Predicting Conditional Branches With Fusion-Based Hybrid Predictors*

Gabriel H. Loh[1] Dana S. Henry[2,1]
gabriel.loh@yale.edu dana.henry@yale.edu

Yale University
[1]Dept. of Computer Science
[2]Dept. of Electrical Engineering
New Haven, CT, USA

Abstract

Researchers have studied hybrid branch predictors that leverage the strengths of multiple stand-alone predictors. The common theme among the proposed techniques is a selection *mechanism that chooses a prediction from among several component predictors. We make the observation that singling out one particular component predictor ignores the information of the non-selected components. We propose* Branch Prediction Fusion*, originally inspired by work in the machine learning field, which combines or* fuses *the information from all of the components to arrive at a final prediction. Our 32KB predictor achieves the same overall prediction accuracy as the 188KB versions of the previous best performing predictors (the Multi-Hybrid and the global-local perceptron).*

1. Introduction

Superscalar processors capable of executing multiple instructions per cycle need accurate branch prediction to provide a steady stream of useful instructions. The cost of a branch misprediction can mean many cycles of wasted work, and many cycles before the functional units resume executing useful instructions along the correct flow of control. The extremely deeply pipelined microarchitectures of future superscalar processors further exacerbate the problem [28].

As the processor clock cycle shrinks to increase processor throughput, there is less time for the branch prediction logic to arrive at a prediction. Large branch predictors provide more accurate branch predictions, but the corresponding slowdown in clock speed can result in lower overall performance. Jiménez and Lin show how large branch predictors can be integrated into an aggressively clocked processor pipeline [14]. In particular, the *overriding predictor* allows a processor with a large, multi-cycle predictor to achieve ILP levels that are close to a processor with an idealized, single-cycle version of the same predictor.

This paper presents a new technique for designing large hybrid branch predictors. Past research in hybrid branch prediction has focused on the problem of how to select a correct prediction from a pool of several component predictors. This approach only makes use of the selected predictor; the information of the other non-selected components remains unutilized. Our technique uses the combination, or *fusion*, of all of the component predictors to improve branch prediction rates.

This paper is organized as follows. Section 2 describes the idea of *Branch Prediction Fusion* at a high-level and explains the original motivation for this work. Section 3 presents a brief overview of previous related work. In Section 4, we detail our simulation environment and explain our predictor optimization methodology. Section 5 describes our fusion-based hybrid branch prediction scheme, the *Combined Output Lookup Table* (COLT) predictor. In Section 6, we provide a classification of correctly predicted branches to determine the importance of the different branch predictor components, and we further explore the design space for the COLT predictor. We draw our final conclusions in Section 7 and also discuss some directions for future research.

2. The Idea

In the pursuit of more accurate branch prediction algorithms, past studies have researched several ways of combining different branch predictors to form hybrid branch predictors [4, 7, 8, 9, 22]. All of the proposed hybrid predictors consist of two or more component branch predictors, and a *meta-predictor* that selects one of the components. We make the observation that selection-based hybrid predictors ignore the information conveyed by the predictions of the non-selected components.

In this paper, we introduce the concept of designing hybrid branch predictors by *branch prediction fusion*. Branch prediction fusion covers any hybrid branch predictor that combines or fuses the predictions of multiple component predictors to form a final prediction. Figure 1 illustrates the difference between branch prediction selection and branch prediction fusion. Instead of having the meta-predictor

*This research was supported by NSF Grant MIP-9702281.

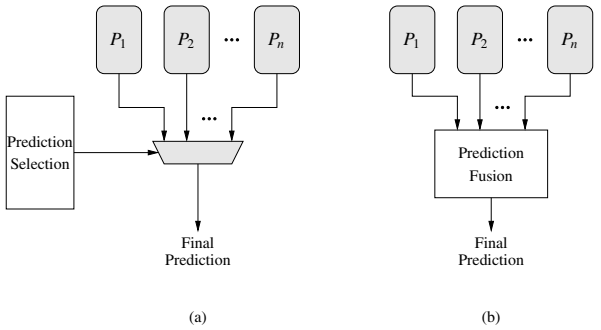

Figure 1: (a) Prediction-selection selects one-of-n predictors. (b) Prediction-fusion incorporates information from all n predictors.

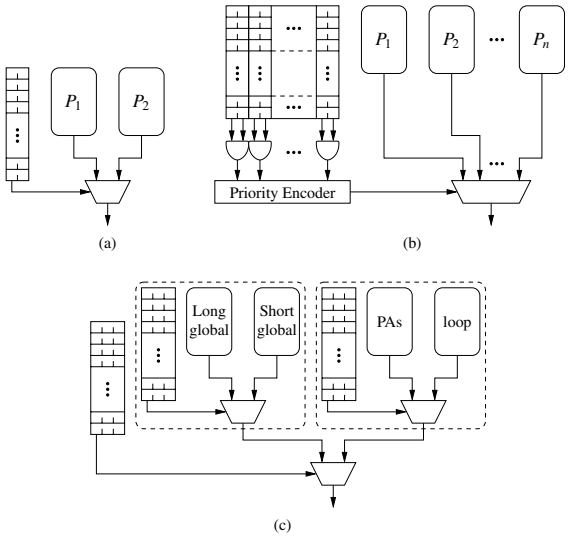

Figure 2: Selection-based hybrid predictors: (a) the Tournament predictor, (b) the Multi-Hybrid from [8], (c) the Multi-Hybrid from [7] (we use Quad-Hybrid in this paper to distinguish between the two).

make a selection among the predictors (Figure 1a), the meta-predictor makes the final prediction itself (Figure 1b).

Our initial fusion-based predictor was inspired by the Weighted Majority Algorithm from the machine learning field [20]. Instead of selecting one particular component to make the final prediction, our Weighted Majority Branch Predictor (WMBP) takes a vote among all components. The vote of each component is weighted depending on the past performance of the predictor. An important fact that we observed is that the combined weights of multiple components frequently override the component with the largest weight to yield a correct final prediction. For example, we found that for the `gcc` benchmark, branches where the weighted majority overrule the best predictor (the predictor with the highest corresponding weight) occur over 80% more frequently than when the weighted majority is incorrect while the best predictor is correct. Due to the complexity of implementing the Weighted Majority Algorithm in hardware, we do not explore the WMBP any further in this paper. Instead, we propose a simpler, and more accurate, fusion-based hybrid predictor called the *Combined Output Lookup Table* (COLT) predictor.

Branch prediction fusion is a new technique for combining predictors. By incorporating the predictions of a variety of predictors, we get the benefits of considering variable branch history lengths [30], dynamically changing history lengths [16], simultaneously using both global and local branch history [26], as well as taking advantage of the best stand-alone (non-hybrid) prediction algorithms available [15]. Another result of this work is that we show that hybrid branch prediction can still provide better prediction rates, despite the continual development of better stand-alone prediction algorithms [15, 18, 23]. This will hopefully encourage researchers to continue the search for better individual prediction algorithms, as well as to develop better hybridization techniques.

3. Related Work

There has been an abundance of research in the area of dynamic branch prediction. In this section, we only review the work most relevant to branch prediction fusion.

The first hybrid branch prediction scheme was the *tournament* predictor proposed by McFarling [22]. The tournament scheme is limited to selecting from only two component predictors. Figure 2a illustrates the tournament selection mechanism. The branch address indexes into a table of saturating two-bit counters. The most significant bit of the counter selects from one of the two component predictors. Chang et al extended the tournament selection table to a two-level table that incorporates branch history to further improve prediction rates [3].

The first hybrid branch predictor to support an arbitrary number of component predictors was Evers et al's Multi-Hybrid predictor [8]. As shown in Figure 2b, the Multi-Hybrid maintains a table of vectors of two-bit counters. Each counter corresponds to one of the component predictors. The selection mechanism chooses the component whose counter has the maximum value of three. In the case that more than one counter equals three, a predetermined precedence ordering breaks the tie. The counter update rules guarantee that at least one counter equals three.

Evers' dissertation provides detailed information and analysis of the behavior of branches [7]. From these results, Evers designed a refined version of the Multi-Hybrid predictor which consists of four component predictors (to differentiate between the two Multi-Hybrids, we will refer to this version as the Quad-Hybrid since all configu-

rations use four component predictors). Figure 2c shows how the Quad-Hybrid prediction scheme is a tournament prediction scheme where each component is also a tournament predictor (highlighted by the dashed boxes). The first sub-tournament consists of two *gshare* predictors that utilize different branch history lengths. A selective update strategy is used on the long history gshare. The second sub-tournament combines a local history predictor (PAs) [32] with a *loop* predictor [2, 8]. A final table of counters selects between the global-history predictors and the local-history/loop predictors. The meta-prediction tables incorporate global branch history similar to the two-level tournament predictor [3]. The counters used in the selection tables are three bits wide instead of two. It was found that three-bit counters yield more stable selections and are less sensitive to fluctuations in the behavior of the branch predictors.

Besides these purely dynamic hybrid branch prediction schemes, compiler and profile assisted approaches have also been proposed. Branch Classification uses profile information to statically assign branches as always taken, always not-taken, or dynamically predicted by a tournament-styled predictor [4]. Static hybrid predictors have also been proposed. The static hybrid predictor is a variant of the Multi-Hybrid (from [8]) where the dynamic selection mechanism has been replaced by a branch hint that chooses which component to use. Static prediction schemes are sometimes undesirable because modifications to the ISA are necessary to convey the static decisions, programs must be profiled and recompiled to receive any benefit and therefore it does not help existing applications, and poor performance may result if the data sets used during the profiling phase are not representative of actual runtime behavior. For these reasons, we only consider purely dynamic approaches in this paper.

Our branch prediction fusion approach combines the outputs of several different kinds of branch predictors. There are some similarities to Skadron et al's *alloyed* history predictors [26]. In the course of categorizing branch mispredictions, Skadron et al discovered that some branches require both global and local history to accurately predict. The alloyed history branch predictor combines multiple types of branch history, whereas our branch prediction fusion approach combines multiple types of branch predictor outputs. The global/local perceptron predictor [13] is basically an alloyed history version of the basic perceptron predictor [15].

The gskewed predictor [23] is related to our Weighted Majority hybrid predictor in that both make use of majority functions to combine multiple inputs. The gskewed predictor uses an unweighted majority to reduce the effects of interference in the pattern history tables whereas the Weighted Majority Branch Predictor uses a weighted majority to combine the predictions of different components.

To optimize our hybrid predictor configurations, we used a genetic search algorithm [11]. Emer et al used a genetic programming approach to optimize branch predictors and indirect branch predictors [6]. They developed a language to generically describe branch predictors, and then used genetic programming to optimize over tree representations of the branch predictor language. In our study, we are interested in selecting components for a hybrid predictor which is easily encoded in a fixed length bit string. This allows us to use a simpler genetic algorithm.

4. Methodology

In this section, we describe our methodology for simulating and evaluating branch prediction algorithms. We also detail our approach for optimizing the components in hybrid branch predictors.

4.1. Simulation Environment

We collected traces of conditional branches from the integer benchmarks of the SPEC2000 suite [31]. Using the functional in-order simulator from the SimpleScalar toolset for the Alpha instruction set [1], we collected 500 million branches from each benchmark using the *train* input sets. We also skipped over the initial 100 million conditional branches to avoid start-up effects. The binaries were compiled on an Alpha 21264 using `cc` with full optimizations. The reported misprediction rates are arithmetic means across all benchmarks, except in the cases where we examine the benchmarks individually.

For our ILP study, we modified the MASE simulator[1] to support the predictors analyzed in this paper [17]. We also simulated an *overriding* predictor configuration to support large branch predictors in a very fast clock speed processor [14]. We fast forward past the initial start-up code, and then we simulate 100 million instructions because the MASE out-of-order processor simulator is much slower than our trace-fed branch predictor simulator. The binaries and input files are identical to those used for the misprediction rate simulations.

4.2. Genetic Search for Hybrid Predictors

Choosing components for a hybrid branch predictor is an enormous search problem. Indeed, even optimizing a single type of branch predictor may require large amounts of computation if the number of parameters is large. We performed all of our tuning and optimization using traces of the first ten million conditional branches from the SPEC *test* inputs to avoid over-training of the predictors.

We first individually optimized the component branch predictors. The component predictors considered are bimodal [27], gshare [22], Bi-Mode [18], enhanced gskewed with partial update [23], YAGS [5], PAs [32], pshare and

[1]We used a pre-release version of the MASE simulator. The MASE simulator will be part of SimpleScalar version 4.0 [1].

gshare		
size	PHT entries	history length
1KB	4096	7
2KB	8192	8
4KB	16384	9
8KB	32768	15
16KB	65536	16
32KB	131072	17
64KB	262144	18

PAs			
size	BHT entries	PHT entries	history length
0.88KB	512	2048	6
2KB	2048	2048	6
3.75KB	2048	8192	7
8KB	4096	16384	8
16KB	8192	32768	8
32KB	16384	65536	8
52KB	16384	131072	10

Bi-Mode		
size	PHT entries	history length
0.38KB	512	6
0.75KB	1024	8
1.5KB	2048	10
3KB	4096	11
6KB	8192	13
12KB	16384	14
24KB	32768	15
48KB	65536	16

Enhanced pskewed			
size	BHT entries	PHT entries	history length
1.88KB	512	2048	6
3.75KB	1024	4096	6
7.75KB	2048	8192	7
16.5KB	4096	16384	9
33KB	8192	32768	9

Enhanced gskewed		
size	PHT entries	history length
0.38KB	512	7
0.75KB	1024	8
1.5KB	2048	9
3KB	4096	12
6KB	8192	13
12KB	16384	14
24KB	32768	15
48KB	65536	16

bimodal	
size	num 2-bit counters
1KB	4096
2KB	8192
4KB	16384
8KB	32768
16KB	65536
32KB	131072
64KB	262144

YAGS		
size	PHT entries	history length
0.63KB	512	8
1.25KB	1024	9
2.5KB	2048	10
5KB	4096	11
10KB	8192	12
20KB	16384	13
40KB	32768	14

loop		
size	num loop counters	counter width
0.75KB	1024	6
1KB	1024	8
1.5KB	2048	6
2KB	2048	8

Alloyed (global/local) perceptron
2KB, 4KB, 8KB, 18KB, 30KB, 53KB configurations in [13].

Table 1: The sizes and parameters of the 59 component predictors considered for inclusion in our fusion-based hybrid predictors.

Name	Hardware Budget	Components	VMT Counters	Counter Width c	History Length h
α	16KB	alpct(8KB) Enh.gskewed(3KB) gshare(2KB)	2048	4	8
β	32KB	alpct(8KB) gshare(8KB) gshare(4KB) PAs(3.75KB)	8192	4	7
γ	64KB	alpct(30KB) gshare(16KB) YAGS(5KB) Enh.pskewed(3.75KB)	16384	4	10
δ	128KB	alpct(30KB) alpct(18KB) gshare(16KB) Enh.gskewed(6KB) YAGS(10KB) PAs(32KB)	16384	4	7
η	256KB	alpct(53KB) alpct(8KB) gshare(64KB) Bi-Mode(48KB) Enh.gskewed(24KB) PAs(32KB)	32768	4	4

Table 2: The COLT configurations chosen by our genetic algorithm. *alpct* stands for alloyed perceptron.

pskewed [7], a local history Bi-Mode, the loop predictor [2, 8], and the alloyed history (global/local) perceptron predictor [13]. For all components except for the alloyed perceptron and the loop predictor, we performed an exhaustive search of the parameter space. For the alloyed perceptron, we used the optimal configurations reported in [13].

After optimizing the component branch predictors, we used these predictors as candidates for inclusion in our hybrid predictor configurations. We considered the 59 different configurations listed in Table 1 with sizes ranging from 1KB to 64KB. There are 2^{59} possible ways the components can be chosen. The search space is even larger when the parameters of the meta-predictor are factored in as well. To optimize our hybrid predictors over such a huge search space, we used a genetic algorithm approach [11].

The encoding of our search problem as a genetic algorithm is straightforward. Each hybrid predictor configuration is encoded as a bit string. The first 59 bits correspond to the potential components, where a 1 denotes the inclusion of the corresponding component. The binary encoded parameters of the meta-predictor are concatenated after the component inclusion bits.

For each execution of the genetic algorithm, we use a fixed hardware budget such that any configuration that exceeds this limit is not considered. The genetic algorithm executes as follows. An initial population of *individuals* is generated at random; invalid individuals (e.g. ones that exceed the hardware budget) are removed and replaced with another random configuration. Each individual of the population is then simulated and we use the average branch prediction rate as the *fitness function*. From this information, we select the best configurations to produce the following generation, and then the process is repeated.

Our rules for generating a new generation of predictors is as follows. We select the best k configurations as potential parents. Out of these k configurations, two parents are selected at random with the constraint that they are not the same individual. A *crossover* point is selected at random to create a new configuration. The new configuration consists of all of the bits from one parent up to the crossover point. All remaining bits are inherited from the other parent. In addition to the crossover operation, a variety of *mutations* may also occur at random. These mutations include randomly flipping bits in the configuration encoding, and randomly incrementing or decrementing the meta-predictor parameter fields.

For our experiments, the population of each generation consisted of 32 configurations, and we ran the search for a total of 20 generations. We chose a value of $k = 10$ to roughly correspond to the top-third of each generation. We used hardware budgets of 16KB to 256KB in factor of two increments. The hardware budget imposes a fairly irregular boundary to the space of allowable configurations. To prevent a population from getting stuck in a local extrema for too long, we used fairly high mutation probabilities. The probability of a random bit flip was 0.2 (independently, per bit), and the probability of incrementing/decrementing a meta-predictor parameter field was 0.5. All of the constants for the genetic algorithm were empirically chosen.

5. The COLT Hybrid Predictor

We found that the Weighted Majority Branch Predictor (WMBP) achieves better prediction rates than a selection-based Multi-Hybrid with the same components. Unfortunately, the WMBP is complex and slow, especially if it must be serialized after the individual component lookups. Computing a weighted majority requires looking up the weights, multiplication (albeit only by 0 or 1), and adding all of the weights together. The Weighted Majority algorithm learns monotone Boolean functions from the components' predictions to the final prediction. A monotone Boolean function is still a Boolean function, and so we use a lookup table instead. A lookup table is much simpler to implement in hardware, and it can also handle non-monotone mappings if they exist. Our proposed lookup table based prediction fusion algorithm is called the *Combined Output Lookup Table* (COLT).

5.1. Predictor Description

The COLT consists of the n component predictors, $P_1, P_2, ..., P_n$, and a collection of mapping tables that maps the predictor outputs to a final overall prediction. This is illustrated in Figure 3. Each entry of the *Vector of Mapping Tables* (VMT) is a 2^n entry mapping table. The entries of the mapping tables are c-bit saturating counters. Similar to the 2-level tournament and the Quad-Hybrid meta-predictors, we also include branch history to correlate mappings to past branch outcomes.

The COLT fusion predictor has three parameters. The first is c, the number of bits per counter for each mapping table entry. The second parameter is a, the number of branch address bits to use when indexing the VMT. The last parameter is h, the number of branch history bits to use when indexing the VMT. These parameters are all illustrated in Figure 3. The total size of the VMT is thus $c \cdot 2^{a+h+n}$ bits.

The lookup phase of the COLT predictor proceeds in two steps. The first step performs the individual lookups on each of the component predictors. Simultaneously, the branch address and branch history select one of the 2^{a+h} mapping tables from the VMT (shown as a bold, dashed arrow in Figure 3). The second step uses the individual predictions to choose one of the 2^n counters of the selected mapping table (the bold, solid arrow). The most significant bit of the selected counter determines the final prediction.

The update phase is similar to most other branch predictors. If the actual branch outcome was taken, then we incre-

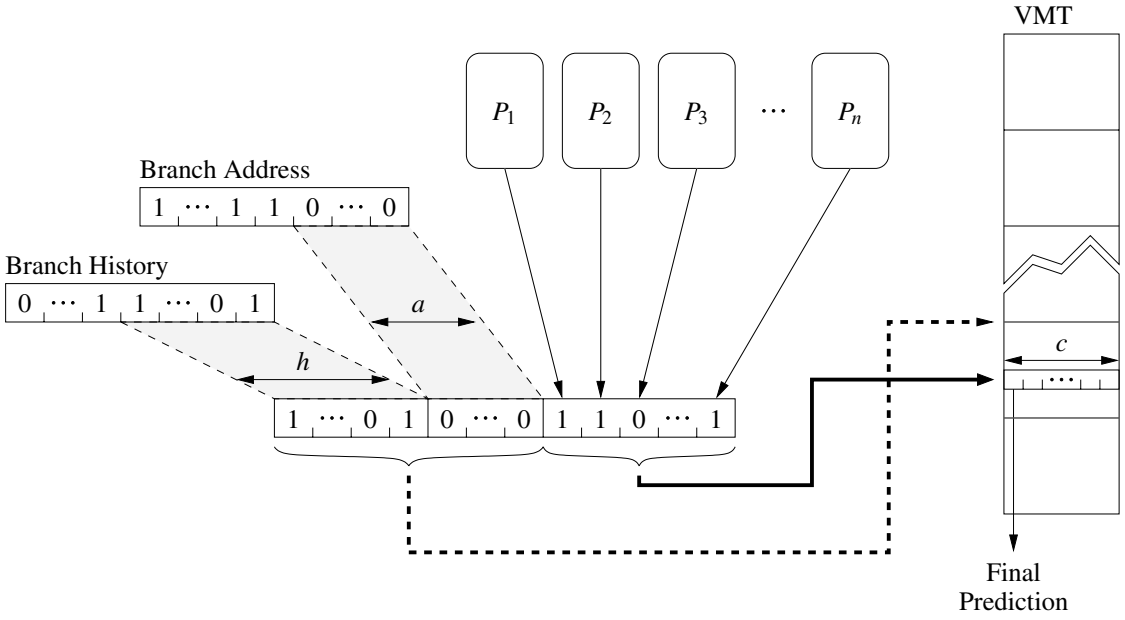

Figure 3: The *Combined Output Lookup Table* hybrid predictor incorporates the outputs of *all* component predictors to arrive at an overall final prediction. The Vector of Mapping Tables (VMT) learns mappings from predictor outputs to the overall branch outcome.

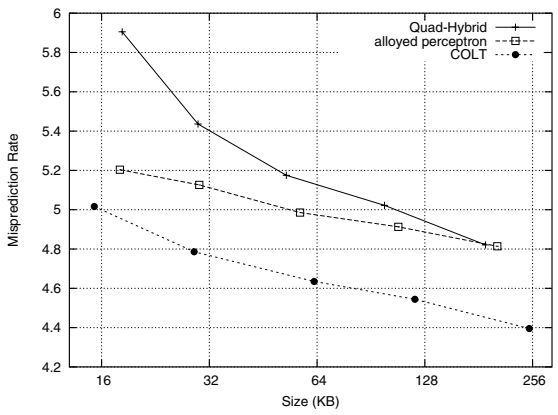

Figure 4: The average SPECint2000 branch misprediction rates of the Quad-Hybrid, alloyed perceptron, and COLT predictors for different hardware budgets.

ment the selected counter, up to a maximum value of $2^c - 1$. If the actual outcome was not-taken, then we decrement the counter, down to a minimum value of zero. This allows the COLT to learn arbitrary patterns, such as a branch that is not-taken only when the exclusive-or of P_1 and P_2 is true.

5.2. Predictor Accuracy

Using our genetic search, we optimized the set of components and the values of the COLT's parameters for different hardware budgets. Although genetic algorithms tend to be an efficient means of searching a large design space, we have no guarantee that the results are the best possible. The results of the genetic algorithm are listed in Table 2. The column labeled "VMT Counters" include the total number of counters across all 2^{a+h} mapping tables. As the hardware budget increases, we find that the genetic algorithm chooses configurations with more components. The common components among configurations of all hardware budgets are an alloyed perceptron predictor and a short history-length global predictor. In Section 6, we categorize which components are useful to the COLT for making predictions and show how this corresponds to the configurations chosen by the genetic algorithm.

We simulated the different COLT predictor configurations, and show the results in Figure 4. We also simulated the Quad-Hybrid predictor, which is the best purely dynamic selection-based hybrid predictor. Furthermore, we included the performance of the alloyed perceptron which, as far as we know, is the best published purely dynamic branch predictor. We did not attempt to further optimize either the Quad-Hybrid or the alloyed perceptron since a significant amount of work has already gone into optimizing these predictors in their original studies. Although this may affect our results a little, we feel that the comparison is still fair because we evaluate all of the predictors on a data set that differs from those used in their respective tuning phases. At 16KB, the COLT predictor achieves conditional branch misprediction rates that are over 15% lower than the Quad-Hybrid; at 32KB, the COLT is over 12% better. As the hardware budget is increased, the prediction accuracies

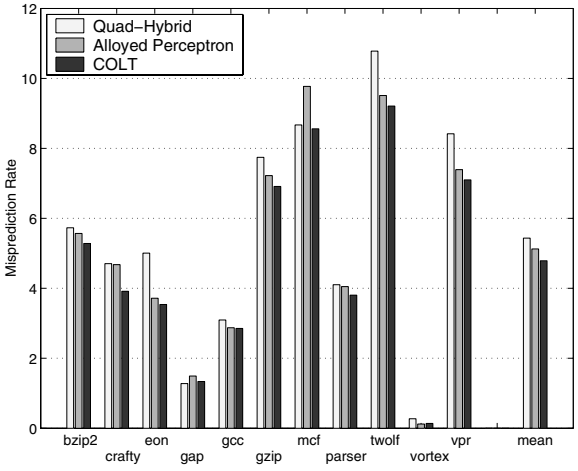

Figure 5: The misprediction rates for the SPECint2000 benchmarks for 32KB branch predictors.

of the Quad-Hybrid and alloyed perceptron tend to converge, while the COLT predictor consistently stays ahead of the pack. Another interpretation of the results is that a 32KB COLT predictor performs better than the 188KB Quad-Hybrid and alloyed perceptron predictors. Based on these results, we claim that our COLT predictors are the most accurate purely dynamic predictors published to date.

The COLT predictor also performs very well on each individual benchmark. Figure 5 shows the branch misprediction rates for each of the SPEC2000 integer benchmarks simulated for predictors at a 32KB budget. For some benchmarks, the perceptron predictor performs hardly better than the Quad-Hybrid predictor. For `gap` and `mcf`, the perceptron actually performs worse than the Quad-Hybrid. The COLT predictor consistently outperforms the Quad-Hybrid and perceptron predictors across all benchmarks, with the exception of `gap`, where the Quad-Hybrid predictor achieves a marginally lower misprediction rate than the COLT predictor.

5.3. Predictor Implementation

The COLT predictor along with the component branch predictors are too large and slow to access in a single clock cycle, especially with the aggressive clock speeds of current and future processors [10]. Jiménez and Lin show how to pipeline and integrate large branch predictors into superscalar processors [14]. Even so, the COLT predictor should not take too many cycles to perform a prediction. As presented in Section 5.1, the individual component lookups are serialized with the mapping table lookup.

The additional delay of choosing one counter from the indexed mapping table must be considered. Since there are n components, there are 2^n possible combinations of predictions. To select from these 2^n entries in the mapping table, we need an additional $O(n)$ gate delays. We can improve this delay by taking advantage of the fact that the component predictors from Table 2 are of different sizes and therefore have different lookup latencies. Before any of the component predictors have returned their predictions, we have 2^n possible entries in the mapping table to consider. Each time a component returns a prediction, the number of possible entries is reduced by one half. If we order the components such that the last input to the selection logic comes from the slowest component, then the additional delay caused by the mapping table lookup is only a single multiplexer delay.

The impact of the branch prediction lookup latency can be further reduced by using other microarchitectural techniques, such as Jiménez and Lin's *overriding predictor* [14]. Each cycle, the fetch engine of a superscalar processor uses the prediction from the small 1-cycle predictor to choose the next fetch target. The processor also starts a lookup on the slower but more accurate predictor. Several cycles later, the large predictor returns its prediction. If the prediction agrees with the original fast prediction, then no further actions are taken. If the predictions are different, then instructions fetched in the mean time are squashed and the fetch is restarted in the direction specified by the large predictor. If the large predictor is correct, this may save the many cycles of a branch misprediction recovery.

We wanted to know how much the additional latency of a large predictor would impact the overall performance of a processor. We assume an aggressive clock speed that only allows about eight levels of logic (gates) per cycle. This limits the small 1-cycle predictor to a table size of 256 entries. We chose a Smith-style table of saturating counters since a gshare predictor would require an extra gate delay to hash the branch address and the global history. Allowing eight levels of logic per cycle, our 32KB COLT predictor has a lookup latency of four clock cycles.

We simulated a six-issue superscalar processor loosely based on the Pentium 4 processor [10] (same caches and same instruction latencies), although our simulator is based on the Alpha instruction set architecture. Similar to the Pentium 4, our branch misprediction pipeline has a minimum latency of 20 cycles.

We simulated four different configurations. The first is a baseline processor with only the 256-entry bimodal predictor and no overriding predictors. The second configuration uses the bimodal predictor with a 30KB overriding alloyed perceptron predictor. The third configuration uses the bimodal predictor with a 32KB overriding COLT predictor. Both overriding predictors have a 4-cycle lookup latency. The last configuration is an ideal 32KB COLT predictor that requires only a single cycle to return its prediction. This helps to illustrate the impact of the branch predictor lookup

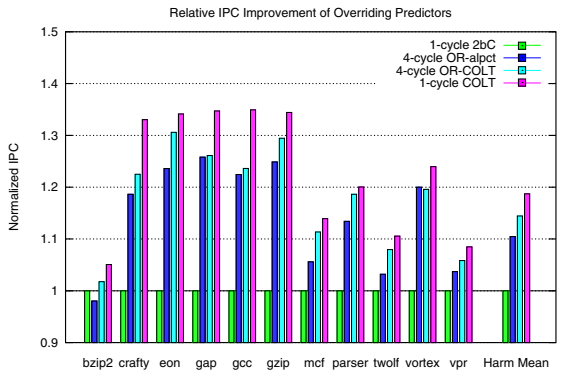

Figure 6: The IPC performance for each benchmark compared to a baseline processor with a single-cycle 256 entry bimodal predictor.

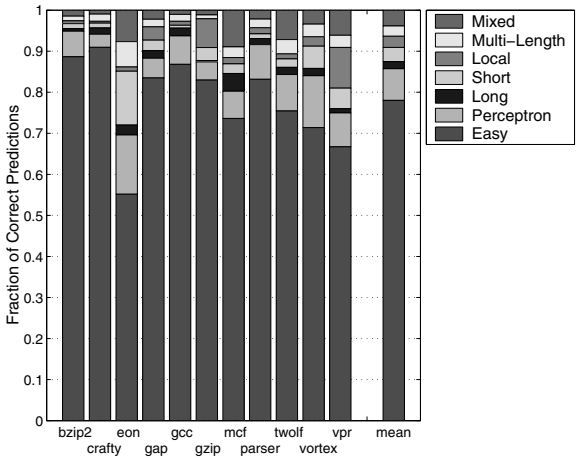

Figure 7: A breakdown of all correct predictions made by the 32KB COLT predictor.

latency.

Figure 6 shows the relative IPC rates for each benchmark as well as the harmonic mean over all benchmarks. All rates are normalized to the bimodal-only processor configuration. The overall performance improvement of the overriding COLT (OR-COLT) over the overriding alloyed perceptron (OR-alpct) varies by benchmark. For example, the OR-COLT shows improvements of 4.6-7.0% over the OR-alpct configuration for `eon`, `gzip` and `parser`. On the other hand, there is almost no difference for `gap` and `gcc`, and even a slight performance drop for `vortex`. The reason why these results do not perfectly correlate with our earlier misprediction rate results is that the branch misprediction penalty varies from branch to branch. This emphasizes the limitations of relying too heavily on branch misprediction rates alone.

6. Performance Analysis

In this section, we first present some data about the mappings learned by the mapping tables in the VMT. The characteristics of the mappings provide evidence that the COLT predictor does incorporate multiple types of information when making some of its predictions. This information also helps to explain the choice of components from our genetic algorithm. We then explore the design space for the COLT predictor, examining the performance trade-offs of varying the parameters of the COLT predictor.

6.1. Explaining the Choice of Components

Branch prediction fusion allows a hybrid predictor to combine the information from several types of branch predictors. In this section, we answer the question of whether the COLT predictor really makes use of more than one component predictor for each branch. Even though the COLT predictor can conceptually make use of all of the information, it could be the case that in practice the mapping tables simply learn mappings that ignore all but one of the components.

To answer this question, we take a closer look at the mappings stored in the VMT. For each successfully predicted branch, we look at neighboring entries in the mapping table. For example, if the four component predictors P_0, P_1, P_2 and P_3 of the 32KB COLT predict 1, 0, 0, 1, respectively (1 denotes a taken prediction), then the COLT lookup uses the counter from entry 1001 of the mapping table. We then compare entries 1001 with 0001. If the two entries yield different predictions, then it means that the prediction from P_0 is needed to determine the final prediction. On the other hand, if the two entries yield the same prediction, then we interpret this as meaning that P_0 does not play a role in this particular mapping. Note that this approach may incorrectly categorize some branches when the counter in a neighboring entry has not completed training.

For each simulated branch, we examine the n neighboring mapping table entries where we invert the outcome of one of the n component predictors. For each branch, we then determine which components played a role in successfully determining the final prediction. Figure 7 shows how the correctly predicted branches were determined by the different components for the 32KB COLT predictor. The figure provides data for each individual benchmark and the arithmetic mean across all benchmarks.

The 32KB configuration uses four components, which gives rise to 16 possible combinations of predictors. The majority of correct prediction are classified as *easy* predictions. These are cases where all neighboring entries in the mapping table provide the correct prediction. The *Perceptron*, *Long*, *Short* and *Local* classifications correspond to branches where only the alloyed perceptron, the long history global predictor, the short history global predictor, or

the local history predictor, respectively, played a role in determining the final prediction. These are the branches that a good selection mechanism should be able to correctly predict. The next group is what we call *Multi-Length* predictions, or predictions that use the input from global history predictors with different global history lengths. We treat the alloyed perceptron as a global predictor in this context because it uses a very long global history register. The final group is called *Mixed*, which includes branches that require both global history (possibly of multiple lengths) and local history.

We first make some observations about the average-case classification results. The first point is that the *Easy* branches comprise the majority of all correct predictions. This is due to the strong bias exhibited by many branches which has been exploited by other branch prediction studies [18, 21]. The data show that the alloyed perceptron covers the next largest fraction of correctly predicted branches. This is expected since the alloyed perceptron predictor is the best stand-alone prediction scheme. The short history length global predictor is the next most important contributor. Shorter history predictors have faster training times, and play an important role when branch behavior changes quickly. The remaining classifications all contribute in roughly equal amounts.

Our correct prediction classification scheme provides a rough ranking of the importance of each component predictor. We would expect this ordering to correspond to the sets of components chosen by our genetic search. The smallest configuration in Table 2 already uses three components, so we re-ran the genetic search algorithm with a hardware budget of 8KB, which resulted in a two-component hybrid. The two stand-alone predictors chosen are an alloyed perceptron and a short history length gshare, which correspond to the two largest classes of correctly predicted branches in Figure 7. For the 16KB configuration, the additional component is a long history length global predictor. At first sight, this may seem to contradict the data in Figure 7 because there are relatively few branches determined solely by the long history length global predictor. However, including the long history length global predictor also allows the COLT predictor to correctly predict the branches in the Multi-Length classification. Finally, for the 32KB configuration, the genetic search includes the local history component which allows the predictor to handle the Local and Mixed branch classes.

As we increase the hardware budget to 128KB and 256KB, the genetic search includes more components of varying history lengths. Stark et al showed that different branches in a program are best predicted with different history lengths, and the results of the genetic search provide further evidence to back this up [30]. Incorporating multiple history lengths can also help to distinguish between aliased branches in a pattern history table. For example, two distinct branches may map to the same entry in the short history-length global predictor, but the alloyed perceptron may provide different predictions in these two cases. Between these two sources of information, a prediction fusion mechanism can potentially learn the difference between the two. Note that the alloyed perceptron need not even give the correct prediction. If the alloyed perceptron provides consistently wrong, but different, predictions for these two branches, the VMT can distinguish between the two cases and provide the correct final prediction.

Juan et al observed that the optimal history length varies between benchmarks as well as during the execution of a single benchmark [16]. The per-benchmark classification data from Figure 7 corroborate the variance in optimal history length. For instance, the fraction of correctly predicted branches in `eon` classified in the *Short* group is about the same as the fraction in the *Perceptron* group, but the fraction of correctly predicted branches that fall into the *Short* class for `gcc` is much smaller relative to the fraction of branches classified as *Perceptron*. Juan et al proposed dynamic history length fitting to address the per-benchmark variance of the optimal history length. Dynamic history length fitting does not address the fact that the optimal history length also varies per branch. Stark et al's variable length path history approach statically assigns the history length per branch, and therefore does not address the fact that the optimal history length changes with time. Our COLT predictor can handle both of these types of variability in branch behavior, although the relatively small number of component predictors limits the number of candidate history lengths. This should not have much affect on overall performance since the data in [16] show that there is little performance difference between using the optimal history length and using a "close-to-optimal" history length.

One disadvantage of prediction fusion is that using a selective update policy for the component predictors may not be effective because the inputs from all predictors are used [23]. In a selection-based approach, components that are never selected for a branch need not be updated, which reduces the amount of interference in that component for other branches. There are some branches that are better predicted by selection-based techniques, while others that require prediction fusion. An extension of this research could involve designing meta-predictors that combine the best attributes of prediction selection and prediction fusion, using each when appropriate. This would also enable the usage of selective update policies whenever prediction-fusion is not used.

6.2. Design Trade-Offs

Besides the choice of components, the COLT predictor has three primary parameters: the width of the mapping table

counters, the number of entries in the VMT, and the number of branch history bits used to index the VMT. The configurations listed in Table 2 have all been "randomly" discovered by our genetic algorithm. In this section, we observe how the accuracy of the COLT predictor changes as the configurations depart from those chosen by our genetic search.

Saturating counters are used in several applications, and the optimal width of the counter also depends on the usage. Smith found that for tracking the direction of branch outcomes, two bits are sufficient [27]. Increasing the counter size beyond two bits yielded rapidly diminishing returns. Similarly, Evers found that three bits worked the best for the counters in selection-based hybrid predictors [7]. We simulated the configurations from Table 2 and varied the width of the VMT counters from 1 to 6 bits. At one or two bits, the counters are still too sensitive to transient variations in the mapping table. Beyond four bits, the incremental improvements are negligible. Instead of dedicating storage to additional counter bits, it is more beneficial to allocate the area to larger component predictors.

For a fixed hardware budget, there is a trade-off between how much real estate is dedicated to the component predictors, and how much to the VMT. Not enough area dedicated to the actual components results in poor individual predictions, which in turn presents less useful information for the hybrid predictor to work with. If the VMT is too small, then interference between different mapping tables will result in poor overall prediction rates. In Figure 8, we plot the performance of the COLT predictors as the size of the VMT is varied. The original configurations from Table 2 are highlighted with dark circles. Each data point to the right of one of the original configurations represents a doubling of the VMT size. The VMT size is halved for each point to the left of an original configuration. The points that represent the best trade-off between the components and the VMT are on the convex hull (from below) of the data points. These configurations all happen to be slightly larger than the cutoffs for the allotted hardware budgets, and so they were not selected by the genetic search.

The amount of branch history used to index the VMT is limited by the number of different mapping tables in the VMT. For the VMT index, different amounts of branch address bits and the global history may be combined. Figure 9 plots the misprediction rates of the COLT predictors as we vary the amount of branch history used to index the VMT. The maximum possible history length varies depending on the hardware budget because the size of the VMT also changes. Using more branch history tends to improve the overall misprediction rate. The amount of improvement gained by using more branch history varies, but the cost in hardware to use additional bits of history is negligible. The genetic search did not always choose the best possible history length. This is because the branch predictors

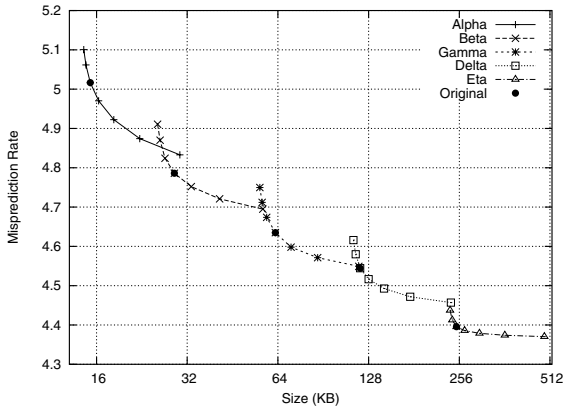

Figure 8: The COLT misprediction rates as the size of the VMT is varied, while holding the configurations of the individual components constant. The solid black circles indicate the original configurations listed in Table 2. In the legend, "Alpha" through "Eta" indicate that the components and all other parameters are identical to configurations α-η in Table 2.

were tuned on a different input set than in the final evaluation. In any case, the difference in performance between the chosen history lengths and the best history lengths is very small. The data from our exploration of the COLT design space gives us confidence that our genetic algorithm has performed a reasonable job at optimizing the COLT branch predictors.

7. Conclusions

In this study, we have proposed prediction fusion as a new approach to the design of hybrid branch predictors. Our experiments suggest that there are branches that can only be predicted if we fuse information from multiple types of predictors. Our Combined Output Lookup Table (COLT) predictor achieves lower misprediction rates than any other published prediction algorithm to date. Using Jiménez and Lin's overriding predictor scheme [14], we also demonstrate that our large, multi-cycle branch predictor can still be gainfully integrated into very aggressively clocked processors.

The Combined Output Lookup Table (COLT) predictor that we presented is but one possible fusion-based predictor. There are other possible variations that we do not explore in this paper, but will briefly mention here. The COLT predictor may be augmented with mechanisms such as *agree* prediction [29] or *selective branch inversion* [21]. Our VMT is indexed with only global branch history, but alloyed branch history may be beneficial as well [26].

The concept of prediction fusion may be useful outside the domain of conditional branch prediction. Prediction and speculation are used in many areas of computer microarchitecture, and some of these may benefit from a fusion of

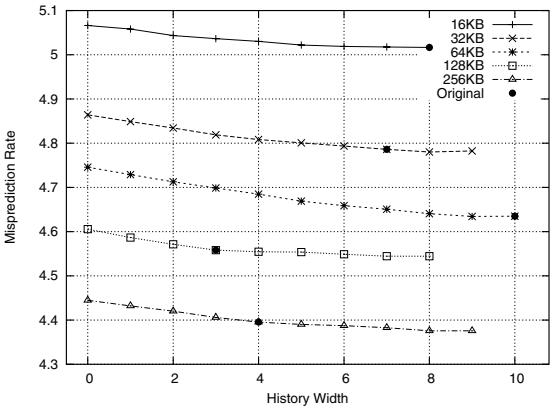

Figure 9: The COLT misprediction rates as the amount of global branch history used to index the VMT is varied. The solid black circles indicate the original configurations listed in Table 2.

prediction techniques. Possible applications are in branch confidence prediction [12], data value prediction [19], and memory dependence prediction [24, 25].

Acknowledgments

Karhan E. Akcoglu (Yale University) first brought the Weighted Majority algorithm to our attention, which was the starting point for this research. We would also like to thank Daniel H. Friendly (Yale University) who suggested valuable improvements to early drafts of this document, and to the anonymous reviewers who provided several useful comments to strengthen the paper. Simon Lok (Columbia University) suggested using the term *fusion*.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin, June 1997.

[2] P. Chang and U. Banerjee. Profile-Guided Multi-Heuristic Branch Prediction. In *Proceedings of the International Conference on Parallel Processing Vol. I*, volume 1, pages 215–218, Urbana-Campaign, IL, USA, August 1995.

[3] Po-Yung Chang, Eric Hao, and Yale N. Patt. Alternative Implementations of Hybrid Branch Predictors. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 252–257, Ann Arbor, MI, USA, November 1995.

[4] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale N. Patt. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 22–31, San Jose, CA, USA, November 1994.

[5] Avinoam N. Eden and Trevor N. Mudge. The YAGS Branch Prediction Scheme. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 69–77, Dallas, TX, USA, December 1998.

[6] Joel Emer and Nikolas Gloy. A Language for Describing Predictors and it Application to Automatic Synthesis. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 304–314, Boulder, CO, USA, June 1997.

[7] Marius Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, University of Michigan, 2000.

[8] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 3–11, Philadelphia, PA, USA, May 1996.

[9] Dirk Grunwald, Donald Lindsay, and Benjamin Zorn. Static Methods in Hybrid Branch Prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 222–229, Paris, France, October 1998.

[10] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Karmean, Alan Kyler, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[11] John H. Holland. *Adaptation in Natural Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[12] Erik Jacobson, Eric Rotenberg, and James E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 142–152, Paris, France, December 1996.

[13] Daniel Jiménez. *Delay-Sensitive Branch Predictors for Future Technologies*. PhD thesis, University of Texas at Austin, January 2002.

[14] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 4–13, Monterey, CA, USA, December 2000.

[15] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 197–206, Monterrey, Mexico, January 2001.

[16] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic History-Length Fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 156–166, Barcelona, Spain, June 1998.

[17] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, USA, November 2001.

[18] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The Bi-Mode Branch Predictor. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 4–13, Research Triangle Park, NC, USA, December 1997.

[19] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 1996.

[20] Nick Littlestone and Manfred K. Warmuth. The Weighted Majority Algorithm. *Information and Computation*, 108:212–261, 1994.

[21] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Branch Prediction using Selective Branch Inversion. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 81–110, Newport Beach, CA, USA, October 1999.

[22] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[23] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, Boulder, CO, USA, June 1997.

[24] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, Boulder, CO, USA, June 1997.

[25] Andreas Moshovos and Gurindar S. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 235–245, Research Triangle Park, NC, USA, December 1997.

[26] Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Alloyed Global and Local Branch History: A Robust Solution to Wrong-History Mispredictions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 199–206, Philadelphia, PA, USA, October 2000.

[27] Jim E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, MN, USA, May 1981.

[28] Eric Sprangle and Doug Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, Anchorage, Alaska, May 2002.

[29] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 284–291, Boulder, CO, USA, June 1997.

[30] Jared Stark, Marius Evers, and Yale N. Patt. Variable Length Path Branch Prediction. *ACM SIGPLAN Notices*, 33(11):170–179, 1998.

[31] The Standard Performance Evaluation Corporation. WWW Site. `http://www.spec.org`.

[32] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, Albuqueque, NM, USA, November 1991.