6 THE *AGBIAS* BRANCH PREDICTOR

Branch prediction accuracy is one of the three main factors of fetch performance, and has a very important impact on the overall processor performance. In Chapter 4 we have evaluated the impact of code layout optimizations on the branch prediction mechanism. However, the use of code layout optimizations usually implies the availability of profile data, or at least, some ahead knowledge on the code behavior. This chapter explores the possibility of using this profile data to improve the branch prediction mechanism.

The first approach to branch prediction were static predictors, which always predicted the same direction for a given branch. The use of profile data and compiler transformations for these predictors is straightforward, and proves very effective at improving prediction accuracy.

We concentrate on the uses of profile data and the possible compiler optimizations that improve the prediction accuracy of dynamic branch predictors. We also propose a novel predictor organization which makes extensive use of profile data, and which proves more accurate than any other examined predictor. The main advantage of our proposed predictor (the *agbias* predictor) is that it does not depend heavily on the quality of the profile data to provide high prediction accuracy.

Our results show that using profile data and compiler transformations can significantly increase the prediction accuracy of most dynamic branch predictors, and that our *agbias* predictor reduces the branch misprediction rate by 14% on a 16KB predictor over the next best compiler-enhanced predictor.

6.1 Introduction

Branch prediction was first approached using static schemes, which always predict the same direction for a branch, like predicting that all branches would be taken, or that only backward branches would be taken [80]. The use of profile data (or heuristics which accurately predict the behavior of the program) to improve the prediction accuracy of such schemes is straight-forward: Semi-static branch predictors use profile feedback information and encode the most likely branch direction in the instruction opcode, obtaining much higher accuracy than the simple static predictors [21]. Profile data can also be used to align branches in the code so that they follow a given static prediction heuristic [9]. In this chapter we try to complete the picture by exploring the usage of profile data on dynamic branch predictors.

First, we describe the role of the compiler in branch prediction accuracy for some of the best proposed branch predictors: gshare, agree, bimode, and gskew. We review already proposed techniques, usually implying the replacement of some predictor component with profile data, and fill in the blanks for those predictors where the compiler involvement had not been explored. For example, the bias bit in the agree predictor can be obtained with profile data, instead of setting it to the first branch execution, increasing the accuracy of a 2KB predictor from 95.3% to 95.7%. Also, a gshare predictor benefits from a code layout optimized binary which makes most branches follow the not taken direction, increasing the accuracy of a 2KB predictor from 94.1% to 94.9%.

Next, we propose the *agbias* predictor, largely based on the static-dynamic predictor combination proposed in [56], which divides branches among four sub-streams: first, among *easy* and *hard* to predict branches; second, among mostly *taken* and *not taken* branches. For example, a branch could be statically classified as a mostly-not-taken but hard-to-predict branch. The *easy* and *hard* sub-streams are predicted using separate pattern history tables (PHT). This eliminates the interference among them, and allows a separate resource allocation, as both sub-streams do not have the same needs. Interference among the mostly *taken* and *not taken* sub-streams in each separate component is also minimized using the agree prediction scheme. The *easy* and *hard* sub-stream division is also used to exclude the *easy* branches from the branch history register (BHR) in both components, which increases the amount of *useful* information available to the predictor.

Our results show that the *agbias* predictor outperforms all other examined predictors (including their compiler enhanced version), reducing branch misprediction rate by 14% on a 16KB predictor.

The use of profile data to replace dynamic predictor components consumes some opcode bits to store the required information. These bits would require ISA extensions in architectures like Alpha [78] or MIPS [37]. But some other ISAs already have space to encode this data: the SPARC [19] has one bit to avoid execution of the delay slot on a given branch outcome, the PA-RISC [31] has one bit to select between static and dynamic branch prediction, and the IA64 [15] has two bits to encode the predicted branch direction, and to select between static and dynamic prediction, the PowerPC [27] also includes support for static prediction. The cost of these ISA modifications has not been considered in this chapter.

6.2 Using profile data in dynamic prediction

In this section, we describe how profile data and compiler transformations can be used to improve the prediction accuracy of some of the best proposed dynamic branch predictors.

Next, we present a novel branch predictor organization, which improves on previously proposed predictor by reducing the dependency of the quality of the profile data.

6.2.1 The gshare predictor

The gshare two-level adaptive branch predictor [42] is one of the most widely referenced branch prediction schemes in the literature, and is counted among the most accurate 2-level schemes. Our compiler optimized version of the gshare predictor is used as the baseline for comparison across the whole chapter.

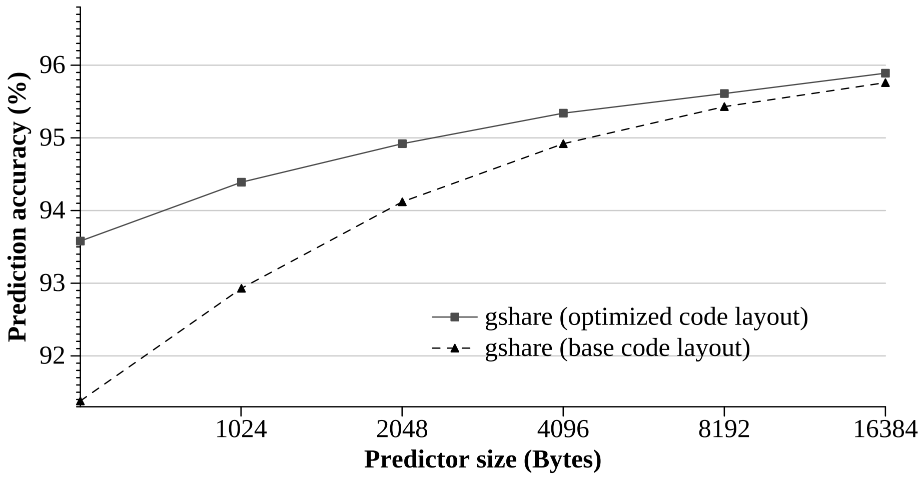


Figure 6.1. Prediction accuracy of the base gshare predictor and a gshare predictor using an optimized code layout.

There is no obvious way for the compiler to influence this prediction mechanism, but it is possible to reduce PHT negative interference by selectively reversing branch direction. Negative interference happens when two branches with opposite behavior map into the same PHT entry. By aligning these branches so that they follow the same direction, we convert a negative conflict into a neutral or positive one.

Figure 6.1 shows the prediction accuracy of the gshare prediction scheme using both the baseline code organization, and an optimized one which reverses most branch directions so that they tend to be *not taken*.

From the results shown it is clear that aligning branches to avoid negative PHT interference can significantly increase the prediction accuracy of this scheme, specially for the smaller predictor sizes, where aliasing is more frequent. A more detailed study of the effects of code reordering on branch prediction can be found in Chapter 4.

6.2.2 The agree predictor

The agree mechanism removes the destructive aliasing in a very similar way to what we did before with the gshare predictor: by selectively reversing a branch direction. But instead of using a compiler transformation, it is done using an additional bit associated to the branch. Each branch has a different direction bit, but both branches will update the PHT counter towards the *agree* state, reducing interference.

Instead of setting the direction bit with the first execution of the branch, the compiler can improve the performance of this predictor by selecting the direction bit based on the most likely branch direction and encoding it in the instruction itself. This approach was already suggested in [82].

Using profile data to select the direction bit is more accurate than simply setting it with the first branch execution, which could just be the odd case for that branch. Also, the bit is always

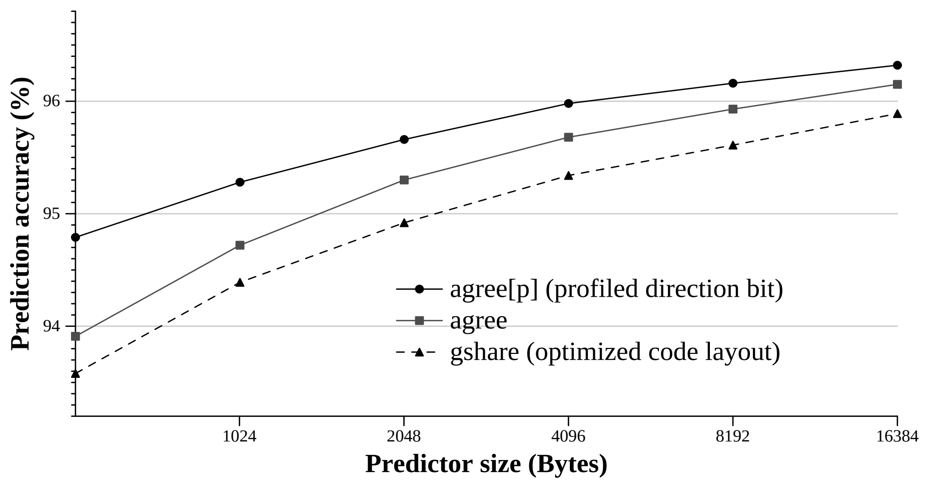


Figure 6.2. Prediction accuracy of the basic agree predictor, the profile-assisted agree and the compiler optimized gshare predictors

present with the instruction and does not have to be recalculated every time the branch leaves the instruction cache.

Figure 6.2 shows the prediction accuracy of a fully dynamic agree predictor and the agree predictor with a direction bit selected with profile information (agree[p]). The compiler optimized gshare curve is shown for comparison purposes.

As happened with the gshare predictor, the results show that the compiler can play a significant role in increasing branch prediction accuracy. In this case by using profile data or static analysis and heuristics to set the direction bit to the most likely branch outcome. Once again, the importance of the compiler is higher for the smaller predictor size, where interference is more frequent.

6.2.3 The bimode predictor

Instead of using a direction bit to separate the two branch sub-streams, the bimode predictor devotes a separate gshare branch predictor to each branch sub-stream, and uses a bimodal predictor to select the sub-stream where the current branch is found.

As most branches in each separate gshare component have the same general direction, it is likely that any conflict will result in positive or neutral interference. To avoid interference between the two sub-streams, only the selected gshare component is updated for each branch.

As we did with the agree predictor, the compiler can use profile data/static analysis to select the sub-stream for each branch, and statically encode it in the instruction itself [25]. This static sub-stream selection may not be as accurate as that of the bimodal component, but does not require the selection table, which represents $1/3$ to $1/2$ of the total bimode predictor size [39].

Figure 6.3 shows the prediction accuracy of the bimode predictor and the bimode predictor using a profiled direction selector (bimode[p]). Both versions compare to the gshare predictor using the optimized code layout.

Both versions of the bimode predictor prove more accurate than the gshare, specially the compiler enhanced version, which benefits from a significant cost reduction: it does not require the bimodal component to separate the two branch sub-streams.

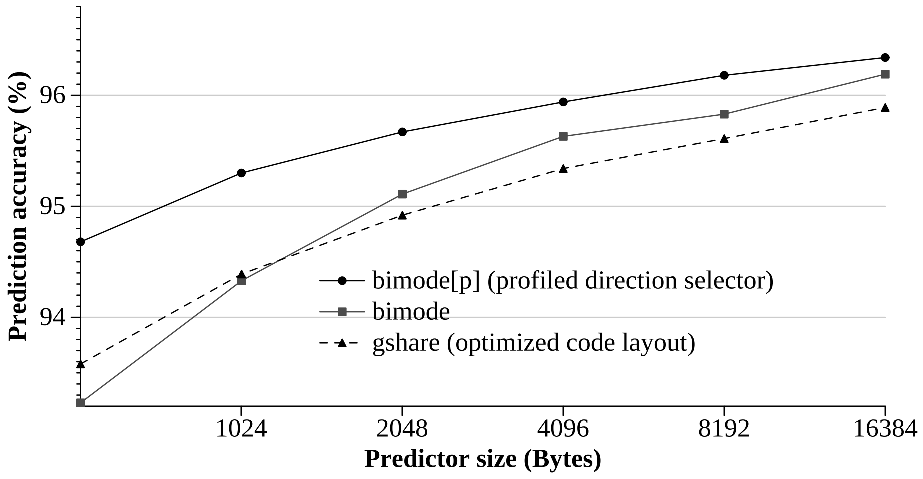


Figure 6.3. Prediction accuracy of the bimode predictor, the compiler-enhanced bimode, and the compiler optimized gshare predictors.

A second role of the compiler on this prediction scheme is ensuring the load balance between the two gshare components. That is, aligning branches so that half of them are usually taken, and half of them usually not taken. If most branches follow the same direction, they will all map to the same dynamic predictor, reducing the efficiency of the scheme.

6.2.4 The gskew predictor

The gskew predictor stores branches in three different tables, accessed using three different functions. The indexing functions are selected so that if a branch conflicts in one of the tables, it will not be corrupted in the other two, leading to un-aliased predictions with a majority vote. If aliasing is detected in one of the prediction tables (a wrong prediction) but the final prediction is correct, only the two un-aliased tables will be updated.

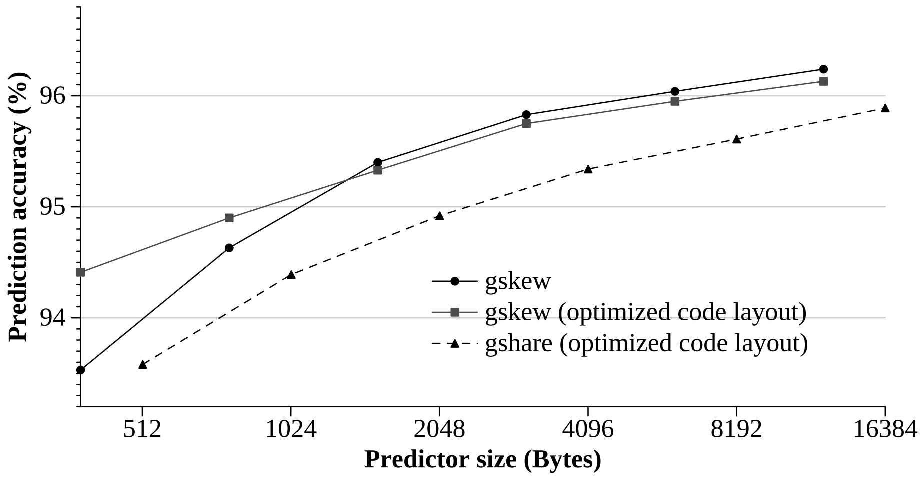


Figure 6.4. Prediction accuracy of the gskew predictor, and the gshare and gskew predictors on the compiler optimized code layout.

As happened with the gshare predictor, there is no obvious way for the compiler to help this prediction scheme, other than the same branch alignment optimization used with the gshare predictor. The compiler can align branches towards the same direction, so that when two branches share the same PHT counter they will both push it towards the same result.

Figure 6.4 shows the prediction accuracy of the gskew predictor, the prediction accuracy achieved using an optimized code layout, and the optimized gshare predictor for comparison purposes.

The use of an optimized code layout proves effective only for the smaller predictor sizes, where interference is more important. For mid to large sizes, the gskew predictor does equally well with both code layouts. This shows that for small predictor sizes, the compiler can play an important role even on purely dynamic schemes designed to remove interference.

6.2.5 Combining dynamic and static predictors

The possibility of using the compiler to help dynamic branch predictors can be taken one step further by using the combined branch predictor scheme [42]. The possibility of using static branch predictors based on profile information to reduce the number of branches that require dynamic prediction, and thus reduce the number of PHT conflicts found has also been studied before [41, 56]. Filtering the strongly biased branches to avoid interference in the PHT was also done before, but using dynamic structures [11, 18, 54].

Figure 6.5.a shows the combination of a dynamic branch predictor, and a static branch predictor based on profile data. The prediction selection is based on the profiled branch bias: strongly biased branches are predicted using the static component, while not strongly biased branches use the dynamic predictor. The dynamic predictor is updated only for those branches requiring dynamic prediction (the not strongly biased group). This reduces the number of branches stored, and thus the number of conflicts (both positive and negative) in the PHT.

The role of the compiler in this prediction scheme is obvious, since we require two bits of information stored in the instruction: the profiled branch direction (taken or not taken), and the branch bias (strongly biased or not) to select between static and dynamic prediction.

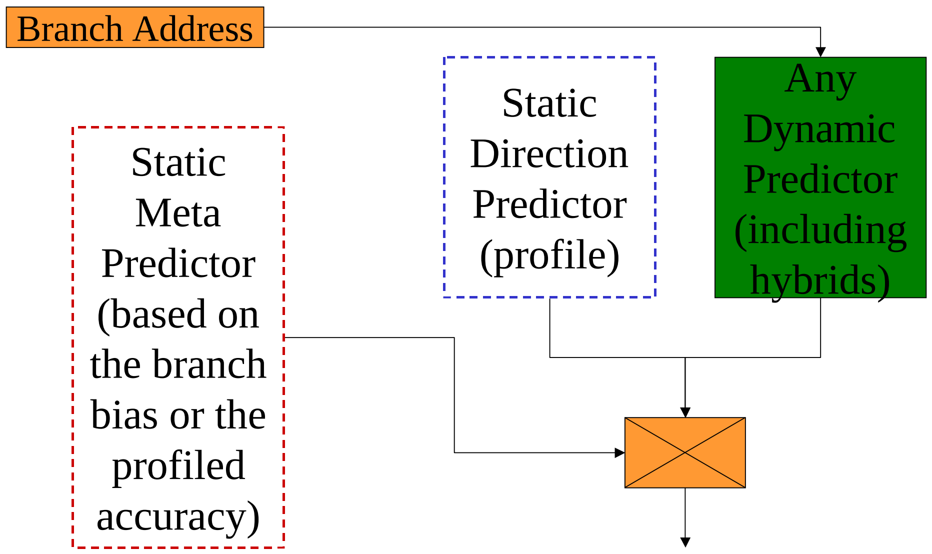
However, the static direction prediction is always stored in the instruction, even for these branches predicted dynamically. We propose to take further advantage of this bit for the dynamic prediction using it to implement the compiler-assisted version of the agree or bimode predictors, instead of using a pure hardware implementation of them. This way, both bits are useful in all possible scenarios (static and dynamic prediction).

Figure 6.5.b shows the prediction accuracy of this mixed static-dynamic predictor using a compiler-assisted agree (agree[p]) for the dynamic component, compared to an isolated compiler-assisted agree. Given the dependence of the profile quality of the static-dynamic combination, we show results for both self-trained (as in [56]) and cross-trained tests.

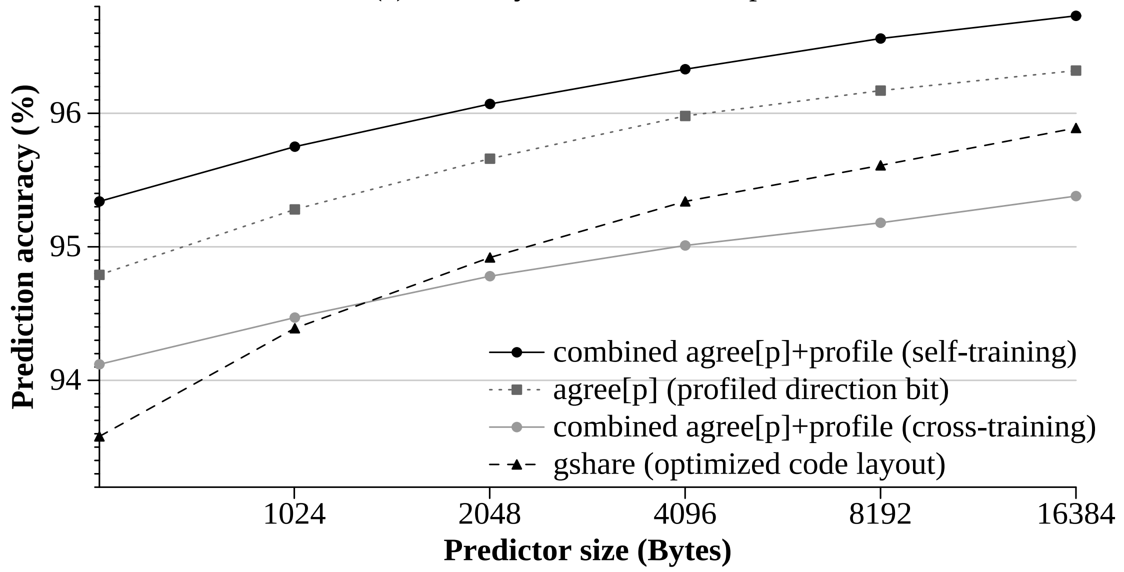
The self-trained test obtains significant improvements over the agree[p] predictor, because it is eliminating nearly 70% of all branches form the PHT, which reduces overall interference.

The main drawback of this prediction scheme is that it depends heavily on the accuracy of the profile data. Nearly 70% of all branches are strongly biased, which means that a loss of 1% in the accuracy for these branches translates in a 0.7% loss in the global prediction accuracy, which can outweight the interference reduction obtained in the dynamic component.

In this case, the cross-trained test drops to the level of a gshare predictor, due to the loss of accuracy of the profile direction predictor (from 99.4% to 97%). This shows that using static


Branch Address
Static Direction Predictor (profile)
Any Dynamic Predictor (including hybrids)
Static Meta Predictor (based on the branch bias or the profiled accuracy)

(a) Static-dynamic combined predictor



(b) Static-dynamic prediction accuracy

Figure 6.5. (a) Combination of any dynamic prediction scheme with a profile-based static predictor. (b) Prediction accuracy of the static-dynamic combination using a compiler-assisted agree for the dynamic component.

data in a too rigid way can lead to performance degradation if the profile data is not completely accurate.

6.2.6 The agbias predictor

Based on the ideas exposed thus far, we propose another combination of static and dynamic branch predictors which we call *agbias*. The agbias prediction scheme is shown in Figure 6.6.a. Largely based on the prediction scheme proposed in [56], it is composed of two dynamic direction predictors, which use some de-aliasing mechanism (we have chosen the agree mechanism for this work), and a static meta-predictor which divides branches between the strongly biased sub-stream (the *easy* branches), and the not so biased sub-stream (the *hard* branches). Both dynamic components share the BHR, which is only updated for the branches belonging to the *hard* sub-stream.

This way, we are dividing branches into four categories: first, using the profiled bias (strongly biased/*easy* or not strongly biased/*hard* branches); second, using their most likely outcome (*taken* or *not taken*). The first division is used to distribute branches in two separate dynamic predictors, which allows an independent resource allocation for the *easy* and *hard* sub-streams. The second division is used to minimize negative interference in the prediction tables of both dynamic components, using a de-aliased scheme (we have chosen the agree[p] scheme, hence the name *agbias*). The classification of branches among strongly biased and non-strongly biased using profile data [12] or dynamic tables [11, 18, 54] has been explored before, but none considered a separate dynamic component for the strongly biased stream.

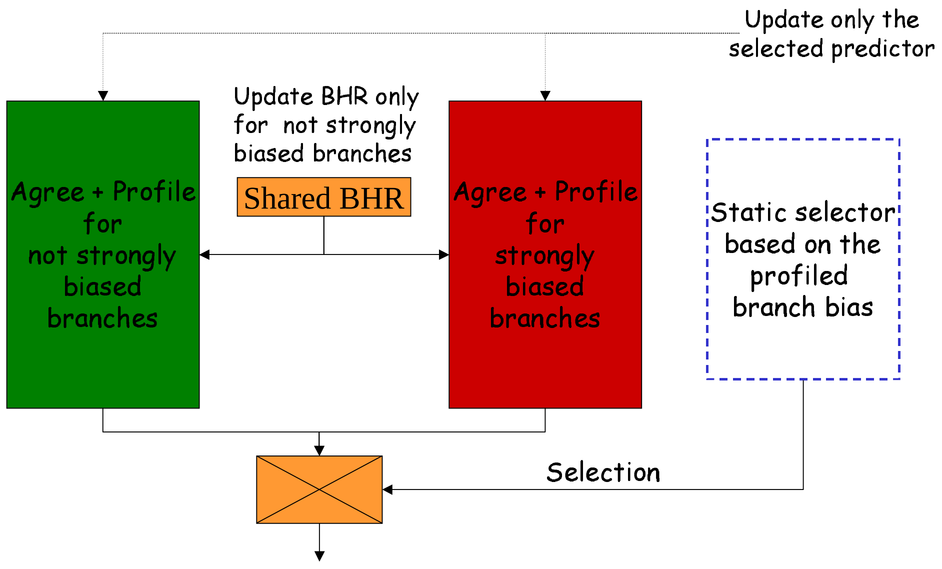
The static-dynamic combination used in [56] does not allocate any dynamic resources to the *easy* sub-stream, relying entirely on the accuracy of the profile data. Our scheme avoids this too strong dependency by using a small dynamic predictor instead. Even if the profile wrongly classifies a branch as belonging to the easy sub-stream, the dynamic component will be more accurate than a pure semi-static predictor. In this work we have used a 512 byte agree predictor with compiler enhancements.

The agbias predictor used in this chapter requires two bits encoded in the instruction: a branch bias bit, and a branch direction bit. An implementation which does not use the compiler-enhanced predictor for the dynamic components would only require one bit to classify a branch in the *easy* or *hard* sub-stream.

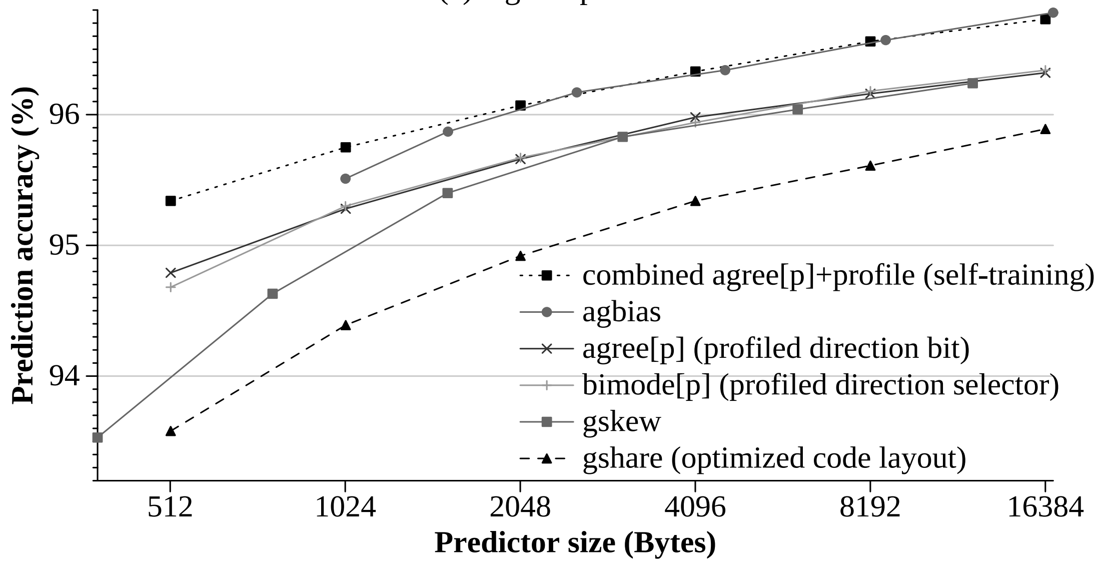
Figure 6.6.b shows the prediction accuracy of the agbias predictor compared to the other compiler-enhanced predictors described. All lines show results for the cross-trained test, except that of the semi static agree-profile combined which shows the self-trained test.

Our results show that the agbias predictor is more accurate than the other predictors examined for all predictor sizes, reaching 96.8% average accuracy for the studied benchmarks, improving on the 96.3% obtained with the compiler-enhanced agree and bimode predictors. Based on an average basic block size of 5 instructions, using the 16KB optimized gshare we have a branch misprediction every 121 instructions. This distance increases to 135 with agree[p], bimode[p] and gskew, and to 156 with agbias.

The only comparable predictor is the static-dynamic combination when running the self-trained test, because it obtains the same prediction accuracy but does not require any hardware resources for the *easy* sub-stream prediction. However, note that the accuracy of that static-dynamic combination drops to the level of a gshare predictor with the cross-trained test (Figure 6.5.b). The advantage of the agbias predictor is that it obtains the same performance in the self and cross-trained tests, being less dependent on the profile data accuracy.


Update only the selected predictor
Update BHR only for not strongly biased branches
Shared BHR
Agree + Profile for not strongly biased branches
Agree + Profile for strongly biased branches
Static selector based on the profiled branch bias
Selection

(a) Agbias predictor scheme



(b) Agbias prediction accuracy

Figure 6.6. (a) The agbias branch prediction scheme. (b) Prediction accuracy of the agbias predictor compared to other compiler-enhanced predictors.

6.3 Performance evaluation

In this section we will analyze the performance results of the proposed agbias predictor, examining in detail the amount of interference encountered in the prediction tables, and the beneficial effect of the BHR filtering used, showing that the agbias predictor proves as accurate ans an unbounded (interference free) predictor.

6.3.1 Prediction table interference

Figure 6.7 shows the percent of dynamically executed branches which cause interference in the dynamic prediction tables for some of the examined predictors. Conflicts are classified in positive (the conflict causes a correct prediction where there would have been a misprediction), neutral (the conflict does not change the prediction), and negative (the conflict causes a misprediction when there would have been a correct prediction). The *easy* dynamic component of the agbias predictor achieves a 99.5% prediction accuracy, thus we concentrate only on the conflicts remaining in the *hard* dynamic component.

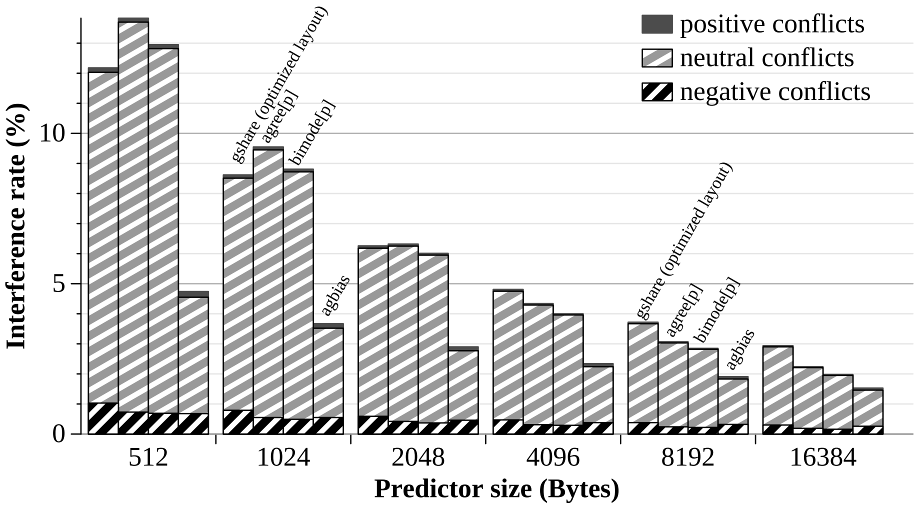


Figure 6.7. Percent of executed branches which cause conflicts in the dynamic prediction tables. Conflicts classified (top to bottom) in positive, neutral and negative, the aggregate corresponds to the total interference rate. There is a separate column for each predictor.

Although both the gshare and the agree[p] predictor use the same base mechanism and indexing function, they do not have the same total number of conflicts because we have used a different code layout with the gshare (an optimized one to align branches towards not taken). But, in any case, the agree[p] mechanism proves more successful at reducing negative interference in the PHT. The bimode[p] predictor is slightly better at reducing negative and overall interference than the agree[p] predictor, but this interference reduction does not translate into prediction accuracy, because it suffers from load imbalance among its two separate prediction tables (the usually taken and not taken components).

The results show that the agbias predictor has a similar amount of negative conflicts to the agree[p] or bimode[p] predictors, but is is much better at reducing overall interference, because only 30% of all branches update the *hard* component. This places it much closer to an interference-free predictor.

As we show next, this global interference reduction is not the only reason for the high prediction accuracy obtained with the agbias predictor, the selective update of the BHR also proves important.

6.3.2 BHR filtering

There is a second difference between the agbias predictor and the other de-aliased schemes: updating the BHR only for branches in the *hard* sub-stream. This selective BHR update achieves an important result: it is increasing the amount of *useful* history information kept in the BHR.

The *easy* branches are those which have the same outcome 95% of the times, that is, they almost never change direction. Knowing the outcome of these branches (having its outcome in the BHR) does not provide the predictor with any extra information, because we *already know* the outcome of the branch before it is executed. The outcome of the next branches does not actually depend on it, because it never changes. Not shifting these outcomes into the BHR prevents other -variable- bits from leaving the BHR. This increases the usefulness of the information stored in the first level table, making it easier for the predictor to guess the correct branch outcome.

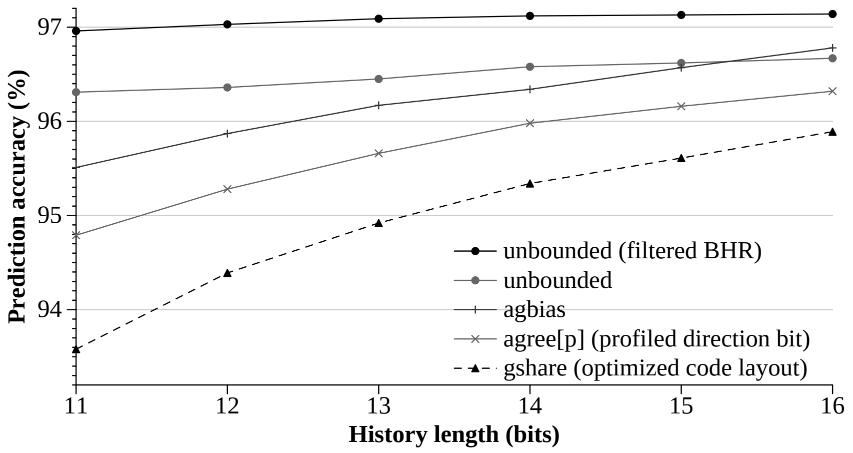


Figure 6.8. Prediction accuracy of the agbias predictor, an unbounded predictor with the same history length, and an unbounded predictor with a filtered history. The agree[p] and gshare predictors shown for reference.

Figure 6.8 shows the prediction accuracy of the agree[p] and the agbias predictors compared to an unbounded predictor using the same history length, and an unbounded predictor with a filtered BHR. The agree and agbias predictors use the history length to determine the PHT size (a 14-bit agree predictor has 2^{14} PHT entries, requiring 4KB of storage, the same 14-bit agbias requires an extra -fixed- 512 bytes for the *easy* sub-stream component). An unbounded predictor has a separate PHT entry for each branch and each possible BHR value, so it is free of interference.

The unbounded predictor does not show large accuracy improvements with increasing history lengths, but it still increases from 96.3% to 96.7% (11 bits to 16 bits), showing that increasing the amount of history information represents a slight advantage to this predictor. The agree and agbias predictors experience larger improvements with increasing history lengths because a longer history also implies a larger PHT, and less interference.

The most remarkable result is that the agbias predictor obtains equivalent performance to an unbounded predictor for 15 bits of history (96.6%), and it actually obtains higher accuracy with 16 bits (96.8% vs 96.7%). But the comparison is not fair, because the agbias predictor also benefits from the BHR filtering: over 70% of all branches do not update the BHR, causing the history length of the agbias predictor to behave like if it had a 70% larger history.

This result shows that it will be very difficult to increase prediction accuracy by reducing PHT interference, as realistic predictors like agbias already reach the accuracy of an interference-free predictor. We will need to explore other techniques, like BHR filtering to further increase prediction accuracy.

But applying BHR filtering to agree[p] actually decreases prediction accuracy due to an aliasing increase (results not shown in Figure 6.8). Only branch predictors with very little interference like agbias, or the interference-free predictor will benefit from BHR filtering. The agbias predictor only increases its prediction accuracy in the *hard* component when using BHR filtering, the *easy* component proves equally accurate under both conditions (over 99.5% accuracy).

As expected, the unbounded predictor also benefits from a filtered history length, increasing accuracy of a 16-bit history predictor from 96.7% to 97.14%, the same performance as a non-filtered predictor of 22 history bits (not shown). This shows that PHT interference is not the only relevant factor to two-level prediction accuracy: the amount of *useful* information stored in the Level 1 tables is also important.

We have shown how the agbias obtains higher accuracy than any other examined predictor (including their compiler enhanced versions): first, it obtains an important interference reduction in the PHT; second, it increases the usefulness of the BHR information, increasing the potential performance of its two-level adaptive predictor components.

6.4 Conclusions

In this chapter we have shown how it is possible to improve the prediction accuracy of dynamic branch predictors using profile information and compiler optimizations.

Also, based on previously proposed predictors and taking full advantage of the compiler, we have presented the *agbias* branch prediction scheme, a static-dynamic hybrid predictor based on the division of the branch stream in four sub streams. A first division among strongly biased (*easy* branches) and not strongly biased branches (*hard* branches), and a second division among mostly *taken* and *not taken* branches.

The agbias predictor uses the *agree* predictions scheme to separate the taken and the not taken sub-streams, and uses two separate dynamic components to separate the easy and the hard sub-streams. Branches are classified using profile information, encoding the class in the instruction opcode.

We outperform all other examined branch prediction schemes for all predictor sizes, obtaining a 96.8% prediction accuracy with a 16KB predictor versus the 96.3% obtained with a compiler enhanced version of agree or bimode, reducing misprediction rate by 14%.