Published in the *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

A Comparative Analysis of Schemes for Correlated Branch Prediction

Cliff Young, Nicolas Gloy, and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138
{cyoung, ng, smith}@das.harvard.edu

Abstract

Modern high-performance architectures require extremely accurate branch prediction to overcome the performance limitations of conditional branches. We present a framework that categorizes branch prediction schemes by the way in which they partition dynamic branches and by the kind of predictor that they use. The framework allows us to compare and contrast branch prediction schemes, and to analyze why they work. We use the framework to show how a static correlated branch prediction scheme increases branch bias and thus improves overall branch prediction accuracy. We also use the framework to identify the fundamental differences between static and dynamic correlated branch prediction schemes. This study shows that there is room to improve the prediction accuracy of existing branch prediction schemes.

Keywords: branch prediction, branch correlation, branch stream characteristics.

1 Introduction

Recent work in branch prediction has led to the development of both hardware and software schemes that achieve good prediction accuracy by exploiting branch correlation [4, 9, 11, 14, 15, 16, 17]. However, little attention has been paid to *why* these schemes behave better than prior ones and to *where* further improvements can be made. In this paper, we describe an analytic framework that helps answer these questions based on the fundamental characteristics of the branch prediction problem. In addition, we use the observations based upon this framework to indicate potentially-fruitful research directions that will allow computer architects to improve branch prediction accuracy. Further improvements in branch prediction accuracy will enhance the effectiveness of global instruction schedulers and the performance of multiple-instruction-issue machines.

Branch prediction addresses two basic problems: predicting the direction of conditional branches, and quickly fetching instructions from the predicted target. These problems can be addressed separately, and in this paper, we limit ourselves to the former. In other words, we consider a *branch prediction scheme* to be a technique for improving performance by anticipating the outcome of conditional branches. Other work has shown how to couple a branch prediction scheme with a branch target buffer to eliminate the performance penalties of branches [7].

Why branch prediction schemes perform differently is just as important as how well they perform. Only after explaining why a scheme works can one understand appropriate ways to improve or alter it. Recent work by McFarling [9] and by Chang et al. [4] uses analysis, reasoning, and experimentation to devise better hardware schemes for correlated branch prediction. In particular, McFarling [9] noticed significant redundancy in the two-level index of the correlation-based branch prediction scheme proposed by Pan, So, and Rahmeh [11]. By hashing the branch history with the branch address, McFarling's *gshare* scheme often improves prediction accuracy under the constraint of a fixed-size table of predictors. Similarly, Chang et al. [4] noticed that, for a fixed-size table of predictors, branches biased to one particular branch direction more than 95% of the time exhibited better prediction accuracies on a two-level adaptive scheme [14] when one decreased the branch history length, while the rest of the branches exhibited better prediction accuracies when one increased the branch history length. This observation led them to propose several new hybrid branch prediction schemes with better overall prediction accuracies.

Still, it is more difficult to understand the actual workings of today's branch prediction schemes than it needs to be. To make it easier to develop optimizations such as those proposed by McFarling [9] and Chang et al. [4], we present a unifying framework that allows one to analyze and categorize branch prediction schemes. Because the framework is based on a theoretical model of the branch prediction problem, it is general enough to encompass all branch prediction schemes proposed to date. The framework focuses attention on how a prediction scheme assigns the dynamic branches of the program to individual predictors. This information then directs our analysis of and our search for weaknesses in a particular scheme, and allows us to isolate and compare different factors that affect prediction accuracy. In particular, we explore the fundamental differences between hardware- and software-based branch prediction schemes that exploit branch correlation. This analysis suggests several ways to improve the overall prediction accuracy of today's branch prediction schemes.

Section 2 describes our framework for classifying and analyzing branch prediction schemes. To demonstrate the generality of our framework, Section 2 presents many of today's popular branch prediction schemes in framework terms. In Section 3, we use the framework to explore the issues in when (and thus why) static schemes for correlated branch prediction work. Section 4 goes on to compare the differences between static and dynamic schemes for correlated branch prediction. As an example of the power of our approach, we also describe changes to correlation-based static and dynamic prediction schemes that improve their overall prediction accuracy. Section 5 summarizes the findings of this work.

2 A Framework for Branch Prediction

Given a conditional branch in a program, the goal of a branch prediction scheme is to predict accurately the outcome of that conditional branch (i.e. that the branch will take or that the branch will fall through).[1] The most accurate branch prediction schemes predict the next action of a branch based on some function of the past actions of that branch and possibly other branches in the program. To understand the capabilities of these branch prediction schemes and to compare competing schemes in a meaningful manner, we must be able to identify and quantify the important properties of branch prediction schemes. To achieve this goal, this section defines a set of mathematical tools that allow us to analyze program and branch behavior in an abstract manner.

2.1 Basic Definitions and Goals

Let a *branch execution* $e = (b, d)$, $e \in \mathbb{Z} \times \{0, 1\}$ be a pair consisting of an identifier $b \in \mathbb{Z}$ and a direction variable $d \in \{0, 1\}$. Intuitively, the identifier uniquely specifies a static branch in a program, and the direction variable indicates the direction that the branch went. We define an *execution stream* or just *stream* as a sequence of branch executions. Intuitively, this corresponds to a branch trace of one invocation of a program, identifying in trace order the conditional branches executed and the directions that they went. A stream can also be formed by concatenating the streams of multiple invocations of a program (possibly with different inputs). We refer to the original stream of all executions in a run of the program as the *program execution stream*. A *substream* of a stream s is a subsequence of s.

A *predictor* is a simple mechanism that predicts the next direction of a stream. A predictor may consider program characteristics (e.g. the opcode of the next branch to predict) in addition to any part of the past program execution stream.[2] The *accuracy* of a predictor is the number of correct predictions divided by the total number of predictions; accuracy measures how closely the predicted stream matches the actual stream.

A *prediction scheme* is a comprehensive mechanism that takes a program execution stream, divides it into substreams, and directs each substream to a unique predictor. Figure 1 illustrates this concept. The objective in dividing the execution stream into substreams is that each substream should be more accurately predictable by its predictor. The accuracy of the prediction scheme is the total number of correct predictions divided by the total number of predictions.

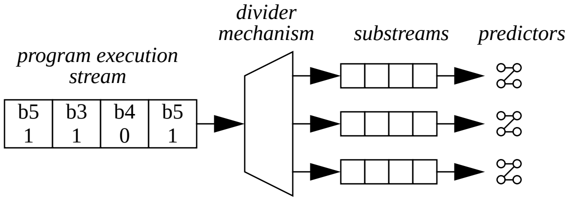

Figure 1. Framework for describing any prediction scheme. The divider mechanism splits the program execution stream into substreams, each of which is predicted by a single predictor.

2.2 Dividing Streams

Based on our formal definition of a prediction scheme, the key to building a more accurate prediction scheme involves the selection of the “right” divider and “good” predictors. In this subsection, we review several current methods for dividing a stream, and we discuss the intuition behind these approaches. Once we have described the important properties of streams that relate to the problem of branch prediction, we then discuss existing predictors and their important characteristics.

Existing schemes divide the program execution stream in a variety of interesting ways. In the simplest case, the divider is the identity function; the program execution stream is fed to a single predictor. The prediction scheme that statically predicts all branches taken [12] and the prediction scheme that uses a single 2-bit saturating up/down counter for all branches [7] are both examples of the identity divider function.

The most popular divider function in today’s microprocessors partitions the program execution stream based on the static branch identifier. This partitioning ideally forms one substream for each static branch in the program (a *per-branch substream*) as shown in Figure 2. Formally, if there are n static branches in the program, then the divider creates n substreams, one for each static branch identifier. The divider assigns the ith execution $e_i = (b_i, d_i)$ to the substream that corresponds to b_i. The intuition behind this divider is that each branch should have its own predictor because the characteristics and past history of this branch are a good predictor of its future behavior. Both the per-branch 2-bit counter scheme[3] [7] and per-branch profile-based prediction scheme [10] partition the program execution stream in this manner.

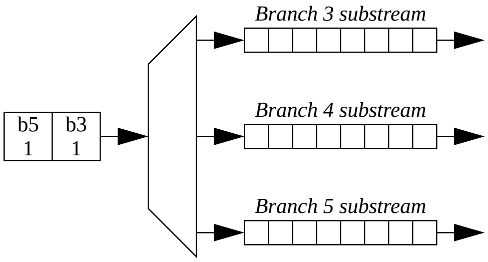

Figure 2. Subdividing the program execution stream into per-branch substreams.

More recent branch prediction schemes further subdivide the per-branch streams. The intuition behind these schemes is that finer decomposition of a per-branch stream can increase the predictability of the individual substreams. For instance, Pan, So, and Rahmeh [11] describe a scheme (which Yeh and Patt call *GAs* [14]) that partitions each per-branch stream based on the pattern of directions of the k preceding branch executions in the program execution stream, as illustrated in Figure 3. The intuition here is that sections of code deal with related information, so tests of one condition are likely to be placed near tests of related conditions. Formally, consider the ith execution in the program execution stream, $e_i = (b_i, d_i)$. The *GAs* scheme considers not just b_i, but also the directions of the k preceding executions d_{i-1}, $d_{i-2}, \ldots, d_{i-k}$. These k bits are called the *pattern history* of preceding branch executions. The k pattern bits are used to further

[1] As a point of interest, the goal of a branch prediction scheme is slightly different than the goal of the computer architect. A computer architect’s goal is to find a branch prediction scheme that provides the best performance (at possibly the smallest cost), and this may not be the scheme with the best prediction accuracy.

[2] Here, we mean past program execution stream in the most general sense so that we can consider branch executions from previous runs of the program (as are required for a profile-based predictor).

[3] In this subsection, we ignore implementation issues that keep us from obtaining a hardware predictor per static branch. These issues are addressed in Section 2.4 and Section 4. Here, we are concerned only with the ideal intent of the branch prediction scheme.

divide a per-branch stream (based on b_i) into 2^k substreams. We refer to these substreams as *per-branch global-pattern streams*.

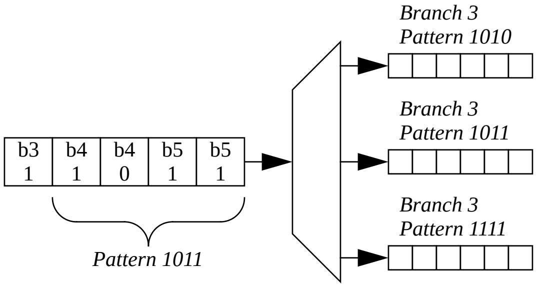

Figure 3. Subdivision in the *GAs* scheme. In addition to branch identifier, the pattern of *k* preceding branches in the program execution stream is used to further divide the branch streams, so there is one stream per pattern per branch.

As another way to subdivide per-branch streams, Yeh and Patt describe a scheme called *PAs* [15] that uses the last k branches in a per-branch stream to further partition that per-branch stream. This leads to a different set of substreams from the *GAs* scheme. Formally, consider the ith branch execution in the program execution stream, $e_i = (b_i, d_i)$ which is an execution of branch b_i. Let l_1, $l_2, ..., l_k$ be the indices of the k previous executions of branch b_i. The *PAs* scheme uses the pattern $d_{l_1}, d_{l_2}, ..., d_{l_k}$ rather than the pattern $d_{i-1}, d_{i-2}, ..., d_{i-k}$ to subdivide the per-branch stream (based on b_i) into 2^k substreams. Since the former pattern is determined only by executions of one branch, b_i, *PAs* does not exploit any inter-branch correlation; instead it is designed to exploit repeating patterns in the execution of a single branch. For example, on a loop branch that iterates a constant $c < k$ times, *PAs* approaches 100% branch prediction accuracy, because it will generate substreams consisting solely of a single branch direction (i.e. it can recognize the pattern of c taken branches that will be followed by a fall-through branch). We refer to these substreams as *per-branch branch-pattern streams*.

As a last example of how to subdivide per-branch substreams, we consider our scheme for static correlated branch prediction (*scbp*) [17]. This scheme divides both by branch and by the *path* of branches that led to the executed branch. A path differs from a pattern because it includes both the branch identifiers and the executed directions, not just the concatenation of direction bits. So our static correlated scheme uses the vector

$$(b_{i-1}, d_{i-1}), (b_{i-2}, d_{i-2}), ..., (b_{i-k}, d_{i-k})$$

to encode the path by which b_i was reached, and it uses this vector to subdivide the per-branch stream (based on b_i) into $(2 \times \text{number of static branches})^k$ substreams. We refer to these substreams as *per-branch global-path* streams.

2.3 Predictors and Streams

Under our framework, the divider presents each substream to a single predictor. Each predictor considers some combination of the program characteristics, the past branch execution stream, and its own internal state (if any) in making a branch prediction. In this subsection, we review the range of existing predictors, and we discuss the characteristics of streams that make them predictable.

Predictors can be classified into two major types: static predictors and dynamic predictors. A static predictor must fix its prediction before the program runs, while a dynamic predictor is allowed to change its prediction during program execution. Streams that are largely invariant in branch direction can be accurately predicted by a static predictor. We say that a stream is *strongly biased* if the frequency of one direction is much greater than the frequency of the other direction, and that it is *weakly biased* if the frequencies are close to equal. We refer to the more prevalent direction of the stream as the *majority* direction; the other direction is conversely the *minority*.

Researchers have investigated a variety of static program and branch characteristics to help determine the appropriate static prediction for an execution stream. For example, the simple static branch prediction scheme that always predicts branches to take [12] uses the statistical fact that branches tend to take more often than they fall through. The “backwards taken forwards not taken” (BTFNT) scheme [12] bases the static prediction on the sign of a branch’s target offset. Other schemes employ a predictor that computes predictions as a function of the opcode of the branch [7]. Finally, methods like those described by Ball and Larus [2] use sophisticated heuristics about the program structure to generate a static prediction for each branch.

Other than the static characteristics of the program and the branches in the program, researchers use a profile of the dynamic behavior of the program branches, gathered during an earlier program run, to set the static prediction of each branch. If the majority direction remains the same from the profile (training) to the testing run, then a profiled static predictor will perform well. To date, researchers have used only the overall bias of the past branch execution to set the static prediction. In our earlier paper [17], we used other characteristics of the past execution stream, but we used this information to reorganize the program so that its individual branch streams are more strongly biased.

In contrast, dynamic predictors can *adapt* to track the bias of a stream during a single execution of a program. This has the added benefit of not requiring any training or profiling before the program run. Surprisingly, there are very few designs for dynamic predictors. By far, the most popular dynamic predictor is the 2-bit saturating, up/down counter [12]. This predictor forms the basis of all of the correlated branch predictors described by McFarling [9], Pan et al. [11], and Yeh and Patt [14, 15, 16].

Lee and Smith [7] observed that the execution streams of most program branches tend to occur in long runs[4] and that an *n*-bit counter predictor can exploit this regularity. Smith [12] further observed that a 2-bit counter empirically provides an appropriate amount of *damping* (or hysteresis) to changes in stream direction. A 1-bit counter has no damping (it simply records the direction of the last branch), and 3-bit and higher counters do not appear to offer large cost/benefit advantages over 2-bit counters [12]. Damping trades off adaptability for vulnerability to short minority runs. A 2-bit counter is excellent at predicting streams with long minority runs, and it is damped enough to ignore minority runs of length 1. This allows loop branches, for instance, to incur just one mispredict per loop, instead of two mispredicts (one on loop exit and one on loop reentry).

The distribution of minority run lengths in a stream strongly relates to the effectiveness of today’s dynamic predictors. Streams with long runs of one direction followed by long runs of the other direc-

[4] A run is a substring of the stream that consists entirely of one direction, and is bounded on either side by executions that go in the opposite direction (or the beginning or end of the stream). Note that a proper substring of a run is not itself a run.

tion can be accurately predicted by a dynamic predictor but not by a static predictor. However, a large distribution of short minority runs can cause a dynamic predictor to exhibit worse accuracy than a static predictor because the dynamic predictor adapts too slowly to the changes in the runs.

One other interesting property is the *frequency of recurrent patterns* in a stream. A *pattern* is a non-empty string $w \in \{0, 1\}^*$. A *recurrent* pattern is a substring that occurs multiple times in a stream. Unlike bias and distribution of runs, which are typically used to predict streams that have been divided, this property is exploited by some dividers (e.g. the *PAs* scheme [15]).

2.4 Implementation Details

To this point, our explanations of existing branch prediction schemes focused on the ideal implementation of a scheme. For example, the explanation above describes a per-branch dynamic prediction scheme based on 2-bit counters as able to assign each per-branch stream to a unique 2-bit counter. In actual implementations of per-branch 2-bit counter schemes, this is believed to be impractical. Implementors usually solve this by using just the j least significant bits of the branch address as an index into a table of 2^j counters. This means that, if two conditional branches have the same j lowest bits, their branch streams will be intermingled and sent to a single 2-bit predictor. We call this effect *aliasing*, as the original intent of the 2-bit counter scheme was to provide a single predictor per static branch.

Issues in aliasing have led researchers to develop different branch prediction schemes that we would classify as based on the same ideal branch prediction model. For instance, the *GAs* scheme [14] and McFarling's *gshare* scheme [9] both ideally divide the program execution stream into per-branch global-history substreams, and both use a 2-bit counter as the base predictor. The *gshare* scheme requires fewer 2-bit counters for fixed values of j and k because it exclusive-ors, rather than concatenates, the k bits of pattern history with the j bits of branch address when indexing into the limited table of 2-bit counters. This gives a requirement of $2^{max(k, j)}$ counters, instead of 2^{k+j} counters. Section 4.2 shows that aliasing potentially limits the effectiveness of the ideal divider by intermingling streams that we would ideally like separated.

Static branch prediction schemes that can fix a prediction to each static branch in the program obviously do not suffer from these effects of aliasing. However, static schemes have their own potential limitations due to implementation details. For example, the implementation of our algorithm for static correlated branch prediction [17] does not distinguish between paths that cross a procedure call or return boundary. In other words, they effectively truncate the vector that is used to divide the stream in the cases where a path crosses a procedure boundary. This truncation *merges* streams that would be separated by a more sophisticated divider. We distinguish aliasing from merging: aliasing combines streams from different static branches, while merging combines streams from one static branch.

2.5 Hybrid Approaches

Recent work in branch prediction by McFarling [9] and Chang [4] has proposed hybrid branch prediction schemes which group together multiple basic prediction schemes. The hybrid schemes, either statically or dynamically, select the basic prediction scheme that performs best on a stream. The model in this section can easily be extended to cover hybrid schemes; however, this paper focuses on the power in our model to analyze and improve the individual prediction schemes. Benefits to basic schemes will of course improve the hybrid schemes that include them.

The framework illustrates two distinct avenues of research for improving the accuracy of a branch prediction scheme: one could attempt to improve the sophistication of the ideal model; or one could attempt to remove limitations imposed by current implementation details. The next two sections give examples of each of these approaches, for both static and dynamic branch prediction schemes.

3 Why Static Correlated Prediction Works

The framework described in the previous section gives us a set of terms that can be used to describe, compare, and contrast the behavior of branch prediction schemes. In this section, we examine a simple application of this framework to a pair of similar prediction schemes: per-branch static profile prediction and our static correlated profile prediction [17]. Per-branch static profiling has been shown to work well in a number of studies [5, 10]. In this section, we show how our code transformation exploits branch correlation to increase branch bias.

As noted in Section 2, bias is key to static branch prediction. Figure 4 plots the distribution of taken branch frequency averaged over all benchmarks and data sets. Table 1 presents a summary of our benchmarks and experimental methodology. The "Self History" bars in Figure 4 show that, even for executables produced by today's compilers, most of the dynamic branches are strongly biased. This U-shaped distribution is what makes per-branch static branch prediction effective.

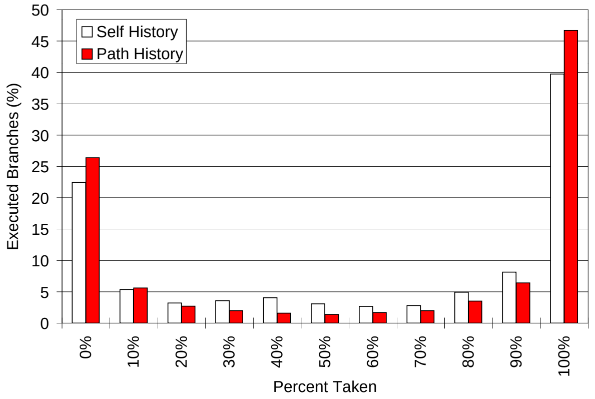

Figure 4. Histogram of branch bias, weighted by execution frequency. This plot averages over all benchmarks, giving equal weight to each data set run. The "Self-History" bars indicate the branch bias in the original executables. The "Path-History" bars indicate the branch bias of executables after transformation to exploit branch correlation with a history depth of 12. The bias values represent the midpoint of a range, e.g. the "10%" bars capture bias values between 5% and 15%. Although this graph averages over all benchmarks and data sets, the trend of increased bias occurred in each individual run. These results train and test on the same dataset.

The effect of exploiting branch correlation is to divide each per-branch stream into several separate streams, discriminating by correlation paths in addition to the static branch identifier. The "Path History" bars in Figure 4 show the distribution of taken branch frequency after our transformation to exploit branch correlation [17]. Compared to the "Self History" bars, the "Path History" bars

	Benchmark and Data Set Descriptions	Total Branches Executed	Static Branches Touched
awk [awk]: pattern-directed scanning/processing, GNU ver. 2.15.5			
a	extensive test of awk's capabilities	2.54M	1393
b	simple scanning and printing	0.62M	835
c	generate max array of 3 arrays	4.99M	968
compress [comp]: compression using adaptive Lempel-Ziv, SPECint92			
in	SPECint92 reference input	11.4M	277
jarg	jargon dictionary (1MB of ASCII)	13.1M	280
ps	15-page postscript paper	2.0M	268
diff [diff]: differential file comparator, GNU version 2.6			
a	two C files with 3 diffs	0.43M	646
b	two latex files with many diffs	0.27M	704
xsim	xsim sources with many diffs	0.72M	711
eqntott [eqn]: boolean equation to truth table conversion, SPECint92			
fx2fp	8-bit fix to floating point encoder	29.4M	533
tbra	MIPS R2000 taken branch decode	19.3M	528
espresso [esp]: boolean minimization, SPECint92			
bca	SPECint92 reference input	73.9M	1722
cps	SPECint92 reference input	83.1M	1845
ti	SPECint92 reference input	87.4M	1899
grep [grep]: pattern searching program, GNU version 2.0			
a	search for a constant string (2 hits)	0.07M	611
khad	complex regular exp. (100% hits)	0.14M	966
re3	search for a regular exp. (21 hits)	0.33M	878
sc [sc]: spreadsheet program, SPECint92			
l1	SPECint92 short input	23.5M	1614
lb1	SPECint92 reference input	179.3M	1642
lb3	SPECint92 reference input	44.4M	1538
xlisp [li]: lisp interpreter, SPECint92			
newt	square root via Newton's method	0.11M	550
q4	4 queens problem	0.41M	605
q7	7 queens problem	32.4M	605

Table 1: Benchmark and data set descriptions. The results in this paper were derived from trace-driven simulations. We collected the traces using ATOM v1.1 [13]. We compiled the SPECint92 benchmarks using cc version 2.0.0 and the optimization level specified in the SPEC makefiles. The additional benchmarks were compiled using gcc v2.6.0 (-O3). All of the experiments were performed on a DEC 3000/400 running OSF/1 version 2.0.

exhibit a larger percentage of strongly biased branches. Over 70% of dynamic branches now occur in streams that are highly predictable. In other words, the more finely subdivided per-branch global-path substreams are more predictable than the coarsely divided per-branch substreams. As we show further in Section 4.3, correlation shifts the distribution of streams and their dynamic branches toward stronger bias.

For static profile prediction to be practical, the static predictions chosen must be valid across invocations of the program. If the majority direction of a stream differs between the profiled (training) data set and the running (testing) data set, then a static predictor will suffer. Fisher and Freudenberger [5] examined a number of different benchmarks and data sets under static profile prediction, and determined that good prediction could be achieved even while training and testing on different data sets. Our experiences so far with various static correlated branch prediction schemes show similar results, although we have not yet done a comprehensive study. An exhaustive treatment of data variance is outside the scope of this paper.

4 Comparing Correlated Schemes

This section uses the framework to tackle the much harder problem of comparing static and dynamic correlated branch prediction schemes. Superficially, one can compare the prediction accuracy reported by the designers of static and dynamic correlated schemes, but this numerical comparison is unenlightening. For example, in an earlier paper [17], we found that our static correlated branch prediction scheme did not achieve as high a prediction accuracy as the published dynamic correlated schemes. We cannot conclude from these results, however, that the dynamic schemes are necessarily better than static schemes since these schemes differ in more than their base predictors.

Aside from the fundamental differences between a static and a dynamic predictor, our framework suggests that there are three major implementation differences in the divider function: the use of path versus pattern history, the aliasing of multiple (possibly unrelated) branches to the same predictor, and the lack of correlation information across procedure call boundaries. Path history is used in our software prediction scheme, while all current correlated branch prediction schemes based on a hardware table of predictors use pattern history. The aliasing of per-branch streams occurs in hardware-based branch prediction schemes but not in the profiled branch prediction schemes. Finally, our software scheme for correlated branch prediction, unlike the hardware-based schemes, does not exploit correlation across procedure call and return boundaries. Each of these implementation differences can be seen as a limitation that keeps the implemented divider from behaving as precisely as an ideal mathematical divider. Sections 4.1 through 4.3 show, by focusing on *gshare* [9], *GAs* [14], and our static correlated branch prediction scheme (*scbp*) [17], that the removal of these implementation differences can improve the prediction accuracy of correlated branch prediction schemes.

Once we have equalized the divider function, an interesting question to ask is how much benefit one gets from the use of a dynamic predictor in a correlated scheme. Section 4.4 presents one answer to this question by comparing the prediction accuracy of a correlated branch prediction scheme that uses either static predictors or 2-bit dynamic predictors. This experiment uses a theoretical divider function that is uninhibited by the implementation effects of Sections 4.1 through 4.3. Through this experiment, we can begin to understand the true need for dynamic predictors. By understanding where a dynamic predictor is beneficial, we expect to understand how to develop new code transformations to improve the static prediction schemes.

4.1 Paths versus Patterns

As explained in Section 2.2, an implementation difference exists between the divider used in our *scbp* scheme and the one used in an ideal *GAs* scheme. Our *scbp* scheme is based on a per-branch global-path divider that uses a history consisting of a path vector (path history), while *GAs* is based on a per-branch global-pattern divider that uses a history consisting of the pattern of the directions of the most-recent branch executions (pattern history). Path history should provide better correlation information than pattern history, because path history is a superset of pattern history. Path information includes the branches by which the current branch was reached, not just the pattern of directions that they went to reach the current branch. For example, path information can differentiate between two streams in which two branches take to reach a third block. Figure 5 shows a simple example where that path history achieves a better prediction accuracy than the pattern history.

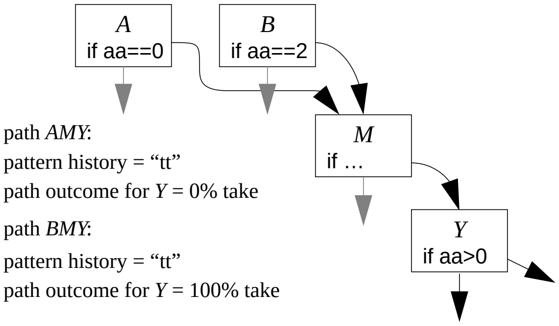

Figure 5. Example illustrating the benefit of path history over pattern history. Path *AMY* and *BMY* are indistinguishable using pattern history, but distinguishable using path history.

To quantify the benefits of path information, we simulated a *scbp* scheme that used only pattern information, and we compared the prediction accuracy of this scheme against the prediction accuracy of the per-branch profile scheme and a *scbp* scheme using path history. These results are summarized in Figure 6. For all benchmarks, the pattern-based *scbp* scheme shows significant improvements over per-branch static profile prediction. There is a small improvement in mispredict rate when path history is used, ranging from negligible in *diff.a* to removing 14% of mispredicted branches in *espresso.bca*. There is a measurable benefit to exploiting path history instead of pattern history, but the majority of advantage is gained just from pattern information.

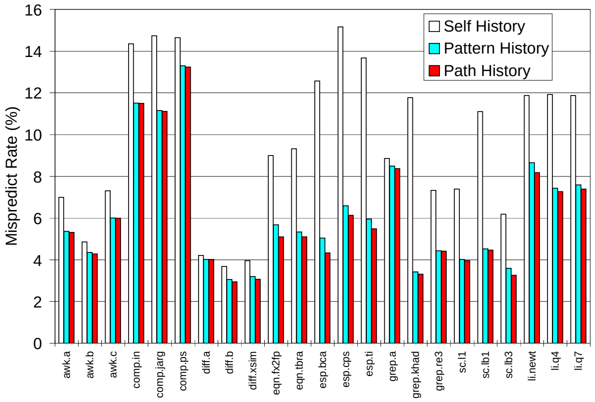

Figure 6. Mispredict rates of per-branch static branch prediction, static correlated branch prediction using pattern history, and static correlated branch prediction using path history. Both correlated schemes use a history depth of $k = 12$. All schemes train and test on the same data set.

From this result in static branch prediction, we would like to generalize to say that all branch prediction schemes could be improved by incorporating path history. But to do this, we need to isolate out the other factors that affect prediction accuracy so that these other factors do not overwhelm the gains due to using path instead of pattern history (and thus cloud the results). We will return to this issue at the end of the next section.

4.2 Aliasing

One important factor that differentiates a static profiled branch prediction scheme from a dynamic branch prediction scheme stems from the fact that dynamic branch prediction schemes map unevenly distributed information like branch address and pattern history into indexed, regular hardware structures (i.e. a table of predictors). In Section 2.4, we defined the term aliasing to describe the situation when, due to implementation limitations, a divider forces streams from different branches to map to the same predictor. A static profiled branch prediction scheme does not suffer from aliasing effects since each branch encodes its predictor function.

Aliasing does not directly imply penalties to prediction accuracy. If two branches with different majority direction alias to the same counter, but one executes 1,000 times followed by the other executing 1,000 times, the loss due to aliasing is negligible. However, if the two branches alternate in trace order, then aliasing may cause significant misprediction. To relate aliasing back to prediction accuracy, we define three kinds of aliasing: If an aliased counter predicts an execution correctly while the corresponding per-stream counter predicts it incorrectly, we call that execution an instance of *constructive* aliasing since the aliasing improves prediction accuracy. Conversely, if the aliased counter mispredicts while a per-stream counter would have predicted correctly, we call that an instance of *destructive* aliasing since the aliasing reduces prediction accuracy. Finally, if the aliased counter predicts an execution correctly (incorrectly) and the corresponding per-stream counter predicts correctly (incorrectly) too, we call that execution an instance of *harmless* aliasing since the aliasing does not change prediction accuracy.

Our intuition is that aliasing is generally bad for prediction accuracy. Since a branch prediction table is a kind of cache, aliasing is analogous to conflict misses in a cache. Instead of suffering conflict misses though, aliased predictors suffer from muddled predictions. As in a cache, increasing the size of the prediction table can help to reduce conflicts (and increase prediction accuracy), as is shown in most of the dynamic branch prediction literature [10, 11, 14, 15, 16]. Chang et al. [4] show benefits to separating out strongly biased branches from weakly biased branches, noting that using static prediction on the strongly biased branches reduces contention (aliasing) in the table of 2-bit counters. Unlike cache conflict misses though, aliasing can be constructive or harmless in addition to destructive. It is important then to understand how aliasing affects the design space of dynamic branch prediction schemes. In particular, we investigate the question of how often aliasing happens in dynamic correlated branch prediction schemes and how this aliasing affects the prediction accuracy.

To see how common aliasing is, we instrumented our hardware simulations to count the number of static branches that map to each 2-bit counter. We examined the usage patterns of the per-branch 2-bit counter scheme, the *GAs* scheme, and the *gshare* scheme for a fixed size table of 4096 2-bit counters. As an example of the type of results we saw, Figure 7 plots the distribution of the number of branches that alias to each counter for each scheme under the *awk.a* benchmark. From Figure 7, one can see that aliasing happens infrequently in the standard 2-bit counter scheme. This makes sense since all benchmarks touch noticeably fewer than 4096 static branches (see Table 1). Aliasing increases in *GAs*, and reaches very high levels under *gshare* since these schemes produce significantly more than 4096 substreams from their dividers. Under *gshare*, running the *espresso* and *sc* benchmarks, aliasing happens so often that in some cases no counters have fewer than three different branches aliased to them. Figure 8 shows detail on constructive versus destructive aliasing in *awk.a* under *gshare*;

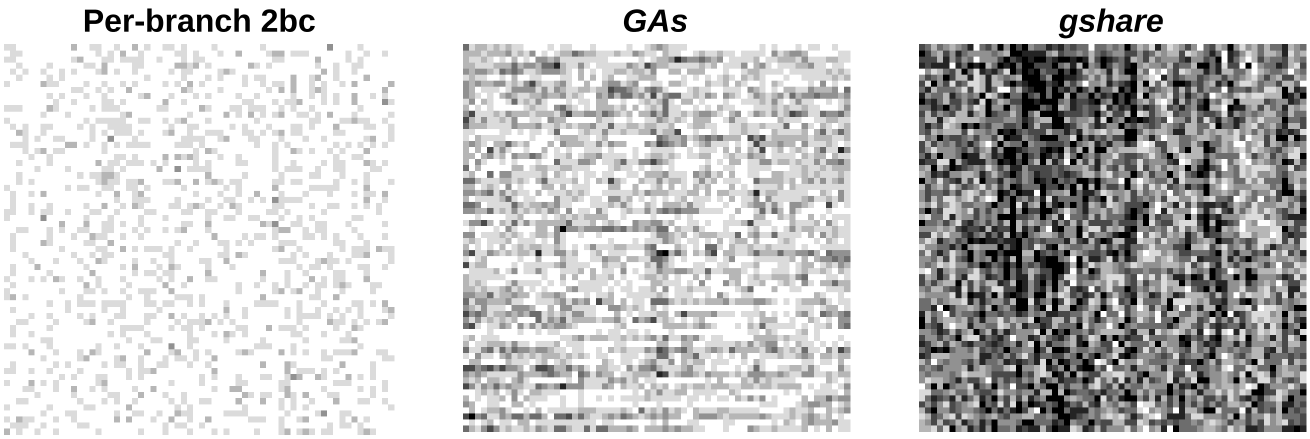

Figure 7. Aliasing in a 4096 counter table for *awk.a* under the per-branch 2-bit counter (12 bits of branch address), *GAs* (6 bits of branch address concatenated with 6 bits of branch history), and *gshare* (12 bits of branch address exclusive-ored with 12 bits of branch history) schemes. White squares represent unused counters; black squares represent counters with seven or more aliased streams.

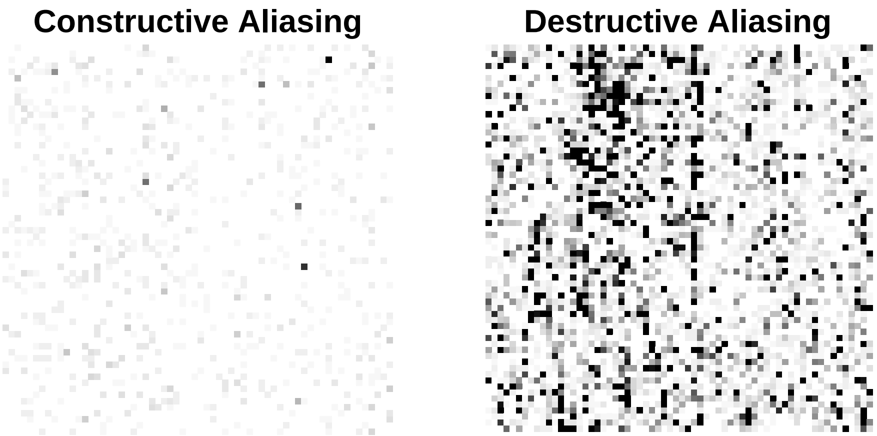

Figure 8. Constructive and destructive aliasing in *awk.a* under the *gshare* scheme. In the left graph, gray scale indicates constructive aliasing, with black representing the maximum attained value of 32. In the right graph, gray scale indicates destructive aliasing, with black representing a difference of 32 or more correct incorrect predictions due to destructive aliasing.

constructive aliasing is both rare and much smaller in magnitude that destructive aliasing.

The more aggressive correlated branch prediction schemes produced more substreams under the assumption that this aggressive subdividing would produce more predictable streams. As shown by the prediction accuracies of the schemes in Figure 9, this decision can lead (though it does not always) to a design with a worse prediction accuracy.

If we remove aliasing from the experiment in Figure 9, an unaliased per-branch global-pattern branch prediction scheme should achieve a higher branch accuracy than either *GAs* or *gshare*. To verify this, we modified our hardware simulation so that each per-branch, global pattern stream was assigned its own counter, then recorded how many executions led to destructive and constructive aliasing. Figure 10 presents the results of this experiment for all benchmarks. Clearly, aliasing happens regularly, and it happens destructively. There is often a significant improvement in the prediction accuracy for removing aliasing effects. Better dynamic prediction schemes are theoretically possible if those schemes can exploit the same pattern and address information as *gshare* without suffering destructive aliasing effects.

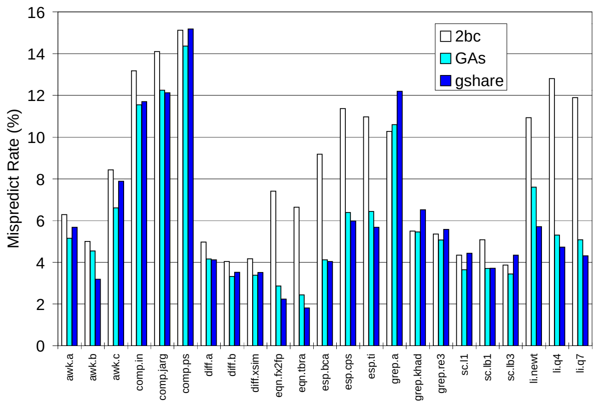

Figure 9. Mispredict rates for the per-branch 2-bit counter (12 bits of branch address), *GAs* (6 bits of branch address concatenated with 6 bits of branch history), and *gshare* (12 bits of branch address exclusive-ored with 12 bits of branch history) schemes; each using a table of 4096 2-bit counters.

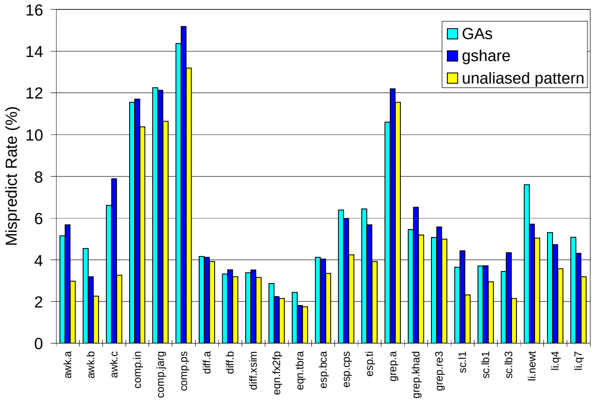

Figure 10. Comparing the mispredict rates of correlated branch prediction schemes that contain aliasing and a branch prediction scheme with a true per-branch, global-pattern divider. All of the schemes use 2-bit counters and a history depth of 12 branches. The *GAs* and *gshare* schemes use 4096 counters; the pattern history scheme uses one counter per stream.

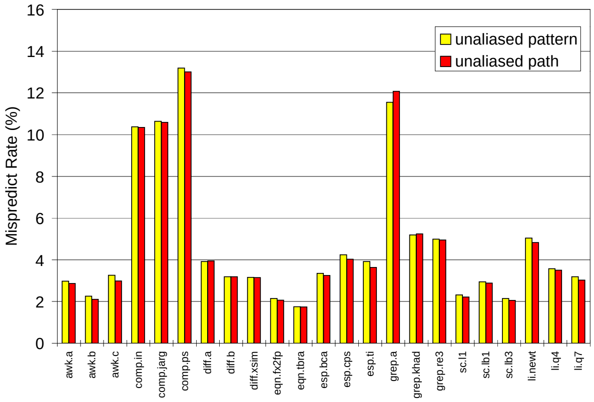

Figure 11. Mispredict rates of schemes using 2-bit counters, a history depth of 12 branches, and a divider without aliasing effects (i.e. one 2-bit counter per stream). “Unaliased pattern” and “unaliased path” depict pattern and path history dividers, respectively.

The *grep.a* bar in Figure 10 is the exception to the trend: the *GAs* scheme shows higher prediction accuracy than the unaliased pattern divider. From Table 1, one can see that *grep.a* executes very few branches. The worse prediction accuracy seems to be a result of the start-up costs of training a 2-bit counter to match a stream’s bias. Since the unaliased divider produces more streams than the *GAs* divider, the unaliased divider pays a larger training cost. This larger training cost is significant on short benchmark runs; it might be reduced if schemes that use dynamic predictors could merge streams with similar initial values.

Once we have removed the effects due to aliasing, we are in position to evaluate the benefit of path history over pattern history in dynamic schemes. We extended our simulator to use an unaliased path-history divider with dynamic predictors. The mispredict rates for this path-based predictor are presented in Figure 11. Using paths improves the mispredict rate on the majority of our benchmarks. As in the *grep.a* case from Figure 10, a few of the short benchmarks exhibit worse prediction accuracy under path rather than pattern history due to start-up training costs. Since the magnitude of benefits from a path-based divider are sometimes small, designers must take care that improvements in prediction accuracy due to path history are not swamped by aliasing penalties introduced as part of the modified scheme.

4.3 Cross-Procedure Correlation

So far, the differences we have explored between static and dynamic correlated branch prediction schemes only hurt the prediction accuracy of the dynamic schemes. Yet the overall prediction accuracy of the dynamic schemes is often better. To explain this disparity, we collected statistics of cases where the hardware prediction schemes achieved better per-branch accuracy and then examined the kinds of correlation that occurred. The vast majority of such cases turned out to be cross-procedure correlation: branches that occurred just after a procedure entry or just after a procedure return.

Our *scbp* scheme [17] cannot preserve correlation information across procedure calls. The scheme encodes correlation history into the program counter by duplicating basic blocks. A particular copy of a basic block implies some set of previous execution paths. The problem is that the value of the program counter is effectively reset on a procedure call or return, eliminating correlation information across procedure calls. In terms of the framework, this means that the static scheme’s divider is not always capable of using all of the components of the path history vector; the portion of the paths in the vector before a call boundary are merged into a single path.[5] In the extreme, a branch just after a call or return will have no history information available. In contrast, hardware schemes ignore procedure call boundaries, since they record conditional branch directions in additional hardware state.

Some examples of cross-procedure correlation are obvious once they are pointed out:

- The *eqntott* benchmark in the SPECint92 suite uses a quicksort routine to sort bit vectors. A variety of different generic bit-vector comparison functions are passed to *qst()*. Each of these compare routines branches to different return points corresponding to equal, less than, or greater than return values; *qst()* then immediately branches based on the return values. The branch that tests the return value is completely determined by that the branch that set the return value.
- The garbage collector’s *mark()* function in the *xlisp* benchmark calls *livecar()* to determine when to follow a node’s left sublist. The switch statement inside *livecar()* returns the constant FALSE in many cases; this FALSE return value is then immediately checked by *mark()*.

These kinds of cross-procedure correlation led us to ask how accurately a static prediction scheme could predict if it were possible to preserve path information across procedure boundaries. We modified our trace and simulation environment to record paths across procedure call boundaries, and to simulate the prediction accuracy that would be obtained if a code transformation could preserve all desired correlation information across calls. The prediction accuracy results where we trained and tested on the same data set are summarized in Figure 12. In these results, *compress* shows very little benefit from cross-procedure correlation, but this makes sense because *compress* is implemented as one large loop in a sin-

[5] Merging is not always harmful. As part of our *scbp* algorithm, we perform per-branch analysis that intentionally merges path streams with the same majority direction. There is no penalty for this kind of merging when using a static predictor; *scbp* exploits this harmless merging to reduce overall code expansion.

gle procedure. In some benchmarks, like *eqntott* and *awk*, more than half of the mispredictions were removed. Other benchmarks showed more modest improvements.

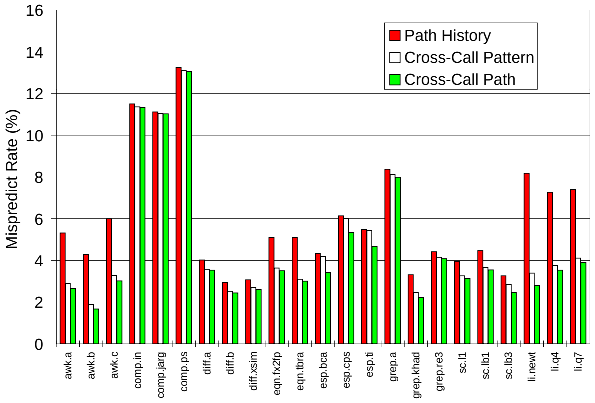

Figure 12. Mispredict rates of scbp using path history (same as the third series in Figure 6) and simulated mispredict rates for cross-call pattern and cross-call path history dividers with static predictors. These results use a history depth of 12 and train and test on the same dataset.

The results in Figure 12 are not necessarily what we would expect from actual implementations of cross-call correlated static schemes, because they train and test on the same data set. This gives best possible static prediction accuracy, rather than what would occur if different training and testing data sets were used. However, these results show that using cross-call correlation we can achieve better static prediction accuracy than was previously believed possible.

Having discussed the implementation differences in dividers, we can now revisit the effect of correlation on bias that we began to explore in Section 3. Figure 13 extends the results shown in Figure 4, adding a new series of columns that shows the bias of streams generated by an unaliased, cross-call, path divider. The improved divider further steepens the U-shaped distribution of bias.

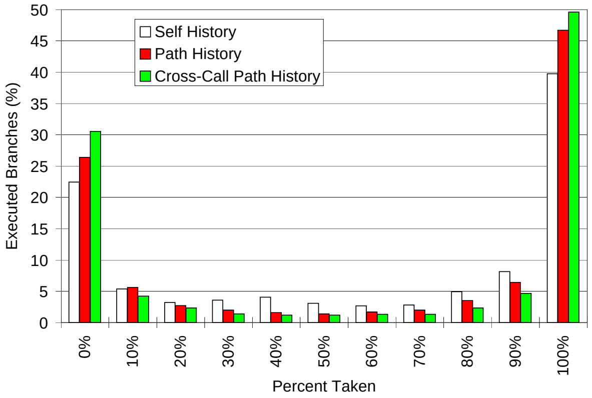

Figure 13. Histogram of branch bias, extending Figure 4. The "Cross-Call Path History" bars show the bias of streams generated by an unaliased, cross-call, path divider. These results use a history depth of 12 and train and test on the same dataset.

Exploiting Cross-Procedure Correlation Statically

We have not yet found a simple code transformation that can generally preserve correlation across calls. However, a number of techniques may be useful: selective inlining [6], template formation, and multiple entry points [1].

Fisher and Freudenberger point out that sophisticated ILP compilers already expect to perform aggressive inlining [5]. Inlining all procedures is impractical, since it is exponential in the depth and degree of the program call graph. But since a small number of procedures make up the majority of program execution cycles [3], it is also likely that a small number of procedures are the best candidates for inlining to extract correlation. The *livecar()* routine in *xlisp* is a great candidate for inlining: it is called in just one place, and it is defined to be local to the *xldmem.c* source file. After inlining *livecar()*, an optimizing compiler could fold the logically correlated branches into a single branch, decreasing the number of static and dynamic branches in the program, and reducing cycle count.

The *eqntott* case, above, is more complicated. Branches in *qst()* correlate into the generic comparison routine that is passed as a function pointer. It is not possible for a compiler to simply inline the comparison routine. However, it would be possible for a compiler or programmer to build different versions of *qst()*, as if *qst()* were a C++ style template function that was instantiated for each comparison function. Since C functions are not first-class types, we could perform function variable propagation analysis to determine all of the possible comparison functions. In fact, in *eqntott*, the comparison functions are constants passed in each call of *qst()*, so we could curry (specialize) the *qst()* call at compile time into a call to the appropriate version of *qst()*.

We can preserve some correlation state across procedure calls by making multiple copies of procedure entry points, one for each relevant past execution history. This allows us to better predict callee branches that correlate back to the caller, but does not help us with the more common case of caller branches that correlate into some utility function.

4.4 Adaptability

The fundamental difference between the static and dynamic correlated schemes is the predictors they use. Dynamic predictors can adapt to track streams during an invocation of the program, while static predictors cannot. This raises the question of whether some streams require the adaptivity of a dynamic predictor to achieve good prediction accuracy. To examine this question, we used the same approach of the previous subsections: subtract out the differences, and see what results. Once again, we used a divider with a path history of length 12 and no aliasing effects. We also made the divider ignore procedure call boundaries like the divider in a hardware implementation.

We classified streams from the divider as "Static Better", "Equal", or "Dynamic Better", depending on whether a static predictor, neither predictor, or a 2-bit counter best predicted the stream. Figure 14 shows the distribution of streams for each benchmark and data set. The "Static Better" bars shows the percentage of streams which were better predicted by a perfectly trained static predictor; the "Static +1" bars show the percentage of streams where the static predictor predicted correctly just one more time than the 2-bit counter. The large number of "Static +1" streams have a majority fall-through direction, and since our simulation initializes 2-bit counters to predict weakly taken, the 2-bit counter incur a mispredict on the first execution in those strongly-biased streams. The

"Dynamic +1" streams are very rare, and there is a small but visible number of "Dynamic Better" streams.

The absolute number of "Dynamic Better" streams is less than 1,000 for all benchmarks except *espresso*. This suggests that there are ways to build better hybrid static/dynamic prediction schemes than that proposed by Chang et al [4]. Their scheme assigns all branches with low bias to dynamic predictors. If we can assign only the rare adaptive streams (which might be aliased together or aliased with statically predictable streams in Chang et al.'s scheme) to their own predictors, while using static predictors for the remaining branches, we should be able to achieve even better prediction accuracy with fewer counters than previous hybrid schemes.

Despite the small percentage of "Dynamic Better" streams in Figure 14, those streams are an important component of overall prediction accuracy. Figure 15 gives details about the *comp.in* bar from Figure 14, plotting the difference in correct predictions. Even though the number of "Dynamic Better" streams is small, the "Dynamic Better" tail is significantly larger than the "Static Better" tail. The integral over the tails gives the differences in correct predictions between schemes using only static predictors and schemes using only dynamic predictors. The "Static Predictor" and "2-bit Counter" bars of Figure 16 compare the mispredict rates of such schemes. Even though we exaggerated the benefit with a static predictor by assuming perfect training, Figure 16 shows that the number of dynamic branches that occur in streams with long runs of the minority branch direction is significant—ignoring them will affect prediction accuracy. However, since the number of static streams requiring an adaptive predictor is very small, the possibility exists for a compiler to selectively apply techniques like predication [8] to these few streams. The vast majority of streams can be handled using simple static branch prediction techniques.

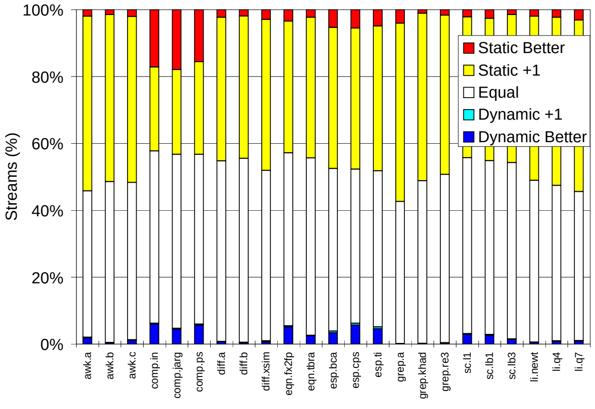

Figure 14. Distribution of streams under an unaliased, cross-call, path divider, depending on whether the streams were predicted better by a perfectly trained static divider or by a 2-bit counter. The "+1" categories contain streams where one of the predictor types correctly predicted just one more execution than the other predictor type. The "Better" categories contains streams where one predictor correctly predicts greater than one more execution than the other predictor.

Hybrid prediction schemes can mix static and dynamic predictors in one scheme. The "Best Predictor per Stream" bars show the mispredict rate as if the best predictor (2-bit counter or static) for a stream was assigned on a per-stream basis, instead of assigning all streams to a single kind of divider. These bars show that schemes using a mix of static and dynamic predictors can achieve very high prediction accuracies.

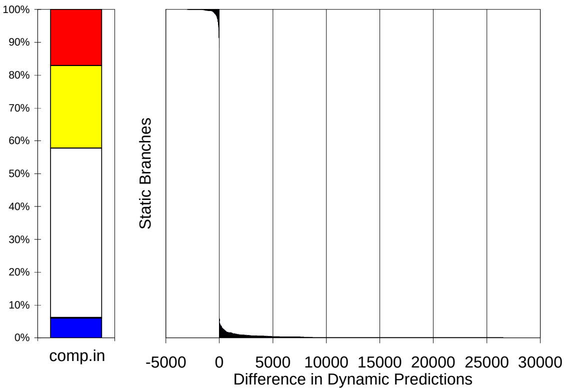

Figure 15. Detailed information on the *comp.in* bar from Figure 14. The horizontal axis shows the difference in correct predictions by the static and 2-bit counter predictors. Positive values correspond to the 2-bit counter predicting more accurately; negative values correspond to the static predictor predicting more accurately.

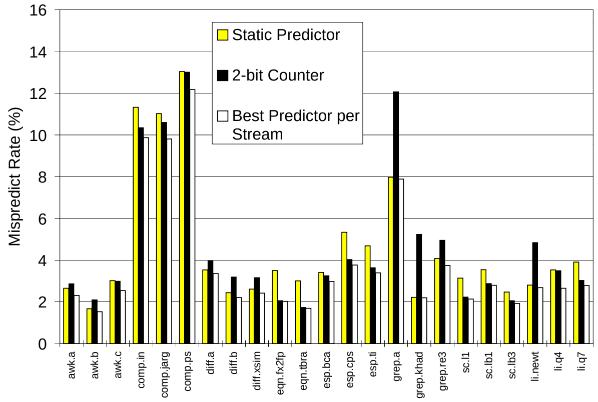

Figure 16. Mispredict rates under an unaliased, cross-call, path divider, comparing assigning all streams to perfectly trained static predictors ("Static Predictor"), all streams to 2-bit counters ("2-bit Counter"), and the best predictor for an entire stream to that stream ("Best Predictor per Stream").

In the long term, adaptability may be the only thing that separates dynamic and static schemes, since static schemes can take cross-call correlation into account, and dynamic schemes can exploit path history and may be able to reduce aliasing problems. Correlation provides a useful tool to reduce the amount of adaptivity (both in dynamic branches and stream distribution) in a program, but no current methods allow us to completely eliminate the need for adaptivity. Hybrid schemes that use the techniques explored in this paper may be able to find efficient ways to separate and handle adaptive streams.

5 Conclusions and Future Work

Before one can build better branch prediction schemes, one must understand how and why existing schemes work. We presented a framework for analyzing and categorizing branch prediction schemes. The framework partitions schemes into two major parts: a *divider* and *predictors*. Dividers attempt to partition the program execution stream into substreams that are individually more predictable than the original stream. All known branch prediction schemes fit into this framework. The framework provided the motivation for all of the studies in this paper, allowing us to practically and systematically analyze the differences between schemes.

Profiled per-branch static branch prediction works because programs have a large percentage of branches that are strongly biased. Correlation changes the distribution of streams to increase the percentage of branches that are strongly biased. Correlation reduces the diversity of branch streams, making profiled static correlated branch prediction more accurate than profiled per-branch static branch prediction.

Under our framework, state-of-the-art static and dynamic prediction schemes differ in four major qualities: use of pattern versus path history, aliasing effects, ability to exploit cross-procedure correlation, and adaptivity.

- Path history is slightly better than pattern history in exploiting branch correlation.
- Correlated dynamic branch prediction schemes utilize more 2-bit counters in their tables, but simultaneously increase the amount of aliasing. Removing the effect of aliasing increases prediction accuracy, suggesting that work should be done to limit aliasing in dynamic branch prediction schemes.
- Cross procedure correlation limits the accuracy of static branch prediction schemes. We showed some large potential benefits to cross-procedure correlation in static schemes. We are pursuing several practical techniques that allow static schemes to exploit cross procedure correlation.
- The percentage of adaptive streams is small, but that the dynamic branches executed in adaptive streams are significant.

We have not reached the limits of existing basic branch prediction schemes. We have demonstrated potential for increased prediction accuracy in each of the areas above. Dynamic branch prediction schemes will benefit from methods to control aliasing and to exploit path history. Static branch prediction schemes will benefit from techniques that exploit cross-procedure correlation and reduce the need for adaptive predictors.

6 Acknowledgments

We thank Hewlett-Packard and Digital Equipment Corporation for their generous donation of several HP 9000 Series 700 and DECstation 3000 Series workstations on which we ran our tracing and analysis tools. We would also like to thank Brad Calder for his useful comments on early versions of this paper. Cliff Young is funded by a Graduate Fellowship from the Office of Naval Research. Michael D. Smith is supported by a National Science Foundation Young Investigator award, grant number CCR-9457779.

An expanded set of the results for this paper can be found in [18].

7 References

[1] V. Bala, personal communication, Oct. 1994.

[2] T. Ball and J. Larus, “Branch Prediction for Free,” *Proc. ACM SIGPLAN 1993 Conf. on Prog. Lang. Design and Implementation*, Jun. 1993.

[3] J. Bentley, *Programming Pearls*, Addison-Wesley, 1986.

[4] P. Chang, E. Hao, T. Yeh, and Y. Patt, “Branch Classification: a New Mechanism for Improving Branch Predictor Performance,” in *Proc. 27th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, November 1994.

[5] J. Fisher and S. Freudenberger, “Predicting Conditional Branch Directions From Previous Runs of a Program,” *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[6] W. Hwu and P. Chang, “Inlining Function Expansion for Compiling C Programs,” *Proc. ACM SIGPLAN 1989 Conf. on Prog. Lang. Design and Implementation*, Jun. 1989.

[7] J. Lee and A. Smith, “Branch Prediction Strategies and Branch Target Buffer Design,” *Computer*, 17(1), Jan. 1984.

[8] S. Mahlke, et al., “Characterizing the Impact of Predicated Execution on Branch Prediction,” *Proc. 27th Annual Intl. Symp. on Microarchitecture*, Nov. 1994.

[9] S. McFarling, “Combining Branch Predictors,” *WRL Technical Note TN-36*, June 1993.

[10] S. McFarling and J. Hennessy, “Reducing the Cost of Branches,” *Proc. of 13th Annual Intl. Symp. on Computer Architecture*, Jun. 1986.

[11] S. Pan, K. So, and J. Rahmeh, “Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,” *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[12] J. Smith, “A Study of Branch Prediction Strategies,” *Proc. 8th Annual Intl. Symp. on Computer Architecture*, Jun. 1981.

[13] A. Srivastava and A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools,” *Proc. SIGPLAN ‘94 Conf. on Prog. Lang. Design and Implementation*, Jun. 1994.

[14] T. Yeh and Y. Patt, “Two-Level Adaptive Branch Prediction,” *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[15] T. Yeh and Y. Patt, “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,” *Proc. 20th Annual Intl. Symp. on Computer Architecture*, May 1993.

[16] T. Yeh, “Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors,” Computer Science and Engineering Div. Tech. Report CSE-TR-182-93, Univ. of Michigan, Ann Arbor, MI, Oct. 1993.

[17] C. Young and M. Smith, “Improving the Accuracy of Static Branch Prediction Using Branch Correlation,” *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, October 1994.

[18] C. Young, N. Gloy, and M. Smith, “A Comparative Analysis of Schemes for Correlated Branch Prediction,” Technical Report 06-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, Mar. 1995.