Code Placement for Improving Dynamic Branch Prediction Accuracy

Daniel A. Jiménez

Deptartment of Computer Science
Rutgers University
Piscataway, New Jersey, USA

and

Departamento de Arquitectura de Computadores
Universidad Politécnica de Cataluña
Barcelona, Cataluña, Spain

djimenez@cs.rutgers.edu

Abstract

Code placement techniques have traditionally improved instruction fetch bandwidth by increasing instruction locality and decreasing the number of taken branches. However, traditional code placement techniques have less benefit in the presence of a trace cache that alters the placement of instructions in the instruction cache. Moreover, as pipelines have become deeper to accommodate increasing clock rates, branch misprediction penalties have become a significant impediment to performance. We evaluate *pattern history table partitioning*, a feedback directed code placement technique that explicitly places conditional branches so that they are less likely to interfere destructively with one another in branch prediction tables. On SPEC CPU benchmarks running on an Intel Pentium 4, branch mispredictions are reduced by up to 22% and 3.5% on average. This reduction yields a speedup of up to 16.0% and 4.5% on average. By contrast, branch alignment, a previous code placement technique, yields only up to a 4.7% speedup and less than 1% on average.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Code generation, Optimization

General Terms Performance, Experimentation

Keywords Compilers, Branch prediction

1. Introduction

As pipeline depths increase to support higher clock rates, the penalty for a mispredicted conditional branch also increases, reducing the number of instructions executed per cycle. Improvements in branch predictor accuracy can have a significant positive impact on performance. Almost all modern microprocessors use a *pattern history table* (PHT) to predict the outcomes of conditional branches by tracking the tendency of a branch to be taken or not taken given a branch address and histories of branch outcomes. Because of strict timing and area limitations, PHTs have relatively few entries compared with the number of branches and branch histories. Thus, mispredictions caused by conflicts in PHTs are inevitable. In this paper, we evaluate *pattern history table partitioning* (PHT partitioning), a code placement technique that places conditional branches at addresses such that destructive interference in the PHT is reduced. The compiler divides an abstract model of the PHT into 2 or more partitions, each intended to predict branches with a particular kind of behavior (e.g. branches that are strongly biased to be taken). Conditional branch instructions are explicitly placed such that their addresses tend to map to the most appropriate PHT partition.

This paper makes the following contributions:

1. We present the first experimental study of an actual implementation of pattern history table partitioning. We evaluate this technique on the Intel Pentium 4, a microprocessor that has an instruction trace cache that significantly reduces the benefit of traditional code placement techniques such as branch alignment [4]. On SPEC CPU integer benchmarks, our technique reduces branch mispredictions by up to 22% and 3.5% on average. This reduction yields a speedup of up to 16.0% and 4.5% on average. By contrast, branch alignment yields only up to a 4.7% speedup and less than 1% speedup on average.
2. We compare our algorithm with a modified version that behaves similarly to address adjustment with branch classification, a previous technique [6]. Address adjustment reduces branch mispredictions but tends to increase instruction counts. When the algorithm is constrained to insert fewer no-ops, it improves average speedup by only 1.5%. When more no-ops are inserted, the technique results in a slight slowdown on average. By contrast, our PHT partitioning achieves the same reduction in branch mispredictions as aggressive address adjustment but delivers a significant speedup.

1.1 Motivation

A first version of our technique builds on the observation that branches tend to be highly biased to be either taken or not taken. It makes sense to partition the PHT into two halves: one for biased taken branches and one for biased not taken branches. This way, branch instructions mapping to the same PHT entry will tend to have similar behavior and will not interfere destructively with one another. This observation has been made in previous work [23] and generalized and exploited in microarchitectural simulation [6].

Figure 1 uses greyscale intensities to show the average value of PHT entries before and after a transformation that maps biased taken branches to even indices and biased not taken branches to odd indices. These results are gathered from trace-based simula-

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

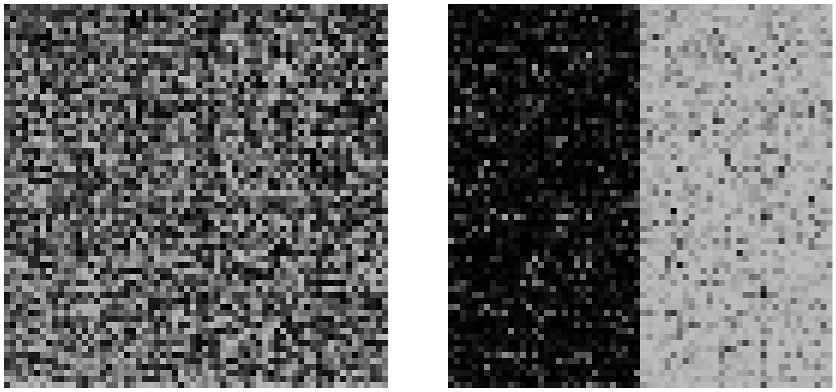


Figure 1. Average PHT entries before and after compiler-based alignment.

tion facilitated by SimpleScalar / Alpha [2][1]. Each pixel represents the average value of an entry of the PHT for the SPEC CPU 2000 benchmark `176.gcc` and a 4,096-entry GAs (Global Adaptive set-based) [31] branch predictor. This predictor concatenates the 4 lower bits of the branch address with the last 8 branch outcomes to form an index into the PHT. The PHT entries are two-bit saturating counters that are incremented or decremented if the corresponding branch was taken or not taken, respectively. Before the optimization the pattern history table is a jumbled mess of counters with a significant amount of destructive interference. After the optimization, the pattern history table is divided into entries that predict mostly taken on the left and mostly not taken on the right. The average PHT entry is brighter on the right and darker on the left, illustrating a reduction in destructive interference. The result of this reduction in interference is that the misprediction rate for `176.gcc` has been cut almost in half, from 13.2% to 7.7%.

This nice result comes from the world of simulation, in which we completely specify the design of the branch predictor. The challenge we address in this paper is to develop a similar optimization for an existing microprocessor whose branch predictor is complex and not well documented: Intel's Pentium 4.

1.2 Paper Organization

This paper is organized as follows. In Section 2, we explore previous related work in optimizing for instruction fetch and branch prediction. In Section 3 we give background into two-level adaptive branch prediction. In Section 4 we describe our optimization. In Section 5 we describe our experimental methodology for evaluating the new optimization for the Intel Pentium 4 microprocessor. In Section 6 we give the results of our experiments showing speedups and reductions in branch mispredictions. In Section 7 we conclude and give directions for future research.

2. Related Work

In this section we discuss related work in code placement techniques.

2.1 Profile Guided Code Placement

Hatfield and Gerald [8], McFarling [20], Pettis and Hanson [25], and Gloy and Smith [11] have presented methods to reorder procedures to improve locality based on profile data. McFarling presents an algorithm for placing code so as to minimize conflict misses in a direct-mapped instruction cache [19].

[1] We use simulation only to generate Figures 1 and 7. All other results in this paper are from a real Intel Pentium 4.

2.2 Code Placement for Reducing Branch Costs

Without respect to branch prediction concerns, it is generally better for a frequently executed branch to be laid out such that it is usually not taken. This way, the instruction cache is used more efficiently as the hot path through the code is laid out sequentially. Also, machines that fetch multiple instructions in a single cycle benefit from not taken branches because they do not prematurely terminate fetching and thus may exploit the full fetch width on every cycle for which there is an instruction cache hit. Calder and Grunwald present *branch alignment*, an algorithm that seeks to minimize the number of taken branches by reordering code such that the hot path through a procedure is laid out in a straight line [4]. Their technique improves performance by an average of 5% on an Alpha AXP 21064. We make use of the "greedy" version of branch alignment to compare against our proposed technique on a modern machine. Young *et al.* present a version of branch alignment [32] based on an algorithm for solving the Directed Traveling Salesman Problem and show that both their algorithm and the greedy branch alignment algorithm presented by Calder and Grunwald are close to optimal with respect to minimizing the number of taken branches.

Software Trace Cache is a code-reordering optimization introduced by Ramirez *et al.* that packs frequently used code in a section of memory so as to minimize conflict misses in a direct-mapped instruction cache [26]. The Software Trace Cache idea also lays out hot paths so that most branches are not taken. Ramirez *et al.* study the effect of such code reordering techniques on branch prediction, showing that branch prediction accuracy can be positively or negatively affected by code reordering depending on the details of the branch predictor [27]. For instance, by increasing the number of not-taken branches, branch alignment techniques can skew the distribution of accesses to PHT entries. Software Trace Cache is evaluated through detailed microarchitectural simulation and assumes a direct-mapped instruction cache.

2.3 Code Placement for Improving Branch Prediction Accuracy

2.3.1 Address Adjustment

Chen and King first propose the idea of using code placement to reduce interference in pattern history tables [6]. They propose *walk-time address adjustment*, a technique that adjusts branch addresses at link-time to avoid interference in branch prediction tables. As in our optimization, no-op instructions are inserted to move branch addresses to positions less likely to interfere with one another and the PHT is divided into partitions representing different kinds of branch behavior. That research considered simple branch predictors evaluated through simulating the first 30 million branches of 8 SPEC CPU benchmarks. The research did not demonstrate an improvement in performance for large or complex branch predictors such as the ones encountered in modern microarchitectures. The address adjustment technique yielding the best improvement in branch prediction accuracy also places many extra no-op instructions on the critical path, a fact whose impact is not realized through simulations that measure only misprediction rate. Extra inserted no-ops with this technique can also hinder performance by violating alignment heuristics. For instance, Chen and King's address adjustment can move branch targets off of fetch block boundaries and cause a decrease in the instruction fetch rate and an increase in instruction cache miss rates. Our technique minimizes both misprediction rate and extra instruction count and is the first such technique to demonstrate significantly improved performance on a real machine. Our technique also respects compiler alignment heuristics designed to maximize the instruction fetch rate.

In Section 6 we configure our algorithm to behave similarly to that of Chen and King's best performing technique, *branch classification*. It shows the negative impact on performance of improving misprediction rate without proper consideraton for keeping instruction count from increasing too much.

The practical impact of Chen and King's work is to suggest that address adjustment be taken into account by microarchitects when designing a branch predictor. That is, address adjustment enables good accuracy with smaller branch predictor. A smaller predictor is desirable because it will have a lower access latency and thus mitigate the negative impact on performance of a high-latency predictor [14]. Our work addresses a different issue: how to optimize for a fixed branch predictor in a real microarchitecture.

In the same spirit as our work and the work of Chen and King, Milenkovic *et al.* suggest that compiler techniques could be used to avoid destructive interference in branch predictors [23]. However, they do not provide an algorithm or an implementation.

2.4 Branch Allocation

Our idea is similar to *branch allocation* [16]. Branch allocation uses the working set characteristics of branches to explicitly assign each conditional branch a set of branch history table resources at compile time. The analysis forms a conflict graph between branches and uses a technique similar to register allocation to allocate branch history table resources among branches such that destructive aliasing is reduced. However, branch allocation modifies the instruction set to allow branch instructions to explicitly specify an index into the pattern history table. Branch allocation is shown to improve misprediction rates through simulation. By contrast, our research implicitly guides branches to destinations in the PHT and thus requires no modification to the instruction set.

2.4.1 Static Correlated Branch Prediction

Young and Smith introduce *static correlated branch prediction* [33], a technique that duplicates basic blocks with branches whose outcomes are highly dependent on the immediately preceding program path. This technique seeks to improve branch prediction accuracy for machines where branches are statically predicted, i.e., the prediction is determined at compile-time. This technique requires path profiling which is more expensive than the edge profiling required by our technique. Static correlated branch prediction duplicated basic blocks resulting in significant code expansion while our technique does not. Most importantly, our technique addresses dynamic branch prediction accuracy used in modern microprocessors, and not older static branch prediction techniques.

2.5 Other Compiler-Related Branch Prediction

Some processors implement static branch prediction either implicitly through heuristics or explicitly through compiler-provided hint bits that are set to indicate the likely direction of each branch. Some branches can be predicted with a static bias bit, while others with less biased behavior can use the dynamic predictor. Since the easily predictable branches are filtered out, aliasing in the dynamic predictor is alleviated and accuracy is improved. This technique, along with a methodology for choosing the bias bits, was introduced by Chang *et al.* [5]. Patil and Emer study the technique, measuring its utility in reducing destructive aliasing and refining the heuristics used to decide which branches should be predicted statically [24]. The Intel Pentium 4 instruction set includes hint bits for conditional branches, but these hints are used only during trace construction [13] and are unlikely to have an impact on dynamic branch prediction accuracy.

3. Branch Prediction Background

In this section, we give background into two-level adaptive branch prediction that is used in many modern microprocessors.

3.1 The Need for Branch Prediction

Modern pipelined microprocessors consult branch predictors to speculatively fetch and execute instructions beyond conditional branches. When a conditional branch is fetched, its outcome and target may not be computed for many cycles. Nevertheless, instruction fetch must continue to feed the rest of the execution engine and sustain performance. Branch predictors quickly predict the likely direction of a branch and allow the processor to continue to fetch instructions down the predicted path. Processors that implement speculative execution execute instructions on the predicted path and have a mechanism for rolling back and restarting execution if a branch is mispredicted. A branch misprediction can be very costly in terms of the number of wasted clock cycles and wasted energy processing wrong-path instructions. As pipelines become deeper to support higher clock rates, the penalty of a mispredicted branch also increases and has a higher negative impact on performance.

3.2 Two-Level Adaptive Branch Prediction

Yeh and Patt observed that the outcome of a given branch is often highly correlated with the outcomes of other recent branches [31]. This history of branch outcomes forms a pattern that can be used to provide a dynamic context for prediction. In the branch prediction scheme of Yeh and Patt, every time a branch outcome becomes known, a single bit (0 for *not taken*, 1 for *taken*) is shifted into a pattern history register. A pattern history table of two-bit saturating counters is indexed by a combination of branch address and history register. The high bit of the counter is taken as the prediction. Once the branch outcome is known, the counter is decremented if the branch is not taken, or incremented otherwise, and the pattern history is updated. Much research in the 1990s focused on refining this scheme of Yeh and Patt. For instance, hybrid predictors that combine branch predictors to improve accuracy have been proposed [21, 10] and implemented [15].

3.3 Destructive Interference in Pattern History Tables

Destructive interference occurs when two distinct branches with opposite behavior coincidentally use the same counter in the pattern history table. Destructive interference will lead to one of the branches being predicted incorrectly. This situation is somewhat similar to conflict misses in data caches where multiple items in memory map to the same set in the cache resulting in cache misses [22]. Several predictor organizations have been proposed to deal with the problem of destructive interference [17, 29, 9], but the problem cannot be entirely eliminated because the number of counters in pattern history tables is much less than the number of branches and branch histories in typical programs. In this paper, we present a software approach to reducing destructive interference.

Almost all approaches to reducing the impact of destructive interference have been studied at the microarchitectural level through simulation. The *agree* predictor uses a PHT to predict whether branch outcome will agree with a bias bit either provided statically or learned online [29]. In this way, the *agree* predictor transforms possibly destructive interference into constructive interference since most branches are likely to agree with their bias. Our technique is similar to this approach in that we statically group branch instructions in a partition of the PHT with other branch instructions with similar behavior. Of course, our technique is a software technique applied to a real machine, not a microarchitectural innovation.

One hardware technique in particular is closely related to our approach. The BiMode predictor divides the predictor into three tables: one PHT for biased taken branches, one PHT for biased not taken branches, and a third chooser PHT that tracks the biases of branches to decide which of the first two PHTs to use for predicting a given branch [17]. This technique does in hardware what our technique does in software. Again, note that our technique is a more feasible approach since it does not require a change to the microarchitecture and does not introduce a level of indirection and delay into the critical path of the branch predictor.

3.4 Branch Prediction in the Intel Pentium 4

For this study, we choose the Intel Pentium 4 as our optimization target. This very popular microprocessor is widely used in a range of applications. It has several features that make it an interesting candidate for our technique:

1. It has a 20-stage pipeline to support a high clock frequency. This means that the Intel Pentium 4 will be sensitive to branch predictor accuracy, as a mispredicted branch will incur a substantial cycle penalty.
2. It has an instruction trace cache [28] that stores decoded instructions in the order they were fetched rather than the order they appear in the program text. This tends to reduce the improvement provided by traditional code placement techniques that seek to improve instruction locality since the trace cache has the effect of dynamically reordering frequently used sections of code. Our code placement technique is still able to improve performance because the instruction fetch engine uses the same branch outcome predictor to guide speculation whether it is fetching from the second level cache or the trace cache [23] (although a different predictor is used to predict the next trace to fetch). This means that even when the Pentium 4 is fetching from the trace cache it is still driving speculation by predicting branches using branch addresses and those predictions are improved by our technique.
3. The branch predictor of the Intel Pentium 4 is not well documented. From published Intel documentation [12], we know that the branch predictor has a 4,096-entry branch target buffer (BTB) with presumably correspondingly many entries in a PHT. The predictor is able to fully predict loop back edges with a trip count of between 1 and 16. It can correctly predict branches with taken/not-taken pattern lengths between 1 and 4. Beyond that information, what little is known publicly about the Intel Pentium 4 branch predictor has been gathered through reverse-engineering [23]. We hypothesize that the branch predictor uses some bits from the conditional branch fetch address as part of an index into a PHT to make a prediction. We use trial and error to determine which bits are the most likely to both be used by the predictor and are feasible to be used by our optimization.

4. Pattern History Table Partitioning

In this section, we describe the pattern history table partitioning (PHT partitioning) technique. We first describe *bimodal* PHT partitioning that divides the PHT into two partitions, one for biased taken branches and the other for biased not taken branches. We then describe a variation called *4-way* PHT partitioning that divides the PHT into quadrants for branches that are combinations of strongly and weakly biased taken or not taken. Figure 2 graphically illustrates the manner in which the PHT is partitioned in the two versions.

4.1 The Basic Idea Behind Bimodal PHT Partitioning

The main idea for bimodal PHT partitioning is to place branches with similar biases (i.e. taken or not taken) such that they use the same half of the PHT. We assume that the branch prediction algorithm uses at least one lower-order bit from the branch address, along with branch history, for indexing the PHT. PHT partitioning ensures that this bit is usually one value, e.g. 0, for biased not taken branches and usually the other value for biased taken branches. This task is accomplished by inserting multiple *no operation* (no-op) instructions between regions in the code identified by the PHT partitioning algorithm. This padding with no-ops changes the bits in the addresses of every instructions; the goal is to maximize the number of conditional branch instructions where the relevant bit matches the branch bias.

4.1.1 Tracking Instruction Address Bits

The PHT partitioning algorithm performs an assembly of each procedure to keep track of the sizes of all instructions so that the algorithm can always compute the lower-order bits address of a given instruction. This tracking of instruction sizes is non-trivial in the Intel Pentium 4 ISA, where jump instructions may change sizes depending on the magnitude of the distance to their targets; after each proposed placement, the offsets are recomputed and the sizes of jump instructions are adjusted accordingly. In addition, the front-end compiler inserts alignment pseudo-ops so that e.g. branch targets are aligned on 16-byte boundaries whenever such alignment results in the insertion of fewer than 8 no-op instructions; our algorithm tracks the address bits generated by this alignment.

There is no documentation explaining which address bits in the Intel Pentium 4 are used in indexing the PHT, but we hypothesize that several of the lower order bits are used. In practice, we find that using the fourth bit of the branch address provides the best trade-off between improved branch prediction accuracy and reduced instruction cache locality. Thus, we need only keep track of addresses modulo $2^4 = 16$. The beginning of each procedure is aligned on a 16-byte boundary so that it begins at address 0 modulo 16. However, for the purpose of describing a general algorithm, let us say that the goal of PHT partitioning is to assign the k^{th} bit of the branch address, where k is a parameter chosen for the given microarchitecture. Thus, our algorithm needs to keep track of branch address bits modulo 2^k.

4.1.2 Profiling Requirements

The PHT partitioning algorithm requires edge profiles, i.e., information from a training run about how often any edge in the control flow graph has been traversed. This way the algorithm can infer the biases of branches to be taken or not taken. Not every branch can be placed with the most appropriate PHT partition, so the frequency information from the edge profiles is used to prioritize the frequently executed branches for placement.

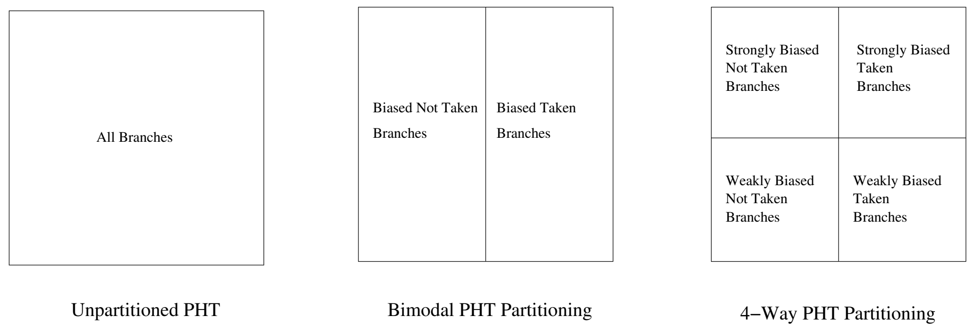


Figure 2. Each box represents the PHT. Bimodal PHT partitioning divides the PHT into two halves; 4-Way PHT Partitioning divides it into four quadrants.

4.1.3 Choosing Regions

Placing multiple no-op instructions in arbitrary locations of a program can have a negative impact on performance because it can decrease useful fetch bandwidth. Thus, PHT partitioning places no-op instructions only where it is unlikely that they will hurt performance. Each procedure is divided into a number of regions between which no-ops may be inserted by PHT partitioning. The basic blocks are scanned in the sequence they appear in the procedure to build regions. The first region begins with the first basic block in the procedure. Regions may be ended by a basic block if it satisfies one of the following rules:

1. The basic block does not fall through. For instance, no-ops inserted after a basic block ending in a jump or a return will not be executed and thus will have no impact on fetch bandwidth.
2. The basic block ends in a conditional branch that is taken at least 99.9% of the time and at least 100 times. This way, no-ops on the fall-through path will be executed relatively infrequently. We require that the branch be taken at least 100 times so that we do not needlessly increase code size by padding very infrequent basic blocks that are unlikely to have any impact on performance. We refer to this rule as the *frequency heuristic*.

4.1.4 Padding Between Regions

Once regions are chosen for a given procedure, the PHT partitioning algorithm visits each region in sequence. It determines the effect of inserting each possible number of one-byte no-op instructions from 0 through $2^k - 1$ on assigning branches to their proper partitions. For each proposed number of no-ops to insert, the algorithm recomputes the addresses of every branch instruction in the region, taking into account changes in branch instruction lengths and compiler-inserted alignment pseudo-ops. It inserts the number of no-ops that maximizes the number of branches assigned to their proper PHT partitions, weighted by edge frequency. That is, it inserts a number of no-ops that maximizes the number of dynamic branches in the profiled run assigned to their proper partition. It is important to note that inserting no-ops between regions composed of multiple basic blocks provides only coarse control over the addresses of branches. For example, in the SPEC CPU 2000 benchmark `254.gap`, 85% of the static branches are assigned to their proper PHT partition, versus 53% before applying PHT partitioning.

4.2 Four-Way PHT Partitioning

Bimodal PHT partitioning uses one address bit to divide the PHT into two partitions. Since the branch predictor presumably uses more than one address bit, it behooves us to investigate using more than one bit to divide the PHT into more than two partitions. But what behavior should each of the new partitions seek to capture?

Four-way PHT partitioning divides the PHT into 4 regions, separating branches into 4 classes of behavior: strongly biased taken, strongly biased not taken, weakly biased taken, and weakly biased not taken. We define strong branch behavior to mean that the behavior is encountered at least 95% of the time, e.g. we say a branch is strongly biased taken if it is taken at least 95% of the time it is executed. We decided on the 95% threshold based on behavior we observed in the profiles we gathered. On average, approximately 50% of the dynamic branches in our benchmark suite were either taken or not taken 95% of the time. Thus, choosing 95% as our threshold for strong behavior should ensure a balanced distribution of branches to PHT partitions.

The details of 4-way PHT partitioning are the same as bimodal PHT partitioning except for the procedure for determining the best number of no-ops to insert between regions. We choose a number of no-ops to insert that maximizes a fitness function defined as follows. The fitness function is the sum over all branches in the region of:

1. The frequency with which the branch agrees with its bias if both the strength and bias match the PHT partition to which it is mapped,
2. Half of the frequency with which the branch agrees with its bias if only the bias matches the PHT partition,
3. 0 if neither bias nor strength match.

Thus, branches that agree in both strength and bias with their assigned PHT partition are most preferred, branches that agree in only bias are preferred less, and branches that agree in neither strength nor bias are avoided.

We found that, for the Intel Pentium 4, using the third and sixth bits of the branch address yielded the best trade-off between improvement in branch prediction and reduced instruction cache locality due to the inserted no-ops. Thus, for 4-way PHT partitioning, from 0 through $2^6 - 1 = 63$ no-op instructions may be inserted between regions.

5. Methodology

In this section, we explain our experimental methodology for evaluating the effect of PHT partitioning.

5.1 Benchmarks

Table 1 shows the benchmarks used for this study. We used the following criteria for choosing the benchmarks for this study:

1. We chose SPEC CPU [30] benchmarks from the 1995 and 2000 suites.
2. We chose the integer benchmarks from SPEC CPU. It is customary to use the integer benchmarks in branch prediction studies because they represent a class of applications sensitive to branch predictor accuracy.
3. We omitted the benchmarks from SPEC CPU 95 that are substantially duplicated in SPEC CPU 2000, e.g. `126.gcc`.
4. We omitted `124.m88ksim` and `252.eon` because they failed to compile and run correctly with our compiler infrastructure.

5.2 The Camino Compiler

In this paper we introduce the Camino Compiler Infrastructure. Camino was developed to study code placement optimizations at the level of assembly language. Camino uses GCC as its front-end to compile C programs to x86 assembly language. Camino reads the assembly language into an internal representation consisting of a list of procedures with control flow graphs (CFG). Camino transforms the code, then uses GCC as a back-end for assembling and linking. Camino is capable of instrumenting code to gather basic block counts as well as edge and path profiles; only edge profiles were gathered for this research. Camino is the infrastructure in which we implement the code placement optimizations mentioned in this paper: greedy branch alignment and pattern history partitioning. Camino was developed using GCC 2.95.4 as the front-end and back-end; however, for this study we use the latest version of GCC available as of this writing, GCC 3.4.2, for most of the benchmarks. We revert to GCC 2.95.4 for `176.gcc`, `253.perlbmk`, and `255.vortex` because the combination of Camino and GCC 3.4.2 results in the failure of those programs to compile correctly.

5.2.1 Optimizations in Camino

Camino implements the greedy branch alignment as described by Calder and Grunwald [4]. The point of the algorithm is to lay out basic blocks in a procedure such that basic blocks incident on frequently executed edges are placed adjacent to one another. Basically, a priority queue of control flow graph edges is generated in order of the frequency with which each edge is traversed according to the edge profiles from a training run. The algorithm removes edges from the queue in descending order of edge frequency. For each edge removed, the basic blocks incident on that edge are laid out adjacent to one another in the program text if no previously chosen edge succeeds the predecessor block or precedes the successor block. Once all basic blocks have been laid out, conditional branch senses are modified so that branches between adjacent basic blocks fall through.

Camino also implements GCC's heuristics for aligning code for maximum performance, e.g. aligning certain branch targets on 16-byte boundaries.

5.3 Compiling and Running the Benchmarks

We compile each benchmark using the edge profiling option of Camino. We run each benchmark on `train` inputs provided by SPEC to gather profiles. We use `ref` inputs to measure the program behavior we report in Section 6. The programs are linked statically. We subsequently compile several versions of each benchmark:

- A baseline using only the GCC optimization flags recommended by SPEC for x86 Linux: `-O3 -fomit-frame-pointer`.
- A version using the baseline optimizations and greedy branch alignment.
- A version using the baseline optimizations and bimodal pattern history table partitioning.
- A version using the baseline optimizations and 4-way pattern history table partitioning.
- Two versions (explained in Section 6.3) in which PHT partitioning is configured to behave similarly to other previous work.

We run the programs on a Dell workstation with 2GB SDRAM featuring an Intel Pentium 4 at a clock frequency of 2.8 GHz. The computer runs the Fedora Core 2 Linux operating system. We kill as many background daemon processes as possible to provide a quiescent system for our runs. We use the Unix `/usr/bin/time` command to measure execution time. We use the OProfile profiling system to measure numbers of branch mispredictions [18]. The reported times and mispredictions are for the measured user process only, not for the whole system. We run each benchmark 5 times and compute speedups based on the median running times. The OProfile system uses the performance counters of the Intel Pentium 4 to count branch mispredictions. We collect branch misprediction statistics in separate runs because the method by which branch mispredictions are counted involves somewhat frequent interrupts of the user process and might have the effect of perturbing timing results.

5.3.1 Code Expansion

Each code placement technique resulted in a very slight expansion of the generated executable. This expansion never exceeded 1.5% of the total size of the executable and was usually much less.

5.3.2 Compilation Times

The PHT partitioning algorithm exhaustively computes a fitness function for each proposed number of no-op instructions to insert for each region. Each evaluation of this fitness function requires that part of the job of assembling the region be done in order to find the addresses (modulo 64) of each branch instruction in the region. We implemented a number of algorithmic optimizations to keep this extra overhead to a minimum. On average, compilation of programs using the PHT partitioning technique takes 16% more time than compilation using only the baseline optimizations.

6. Results

In this section, we give the results of experiments showing the benefit of PHT partitioning. We begin by showing the speedups achieved on the benchmark. We then give insight into how these speedups were achieved by showing the effect of the technique on reducing mispredictions.

6.1 Speedup

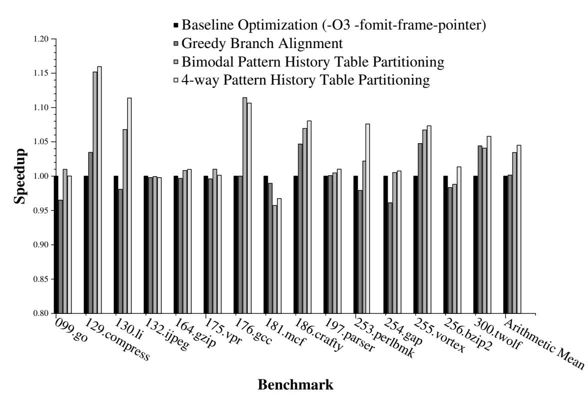


Figure 3. Speedup for SPEC CPU integer benchmarks.

Benchmark	Description	# Branches
`099.go`	Plays the game of *go*. Pattern matching.	9,511
`129.compress`	Compresses files with Lempel-Ziv adaptive encoding.	205
`130.li`	Lisp interpreter running the Gabriel benchmarks.	967
`132.ijpeg`	Compression/decompression for JPEG images.	1,703
`164.gzip`	Compresses files with Lempel-Ziv coding.	794
`175.vpr`	Placement and routing program for FPGAs.	1,248
`176.gcc`	C compiler (gcc 2.7.2.2) for the Motorola 88100.	27,961
`181.mcf`	Single-depot scheduling for mass transportation.	294
`186.crafty`	Plays chess using alpha-beta search.	3,787
`197.parser`	Parses English text to produce grammar analysis.	3,869
`253.perlbmk`	Stripped-down version of Perl v5.005_03.	9,007
`254.gap`	Language for group-theoretic computation.	6,263
`255.vortex`	Object-oriented database program.	8,644
`256.bzip2`	Compresses files with block-sorting compression.	756
`300.twolf`	Standard-cell placement and routing.	3,052

Table 1. Description of SPEC CPU integer benchmarks with the number of conditional branches executed at least once.

Figure 3 shows the speedups achieved for each benchmark and on average by greedy branch alignment, bimodal PHT partitioning, and 4-way PHT partitioning. On average, 4-way PHT partitioning performs the best at an arithmetic mean 4.5% speedup. This technique yields a speedup of up to 16.0% in the case of `129.compress` and 11.4% for `130.li`. The speedup is greater than 1.0 on 13 of the 15 benchmarks.

Bimodal PHT partitioning, a simpler technique, yields a 3.4% speedup on average. It speeds up `176.gcc` by 11%. The speedup is greater than 1.0 on 12 of the 15 benchmarks. In a few instances, bimodal PHT partitioning outperforms 4-way PHT partitioning, but the magnitude of the aggregate speedup of 4-way PHT partitioning leads us to prefer it over bimodal.

Greedy branch alignment, a code reordering technique that lays out basic blocks so that most branches are not taken, achieves only a 0.2% speedup on average. The speedup is greater than 1.0 on only 7 of the 15 benchmarks. The lack of improvement from greedy branch alignment is almost certainly due to the fact that the Intel Pentium 4 uses an instruction trace cache that has the effect of reordering code dynamically, negating much of the benefit to instruction fetch of branch alignment.

6.2 Reduction in Branch Mispredictions

Figure 4 shows the number of branch mispredictions per 1000 instructions (MPKI) for each benchmark. Clearly, there is a wide variation in branch prediction accuracy over all the benchmarks. For instance, `099.go` incurs about 37 mispredictions for every 1000 instructions. On the other hand, `255.vortex` incurs less than one misprediction for every 1000 instructions. **Note**: the MPKI figures are all computed with respect to the instruction count of the original program, so e.g. the extra no-ops introduced by PHT partitioning do not artificially decrease MPKI.

The highly variable MPKI can obscure the effect on mispredictions of the various techniques. Figure 5 shows the MPKI for each benchmarks normalized to the MPKI for the baseline optimizations. This gives a more clear and balanced picture to explain why PHT partitioning provides superior performance. For instance, when 4-way PHT partitioning is used for `130.li`, this benchmark incurs 16.2% fewer mispredictions that it does with the baseline optimizations. On average, normalized MPKI is reduced by 3.5%.

Note that the magnitude of the improvement in MPKI does not always correspond to the magnitude of the performance improvement. This is due to the fact that there are many other components to the microarchitecture than just branch prediction, and the extent

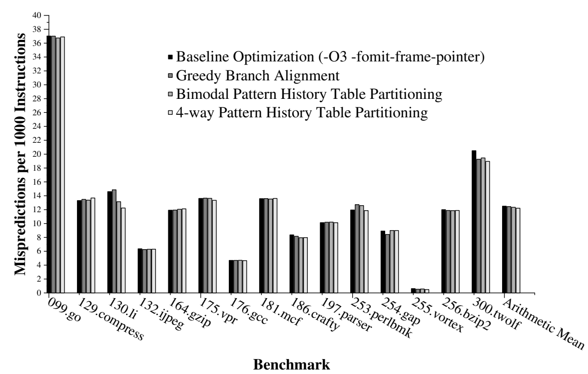


Figure 4. Mispredictions per 1000 instructions (MPKI) for SPEC CPU integer benchmarks.

to which these factors affect performance varies from one benchmark to another. For instance, a benchmark with very few cache misses might be very sensitive to branch prediction accuracy, while another with a high number of misses might not be improved at all by branch prediction optimizations. Also, some branches are more important than others in terms of their ability to affect performance. The minimum penalty of a mispredicted branch on the Intel Pentium 4 is 20 cycles, but the maximum penalty is much higher due to wrong-path effects such as cache pollution and resource contention with right-path instructions. Thus, it might be more important to predict certain branches correctly than others; we plan to research this issue in future work.

For instance, although the best speedup for 4-way PHT partitioning is achieved on `129.compress`, this benchmark incurs slightly more mispredictions that it does with the baseline optimizations. We hypothesize that this apparent paradox is due to the fact that 6 static branches account for over 50% of all the branches executed during a run of the program. With so few branches dominating the program, a small variance in the misprediction penalty in one of these branches can have a large impact on performance. That is, it is likely that one or more of these 6 branches has a high misprediction penalty, and our optimization removes this high penalty by removing the mispredictions. In retrospect, `129.compress` might be considered a poor benchmark for branch prediction studies because of its very small working set of branches, but we include it in this study for completeness. Nevertheless, we are confident that we have demonstrated that our technique achieves a significant speedup: when we exclude `129.compress` as well as `181.mcf` which is the outlier with the lowest speedup, the arithmetic mean speedup is 4.2% and the average improvement in normalized MPKI is 41%.

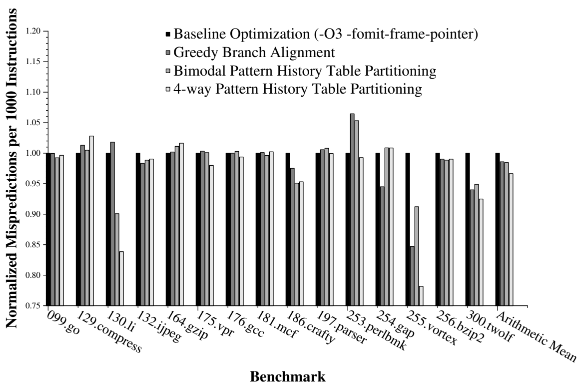


Figure 5. Normalized mispredictions per 1000 instructions for SPEC CPU integer benchmarks.

6.3 Impact of Minimizing Extra No-Op Instructions

There is an important trade-off between reducing the number of mispredicted branches and increasing the number of no-op instructions fetched. The previous work of Chen and King achieved a significant reduction in misprediction rate using an optimization similar to PHT partitioning; however, this reduction comes at the cost of many extra no-op instructions being executed.

Chen and King define the *Maximum Motion Distance* (MMD) as the maximum number of no-op instructions that may be inserted after a branch instruction [6]. A lower MMD will tend to reduce the number of extra no-ops fetched while a higher MMD provides more flexibility for placing branches to reduce destructive interference. Our technique uses a frequency heuristic to decide whether to insert no-ops after a conditional branch: if the branch is taken less than 99.9% of the time, then we do not place no-ops after it because they have the potential to decrease fetch bandwidth with extra fetched no-ops.

Figure 6 illustrates the impact of extra no-op instructions. For this experiment, we modify the PHT paritioning algorithm to ignore the frequency heuristic and accept an MMD parameter that will be used to determine how many no-ops may be placed after a branch. The resulting algorithm is similar to the link-time algorithm of Chen and King. (However, the modified algorithm still respects alignment heuristics as described before.)

Figure 6 (a) shows the normalized MPKI for the unmodified and modified PHT partitioning algorithms. With an MMD of 4, the modified algorithm yields an average normalized MPKI of 0.991, a reduction of less than 1% in the number of mispredictions. With an MMD of 32, the modified algorithm yields a normalized MPKI of 0.966, a reduction of 3.5%. The original algorithm also yields a normalized MPKI of 0.966.

However, the original algorithm yields a much better speedup than the modified versions because it uses the frequency heuristic. Figure 6 (b) shows that the average speedup of the original algorithm is 4.3%, while the speedups for the modified algorithm with MMDs of 4 and 32 are 1.5% and -0.4% respectively. Indeed, the most aggressive form of the modified algorithm with the highest reduction in mispredictions yields a negative speedup. Thus, it is imperative that a PHT partitioning algorithm avoid placing no-op instructions on frequently executed paths.

6.4 Potential for Influencing Branch Predictor Design

PHT partitioning has the potential to simplify the design of future branch predictors. A branch predictors must supply its prediction within a single clock cycle [14]. However, as clock rates increase, the amount of time available to make a prediction becomes shorter. One option available to microarchitects is to reduce the size of the PHT and thus decrease its access latency, allowing the branch predictor latency to fit in a single clock period. For instance, the aggressively clocked AMD Athlon uses a small 2K-entry GAs predictor, four times smaller than the 8K-entry GAs used in the previous generation AMD K6 core [7]. However, the resulting decrease in branch predictor accuracy can decrease some of the performance advantage gained by increasing the clock rate. PHT partitioning can allow designers to use smaller tables but sustain the same misprediction rates as previous generation processors. Figure 7 shows the effect on misprediction rate of PHT partitioning with PHTs varying from 256 to 64K entries for the `176.gcc` benchmark using a simulated GAs predictor with an 8-bit history length. For reference, the Intel Pentium 4 branch predictor has 4,096 entries [12]. A predictor with only 256 entries and bimodal partitioning achieves a misprediction rate of 4.3%, while a predictor with 8,192 entries with no partitioning achieves a higher misprediction rate of 4.8%. Thus, accuracy can be sustained or even improved even though technological contraints may force the PHT to become smaller.

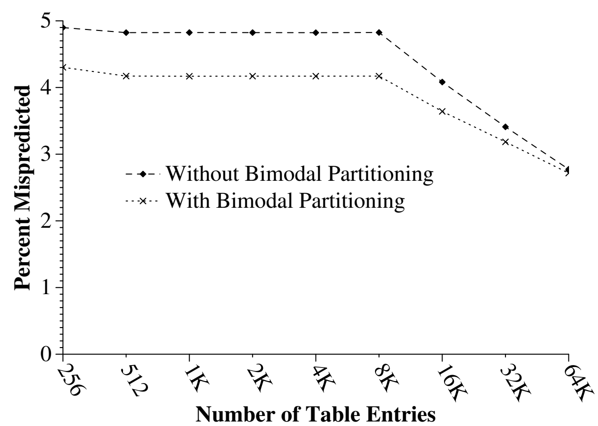
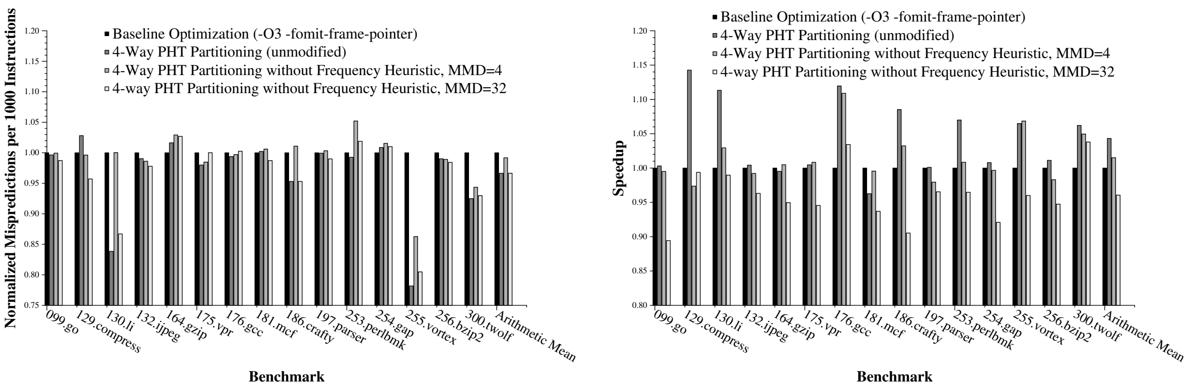


Figure 7. Simulated Misprediction Rates for Various Sized PHTs on `176.gcc`.

7. Conclusion

This paper has introduced a technique for pattern history table partitioning for reducing branch mispredictions. By dividing the branch predictor's pattern history table into partitions and grouping branches with similar behaviors into the same partition, our technique reduces destructive interference in the PHT and decreases branch mispredictions, resulting in an improvement in performance. Our technique requires only edge profiles which can be collected by many popular compilers. Thus, our technique can be readily incorporated into compilers that exploit feedback directed optimization.

We plan to extend our work in PHT partitioning in several ways. We believe that the full potential of our technique has not been demonstrated due to the limited public knowledge of branch predictor implementations in industrial CPUs. We are attempting to reverse-engineer branch predictors of several platforms so that we



(a) (b)

Figure 6.

can better target our optimizations for the characteristics of those microarchitectures. We are adapting our algorithm to microarchitectures that use global history but not branch address in indexing the PHT; this entails controlling branch senses (e.g. `jge` vs. `je`) rather than branch addresses to partition the PHT. By predicting more branches correctly, our technique reduces the amount of energy wasted on wrong-path computations. We are planning to measure the benefits to energy reduction of branch prediction optimizations. We also plan to explore the use of static branch prediction heuristics [1, 3] to avoid the costly profiling step.

Although microarchitectural research in improving branch prediction accuracy has been heavily explored, relatively little work has been done to improve accuracy with software-only techniques. We believe that this approach has great potential to improve performance in current and future microprocessors.

8. Acknowledgements

This research is supported by a grant from the Ministerio de Educación y Ciencia (Spanish Ministry of Education and Science), SB2003-0357.

References

[1] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[3] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, 1997.

[4] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[5] P.-Y. Chang and U. Banerjee. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[6] C.-M. Chen and C.-Ta King. Walk-time address adjustment for improving the accuracy of dynamic branch prediction. *IEEE Transactions on Computers*, 48(5):457–469, May 1999.

[7] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.

[8] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[9] A. N. Eden and T. N. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–80, November 1998.

[10] M. Evers, P.-Y. Chang, and Y. 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*, May 1996.

[11] N. Gloy and M. D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[12] B. Hayes. Differences in optimizing for the Pentium 4 processor vs. the Pentium III processor. *Intel Developer Services, http://www.intel.com/ cd/ ids/ developer/ asmo-na/eng/44010.htm*.

[13] Intel Corporation. Intel Pentium 4 processor optimization. Technical Report Order Number: 248966, Intel Corporation, 2001.

[14] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, pages 67–76, December 2000.

[15] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[16] S. P. Kim and G. S. Tyson. Analyzing the working set characteristics of branch execution. In *Proceedings of the 31rd Annual International Symposium on Microarchitecture*, November 1998.

[17] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 4–13, November 1997.

[18] J. Levon. Oprofile - a system profiler for linux. Technical report, http://oprofile.sourceforge.net/ (Current on September 23, 2004).

[19] S. McFarling. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 183–191, 1988.

[20] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[21] S. McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[22] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[23] M. Milenkovic, A. Milenkovic, and J. Kulick. Microbenchmarks for determining branch predictor organization. *Software Practice and*

Experience, 34:465–487, April 2004.

[24] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000.

[25] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[26] A. Ramírez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proceedings of the 13th International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[27] A. Ramirez, J. L. Larriba-Pey, and M. Valero. The effect of code reordering on branch prediction. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society Press, October 15–19, 2000.

[28] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[29] E. Sprangle, R.S. Chappell, M. Alsup, and Y. 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, June 1997.

[30] Standard Performance Evaluation Corporation. *SPEC CPU 2000, http://www.spec.org/osg/cpu2000*, April 2000.

[31] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.

[32] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of the SIGPLAN'97 Conference on Program Language Design and Implementation*, June 1997.

[33] C. Young and M. D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, May 1999.