Branch Path Re-Aliasing

Daniel A. Jiménez   Calvin Lin
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{djimenez,lin}@cs.utexas.edu

Abstract

Deeper pipelines improve overall performance by allowing more aggressive clock rates, but some performance is lost due to increased branch misprediction penalties. Ironically, with shorter clock periods, the branch predictor has less time to make a prediction and might have to be scaled back to make it faster, which decreases accuracy and reduces the advantage of higher clock rates. We show how the lost accuracy and performance can be reclaimed and even increased, and we show how the predictor can be simplified in the process. We present a new technique, branch path re-aliasing*, that moves complexity off of the critical path for making a prediction and into the compiler. The key component of our technique is a hint bit set in each branch instruction that aids in reducing destructive aliasing; unlike other such techniques, the hint bit is not needed until the less critical update stage, so there is no extra delay in accessing the branch predictor. When augmented with our approach, a simple 2048-entry GAg predictor achieves a misprediction rate of 6.5%, 21% lower than the 8.2% misprediction rate of a more complicated gshare predictor of the same size.*

1 Introduction

The branch predictor for the AMD Athlon microprocessor represents a step backward when compared to its predecessor, the K6. While the K6 has a highly accurate 8K-entry GAs predictor, the Athlon uses a less accurate 2K-entry GAs predictor [6]. This change reduces the delay and real estate costs of the branch predictor and could be one reason why the Athlon is able to achieve an aggressive clock rate of 1.4 GHz. Could AMD have reduced the size of their branch predictor without sacrificing accuracy? This paper argues that the answer is yes.

The above example reveals a larger trend, namely, that the desire for ever higher clock rates has ramifications on branch predictor design. High clock rates motivate smaller branch prediction tables in two ways. First, as clock rates increase and feature sizes shrink, wire delay makes it increasingly expensive—in terms of clock cycles—to access large structures such as branch prediction tables. Thus, higher clock rates favor smaller prediction tables. Second, recent studies have shown that for today's high prediction rates, it is never a good idea to increase predictor latency above 1 cycle in exchange for increased accuracy [12]. Thus, the smaller clock periods imply the need for smaller tables that can be accessed faster.

At the same time, higher clock rates also increase the need for higher branch prediction accuracy. As pipelines become deeper to create less work per cycle, the penalty of a misprediction increases. For example, the Pentium 4 has a 20 stage misprediction pipeline [9], and the next generation Pentium is said to have a 36 stage pipeline. Table 1 shows the clock rates and pipeline depths of several current microprocessors.

The general problem is to design accurate branch predictors while providing low branch predictor latency. There are two possible approaches to this solution. The first is to devise complex mechanisms, such as the Alpha 21264's overriding hybrid predictor [14] and the cascading and overriding approaches proposed by Jimenez, *et al.* [12]. The second approach is to favor simpler branch predictors with minimal complexity. This paper focuses on the second approach, which is a design space that has been largely ignored in the literature, but which has important engineering advantages because of the decreased design time and potentially shorter time to market. In particular this paper proposes a method of building small, simple predictors that have high accuracy.

In this paper we introduce the concept of *branch path re-aliasing*, which enlists the compiler's help in moving important functionality off of the critical path to making a prediction. In particular, our scheme gives the compiler the task of decreasing destructive aliasing and increasing constructive aliasing, so that the branch predictor hardware can be simplified. While other approaches have used the compiler to provide hints which decrease aliasing, our scheme is unique in that the hint bits are kept off the critical path for prediction. Furthermore, our path-profiling algorithm allows us to detect and prevent

Microprocessor	Integer Pipeline Depth	Clock Frequency (MHz)
PowerPC 7400	4	733
HP PA-8700	7	800
Alpha 21264	7	833
AMD Athlon	9	1400
Intel Pentium 4	20	1760

Table 1: Pipeline depth vs. clock rate. This table shows the depth of the integer pipeline and the clock frequency for several modern microprocessors. As clock frequencies become more aggressive, pipelines become deeper and the penalty for a mispredicted branch increases.

path-aliasing as well as *pattern-aliasing* [20].

In our scheme, the compiler uses path profiling information to provide hints to branch instructions so that paths with different outcomes will have histories that map to different locations in the branch predictor's tables. A small, simple predictor is used to make a branch prediction, after which the branch history is updated so that destructive aliasing is decreased. Our scheme places a *branch inversion bit* in each branch instruction to indicate whether the branch outcome should be inverted before it is recorded in the global history register. Even in CPUs with multi-cycle instruction caches, our scheme can deliver a prediction in parallel with the instruction cache access, and only needs to read the hint bit to update the branch predictor.

Our simulations show that a 2048-entry GAg predictor enhanced with branch path re-aliasing has a misprediction rate of 6.5%, 21% lower than the misprediction rate of 8.2% for the same sized, but more complicated, *gshare* predictor, and equivalent to the misprediction rate of a *gshare* predictor with twice the size. We also show that our predictor can improve accuracy for other PHT-based predictors.

This paper makes the following contributions:

- We present *branch path re-aliasing*, a technique in which the compiler reduces destructive aliasing by setting a hint bit in the ISA, thereby allowing dynamic predictors to use smaller tables more effectively.
- We describe an algorithm for using path profiles to set these hint bits.
- We present experimental evidence that branch path re-aliasing allows small branch predictors to achieve greater accuracy than other, slower predictors.
- We show that our technique improves accuracy even for the *agree* predictor, which was designed to convert destructive aliasing into constructive aliasing, and we show that our technique can improve the accuracy of complex predictors, such as the Alpha 21264.

This paper is organized as follows. Section 2 describes the problem of delay in branch predictions. In Section 3, we present background and related work. We describe branch path re-aliasing in detail in Section 4. Then, we present our experimental methodology and results in Section 5, and finally we conclude.

2 Delay in Branch Predictors

This section describes some of the details behind delay in branch predictors, and explains delay in technology-independent terms.

Branch prediction should take at most one cycle, so that the result from a branch prediction for a branch fetched in one cycle can be fed into the fetch stage of the next cycle. Current trends in clock scaling make it difficult to access the branch predictor in a single cycle. It is estimated that, with an aggressive clock period of 8 fan-out-of-four (FO4) inverter delays, the largest PHT that can be read in a single cycle in current and future process technologies has only 1024 entries [12]. Complex predictor organizations and dependences on microarchitectural state further exacerbate delay. Moving to smaller process technologies with less gate delay will not help for two reasons: clock scaling is increasing at a faster rate than the increase in gate switching speeds; and wire delay is increasing relative to gate delay, making it relatively harder to access large SRAM structure, such as branch predictors, in future technologies.

As mentioned in the Introduction, the AMD Athlon's 2K-entry GAs branch predictor [13] is a step back from the previous AMD core, the K6, which used a more accurate 8K-entry GAs [6]. At the time of this writing, the fastest Athlon chip available is clocked at 1.4GHz in $.18\mu$ technology, an estimated clock period estimated of about 11 FO4 delays. The fastest K6 available is clocked at 550MHz in $.25\mu$ technology, an estimated 20 FO4 delays. Even though the Athlon is fabricated in a smaller technology, the clock is relatively more aggressive, since FO4 is a technology-independent measure of delay. Clearly, reducing the length of the critical path to allow for more aggressive clocking was a motivation in the decision to reduce the size of the predictor. Unfortunately, the Athlon's branch predictor is less accurate, and its misprediction penalty is higher. Ideally, the goal is to design a highly accurate branch predictor that has a small delay, as this would allow for both aggressive clocking and a higher rate of instructions per cycle (IPC).

3 Background and Related Work

In this section, we review some of the concepts of branch prediction, and describe related research.

3.1 Branch Prediction

Branch prediction is a form of speculation that breaks control dependences. When a branch is encountered, a branch predictor is consulted to predict whether the branch will be taken or not. Instructions are speculatively fetched and executed down the predicted path. If a misprediction occurs, the speculatively executed instructions are squashed and fetching and execution continue from the correct path. A misprediction incurs a several-cycle cost, which increases with pipeline length.

3.2 Dynamic Branch Prediction

Most modern microprocessors use *two-level adaptive branch prediction*, introduced by Yeh and Patt [25]. Two-level predictors keep track of the recent history of branches in a first-level table of shift registers. A second-level pattern history table (PHT) of two-bit saturating counters is indexed by a combination of a first-level register and branch address. The high bit of the references counter is used as the prediction. When the table is updated, either speculatively or when the branch retires, the corresponding counter is incremented if the branch was taken, decremented otherwise. Thus, the PHT keeps track of the correlation between branch history and outcome.

3.3 GAg and GAs Predictors

Our work focuses on improving the accuracy of GAg branch predictors. Yeh and Patt taxonomize two-level branch predictors using a three-letter naming scheme [26]. The first letter represents how the first level branch history is kept. G means a single global history register is used. The second letter denotes the prediction mechanism: A means that a two-bit saturating counter is used. The third letter indicates how the second level table is indexed; g means a single column of counters is used for all addresses while s means that bits extracted from the branch address are used to select a set of counters, and the set is indexed by the history register. Thus, a GAs predictor selects a set of counters from a PHT using bits from the branch address, and chooses a particular counter from that set using bits from the global history. A GAg predictor uses only the global history to index the PHT.

3.4 Aliasing in Branch Predictors

Recent efforts to improve branch prediction focus primarily on eliminating *aliasing* in variants of two-level adaptive predictors [18, 16, 22, 7], which occurs when two unrelated branches destructively interfere by using the same prediction resources. With a GAg or GAs predictor, two unrelated branches with the same branch histories might lead to different branch outcomes. If these branches map to the same entry in the PHT, they will destructively interfere with one another, leading to poor prediction accuracy. All of the proposed methods for reducing aliasing put some extra complexity in the critical path for branch prediction. In the context of aggressive clock rates, the cost of reducing aliasing must be weighed against the extra delay and complexity of these schemes. Some branch predictors use pattern history information and experience *pattern-aliasing*, while others use path history to correlate with branch outcome, and experience *path-aliasing* [20]. Our idea combines path and pattern history information to reduce both kinds of aliasing.

3.5 Branch Predictors in Current CPUs

Current microprocessors use two-level branch predictors. The following are three notable examples:

- The AMD K6 and K7 (Athlon) processors use GAs predictors [6].
- The HP-PA 8700 uses a 2048-entry GAs with the *agree* mechanism [17, 24]. Rather than correlating with branch outcome, the PHT entries keep track of whether a branch outcome will agree with a bias bit set in the branch instruction. The *agree* mechanism turns destructive interference into constructive interference, increasing accuracy. However, since the

branch instruction opcode must be read and combined with the PHT prediction, the instruction cache is on the critical path for branch prediction.

- The Alpha 21264 core uses a hybrid predictor composed of two two-level predictors [14]: a 4K-entry GAg is indexed by a 12-bit global branch history while a 1K-entry PHT of 3-bit saturating counters is indexed by one of 1024 local 10-bit branch histories. The final branch prediction is chosen by indexing a third predictor that keeps track of the relative accuracies of the two predictors for a particular global history. The Alpha predictor is very accurate; indeed, it is the most accurate of implemented branch predictors that we have observed. However, its implementation complexity comes with a cost. The Alpha branch predictor overrides a less accurate instruction cache line predictor, introducing a single-cycle bubble into the pipeline whenever the two disagree [14].

3.6 Hint Bits in Branch Predictors

Our scheme is one of many that provides hints through the ISA to the branch instruction. One highly successful technique is *branch classification* [5], in which a branch instruction specifies which predictor is best for that branch. Many branches are predicted well with a static prediction; these branches can be "filtered" out of the stream of branches that are allowed to update the PHT, thus reducing aliasing. A version of the *agree* predictor predicts whether a branch outcome will agree with a bias bit set in the branch instruction [22].

Unfortunately, for any of these techniques to work, the branch instruction has to have been at least partially decoded before the branch prediction can be made. These techniques will not be feasible in aggressively clocked CPUs with multi-cycle instruction cache latencies, since the predictor is in series with the instruction cache. Our predictor is different; it uses a hint bit in the branch instruction, but the hint is not needed until the branch predictor is updated.

3.7 Compiler-Assisted Branch Prediction

Several schemes use the compiler to assist in branch prediction. The variable length path branch predictor [23] encodes profiling information in branch instructions. This information guides a dynamic predictor, increasing accuracy by choosing the best history length and hash function to form an index into the PHT. Jimenez *et al.* propose changing the ISA to allow each branch instruction to represent a Boolean formula chosen at compile-time used to guide branch prediction; this work also considers branch predictor delay [11]. August *et al.* [1] propose placing hint bits in each branch instruction that tell a dynamic predictor what kind of state to examine to make a prediction. Each of these schemes places the contents of the branch instruction on the critical path for branch prediction, which will cause problems in CPUs with multi-cycle instruction caches.

Other techniques use the compiler to help with branch prediction without changing the prediction mechanism. For instance, *branch alignment* [4, 28] increases instruction fetch bandwidth by minimizing the number of taken branches in a program. *Static correlated branch prediction* [27, 29] is another optimization that introduces duplicate basic blocks, encoding in the program counter information about the path taken to reach a particular static branch and increasing the accuracy of static prediction.

Our idea works on the same principle as *branch allocation* [15]. Branch allocation uses the working set characteristics of branches to explicitly assign each conditional branch a set of PHT resources at compile time. The analysis forms a conflict graph between branches and uses a technique similar to register allocation to allocate PHT resources among branches such that destructive aliasing is reduced. With branch allocation, the process of reading the contents of the branch instruction is on the critical path to making a prediction. Also, branch allocation sets aside many bits in the branch instruction, requiring a significant change to the ISA. Our predictor has neither of these undesirable properties.

4 Branch Path Re-Aliasing

In this section, we describe the problem of history aliasing, which is common to many two-level branch predictors. We then describe a technique that increases accuracy by decreasing aliasing.

4.1 Path and Outcome Histories

Branch path re-aliasing gives the compiler explicit control over how paths through the program are mapped to PHT entries. Branch outcomes are highly correlated both with path and pattern histories [20, 27, 23]. Pattern histories are easier to use than path histories since they require recording only a single bit for each branch. However, pattern histories are highly susceptible to aliasing, both between different static branches and within the same branch. That is, several different paths correlated with different branch behaviors may all induce the same pattern history, leading to destructive aliasing. Our optimization *re-aliases* pattern histories to better reflect path histories, improving accuracy by decreasing destructive aliasing.

4.2 History Aliasing in a Global Predictor

Several types of aliasing have been identified in branch predictors [19]. Our focus is on *conflict aliasing*. Consider a GAg predictor, which consists of a PHT indexed by a global history register. Two different paths in the program may coincidentally lead to the same global history, even though the code being executed is unrelated. In this case, the same PHT entry will be used for both branches, but the prediction will not correlate highly with the outcome of either. Thus, the branch predictor will have poor accuracy for these branches.

4.3 Our Solution: Branch Path Re-Aliasing

Our approach to solving the history aliasing problem is to insert a hint bit into each instruction that tells the branch history update mechanism whether or not to invert the branch outcome before recording it in the history register. We choose the hint bits, which we call *inversion bits*, such that paths leading to branches with opposite outcomes will have different histories. Essentially, by changing the way paths alias one another in the PHT, we reduce destructive aliasing.

We introduce our idea by modifying the simplest possible two-level branch predictor: the GAg. A global history register is used to index a PHT of two-bit saturating counters, from which the prediction is directly read. Once the prediction is read and made available to the fetch engine, the critical time to make a prediction is over, so the predictor is no slower than a normal GAg. The branch prediction is then used to speculatively update the global history register, which is backed up and corrected after a misprediction. With branch path re-aliasing, the difference comes in the how the history register is updated. Each branch instruction encodes an inversion bit. If this bit is set, then the branch outcome is inverted before it is recorded in the global history register. In short, the value recorded in the history register is the exclusive-OR of the inversion bit and the branch outcome.

At first glance, it might seem that this technique could be implemented by simply changing branch senses and reordering code; however, this transformation would be at odds with techniques such as branch alignment [4] that seek to minimize the number of taken branches to increase fetch bandwidth. Branch alignment can increase performance, even though it may decrease prediction accuracy [21]. Our technique can nicely complement branch alignment by decreasing the destructive aliasing introduced by alignment.

4.3.1 Path Profiles

Path profiling collects information on the paths taken during the execution of a program [2]. Branch path re-aliasing uses path profiles to determine which branches should have their inversion bits set. For a history length of N, i.e., a GAg with an N-bit history, each path profile stores the following information for a path p:

1. The addresses of the last N branches encountered.

2. The outcomes (*taken* or *not taken*) of the last N branches encountered.

3. $freq(p)$, the frequency with which this path was executed.

4. $ntaken(p)$, the number of times this path led to a taken branch.

4.3.2 Algorithm

Once the path profiles have been collected, we use a two-phase algorithm to set inversion bits. In the first phase, the algorithm tries to map paths to PHT entries by setting the inversion bits of certain branches, causing constructive aliasing between paths that agree on branch outcome and choosing different PHT entries for paths with different outcomes. In the second phase, a hill-climbing heuristic sets the inversion bits of each branch sense one at a time, keeping the set of inversion bits that maximizes a fitness function based on the estimated amount of constructive and destructive interference. The details the two phases are as follows:

1. The first phase of the algorithm maps paths to PHT entries by inverting or not inverting branches along the path. The algorithm considers each path profile in descending order of frequency. For each profile p, the algorithm looks for an entry i in the PHT to which similarly biased paths are mapped, or to which no paths are mapped at all. If one is found, then path p is mapped to PHT entry i; otherwise, the inversion bits of the path p are left the same.

2. The second phase considers each static branch, choosing the inversion hint bit for that branch that maximizes a fitness function over all branches. Let P_i be the set of paths all mapped to PHT entry i,

and let n be the history length, so that there are 2^n counters in the PHT. Let a Boolean $taken_i$ be the aggregate bias (i.e. true for *taken* or false for *not taken*) of all the paths mapped to PHT entry i, i.e., $taken_i$ is true if and only if:

$$\sum_{p \in P_i} ntaken(p) \geq \frac{1}{2} \sum_{p \in P_i} freq(p)$$

In other words, $taken_i$ is true if and only if all the paths mapped to PHT entry i lead to taken branches at least half the time. For a path p, let a Boolean $bias_p$ be true if and only if $ntaken(p) \geq freq(p)/2$, i.e., $bias_p$ is the bias of an individual path. Then the value of the fitness function is:

$$\sum_{0 \leq i \leq 2^h} \sum_{p \in P_i} \begin{cases} freq(p) & \text{if } taken_i = bias_p \\ -freq(p) & \text{otherwise} \end{cases}$$

Each path is mapped to a particular PHT entry. Intuitively, the fitness function is the sum, over all paths, of the frequencies of paths mapped to PHT entries with the same bias, minus the frequencies of paths mapped to PHT entries with different bias. The higher the fitness function, the more constructive and less destructive interference there is.

4.4 Implementing Inversion Bits

An important consideration for branch path re-aliasing is the representation of the inversion bits. Each branch instruction encodes an inversion bit, which is reasonable since several existing ISAs already dedicate one or two bits in each branch instruction to managing branch prediction. For example, the HP/PA-RISC architecture allows each branch to encode a bias bit [17], which is used either for static or *agree* branch prediction. The Pentium 4 microprocessor extends the IA-32 instruction set to include branch hints [10]. The IA-64 architecture encodes several hint bits in branch instructions [8]. These extra bits in the ISA could be re-used to represent inversion bits. Old binaries would still run with reduced performance, and newer ones could be optimized to use the inversion bits for branch path re-aliasing.

5 Experimental Results

In this section, we give the results of branch path re-aliasing on the SPEC 2000 integer benchmarks, measuring the decrease in misprediction rates on several branch predictors. We show that our optimization also helps more complex *agree* and hybrid predictors. Finally, we measure the decrease in aliasing responsible for the improved accuracy.

5.1 Predictor Simulation Methodology

We use the 12 SPEC 2000 integer benchmarks running under SimpleScalar/Alpha [3] to collect traces. For each benchmark, we gather traces giving the branch address and outcome for 300 million branches for both `train` and `ref` inputs. Each benchmark executes over one billion instructions before the simulation ends.

We use the `train` inputs for collecting the path profiles, and we use the `ref` inputs to evaluate the accuracy of the predictors. We use traces to gather our path profiles. This method is costly, but there are techniques in the literature that would easily make this task much more efficient, e.g. the efficient algorithm of Young [30], which gathers bounded-length paths with both forward and backward edges, or the forward-path profiling of Ball and Larus [2]. We consider path profiles with history lengths of 8 to 15.

We use branch path re-aliasing to decrease the misprediction rates of three dynamic branch predictors: GAg, an *agree* predictor, and a hybrid predictor. We compare our improved predictors with several other predictors. We first tune each predictor for optimal history length using the traces collected with the `train` inputs.

5.2 Algorithm Implementation

We measure misprediction rates using a trace-driven simulation program. For our simulations, we use a 733MHz Pentium III that reads compressed traces from an NFS server. On this machine, the branch path re-aliasing algorithm takes from 5 to 30 minutes, depending on the history length, number of paths in the program, and compression ratios of the traces. We did not pay particular attention to the efficiency of the program, using C++ and STL for rapid development. However, we are confident that a production version of branch path re-aliasing using Young's path profiling algorithm would be reasonably quick.

5.3 Simple Two-Level Predictors

Figure 1 compares our basic scheme, GAg with branch path re-aliasing, against three simple two-level predictors: GAg, GAs, and *gshare*. The graph shows misprediction rates for hardware budgets ranging from 256 to 8K bytes. At all hardware budgets, our basic scheme achieves the lowest misprediction rate. The graph does

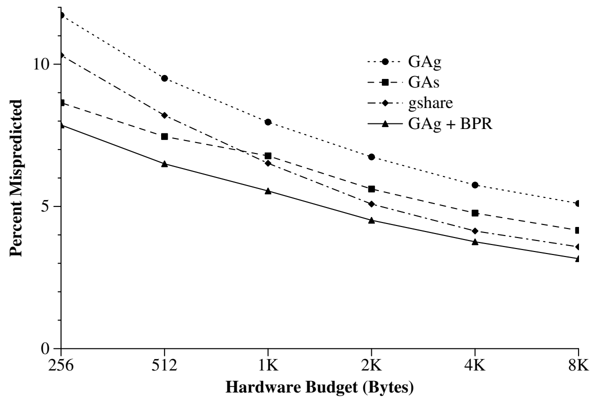


Figure 1: Branch misprediction rates on the SPEC 2000 integer benchmarks. Branch path re-aliasing is able to improve GAg beyond the accuracies of GAs and *gshare* while keeping the critical path for prediction short.

not show, of course, that our scheme allows faster clocking by removing work from the critical path. For a branch predictor with 2K-entries, the same hardware budget used in the AMD Athlon, branch path re-aliasing reduces the misprediction of GAg by 32%, from 9.5% down to 6.5%. The misprediction rates for a 2K-entry GAs and *gshare* are 7.5% and 8.2%, respectively; for 2K-entries, our basic predictor sees misprediction rates that are lower than GAs and *gshare* by 13% and 21%, respectively.

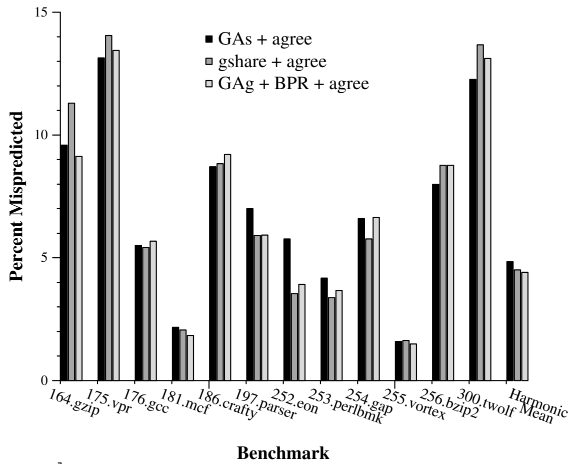
To see how these numbers might be used to design future predictors, suppose the microarchitects of a CPU core that uses a 4K-entry GAs predictor decided it was necessary to shrink the branch predictor to 2K entries to allow for more aggressive clocking. Our simulations show that the misprediction rate would increase by 12%, from 6.7% to 7.5%. Instead, the microarchitects could replace the 4K-entry GAs with a 2K-entry GAg and provide inversion bits. Branch path re-aliasing could achieve a misprediction rate of 6.5%, decreasing the misprediction rate of the larger predictor by 3%.

5.4 More Complex Predictors

We have argued that high-latency, complex predictors will become less feasible as clock rates increase and pipelines get longer. Nevertheless, some CPU designs will continue to keep shorter pipelines and less aggressive clock rates. Even with more complex predictors, branch path re-aliasing offers higher accuracy.

5.4.1 Agree Predictors

The *agree* predictor achieves increased accuracy by turning the destructive aliasing of a normal PHT predictor into constructive aliasing. Rather than predicting the outcome of a branch, the PHT is used to predict whether the outcome will agree with a bias bit. Still, there is a different kind of destructive aliasing to which *agree* predictors are susceptible. Instead of paths that lead to taken and not taken branches colliding in the PHT, we may have paths that lead to agreement and disagreement with the bias bit aliasing each other. We modify the branch path re-aliasing algorithm to reduce aliasing in a GAg-based *agree* predictor that uses bias bits set in each branch instruction. Instead of keeping track of the taken/not taken bias of a particular path, the new algorithm keeps track of the agree/disagree bias of a path. That is, for each PHT entry, the algorithm determines whether each path leading to that entry usually agrees or disagrees with the corresponding bias bit.



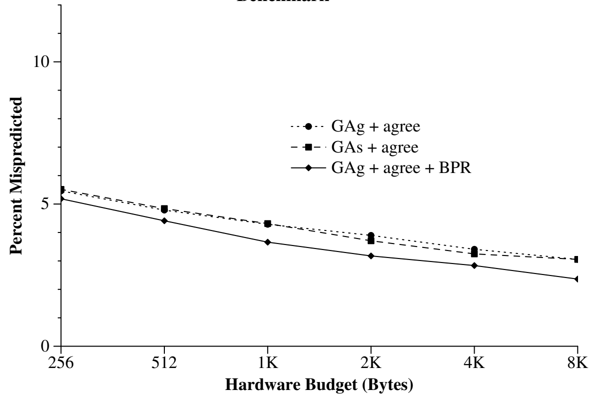


Figure 2: Branch misprediction rates on each SPEC 2000 integer benchmarks for *agree* predictors.

Figure 2 shows harmonic means of misprediction rates for several hardware budgets, as well as the misprediction rates on each SPEC integer benchmark means for 2K-entry GAs, *gshare*, and GAg predictors with branch path re-aliasing, each using the *agree* mechanism. These predictors use the same size table as the *agree* predictor of recent HP-PA/RISC cores such as the 8700 [17, 24]. Branch path re-aliasing achieves the lowest harmonic mean misprediction rate of 4.4%, compared with 4.8% for GAs with *agree* and 4.5% for *gshare* with *agree*.

5.4.2 Hybrid Predictors

One of the components of the Alpha 21264 hybrid branch predictor is a 4K-entry GAg predictor. The choice predictor, which predicts whether the global or per-branch component will be more accurate, is also a 4K-entry table of 2-bit counters indexed by the global branch history. We modified the branch path re-aliasing program to measure the bias of a particular branch to be predicted better by a global or per-branch predictor by tracking the misprediction rates of both prediction components. We modified the fitness function to take into account both taken/not taken and global/per-branch biases. This way, aliasing is reduced both in the global PHT as well as in the choice table.

We simulate the unmodified Alpha 21264 hybrid predictor, as well as a version of the Alpha predictor augmented with branch path re-aliasing. We allow the global and chooser PHTs to range in size from 256 to 32K entries, scaling the per-branch table of histories and PHT with 1/4 the entries as the global PHT, yielding a sequence of Alpha-like predictors at increasing hardware budgets. Figure 3 shows a plot of the harmonic means of misprediction rates as a function of hardware budget for the hybrid predictors as well as two *agree* predictors, one with branch path re-aliasing. Figure 3 also shows a bar graph for the 4K-entry global PHT versions of the hybrid predictors, using the same configuration as the Alpha 21264 predictor. The bargraph shows a 16K-entry *agree* predictor using branch path re-aliasing. This *agree* predictor uses about the same hardware budget (4096 bytes) available to the Alpha 21264 (3712 bytes). Using branch path re-aliasing with the hybrid predictor reduces the harmonic mean of the misprediction rate by 10%, from 3.1% to 2.8%. Using GAg with branch path re-aliasing and the *agree* mechanism, the misprediction rate is 3.0%, slightly better than the original hybrid predictor and with reduced complexity.

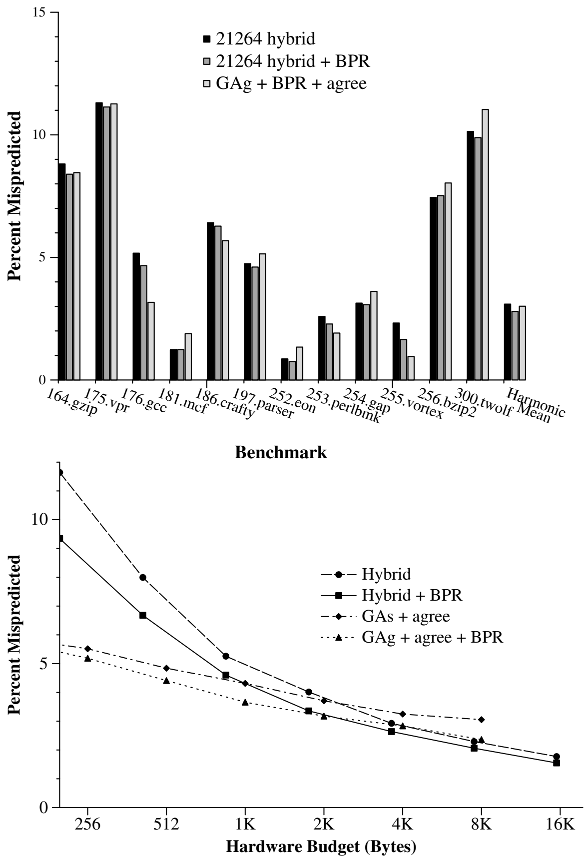


Figure 3: Branch misprediction rates on each SPEC 2000 integer benchmarks for hybrid and *agree* predictors.

5.5 Aliasing Rates

The purpose of branch path re-aliasing is to reduce destructive aliasing in the PHT for a GAg predictor. In our experiments, we model a "de-aliased" predictor, i.e., a predictor where different paths cannot alias the same PHT entries. We use this predictor to measure three kinds of aliasing [19]:

- Destructive aliasing occurs when PHT aliasing leads to a misprediction in GAg where the de-aliased predictor has no misprediction.
- Constructive aliasing occurs when PHT aliasing leads to a correct prediction where the de-aliased predictor mispredicts.
- Harmless aliasing occurs when aliasing in the PHT has no effect on whether or not a prediction is correct.

Note that these cases are mutually exclusive and account for all aliasing in the PHT. Figure 4 shows these different types of aliasing rates in a 2K-entry GAg predictor for the SPEC 2000 integer benchmarks, before and after applying branch path re-aliasing. The harmonic mean of the destructive aliasing rate is reduced by 21%, from 6.1% before re-aliasing to 4.8% after re-aliasing. Constructive aliasing is also reduced slightly, from 0.41% to 0.31%. Total aliasing is reduced by 48%, from 18.3% to 9.5%.

On `181.mcf`, re-aliasing reduces destructive aliasing by 30%, from 16.6% down to 11.5%, explaining the 64% decrease in the misprediction rate, from 7.9% for GAg down to 2.8% for GAg with branch path re-aliasing.

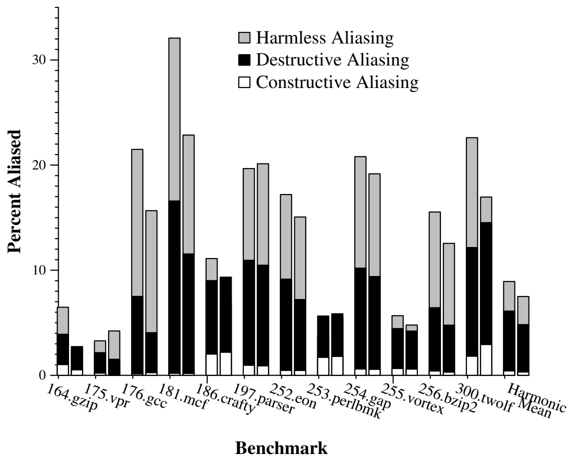


Figure 4: Branch aliasing rates on the SPEC 2000 integer benchmarks. For each benchmark, the left bar shows the aliasing rates before applying re-aliasing, and the right bar shows the aliasing rates after the transformation is applied.

6 Conclusion

Branch path re-aliasing is a new branch prediction technique that improves accuracy. By using path profiles to map paths leading to different outcomes to different PHT locations and paths with similar outcomes to the same PHT locations, re-aliasing decreases destructive aliasing.

An advantage of our idea is its simplicity and low delay. The time to access the predictor is limited only by the time to access the PHT; there is no other logic on the critical path. Moreover, variations of our technique improves accuracy in complex predictor organizations such as *agree* and hybrid predictors.

Our approach fits in with the general trend towards moving more work out of the processor and into the compiler. By making prediction simpler without reducing accuracy, we can enjoy the benefits both of high IPC and high clock rates.

References

[1] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, February 1997.

[2] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[3] Doug Burger and Todd M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[4] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the Sixth 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] Keith Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.

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

[8] Linley Gwennap. Ia-64: A parallel instruction set. *Microprocessor Report*, 13(7):6–11, May 1998.

[9] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The

microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

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

[11] Daniel A. Jiménez, Heather L. Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[12] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 67–76, December 2000.

[13] Andreas Kaiser. K7 branch prediction. *http://www.s.netic.de/ak/k7doc.pdf*, December 1999.

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

[15] Sangwook P. Kim and Gary S. Tyson. Analyzing the working set characteristics of branch execution. In *Proceedings of the* 31^{rd} *Annual International Symposium on Microarchitecture*, November 1998.

[16] 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*, November 1997.

[17] Gregg Lesartre and Doug Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *42nd IEEE International Computer Conference*, February 1997.

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

[19] 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*, June 1997.

[20] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, December 1995.

[21] Alex Ramirez, Josep L. Larriba-Pey, and Mateo 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.

[22] 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*, June 1997.

[23] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[24] Li C. Tsai. A 1GHz PA-RISC processor. In *Proceedings of the 2001 International Solid State Circuits Conference (ISSCC)*, February 2001.

[25] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the* 24^{th} *ACM/IEEE Int'l Symposium on Microarchitecture*, November 1991.

[26] T.-Y. Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[27] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of ASPLOS VI*, pages 232–241, 1994.

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

[29] Cliff Young and Michael D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, May 1999.

[30] Reginald Clifford Young. *Path-based Compilation*. PhD thesis, Harvard University, Department of Computer Science, January 1998.