The Effect of Code Reordering on Branch Prediction *

Alex Ramirez, Josep L. Larriba-Pey and Mateo Valero
Universitat Politecnica de Catalunya
Jordi Girona 1–3, D6
08034 Barcelona (Spain)
{aramirez,larri,mateo}@ac.upc.es

Abstract

Branch prediction accuracy is a very important factor for superscalar processor performance. The ability to predict the outcome of a branch allows the processor to effectively use a large instruction window, and extract a larger amount of Instruction Level Parallelism (ILP).

In this paper we will examine the effect of code layout optimizations on branch prediction accuracy and final processor performance. These code reordering techniques align branches so that they tend to be not taken, achieving better instruction cache performance and increasing the fetch bandwidth. Here we focus on how these optimizations affect both static and dynamic branch prediction.

Code reordering mainly increases the number of not taken branches, which benefits simple static predictors, which reach over 80% prediction accuracy with optimized codes. This branch direction change produces two effects on dynamic branch prediction: on the positive side, trades negative interference for neutral or positive interference in the prediction tables; on the negative side, it causes a worse distribution of the Branch History Register (BHR), causing many possible history values to be unused.

Our results show that code reordering reduces negative Pattern History Table (PHT) interference, increasing branch prediction accuracy on small branch predictors. For example, a 0.5KB gshare improves from 91.4% to 93.6%, and a 0.4KB gskew predictor from 93.5% to 94.4%. For larger history lengths, the large amount of not taken branches can degrade predictor performance on dealiased schemes, like the 16KB agree predictor which goes from 96.2% to 95.8%.

But processor performance not only depends on branch prediction accuracy. Layout optimized codes have much better instruction cache performance, and wider fetch bandwidth. Our results show that when all three factors are considered together, code reordering techniques always improve processor performance. For example, performance still increases by 8% with an agree predictor, which loses prediction accuracy, and it increases by 9% with a gshare predictor, which increases prediction accuracy.

*This work was supported by the Ministry of Education and Science of Spain under contract TIC-0511/98 and by CEPBA. Alex Ramirez is also supported by Generalitat de Catalunya grant 1998FI-003060-26.

1. Introduction

Fetch performance broadly depends on three factors: the number of instruction cache misses, the width of instructions fetched each cycle, and the branch prediction accuracy. The first two factors determine the speed at which instructions are provided to the processor, the third determines the quality of the instruction provided, that is, how many instructions will be provided between instruction window squashes, limiting the amount of ILP that the processor is able to exploit.

Code reordering techniques are a known approach to the first two factors. The number of instruction cache misses depends on the code layout, by mapping the routines in a program so that they do not conflict with each other, we can reduce the number of cache misses by almost an order of magnitude [17, 7, 6]. By aligning basic blocks so that they execute sequentially, we can further increase spatial locality increasing both cache performance and fetch bandwidth [8, 17, 25, 19]. The third factor has motivated the search of more accurate branch predictors.

The performance loss due to branch instructions was first approached with static branch predictors, which always predict the same outcome for a given branch. This prediction was obtained either using very simple heuristics [23], static analysis [1], or profile information [5, 4].

The accuracy of static branch predictors can be increased using code transformations, which usually imply code replication [14, 27, 9, 13, 16], and branch alignment [3]. This branch alignment is nothing but a code reordering optimization which targets an increase in the static branch prediction accuracy: knowing the branch outcome, it is aligned to follow the heuristic implemented by the static predictor.

As the transistor budget in the processor increased, branch prediction moved to the more accurate dynamic branch predictors. These store the recent branch behavior,

0-7695-0622-4/00 $10.00 © 2000 IEEE

and lookup the data each time the branch executes to produce a direction prediction [23, 26].

But the size of these dynamic tables is limited, and sometimes two different branches end up sharing the same PHT entry. This is called prediction table interference, and is the main cause for decreased prediction accuracy [28].

Dynamic prediction tables can be organized in a clever way to reduce prediction table interference, leading to the recently proposed dealiased schemes [10, 12, 24].

In this work we examine the effect on branch prediction accuracy of the code reordering optimizations which target the instruction cache. We examine the interaction of these optimizations with both static and dynamic branch predictors using the Software Trace Cache layout optimization [19].

The main effect of these code reordering techniques is an increase in the fraction of not taken branches. This increase favors static predictors which predict that all branches will be not taken, or that all forward branches will be not taken, going from 60% to over 80% prediction accuracy.

Such an increase in the number of not taken branches also favors neutral or positive interference, because branches sharing the same PHT entry are likely to exhibit the same behavior, and will update the counter in the same direction. This interference reduction is specially significant in small predictors, and increases accuracy in a 0.5KB gshare from 91.4% to 93.6%, and a 0.4KB gskew predictor from 93.5% to 94.4%.

As larger tables are used, prediction table interference naturally decreases, reducing the benefits of an optimized layout. As history length increases, the large number of not taken branches produces a worse distribution of the BHR values, increasing interference in the dealiased predictors. The negative BHR effect decreases performance in mid to large sized dealiased predictors, like the 16KB agree predictor which goes from 96.2% to 95.8%.

Finally, we show results on the overall processor performance because not only branch prediction accuracy affects IPC. Instruction cache performance and fetch bandwidth also play an important role, and more than compensate for the possible degradation in prediction accuracy. Processor performance still increases by 8% with an agree predictor w/out filtering (which loses prediction accuracy), and increases by 9% with a gshare predictor (which increases prediction accuracy).

1.1. Simulation setup

All the results in the paper were obtained using a simulator derived from the SimpleScalar 3.0 tool set [2]. We run most of the SPECint95 benchmarks plus the PostgreSQL 6.3 database system running a subset of the TPC-D queries. All programs were compiled statically and with -O4 optimization level using Compaq's C compiler.

Benchmark	Train	Test
go	UNUSED: Profile data unavailable, crashed with pixie and ATOM	
m88ksim	train	test
gcc	train	cccp.i
compress	UNUSED: Considered too small to be representative, has too few branches	
li	train	test
ijpeg	vigo.ppm	specmun.ppm
perl	UNUSED: Simulation time too long.	
vortrex	train	test
postgres	$Q_{3,4,5,6,9,15}$	$Q_{2,3,4,6,11,12,13,14,15,17}$

Table 1. Simulated benchmarks and their training and test inputs.

Table 1 shows the six benchmarks used and the input sets used to obtain the profile information and for testing, and the reasons for not including the remaining 3 SPECint95 codes. All simulations were run to completion. All figures in the paper present the arithmetic average of all executed benchmarks, where all benchmarks have the same weight.

In order to simulate the optimized code layout we generate an address translation table using the Software Trace Cache algorithm [19] and feed the simulator with translated PC's and recomputed branch outcomes.

1.2. Paper structure

The rest of this paper is structured as follows: In Section 2 we present previous related work regarding both code layout optimizations and branch prediction, we also describe the dynamic branch prediction schemes used in the paper. Section 3 examines the effect of code layout optimizations on static branch prediction accuracy. Section 4 does the same for dynamic branch predictors, including dealiased prediction schemes. In Section 5 we measure not only branch prediction accuracy, but overall processor performance in order to account for all the effects of code reordering, both positive and negative. Finally, in Section 6 we summarize the influence of code layout optimizations on branch prediction and present our conclusions.

2. Related work

We can classify related work in two main groups: code layout optimization techniques, and branch prediction techniques.

Code layout optimizations usually target a better utilization of the instruction cache, and use profile data or heuristics to lay out the routines in a program [17, 7, 6], and the basic blocks in a routine [8, 17, 25, 19] to minimize the number of conflict misses. Reducing the number of conflict misses in the instruction cache, code reordering increases fetch performance, and overall processor performance. The use of both routine placement and basic block reordering

can also increase the effective fetch bandwidth provided by increasing code sequentiality (reducing the number of taken branches). Both factors prove important at increasing the fetch performance, as shown in [19, 18].

Code layout optimizations have also been used to increase the static branch prediction accuracy, using profile data [5, 4] or complex static analysis techniques [1] to predict the branch direction, and then align the branch so that it follows a more simple heuristic [3], like making all branches usually taken (or usually not taken), or aligning branches so that only a forward branch is usually not taken [23]. In this work we examine how code layout optimizations targeting the fetch engine affect both static and dynamic branch prediction.

There have been other code transformations proposed to improve static branch prediction accuracy, usually implying code replication [14, 27, 9, 13, 16]. These code transformations are beyond the scope of this work.

Basic branch prediction techniques can also be broadly classified in three groups: static, semi-static, and dynamic predictors. Static prediction techniques are based solely on static analysis and simple prediction strategies, and always predict the same outcome for a given branch. Semi-static branch predictors improve on static techniques by using profile data obtained at run-time to replace the static analysis and heuristics used, but still predict always the same outcome for a given branch. The more accurate dynamic branch predictors store this run-time information in dynamic tables, and lookup this data every time the branch is executed to make a direction prediction. The different dynamic branch predictors differ in the way they store the past behavior of a branch.

The Software Trace Cache

The code layout optimization used in this paper is the Software Trace Cache (STC) [?]. The STC maps basic blocks so that sequentially executed basic blocks tend to be in consecutive memory positions, building basic block chains than may span multiple routines. The generated chains are then mapped in memory trying to minimize conflicts among them, by mapping two popular chains next to each other, and mapping the most heavily used chains to a specially reserved area of the instruction cache that we call the Conflict Free Area (CFA).

The chain mapping algorithm should have little or no influence on the branch prediction mechanism, only the basic block chaining is relevant for that purpose. The results obtained in this paper should be valid for any other code layout optimization which aligns branches towards their not-taken target.

Two-level adaptive predictors

The more simple dynamic branch predictor (the bimodal branch predictor [23]) simply keeps a saturating two-bit counter for each branch, increasing the counter if the branch is taken, and decreasing the counter if it is not taken. The branch is predicted to behave as the high bit of the counter says (taken if it is 1, not taken otherwise).

But a branch outcome not only depends on the branch itself, it also depends on the outcomes of the previously executed branches, and on the past outcomes of the same branch.

As shown in Figure 1, two-level adaptive branch predictors [26] keep two levels of data about the branch behavior. The Level 1 table keeps information about the past branch outcomes. These table can store the outcomes of all branches in a single register (global history, named PA$p, s, g$, shown in Figure 1.a), or it can have a separate register for each branch (private or self history, named GA$p, s, g$, shown in Figure 1.b). The Level 1 table is usually referred to as the Branch History Register (BHR). The BHR is used to index into the Level 2 table, composed of two-bit saturating counters managed as in the bimodal predictor. The Level 2 table is usually referred to as the Pattern History Table (PHT).

By storing data this way, any given entry in the PHT corresponds to a branch address in a given history situation, which allows the predictor to make a more informed decision, achieving higher accuracy.

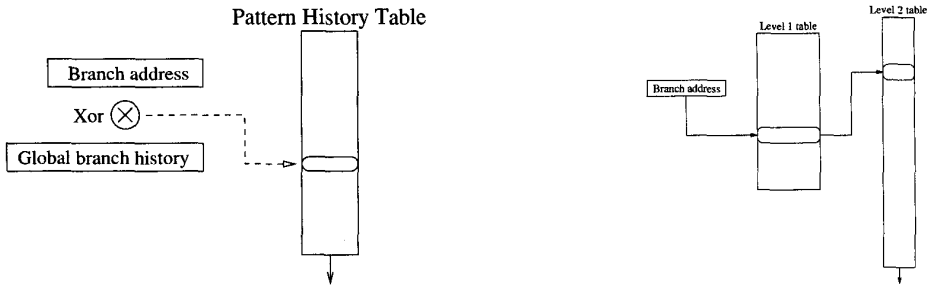
It is possible to improve the Level 2 indexing function by using a hash function of the branch address and the BHR, like an XOR [11]. This function distributed branches in the PHT in a better way, increasing the accuracy of global history predictors. The resulting scheme (shown in Figure 1.a) is the gshare branch predictor.

Dealiased predictors

Two-level adaptive branch predictors distribute data so that each branch has a separate PHT entry for each different history situation. But the prediction tables are finite, and sometimes two different branches end up sharing the same PHT entry.

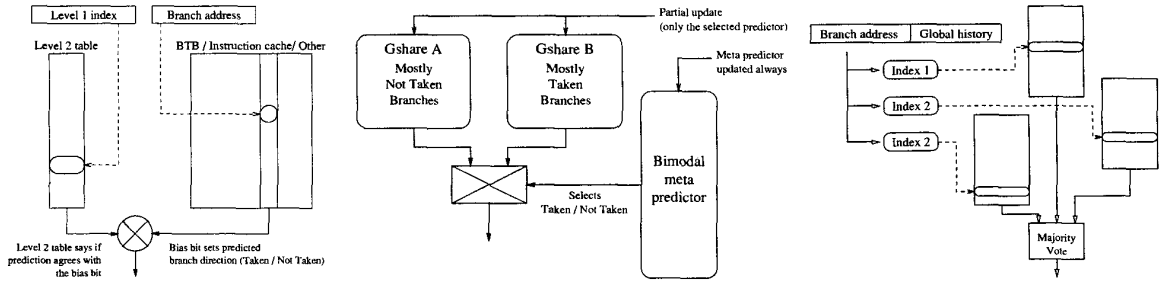
We classify PHT interference in three types: when the conflict does not change the 2-bit counter value, we talk about neutral interference; if the changed counter value produces a correct prediction where there would have been a misprediction, we talk about positive interference; if the conflict causes a misprediction when the old counter was correct, we talk about negative interference. Negative interference happens more often than positive interference, and is the main cause of decreased prediction accuracy [28, 20].

Dealiased branch predictors reduce negative PHT interference by changing the way they store data in the predic-



(a) Gshare predictor (global history) (b) PAg predictor (private history)

Figure 1. Two-level adaptive branch predictors store data in two separate tables, using the first table to index into the second level.



(a) Agree predictor (b) Bi-mode predictor (c) Gskew predictor

Figure 2. Dealiased branch prediction schemes.

tion tables.

Figure 2.a shows the agree prediction scheme [24]. The agree predictor adds an extra bit of information associated to each branch into the BTB/instruction cache: the *bias* bit. This bit predicts the branch direction. The meaning of the PHT counter changes: the two-bit counter now predicts if the branch behavior will *agree* with the bias bit, or not. This allows two branches with opposite behavior (a mostly taken and a mostly not taken branch) to use the same PHT entry, without creating a negative conflict because both branches will push the counter towards the agree position, being the bias bit what differentiates them.

The bi-mode branch predictor [10] (shown in Figure 2.b) is based on the same principle as the agree predictor: separating branches among usually taken and usually not taken sub-streams. The bi-mode predictor uses a separate gshare component to keep track of each sub-stream, avoiding interference among them, and uses a bimodal branch predictor to classify a branch into each sub-stream. Interference among the two sub-streams is avoided because each branch only updates the gshare which keeps track of its sub-stream.

The gskew branch predictor [12, 22] (Figure 2.c) is based on the fact that most aliasing in the prediction tables is due to conflict aliasing, not capacity problems. Derived from the skew-associative caches [21], the gskew predictor stores branches in three separate tables, which are accessed with three different indexes. If a branch data is aliased in one of the tables, it is expected that it will not be so in the other two, obtaining a correct prediction with a majority vote.

Code reordering techniques are known to improve the instruction cache miss rate and the fetch bandwidth. Next, we examine how they interact with the third factor in fetch performance: the branch prediction mechanism.

3. Effect on static prediction

In this section we will examine the prediction accuracy that some simple static branch prediction schemes achieve for the examined benchmarks. The static strategies examined are: predict that all branches will be taken, predict that all branches will be not taken, predict that backwards branches will be taken and forward branches will not, and predict that a branch will always take its most usual direction based on profile information.

Figure 3 shows the branch prediction accuracy of some simple static branch prediction strategies (always taken, always not taken, backwards taken forward not taken) and the profile based predictor for both the original code layout and the compiler optimized layouts. For the optimized layout, we show results for the same input set used for

training (self-optimized) and for a different input set (cross-optimized). The prediction accuracy of an 8KB Gshare predictor is shown for comparison purposes.

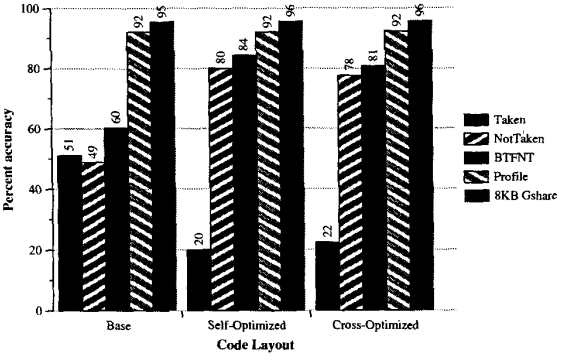


Figure 3. Static branch prediction accuracy for the original and optimized code layouts (self and cross trained).

The simple static prediction approaches prove quite useless for the baseline code layout with near 50% prediction accuracy, only the BTFNT predictor reaches 60%, and doesn't go under 50% for any of the studied benchmarks (individual benchmark results not shown). On the other hand, the profile static predictor proves very accurate, predicting correctly over 90% of the branches. This shows that branches can be predicted statically, but not with this simple strategies.

We optimize the code layout using the Software Trace Cache (STC) algorithm [19], which targets an increase in the sequentiality of the code, that is, it reorders basic blocks so that branches tend to be not taken.

Once we have optimized the code layout, the static branch prediction accuracy changes dramatically. The Not Taken and the BTFNT predictors now predict correctly over 80% of the branches, losing some accuracy in the cross-trained test. This 80% prediction accuracy shows that static branch prediction can be very accurate for these optimized code layouts; but it is still much lower than what can be achieved with modern two level adaptive branch predictors like the Gshare.

To gain further insight on this high predictability of optimized binaries, we explore in depth the changes in branch behavior introduced by the code layout optimization. Figure 4.a shows a classification of all dynamic branches by the percentage of times they are taken or not taken for both the original and the optimized code layouts. Branches to the left of the plot are always not taken, while branches to the right are always taken.

Examining the branch classification for the original code layout, we observe that 36% of the branches are always

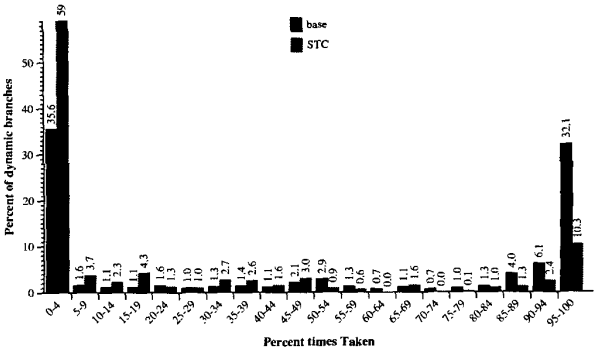


Figure 4. The use of optimized code layouts reverses branch direction, so that they tend to be usually not taken.

not taken, while 32% are always taken. The rest of the branches are evenly spread across all taken percent values, with a slightly higher peak for branches that are 50% taken. This explains the low prediction accuracy obtained, because branches do not seem to follow such simple behavior rules.

By optimizing the code layout, we can reverse the direction of those branches which are taken more than 50% of the times. This way, a branch which was taken 80% of the times will now only be taken 20% of the times.

The classification for the optimized code layout shows that we were quite successful at reversing the branch direction for those usually taken branches. The fraction of always taken branches is reduced from 32% to 10%, and most categories over 50% taken also present reductions in the number of branches. This leads to a significant increase in the number of always not taken branches, from 36% to 59%. With most highly biased branches in the not taken side, and most other branches moving from over 50% taken to mostly not taken, the prediction accuracy of an always not taken (or BTFNT) predictor, increases significantly, as we have seen in Figure 3.

The increase in the number of usually not taken branches explains the different behavior of the two code layouts regarding static branch prediction. Further increases in static prediction accuracy can be expected of a code layout optimization that explicitly targets a specific branch predictor, like the BTFNT predictor, or uses code replication techniques to use path information in its static predictions.

Next, we will examine how this change in branch direction affects dynamic branch prediction.

4. Effect on dynamic prediction

4.1. Two-level adaptive predictors

Figure 5 shows the effect of code reordering on dynamic prediction accuracy for the Gshare, PAg, and bimodal predictors. Predictor sizes from 512 bytes to 16KB are explored for both the baseline (dotted line) and the optimized code layout (solid line).

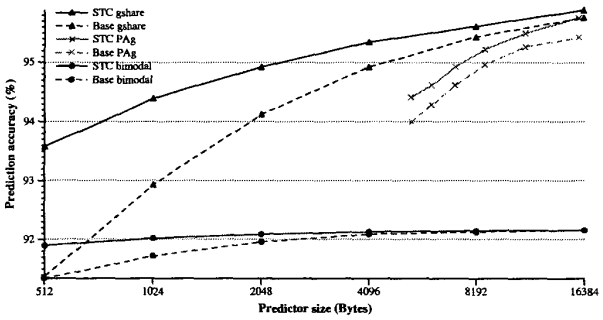


Figure 5. Dynamic prediction accuracy for both the base and the STC optimized code layouts using two-level adaptive prediction schemes.

Clearly, the STC increases the prediction accuracy of the examined branch predictors, specially for the smaller predictor sizes. Both the Gshare and the bimodal predictors seem to converge at infinite predictor size, which points that the benefits of using the STC are related to prediction table interference. The larger the table, the less interference, the closer the prediction accuracy for both layouts.

Prediction table interference

Figure 6 shows the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with both the baseline and the optimized code layouts. We classify conflicts in three groups: neutral interference when the conflict does not change the prediction, and positive or negative if the conflict changes the prediction for good or bad.

As expected, there is a significant reduction in the number of negative conflicts when the STC layout is used with the Gshare branch predictor. For example, a 1KB gshare goes down from 1.45% of negative conflicts to 0.79% using the optimized code layout.

Intuitively, the increase in the number of not taken branches favors positive interference, because it is more likely that when two branches interfere, they both behave the same way (both not taken) resulting in a positive or neutral conflict.

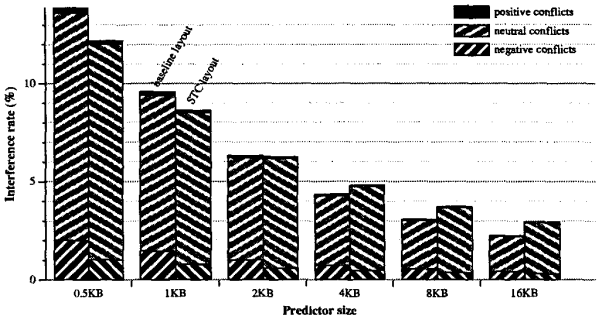


Figure 6. Percent of dynamic branches which cause interference in the gshare prediction tables for the baseline and optimized code layouts.

The total amount of conflicts shows a different behavior. The optimized code layout has fewer neutral conflicts for small predictor sizes, but it ends up with a larger amount of neutral interference for the largest configurations.

We will look further into this neutral interference increase in the next section, where we will examine dealiased branch prediction schemes.

4.2. Dealiased branch predictors

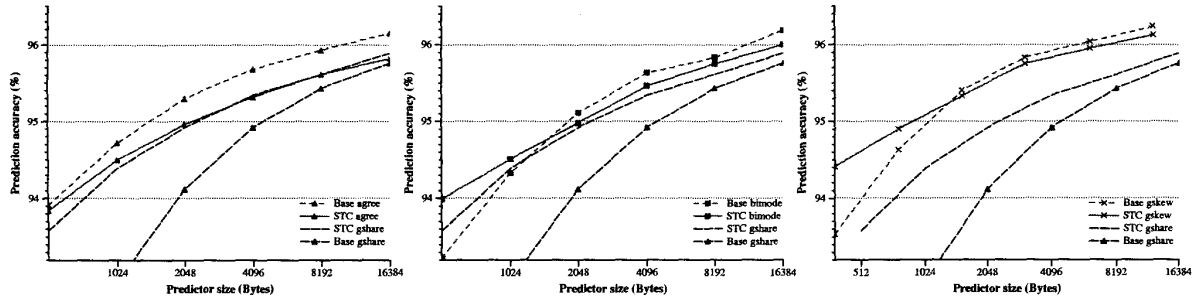
Given that the use of an optimized code layout is reducing the negative interference found in the dynamic prediction tables, it is interesting to examine what happens with modern branch predictors that are already organized to minimize such interference like the agree [24], bimode [10], and gskew [12, 22] predictors. We will refer to these predictors as dealiased branch prediction schemes.

Figure 7 shows the prediction accuracy of the dealiased predictors with both the baseline and the optimized code layouts. The prediction accuracy of the gshare predictor with the optimized layout is shown for reference purposes.

These results show that for small predictor sizes, the use of optimized code layouts obtains equivalent or higher accuracy even in the dealiased branch predictors. The advantage of the optimized layouts is specially clear in the 0.4KB gskew predictor, which increases prediction accuracy from 93.5% to 94.4%.

For medium and large predictor sizes, all dealiased branch predictors obtain higher accuracy with the baseline code layout, being the difference specially significant with the 16KB agree predictor, which obtains a 96.2% accuracy with the baseline layout and a 95.8% with the optimized code.

A more important result shows that the use of a large agree or bimode predictor with the optimized code layout does not yield significant improvements over a gshare pre-



(a) Agree predictor (b) Bi-mode predictor (c) Gskew predictor

Figure 7. Effect of the optimized code layout on dealiased branch predictors.

dictor. Only the gskew predictor obtains significantly better results than the gshare predictor when using the optimized code layout.

Prediction table interference

Figure 6 shows the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with the optimized code layout and the agree predictor using both code layouts.

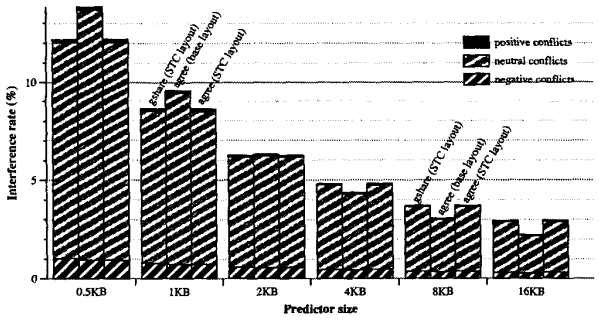


Figure 8. Percent of dynamic branches which cause interference in the gshare prediction tables optimized code layout and the agree predictor using both code layouts.

These results show that the agree prediction scheme with a non optimized layout obtains a slightly better negative interference reduction than the optimized code layout. It is surprising that using the agree predictor, the optimized code layout has more negative conflicts than the baseline.

From these results it seems that the dealiased predictors prove more effective at reducing interference than the optimized code layout, but the more important result is that it seems more difficult to reduce conflicts in an optimized binary. The fact that the optimized code layout has more total interference for the larger predictor sizes can explain this higher fraction of negative conflicts.

Branch history register distribution

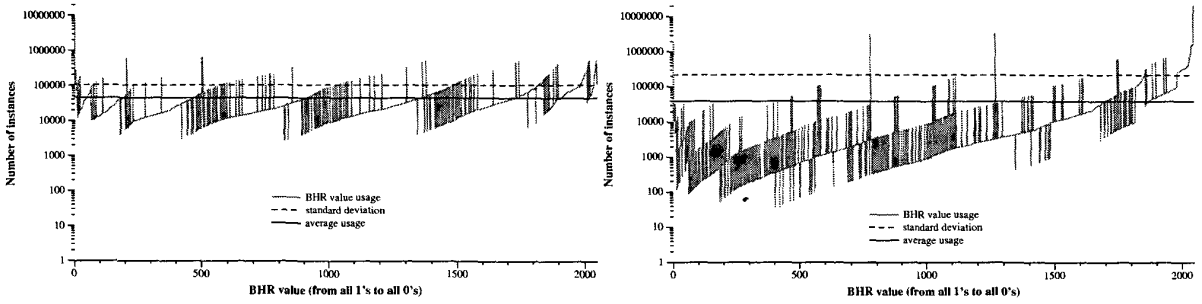
The fact that dealiased predictors using an optimized binary obtain worse results than a gshare predictor points to some other factor hindering the performance of these predictors.

The high fraction of not taken branches found in the optimized code layout (80% of all branches are not taken) may be hindering the branch distribution in the BHR. When working with an optimized binary, the BHR will tend to be full of zeros, causing many possible BHR values to be never or rarely used, leading to a worse branch distribution and a loss of *useful* information to make a correct prediction.

The dealiased predictors do not benefit from the interference reduction effect, because they are quite good at reducing it themselves, thus they only suffer the negative BHR effect and loose accuracy with the optimized code layout.

To analyze this BHR distribution factor, Figure 9 shows the number of times each possible history value was found in an 11-bit global history predictor for both code layouts. The BHR values are sorted by the number of zeros their binary value contains (from all 1's to all 0's). In addition to the BHR value usage, the figure shows the average usage, and the average + standard deviation. The average usage is the same in both code layouts. Note the Y axis is in log_{10} scale.

The first remarkable aspect of these plots is the position of the highest peak. The most popular history value for the baseline layout is a BHR full of 1's (leftmost value), while the highest peak of the STC layout corresponds to a BHR full of 0's (rightmost value). Aside from that, the BHR value usage in the baseline layout is mostly spread across 1–2 orders of magnitude. Meanwhile, the STC layout has its BHR value usage spread across 4–5 orders of magnitude, with very high peaks on a reduced set of values. It is clear that values having mostly 1's are less used than those having mostly 0's.



(a) Base layout (b) STC layout

Figure 9. Branch history register value distribution for the baseline code layout (a), and the STC optimized layout (b).

To summarize these observations, we can just look at the distance between the average usage and the standard deviation lines. The more distance between them, the worse the BHR value distribution. In this case, the distance between both lines in the STC layout is 2.5x larger than in the baseline code layout.

5. Processor performance

The complexity of current processors is already very high, and keeps increasing with each generation. Simulating such complex designs is not always feasible, specially if the design space to explore is large. This leads to many studies in which only isolated components are examined, on the basis that if that component works better, then overall performance will also increase.

We have shown that the performance impact of the branch predictor is heavily dependent of the instruction cache performance [15]. New results shown here in this paper point that branch prediction accuracy can decline when optimized code layouts are used, but we know that those same layouts also increase the instruction cache performance.

The performance benefits of an instruction cache miss reduction could compensate for the performance loss due to reduced branch prediction accuracy. In order to explore this possibility, we simulated a whole out of order processor using the sim-outorder simulator of the SimpleScalar 3.0 Tool set. The detailed simulation setup for our 4-wide processor is shown in Table 2.

Figure 10 shows processor performance measured in IPC for both the baseline and the STC code layouts using two different branch predictors: the gshare predictor, which proves more accurate with the optimized layout; and the agree predictor, which proves more accurate with the baseline layout. We simulated both a small 16KB instruction cache and a larger 64KB cache.

Item	Value	Item	Value
Int ALU	3	L1 Data cache	64KB, 2-way
Int MUL/DIV	1	L1 Inst. cache	16KB *or* 64KB, 2-way
FP ALU	1	L1 latency	1 cycle
Mem ports	2	L2 cache	2MB, 2-way
Window size	64	BTB	4096 entries, 4 sets
LSQ size	16	RAS	64 entries
		BPred	gshare *or* agree 12, 14 and 16 bits

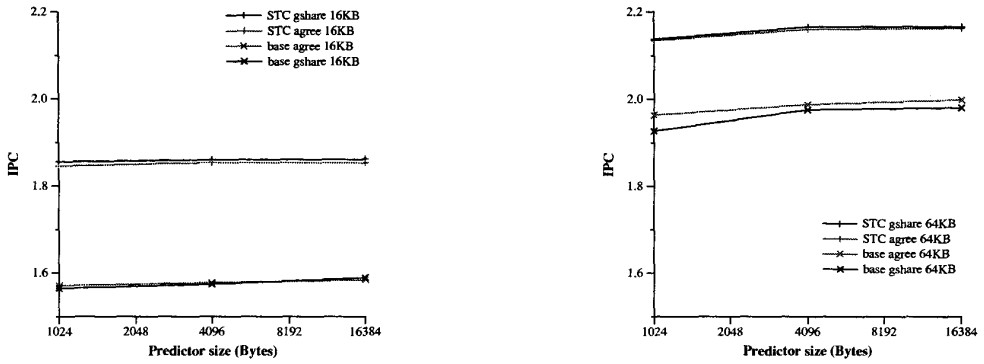
Table 2. Setup description for the 4-way out of order processor examined.

These results show that the instruction cache miss reduction more than compensates for the loss of branch prediction accuracy, as the STC layout always performs better than the baseline, even with the agree predictor, with a 17% improvement on the 16KB cache, and a 9% on the 64KB cache.

Examining the results for each individual code layout on the 16KB instruction cache, we observe that the branch predictor used does not make a significant difference for the baseline layout. Meanwhile, the optimized layout does 0.5% better with the gshare predictor than with the agree predictor.

When a 64KB instruction cache is used, the baseline layout obtains a 2% improvement using the agree predictor for the smaller predictor size, and a 1% improvement for the larger setup. The optimized code layout still does slightly better with the agree predictor, but the difference is not significant. In any case, the optimized layout still obtains an 8% improvement over the baseline layout with the agree predictor.

To gain further insight on why the optimized code layout obtains better performance, even when it has lower prediction accuracy, Table 3 shows a comparison of all three fetch performance factors for both code layouts and the 16KB a-



(a) 16KB Instruction cache (b) 64KB Instruction cache

Figure 10. Processor performance measured in IPC for the baseline and STC code layouts using gshare and agree branch predictors. Results shown for (a) 16KB and (b) 64KB instruction caches.

gree branch predictor. We show the total number of misses (in millions), the average fetch width (in instructions per cycle), the branch prediction accuracy (in percent), and the processor performance (IPC).

I$ size	Layout	I$ misses	Fetch width	BP accuracy	IPC
16KB	base	13 mil.	1.8 IPC	96.1 %	1.58
16KB	STC	8 mil.	2.1 IPC	95.6 %	1.85
64KB	base	4.5 mil.	2.3 IPC	96.1 %	2.00
64KB	STC	2.5 mil.	2.5 IPC	95.6 %	2.16

Table 3. Instruction cache (I$) misses, fetch width, prediction accuracy, and IPC for both layouts using a 16KB agree predictor.

The lower prediction accuracy of the STC layout translates into smaller sequences of valid instructions, because a branch misprediction is encountered sooner. But the smaller distance between mispredictions is compensated by the smaller perceived latency of the instruction cache, and the higher rate at which these instructions are provided.

These results show that reducing the number of branch misprediction does not necessarily mean increasing processor performance. Code transformations such as basic block reordering may decrease branch prediction accuracy, but still increase performance due to other effects, like an instruction cache miss reduction and an increase in the fetch bandwidth.

6. Conclusions

To our knowledge, this is the first paper showing the effects of code reordering on branch prediction accuracy. These are summarized in Figure 11.

Summarizing, optimizing the code layout for higher fetch rate will:

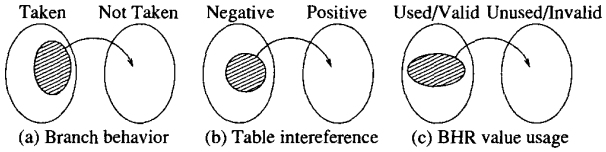


Figure 11. Effect of code reordering in (a) static branch prediction (branch direction), (b) dynamic prediction table interference, and (c) branch history register value usage.

Change branch direction: Most branches tend to be not taken, and most highly biased branches are now always not taken branches.

Reduce negative interference: As most branches are now not taken, it is more likely that when two branches map to the same two-bit counter, they push the counter in the same direction (towards not taken).

Generate a worse BHR value distribution: The high proportion of not taken branches causes many BHR values to be not used, concentrating branch history on a smaller set of values. This reduces the amount of useful information the predictor has to take a decision.

The overall effect of code reordering on a given branch predictor will depend on which of these effects dominates. Predictors which do not use global history registers (bimodal and PAx), or which hash the global history register with the branch address or other values (gshare) will benefit from the table interference reduction, while they mitigate or ignore the BHR value effect. Predictors which heavily depend of the global history register, or which already have their own interference avoiding mechanism will feel

the negative BHR value effect, without obtaining a large benefit form the interference reduction offered by optimized layouts.

Second, we have shown that increasing branch prediction accuracy does not necessarily mean higher processor performance. For example, optimizing the code layout for better instruction cache performance may decrease prediction accuracy, but the reduced distance between branch mispredictions is compensated by a lower cache miss rate, and a higher fetch width, which increase the speed at which instructions are provided.

References

[1] T. Ball and J. R. Larus. Branch prediction for free. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 300–313, June 1993.

[2] D. Burger, T. Austin, and S.Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report TR-1308, University of Winsconsin, July 1996.

[3] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, Oct. 1994.

[4] B. Calder, D. Grunwald, and D. Lindsay. Corpus-based static branch prediction. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 79–92, 1995.

[5] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. *Proceedings of the 5th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 1992.

[6] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[7] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 171–182, June 1997.

[8] W.-M. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 242–251, June 1989.

[9] A. Krall. Improving semi-static branch prediction by code replication. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 97–106, 1994.

[10] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 4–13, Dec. 1997.

[11] S. McFarling. Combining branch predictors. Technical Report TN-36, Compaq Western Research Lab., June 1993.

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

[13] F. Mueller and D. A. Whalley. Avoiding conditional branches by code replication. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 56–66, 1995.

[14] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 322–330, 1992.

[15] C. Navarro, A. Ramirez, J. L. Larriba-Pey, and M. Valero. On the performance of fetch engines running dss workloafds. *Proceedings of the Intl. Euro-Par Conference*, page to appear, Aug. 2000.

[16] J. R. C. Patterson. Accurate static branch prediction by value range propagation. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 67–78, 1995.

[17] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 16–27, June 1990.

[18] A. Ramírez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Optimization of instruction fetch for decision support workloads. *Proceedings of the Intl. Conference on Parallel Processing*, pages 238–245, Sept. 1999.

[19] A. Ramírez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *Proceedings of the 13th Intl. Conference on Supercomputing*, June 1999.

[20] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. *Proceedings of the 23th Annual Intl. Symposium on Computer Architecture*, pages 22–32, 1996.

[21] A. Seznec. A case for two-way skewed-associative caches. *Proceedings of the 20th Annual Intl. Symposium on Computer Architecture*, May 1993.

[22] A. Seznec and P. Michaud. D-aliased hybrid branch predictors. Technical Report PI-1229, IRISA, Feb. 1999.

[23] J. E. Smith. A study of branch prediction strategies. *Proceedings of the 8th Annual Intl. Symposium on Computer Architecture*, pages 135–148, 1981.

[24] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. *Proceedings of the 24th Annual Intl. Symposium on Computer Architecture*, pages 284–291, 1997.

[25] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, Jan. 1995.

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

[27] C. Young and M. D.Smith. Improving the accuracy of static branch prediction using branch correlation. *Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, Oct. 1994.

[28] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, June 1995.