynamic predictor can adapt to changing run-time behavior of branches and there is hardly any aliasing in bimodal tables of sizes we simulated. On the other hand, 'ghist' consistently improves with static prediction of highly biased branches as 'ghist' works on the principle of correlation that nicely complements the 'bimodal' nature of static prediction of highly biased branches. Combining 'ghist' with *static_95* is effectively like a 'gshare' scheme.

Table 3. 2bcgskew :Improvements in MISPs/KI with two static prediction schemes for go & gcc

2bcgskew Size	Go: Static_ 95	Go: Static_ Acc	Gcc: Static_ 95	Gcc: Static_ Acc
2 Kbytes	2.8%	7.7%	13.4%	14.1%
4 Kbytes	1.1%	4.1%	11.1%	12.0%
8 Kbytes	-0.1%	2.2%	8.0%	8.2%
16 Kbytes	-1.1%	0.4%	3.7%	5.2%
32 Kbytes	-2.3%	-1.4%	1.8%	4.2%

For m88ksim statically predicting highly biased branches (*static_95*) is better than statically predicting difficult to predict branches (*static_Acc*) for all dynamic predictors (except, of course, bimodal). Table 2 shows that most of the branches in m88ksim are highly biased and predicting them statically drastically reduces collisions in the dynamic predictors showing improvements. On the other hand, for programs with relatively few highly biased branches such as go and gcc, statically predicting hard to predict branches is better than statically predicting highly biased branches.

Ijpeg shows hardly any improvement with either static prediction scheme for any dynamic predictor; in fact, it shows slight degradation for 2bcgskew. One reason may be the lower frequency of occurrence of conditional branches in ijpeg -- 69 CBRs/KI as opposed to values double that for other benchmarks (Table 1). This suggests that aliasing may not be a problem for ijpeg. This jibes well with our finding that increasing predictor size, which is one way of reducing aliasing, benefits ijpeg very little for any dynamic predictor. On the other hand, gcc, which has the highest CBRs/KI (and also the highest number of static CBRs) among all the test programs, does consistently better with static prediction. In fact, we have observed that MISPs/KI improve for gcc with increasing capacity for all predictors (except bimodal) suggesting the presence of a lot of aliasing.

Among the dynamic predictors we simulated, we found 2bcgskew yields the best MISPs/KI. Unlike the other dynamic predictors it employs two techniques to remove aliasing: a sophisticated indexing scheme and the use of bimodal and gshare component predictors that target different set of branches. These two techniques seem to be quite effective in removing aliasing leaving very little room for improvement with static prediction. Although at smaller predictor sizes (2-4Kbytes) 2bcgskew did benefit from static prediction (see Table 3). On the other hand, we consistently saw improvement for 'gcc' which has the highest CBRs/KI among our test programs even at larger sizes of 2bcgskew.

Table 4. 2bcgskew: Effect of shifting history for statically predicted branches

	Size (bytes)	Static_ 95	Static _95_ Shift	Static_ Acc	Static _Acc_ Shift
go	32768	-2.3%	3.9%	-1.4%	5.8%
	65536	-2.4%	3.6%	-1.5%	5.0%
gcc	32768	1.8%	5.8%	4.2%	9.4%
	65536	-1.8%	5.3%	2.1%	8.9%
perl	32768	5.5%	1.0%	8.8%	4.4%
	65536	0.2%	0.2%	3.2%	4.0%
m88ksim	32768	14.9%	-5.2%	-8.1%	5.4%
	65536	19.0%	-6.0%	-14.6%	6.7%
compress	32768	6.2%	0.2%	4.5%	3.3%
	65536	11.7%	0.2%	5.0%	3.1%
ijpeg	32768	-2.8%	0.6%	-3.5%	2.0%
	65536	-1.7%	0.6%	-3.8%	1.5%

For predictors that use a global history of branch outcomes for indexing, shifting or not shifting outcomes of statically predicted branches will change aliasing. So we experimented with optionally shifting those outcomes in the global history register. Our findings for various sizes of 2bcgskew are tabulated in Table 4.

Columns marked Static_95 and Static_Acc show percentage improvement in MISPs/KI over the basic dynamic predictor with our two static selection schemes. Columns marked Static_95_Shift and Static_Acc_Shift show improvements (again w.r.t. the basic dynamic predictor) when each of our static prediction schemes is accompanied with shifting outcomes of statically predicted branches in a global history register. We notice that not all programs benefit from shifting. However, it is interesting to note that whenever a static prediction scheme shows degradation (e.g., ijpeg with Static_Acc) shifting shows improvement (see ijpeg with Static_Acc_Shift). Also both go and gcc show excellent improvement with shifting for both our static schemes even at very large predictor size (64Kbytes).

5.1 Profiling Technique for Static Prediction

All the numbers reported in this section so far were reported with "self-trained" profiling, i.e., profiling and measurements were done with the same input to our test programs. We also repeated our experiments with "cross-trained" profiling. Before we present the results we show some statistics on "cross-training" in Table 5. The table shows that when input is changed from 'train' to 'ref' two things can be noted (1) a different number of branches are executed and (2) even though many branches are common to the executions with the two inputs, the behavior of those branches changes widely at times.

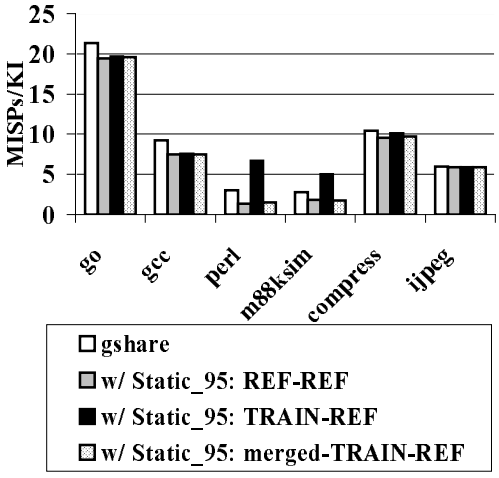

Figure 13. Effect of cross-training on profile-based static prediction: GSHARE (16KBytes) + static prediction (bias > 95)

All the columns in Table 5 show static/dynamic percentage of branches executed with 'ref' input. The column marked "Seen with .." shows data on branches that are executed with both the 'train' and 'ref' input, i.e., it shows the 'coverage' obtained by the 'train' input. Except in case of 'perl', the 'train' input executes almost all the branches the 'ref' input does. However, the behavior of the common branches changes quite a bit while going from 'train' to 'ref' input. The column marked "Majority direction change" shows data on branches that reverse majority direction when input is changed from 'train' to 'ref'. In addition statistics on branches whose bias changes by <5% and >50% is shown. We see that branch behavior for most programs varies with input and there is a non-trivial number of branches showing complete reversal of behavior. This suggests that inferences based on a 'train' profile will not always hold for 'ref' runs.

Profile-based optimizations make inferences about program behavior based on data from profiling runs. If the behavior of the program changes with input, the inferences based on the training input become less valid and sometimes can become completely wrong. In those cases static prediction can dramatically increase mispredictions. We observed few such cases in our experiments on profile based static prediction. Although the problem caused by change in behavior of programs with input can be alleviated by modifying profile data as programs are run with different inputs.

We envision profile-based static prediction being implemented using a binary re-writing tool such as Spike[1]. Spike maintains a database of profile data for every program. As a program runs with different inputs in 'instrumentation' mode, Spike collects execution profile for the program and updates the profile database. The program is later modified during the optimization phase based on the profile database.

For static prediction the profile data will consist of biases of different branches. We can imagine that as the profile database is updated anomalies in branch biases can be removed. For example the profile updating can filter out profile data about branches that change bias by, say, more than 5%. As seen in the column marked 'Bias change < 5%' in Table 5 this filtering could still retain most of the profile data.

Figure 13 shows results from performing static prediction using cross training. The bars show MISPs/KI for our test programs for a 'gshare' of size 16Kbytes. There are four bars for each test program for the following cases 1) No static prediction 2) Static prediction with "self-trained" profiling 3) Static prediction with naïve "cross-trained" profiling and 4) Static prediction with "cross training" with branch

Table 5. Branch behavior: training vs reference input.

		Seen with TRAIN/REF	Majority direction change	Bias change < 5%	Bias change > 50%
go	Static	90.1%	9.6%	53.3%	3.0%
	Dynamic	100.0%	5.1%	70.0%	0.1%
gcc	Static	97.6%	3.3%	83.7%	0.7%
	Dynamic	100.0%	1.7%	91.3%	0.0%
perl	Static	66.7%	7.5%	51.2%	3.8%
	Dynamic	89.1%	9.0%	60.4%	7.7%
m88ksim	Static	80.3%	3.1%	69.3%	1.4%
	Dynamic	99.1%	5.1%	84.9%	3.8%
compress	Static	95.1%	2.3%	88.1%	1.0%
	Dynamic	99.9%	0.0%	58.6%	13.3%
ijpeg	Static	99.5%	1.4%	94.7%	0.7%
	Dynamic	100.0%	0.1%	97.8%	0.0%

profiles merged by removing from the reference profile the data for branches with a bias change >5%. As the third bar in each set indicates naïve cross training can cause severe performance degradation in some cases. In particular, MISPs/KI degrade for perl and m88ksim by a large amount. Looking at the profiles from runs with 'train' and 'ref' inputs for these programs we found that there are some branches whose bias changes widely and those branches are also executed quite frequently. Thus for perl and m88ksim static prediction based purely on 'train' input is not so valid for 'ref' input and it hurts MISPs/KI drastically. As expected, using a merged profile alleviates the problem as seen by the 4th bars for perl and m88ksim.

6. Conclusions

We have analyzed a way to address the problem of aliasing in dynamic branch predictors. We examined combining static and dynamic prediction in such a way that predicting certain branches statically relieves the aliasing in the dynamic predictor. We measured the change in collisions in many well-known dynamic predictors when combined with static prediction. We studied two ways of using profiling to select branches for static prediction – first using a fixed cutoff bias to target easy to predict, highly biased branches and second using a per branch comparison of bias with prediction accuracy of a given dynamic predictor to target hard to predict branches. We found that static prediction can be quite effective for a) simple dynamic branch predictors for varying sizes, b) hybrid branch predictors of relatively small sizes, and c) for programs with a lot of branches irrespective of the type/size of the dynamic predictor used. We also observed that shifting outcomes of statically predicted branches in global history register has a noticeable impact on the overall effectiveness of a combined static-dynamic predictor.

7. Acknowledgments

We thank Andre Seznec, Venkat Krishnan, and John Edmondson for their consultation and help during this work; Geoff Lowney for supporting this effort; and Srilatha Manne for her suggestions and corrections after reviewing this paper.

8. References

[1] Robert Cohn, David Goodwin, P.G.Lowney, and N. Rubin, *Spike: An Optimizer for Alpha/NT Executables*, The USENIX Windows NT Workshop Proceedings, Seattle, Wash. (August 1997):17-24.

[2] Andre Seznec, Pierre Michaud, *De-aliased Hybrid Branch Predictors*, Research Report number 1229, IRISA, France. February 1999.

[3] Cliff Young, Nicolas Gloy, and Micheal D. Smith, *A Comparative Analysis of Schemes for Correlated Branch Prediction*, Proceedings of ISCA 1995: 276-286.

[4] Eric Sprangle, Robert Chappell, Mitch Alsup, and Yale Patt, *The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference,* Proceedings of ISCA'24 1997.

[5] IA-64 Application Developer's Architecture Guide Intel corp. http://developer.intel.com/design/ia64/index.htm

[6] John Henessy and David Patterson, *Computer Architecture a Quantitative Approach*, 2nd edition, Morgan Kaufmann publishers 1996.

[7] Scott McFarling, *Combining Branch Predictors*, WRL TN-36, Digital Western Research Lab, June 1993.

[8] Toni Juan, Sanji Sanjeevan, and Juan Navarro, *Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction*, Proceedings of ISCA'25, 1998.

[9] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor Mudge, *The Bi-mode Branch Predictor*, Micro-30, 1997.

[10] P. Chang, E. Hao, T. Y. Yeh, and Y. Patt, Branch Classification: a new mechanism for improving branch predictor performance. Micro-27, 1994.

[11] Dirk Grunwald, Donald Lindsay, and Benjamin Zorn, *Static Methods in Hybrid Branch Prediction,* Proceedings of PACT'98 1998.

[12] Cliff Young and Michael Smith, Improving the Accuracy of Static Branch Prediction Using Branch Correlation, Proceedings ASPLOS-VI, 1994.

[13] Amitabh Srivastava and Alan Eustace, ATOM: *A system for building customized program analysis tools*. Proceedings of PLDI'94, 196-205, 1994.

[14] SPEC CPU'95 information at http://www.spec.org

[15] Jeffrey Dean, James Hicks, Carl Waldspurger, William Weihl, and George Chrysos. *ProfileMe: Hardware support for instruction-level profiling on out-of-order processors*. Micro-30, 1997.

[16] Josh Fisher and Stefam Freudenberger, *Predicting conditional branch directions from previous runs of a program*, Proceedings ASPLOS-V, 1992.

[17] Jim Smith, *A Study of Branch Prediction Strategies*, Proceedings of the 8th ISCA, 1981.

[18] Tse-Yu Yeh and Yale Patt, *Alternative Implementations of Two-Level Adaptive Branch Prediction*, Proceedings of the 19th ISCA, 1992.

19] Donald Lindsay, *Static Methods in Branch Prediction*, Ph.D. thesis, Department of Computer Science, University of Colorado, 1998.