Dynamic Branch Prediction Study

Combining Perceptrons and Bit-Counter Predictors

Lisa M. Alano, Brad S. Boven, Andy R. Terrel

August 19, 2007

Abstract

Current processors use two bit counters for branch prediction; however, recent literature has established that the perceptron learning algorithm, a supervised learning technique, render more accurate predictions. We attempt to integrate both approaches to improve prediction accuracy through four hybrid predictors based on the 2bc-gskew predictor. In all configurations, a single perceptron predictor is still more accurate leading to an analysis of how the technologies behind perceptrons and bit counters interact.

1 Introduction

Modern processors rely on sophisticated mechanisms to exploit instruction level parallelism (ILP). For example, processors resolve control hazards by attempting to predict whether a branch will be taken or not. This technique, which is called branch prediction, which decreases potential stalls caused by control dependences. But as hardware designers pave the way for even more ILP through pipelining and other dynamic schemes, the impact of control dependencies on the effectiveness of a processor grows. Furthermore, as pipeline depths increase, the penalty for a missed prediction becomes more severe, heightening the demand for accurate predictions. However, as clock cycle time decreases and processor speeds

increase, the possibility of using complicated logic to predict these branches in one cycle is elimintated. For these reasons, fast and accurate branch prediction becomes extremely important. Given the current state of the industry, branch prediction is the number one bottleneck in ILP today [17].

Several factors influence how well branches can be predicted: how often they are taken, and whether or not nearby branches correlate. These factors determine what trade-offs are important for a given branch prediction engine, and by understanding these factors we can design branch predictors for specific classes of problems.

Early work in branch prediction [4, 14, 1, 2] focused on the use of bit counters. Manipulating these counters and incorporating ideas involving prediction history, address tagging , and branch correlation drove the design of branch predictors [4, 2]. In addition to these simple models, hybrid models were created by combining two or more of these simpler schemes. These hybrid models performed better than the single models alone. However, in recent research findings [11, 6, 5, 7], a technique based on the perceptron, a supervised learning model, has outperfromed its predecessors. In this paper, we present predictors composed of both bit-counters and perceptrons.

Once the foundation for the simple predictors had been established with two-bit counters and perceptrons, the natural progression is to implement a hybrid taking advantage of the fact that the strengths of some approaches could cover up weaknesses of others. Thus, we believe these hybrid prediction schemes will outperform existing techniques based solely on one framework or the other. We have invented four such hybrids to test this hypothesis, these include:

- 2bc-pgskew
- 2bc-ppskew
- Weighted Perceptron
- Meta Perceptron

In Section 2, we present several of the most commonly implemented predictors based on two-bit counters. We also introduce the main problems which branch prediction schemes must overcome to be effective, and how each of these bit counter schemes attempts to handle these problems. In Section 3, we introduce the idea of using the perceptron for doing dynamic branch prediction. For reference material on the perceptron, please see Appendix A. We then focus on the differences between bit-counters and perceptrons in Section 3.2, and discuss why perceptrons achieve superior performance. We then explain the ideas behind our experimental predictors and what we think they will show in Section 4. The methods for implementing and testing these new predictors can be found in Section 5. We focus on the accuracy of these predictors, leaving discussion of hardware costs of these predictors to another paper. We compare the performance of these new predictors against simple predictors and other current predictors, in Section 6. Many of these prediction schemes try to minimize error by fractions of a percent, which may seem futile. As you will see, in a large program with millions of instructions, a small improvement could impact thousands of branches and have a drastic impact on performance. Finally, we conclude in Section 7 with a look at the possibilities for extending our work in the future, and a summary of the prediction schemes presented.

2 History of Branch Prediction

Branch prediction comes in two different forms: static and dynamic. Some of the earliest Reduced Instruction Set Computers used trivial static branch prediction to improve program throughput. These architectures used a static not taken branch predictor, which fetches the next sequential instruction assuming the branch will not be taken. Another obvious alternative is to consistently predict taken. Both of these schemes fail to predict branches with an accuracy rate acceptable for existing processors, often performing at only 50-60%

accuracy [2]. A more complicated static branch prediction method called Backward Taken, Forward Not Taken (BTFNT), predicts that branches with branch targets located before the branch will be taken, and not taken otherwise. The taken predictors work better for loops because all but one of the predictions (the last one) will be correct. Forward branches usually indicate some type of exception that cause a program to jump out of a code block early. These situations happen infrequently, and thus a not taken predictor does well. In addition, the compiler can use profiling techniques to generate the branch instructions so that the most commonly taken control path is always on the taken path. Thus, the BTFNT predictor combines the strengths of both trivial predictors. While static predictors are very simple to implement, the performance benefits from these schemes pale in comparison to the dynamic schemes we introduce next. However, when no dynamic prediction information is available yet, many dynamic predictors default back to static schemes.

Dynamic branch predictors use previous branch results to increase branch prediction accuracy. By using the address of the branch, coupled with a history of recent predictions, a branch predictor can predict branch outcomes accurately. The premise of these methods is that a pattern will arise that may help in future predictions. We present a selection of many different branch prediction schemes in order of increasing complexity.

There are three different classes of branch predictors: local, global, and the combination of both local and global. Local branch prediction works well for repetitive behavior such as loops, because a specific loop will usually behave the same way each time it is executed. On the other hand, global prediction is important in understanding correlation between branches within close proximity, because branches near one another often behave similarly each time that block is executed.

2.1 One-Bit Counters

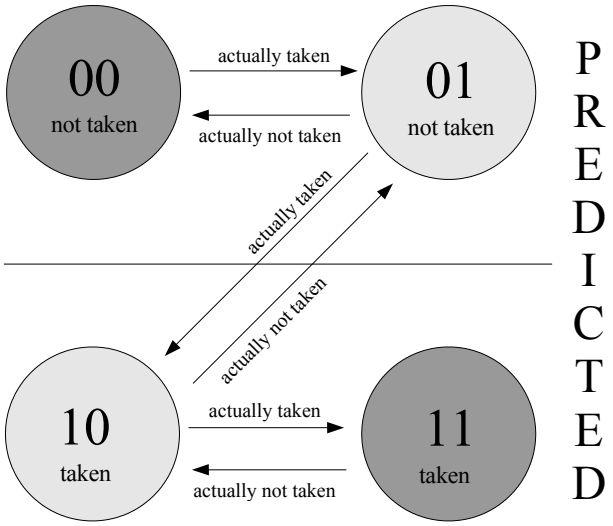
A single one-bit counter [4] is the simplest dynamic predictor. This method operates under the assumption that given any branch in a program, it is best to mimic the outcome for the most recent branch. For example, this type of predictor only misses twice for any given loop, once at the beginning of a loop and once at the end. However, the presence of other branches inside the loop could be detrimental. Suppose there is a branch inside the loop which was very rarely taken. Then each time the loop is entered, the history bit would be set to one because the loop branch was taken, and the predictor would guess that the branch inside the loop should be taken. However, it is not taken, and the history bit is set to zero. Now the processor reaches the end of the loop and the history bit is zero, so the processor predicts not taken. As one can see, this scheme would flip-flop through the entire execution of the loop, with prediction accuracy close to zero.

An extension of this idea includes a history bit for each branch in the program [4]. This would eliminate the flip-flop problem described above. However, for very large programs this table could be enormous. Instead each address is used to hash into a smaller history table. It is then important that the hash function separate addresses that are close together so that they do not collide in the table. In the event that two nearby branches collide, the type of thrashing seen in the example above is not avoided. This method is an example of a local branch predictor, where individual branch information is used.

2.2 Two-Bit Counters

Two bit counters [4, 2, 1] are logical extensions of one-bit counters, figure 1 illustrates this method.. If a branch is taken, the predictor increments a counter (initialized to 01). There are four different states: weakly taken, strongly taken, weakly not-taken, and strongly not-taken. The state the predictor is currently in at the time of the branch defines what the

Figure 1: A Finite State Machine for the bi-mod Predictor



prediction will be. Each prediction prompts a change in the state that depends on whether the prediction was correct. If the prediction was correct, the state moves from weakly taken or not-taken to strongly taken or not-taken. Conversely, incorrect predictions move the machine away from the strong state. Using this scheme, the predictor relaxes its response to incorrect predictions by allowing for two incorrect predictions before guessing the opposite choice. Notice that this uses a saturating counter where the lowest value can be zero and the largest value is limited to three (i.e 11).

2.3 The Bi-mod Predictor

The bi-mod predictor [2, 4] is an example of a local prediction scheme which uses a table of two-bit counters. The table is indexed by the address of the branch, making each entry local to that particular branch. However, conflicts exist if two addresses hash to the same spot in the table. This problem is referred to as aliasing and will be discussed in more detail later as it is one main source of problems in branch prediction. Such predictors are more

accurate than one bit predictors for loops, since they only miss once per loop if initialized to the weakly taken state.

2.4 Two-Level Adaptive Predictor

Another example of a local prediction scheme is a more complicated version of the bi-mod predictor. This scheme uses two levels of prediction. The first level table, indexed using the branch address, records k bits in some number of branch history registers, where each new bit shifts the least recent bit out. The second level table, the Pattern History Table (PHT), is indexed based upon the first level branch history pattern, where each entry in the PHT has its own two-bit counter. The two-bit counter selected based on this index is then used to predict the branch, and then this same counter is updated after the branch outcome is determined.

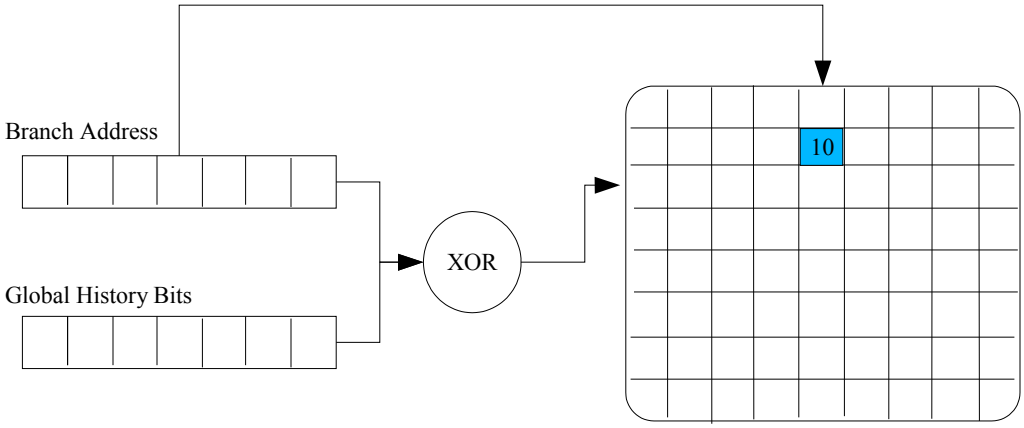
2.5 GA: Prediction Based on History

A GA predictor [2, 1] is a global two-level adaptive predictor. This method uses only one branch history register to record the outcome of all branches. The branch history register is then used as the index into a PHT. This technique takes advantage of correlations between the current branch and the other branches in the history to predict the result. It searches for repetitive patterns of global branch outcomes as opposed to keeping track of individual branch histories.

2.6 PA: Prediction Based on Address

A PA predictor [2, 1] is a per-address two-level adaptive predictor. This method uses one branch history register per branch. For a given branch, its branch history register is used as the index into a PHT just as in the GA predictor. The selected predictor is local to

Figure 2: The Gshare Predictor



the branch, so it works very well for branches that repeat the same type of behavior. For example, if every fifth branch is not taken, and taken otherwise, a per address predictor would perform well because of this repetitive behavior.

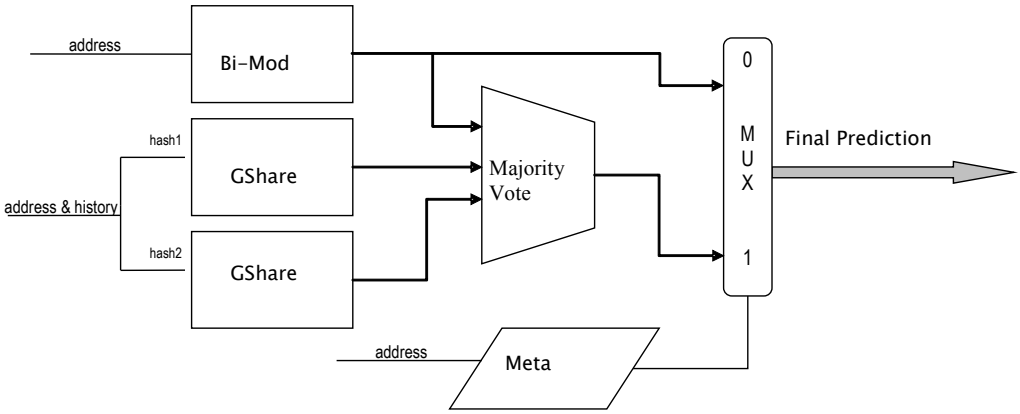
2.7 Gshare: Prediction Based on Both Address & History

The Gshare predictor [2, 1] is a variation of the GA predictor and is shown in figure 2. Gshare takes the global branch history register and XORs it with a portion of the fetch address to compute the index into the PHT. This method works better than a GA, because it reduces the chance that two identical patterns generated by different blocks of code will collide in the PHT.

2.8 De-Aliasing Predictors

De-aliasing predictors try to solve the problem of multiple branches being hashed to the same entry in a table. To avoid this conflict, we can skew two identical predictors, that is

Figure 3: The 2bc-gskew Predictor



hash the inputs to these predictors to different spots in the PHT. Thus, if two addresses map to the same entry in one predictor, we can be fairly certain that they will not map to the same entry in the second predictor.

2bc-gskew [14] uses two Gshare predictors, and one bi-mod predictor as shown in figure 3. The input to the two Gshares is the same global history information, however, the history is hashed into the PHT using two different hashing functions. The predictions of the bi-mod and both gshare predictors are then subject to majority voting. If all three predictors do not agree, the prediction with two votes is what is trusted. If the gshare predictors disagree, this means that one of the predictors must have had a collision in the table, and thus the bi-mod predictor will be the deciding vote. A meta predictor is also present to officiate between the output of majority vote and the bi-mod predictor. The meta predictor is used to solve the learning latency problem present in the gshare predictors. Since these predictors are based on global history, it takes k number of branches for the history to fill up, where k is the number of history bits being used.

2.9 Hybrid Predictors

Hybrid predictors capitalize on all the predictors discussed up to this point. Each different predictor has different strengths and weaknesses. Hybrid predictors attempt to fill in the gaps by combining simpler predictors. Notice that the 2bc-gskew is a hybird predictor. However a problem of the predictors is feasibility. How to capture the best aspects of a number of predictors, while at the same time keeping computational and hardware costs to a minimum. For instance, combining all of the predictors discussed up to this point way might yield the best predictor yet, but how long would it take to make a prediction, and would the accuracy achieved really be profitable compared to what you might achieve from a much faster prediction technique? This is the direction that dynamic branch prediction has been moving, and we investigate and propose several new hybrid schemes in the following sections.

2.10 Performance

Each of these bit counter techniques: bi-mod, PA's, GA's, Gshare, and 2bc-gskew all try to solve a different problem in branch prediction. While some use local prediction techniques, other use global, and predictors like Gshare and 2bc-gskew use both. While bi-mod is efficient and very fast, it is the least accurate predictor. Gshare and 2bc-gskew take more hardware to implement, but are much more accurate. The accuracy of these predictors is somewhat application specific, but generally, bi-mod performs worst, while gshare and 2bc-gskew perform the best. This is intuitive because both Gshare and 2bc-gskew have more information at their disposal. The PA and GA predictors fall somewhere in between [2].

3 Branch Prediction with Perceptrons

In recent literature the perceptron predictor has been shown to be the most accurate branch predictor [10]. The perceptron learning algorithm, a method for supervised learning, enables a perceptron to learn whether a branch is taken or not taken, for more detail see Appendix A. Following the initial proposal by Jimenez and Lin, a large number of combinations of perceptron predictors have been developed. This section discusses both issues with implementing the initial perceptron predictor and combinations of the perceptron predictor.

3.1 Implementing a Perceptron Predictor

Perceptron predictors keep a weight table, indexed by the branch address. Each line holds the weights corresponding to the input of the predictor necessary to comput the output. To implement the predictor on the chip, it is important to note that the dot product of the weight and history vectors will compute a sum of weights chosen by the history bits. To sum these values in parallel, a Wallace adder tree is used [8]. The update function can be applied later in the pipeline, after the branch condition has been computed.

Jimenez's original work [7] proposed using only the global history for the input to the predictor. But a perceptron predictor can take the global history, the local history, or a combination of the two as input. The perceptron is able to learn which bits are the most important for the prediction and develop a hyperplane off of those bits, effectively correlating branches when the perceptron updates itself. This feature allows the percepton to make use of longer histories than what is used by the two bit counter perdictors presented in section 2. This is because the history becomes an input to the perceptron instead of a way to pick out a counter as done in Gshare.

3.2 Bit Counters versus Perceptrons

Both bit counter schemes and perceptrons try to accurately predict the future of a branch based on its past history, e.g. two-bit counters predict based on the last three outcomes of a branch, while perceptrons train on the entire history of a branch. These differences affect how the different paradigms handle two major branch prediction problems: branch correlation and aliasing.

3.2.1 Branch Correlation

Two level bit counter schemes use methods to distinguish the branch according to its global and/or local history and then predict the current branch based on the last two outcomes of the branch in a specific situation. This strategy relies on having the distinguished branch learn the situation where the branch is correlated to another branch and then letting this distinguished branch point to a spot in the PHT. This method is very inefficient. Consider a gshare predictor that uses fifteen bits of global history to address the second level of the PHT. If there are only three bits out of the fifteen that are correlated to the branch, then the values of the twelve other bits should not matter; however, the branch could be represented 2^{12} different ways and point to different slots in the PHT.

Perceptrons, on the other hand, use a learning algorithm to learn what the prediction should be based off of, either the global and/or local history. Branches that correspond to different bits in the input will be appropriately weighted. In the above example, the branch will only need one line of the weight table to learn the correlations.

3.2.2 Aliasing

For a two-bit counter, two branches can interfere with each other when they map to the same spot in the PHT. This makes aliasing a non-recoverable error. If these conflicts produce the wrong prediction, it will reproduce that wrong prediction every time. To de-alias a

bit counter predictor, there are two mechanisms: add more slots in the PHT or skew two predictors. Adding more slots reduces the number of conflicts and by skewing the conflicts can be overrided by other predictors that hopefully do not have the same conflict.

Perceptrons are less susceptible to aliasing problems because the perceptron learns the input of both branches. As long as the outcomes (taken or not taken) between the conflicting branches are linearly separable, the perceptron will be able to learn the conditions for both branches. This affect is most apparent in the more accurate 64Kb perceptron predictor with 250 lines in its weight table as opposed to the gshare with 32,000 slots in its PHT.

3.3 Perceptrons in other predictors

Perceptron predictors have two major drawbacks: latency in the learning algorithm and complexity of the added hardware. But even with these drawbacks, perceptrons can generally outperform both gshare and 2bc-gskew on similar hardware budgets [8].

Perceptrons have been recently used within a number of other predictors [11, 3, 13, 16, 9] and appeared in all of the predictors in the Championship Branch Prediction Competition [15]. While the perceptron is a relatively new idea in branch prediction, coupling predictors is not. One natural extension of the perceptron predictor is to see how many ways it can be used in hybrid predictors and how that will effect both the results of the predictor and the hardware costs.

One interesting use of perceptrons is to use a better weighting mechanism. Currently, hybrid predictors are combined using either majority vote or a meta predictor [2] that uses counters to predict which predictor will be correct. We propose using several predictors that feed input into a perceptron, which weights these predictions and predicts accordingly. Loh and Harry [10], describe such a predictor. Another interesting approach is to use gshare predictors with global histories of various lengths as input to a perceptron [13].

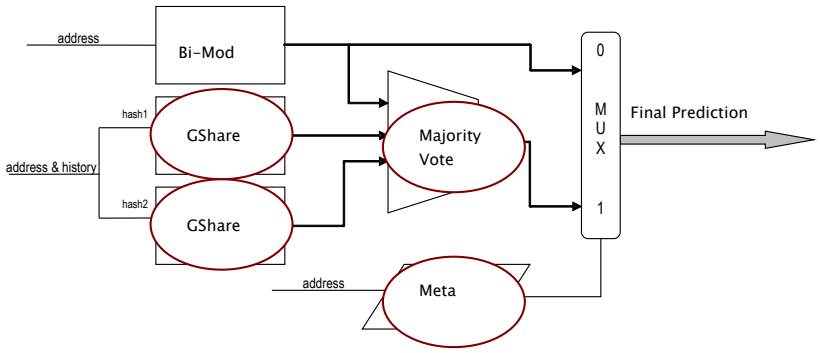
Perceptrons have been used in a number of hybrids [9, 11]. These predictors alleviate

aliasing, which increase the probability of non-linearly separable. Such a predictor as the Frankenpredictor [9] achieves de-aliasing by the skewing technique and by combining with other predictors. By using different inputs such as the address and history, the problem of non-linearly separable data can be slightly alleviated since the data may be linearly separable when using another input [11].

For certain hardware budgets, each of these methods have made improvements over the original perceptron predictor. We propose predictors to use some of these different ways of combining perceptrons to develop a better prediction scheme.

4 Proposed Branch Predictors

Figure 4: The structure of 2bc-gskew predictor and the encircled components where perceptrons may be used.

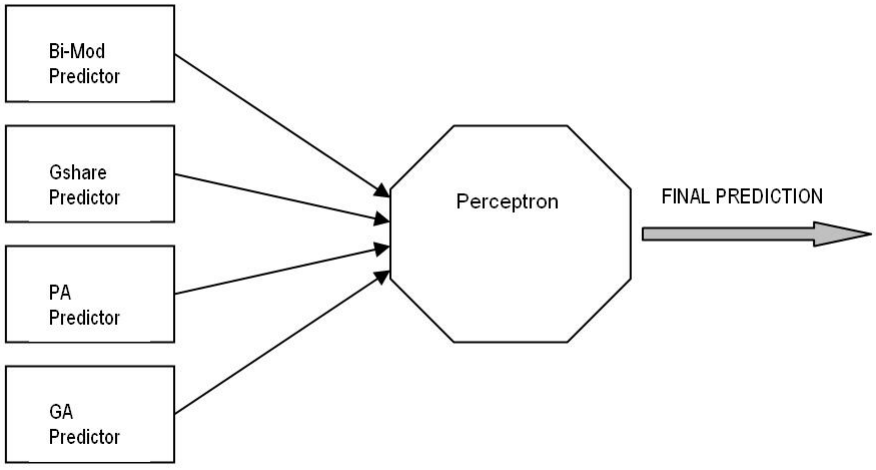


In our work, we combine the well-established bit counter schemes with the novel perceptron model. We hypothesize that combining both ideas will produce a hybrid of predictors that will have better prediction accuracy. Specifically, we manipulate the best performing two-bit counter predictor, the 2bc-gskew, which was introduced by Seznec [14]. We propose four components where perceptrons can be inserted, (see figure 4) and we shall discuss below

the motivation for such changes.

4.1 weighted-voting predictor

Figure 5: The structure of the weighted predictor.

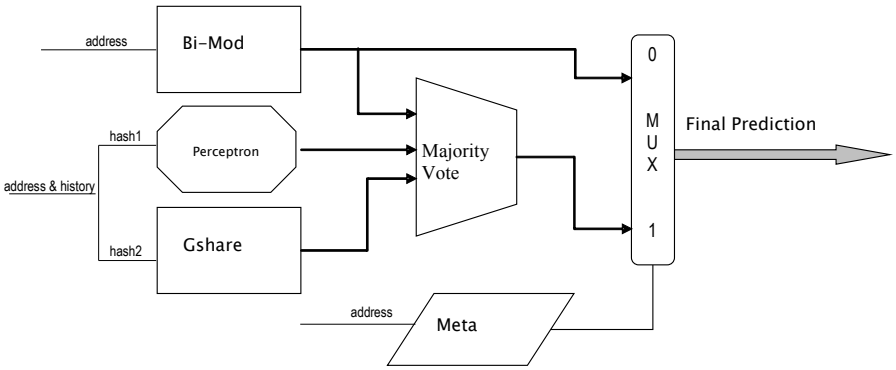


Our first proposed predictor, the weighted predictor, replaces the majority voting mechanism of 2bc-gskew with a perceptron. The motivation for this predictor is that for each branch, one predictor will be more accurate. Therefore, we use the perceptron to identify which predictor best suits the branch. As an analogy, consider the situation where a person has to make a decision and seeks recommendation from a group. It is common to give more importance to the person that has given correct advice in the past. In the weighted predictor, as before, each predictor gives its prediction but more consideration is given to the predictor that has been more accurate in past predictions. Notice that one predictor may do well on one branch, but perform poorly on another. This situation correlates to getting good advice from a math professor about calculus and obtaining bad advice from the same person about salsa dancing. To capture this information, the perceptron adjusts it weights to give appropriate weights to each predictor based on accuracy. As show in figure 5, each predictor

gives its prediction to the perceptron, and the perceptron determines the final outcome.

4.2 pgskew predictor

Figure 6: The structure of the pgskew predictor.



Our second proposed predictor is the pgskew predictor. This predictor is based on the 2bc-gskew predictor, but we do not use two gshare predictors that skew their hashed addresses. Instead, we use one gshare predictor and one perceptron predictor. Since perceptrons outperform the two-bit counters, giving the perceptron a vote may make the hybrid perform better.

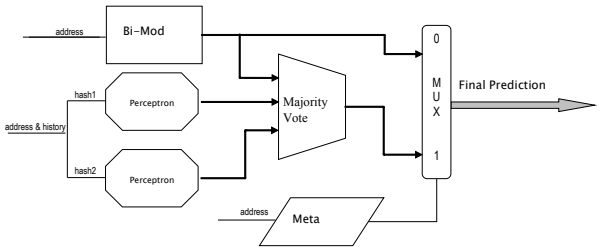
The resulting predictor is a hybrid composed of bi-mod, gshare, and perceptron predictors, that all make individual predictions. The final outcome is determined by the dominant prediction. Using a majority vote to select the outcome, common mispredictions will be eliminated, increasing the accuracy of the prediction [14].

This combination integrates each predictor's strengths. Bi-mod specifically works well with biased branches that are usually the same (taken or not taken), and has a low learning latency. However, it fails to do well for branches that have an alternating pattern of taken and non-taken branches. Gshare on the other hand, can detect such a pattern, but has

aliasing problems. The perceptron predictor can detect this pattern as well, but handles aliasing better. While bi-mod takes advantage of local history, and both the perceptron and the gshare use both local and global history, these last two predictors have been shown to perform more accurately in general. However the bi-mod prediction, having a low learning latency, will be used until the other two are ready. In the case of tie, bi-mod shall be the deciding vote as before.

4.3 ppskew predictor

Figure 7: The structure of the pgskew predictor.



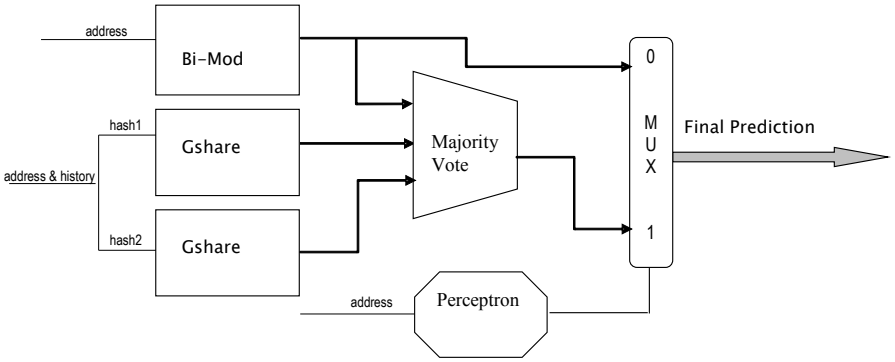
Ppskew is our third predictor, based on the same 2bc-gskew structure but with perceptrons instead of gshare predictors. Gshare was used in 2bc-gskew because it was the best performing two-bit counter predictor at that time, but now that the perceptron has been shown to outperform its predecessors. The mechanics of the hybrid predictor remain the same. Bi-mod predictions are used while the perceptrons warm up. The meta predictor functions as previously discussed as well. The two perceptron predictors are skewed, to avoid accessing the same entry of weights for each perceptron predictor. This causes collisions to hit different perceptrons in each perceptron predictor.

An important feature about ppskew, as opposed to pgskew is the perceptron cannot be outvoted by bit counter. The event may occur in pgskew, where the perceptron produces a

correct prediction, but disagrees with both predictions of bi-mod and gshare. This situation is avoided in ppskew.

4.4 meta-perceptron predictor

Figure 8: The structure of the meta-perceptron predictor.



This last predictor, the meta-perceptron predictor, uses a perceptron instead of a meta-predictor. Its role is to choose between the bi-mod and the majority vote predictions, with the goal to more accurately learn the precise time when gshare is ready to produce accurate results. In 2bc-gskew and these other proposed predictors, there is no specific rule indicating how many history bits gshare should see before it produces good predictions. This highly depends on the branch pattern, making it difficult to speculate when the warm-up phase ends. In our implementations, we decided to give gshare at least two history bits before it is considered by the meta-predictor. With the perceptron working as the meta-predictor, we do not have to make this decision. The perceptron will identify based on accuracy, when gshare should be trusted. As the majority vote gives more accurate predictions, the appropriate weights will be adjusted.

A major concern for the setup of this hybrid predictor is the number of inputs into the

perceptron. Having only two inputs that each have only two possible values, determining a separating hyperplane for all branch inputs is improbable. But for completion, we implement this predictor as well.

5 Methods

5.1 Testing Framework

After implementing several published predictors and our proposed predictors, we used the Championship Branch Prediction (CBP) framework [15] to test the accuracy of each predictor. This framework was used in a 2004 branch prediction competition held by the Journal of Instruction Level Parallelism in cooperation with IEEE and Intel. Current literature introducing predictors with perceptrons use this framework [15] because it provides a common evaluation tool that allows independent verification of results. Benchmarks are provided with twenty trace files of four applications areas: floating point, integer, mutlimedia, and server.

While each trace file has thirty million instructions, we do not claim these traces to represent all types of programs. However, it is a reasonable basis to compare our work with published predictors.

5.2 Metrics

We consider two standard metrics for our comparisons. We measure the number of mispredictions per 1,000 instructions and measure the ratio of mispredictions to the total number branches. The first metric is the standard metric used in formal literature, but the second metric gives a more tangible assessment of how many more (or less) branches we accurately predict.

We report the mean of the five traces given for each application area mentioned, and the overall mean. This latter mean may not reflect individual traces that may generate opposite results, but it shows general performance. Certain predictors may perform best with certain applications, and we shall recognize them as such, but our studies focus on the predictor that would be best suited for a general-purpose machine.

5.3 Optimized Settings

To compare our proposed predictors with the bit-counter prediction scheme, we have implemented both gshare and 2bc-gskew predictors. Gshare is the best 2-level adaptive branch predictor, and the 2bc-gskew is the best hybrid predictor using bit counters. Also, we implemented the basic perceptron, the current front-runner in branch prediction.

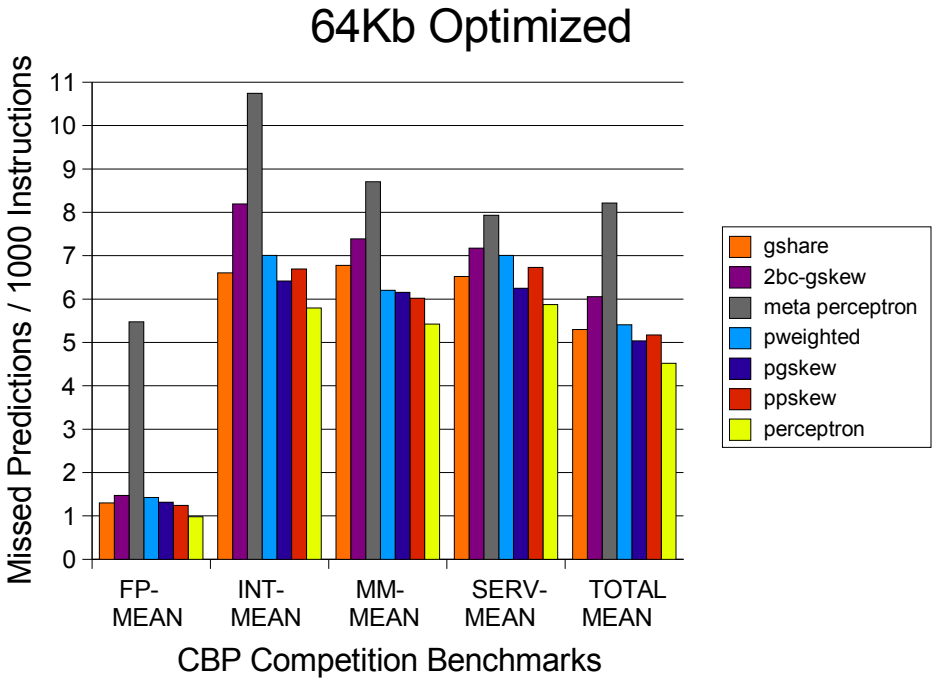
These predictors as well as our proposed predictors were all implemented with a fixed 64 Kb hardware budget to work with. However, one major concern is finding the optimal distribution of the budget per predictor to allow each predictor to perform at its best. We had to conduct additional tests per predictor in order to find the optimal settings, and this can be seen in the following table. Other hardware budget sizes were also studied, and this can be viewed in appendix B.

Figure 9: Optimized hardware distribution per predictor.

PREDICTOR	***gshare A table size***	***gshare B table size***	***bimod table size***	***meta table size***	***perceptron A table size***	***perceptron A history bits***	***perceptron B table size***	***perceptron B history bits***
gshare	32,000	-	-	-	-	-	-	-
2bc-gskew	8192	8192	8192	8192	-	-	-	-
pweighted	8192	8192	4096	-	1000	3	-	-
pgskew	8192	-	2048	2048	300	24	-	-
ppskew	-	-	8192	4096	124	20	124	20
meta-perceptron	8192	8192	2048	-	1695	8	-	-
perceptron	-		-	-	250	32	-	-

6 Results

Figure 10: Performance of all predictors.



6.1 Analysis of Results

Having simulated all predictors with 64Kb, the perceptron continues to dominate prediction accuracy. Ppskew, pgskew, and gshare perform similarly, with pgskew generally doing better as reflected in the total mean of figure 10. Pweighted does worse than pgskew and ppskew in all traces, and the meta perceptron predictor predicts least accurately across all tests. For specific statistics of the number of misses per 1000 instructions and the ratio of misses to total number of branches, refer to figure 13 and figure 14.

The perceptron predictor's precision is attributed to the learning nature of its update function. As explained, this predictor only has 250 entries in its table compared to the 32,000 entries of gshare. However, their bar graphs display the perceptron leading with a

significant margin, affirming that perceptrons deal with aliasing well.

The perceptron also outperforms pgskew and ppskew because these two predictors had to decrease the number of entries in the perceptron tables to make room for other components in the hybrids. Pgskew had to divide the 64Kb budget between the bi-mod, gshare, perceptron and meta perdictors. Consequently, the perceptron encountered more aliasing problems. Although still expected to outperform the other predictors in the hybrid, its more accurate results could also have been outvoted.

Ppskew, competitively performing with pgskew, had to divide its hardware allowance as well. Unlike in the pgskew, the perceptron predictions will never be outvoted on the majority vote. With skewing, each perceptron predictor encounters a different set of collisions, thus learning on a different set of data. This increases the probability that at least one perceptron is learning on linearly separable data. Although it should have performed worse than pgskew because the number of entries per perceptron table had to be decreased yet again, skewing improved the overall performance. In a later section, this shall be verified. For these reasons, ppskew and pgskew are comparable, both doing better than gshare and 2bc-gskew.

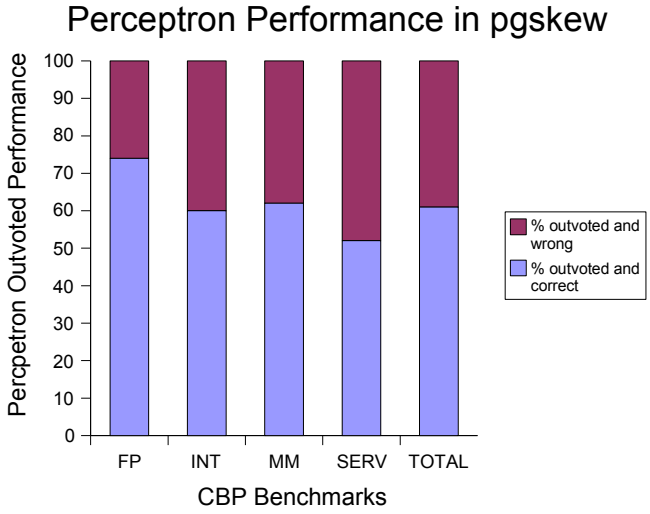
The weighted voting predictor did not behave as expected because the perceptron only took three data inputs per branch. With three data inputs and each input only having two possible values, there are only eight possible sets of data inputs. This observation quickly affects the perceptron's ability to find a separating hyperplane for all data sets, especially with collisiong occur more frequently. This situation explains the extreme behavior of the meta-perceptron predictor as well. Perceptrons clash with the simple mechanisms of the meta-predictor and majority voting.

6.2 Verification

To verify that the perceptron predictor in pgskew was indeed being outvoted, we ran the benchmarks and collected statistics specific for perceptron accuracy. As seen in figure 11

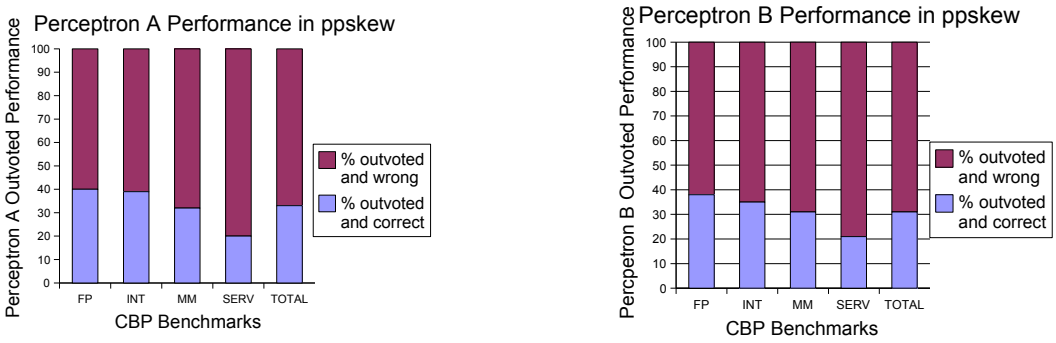
60% of the time that the perceptron predictor is outvoted, it is actually giving a correct prediction. The perceptron was outvoted 2% of the time, which is roughly 69,600 branches. This implies that the perceptron could have increased the accuracy by 38,800 correct branch predictions.

Figure 11: Performance of the Perceptron predictor in pgskew.



Replacing the two gshare predictors with perceptrons helped improve 2bc-gskew. The negative effect of more branch collisions was countered because skewing segmented the data to be more linearly separable. Compared to the perceptron in pgskew, figure 12 shows that the perceptrons in ppskew drop to 33% and 31% of being outvoted but predicting correctly. Despite this decrease, their overall accuracy does not differ by much with pgskew. Notice also that each perceptron behaves independently, affirming that each perceptron is encountering different aliasing problems. This explains why ppskew is performing better than it should.

Figure 12: Performance of both Perceptron predictors in ppskew.



7 Conclusion

Branch prediction is the major limiting factor in Instruction Level Parallelism today. Therefore, fast and accurate branch prediction is crucial in designing a processor. Although older bit counter schemes have merit in that they are very fast, they are not as accurate as the emerging perceptron predictors.

Because bit counters and perceptrons are based on very different technologies, each handles aliasing and correlation in different ways. Perceptrons are most effective; their supervised learning technique is able to learn the correlation between branches and train on several branches at once reducing the error from aliasing conflicts. Thus, the perceptron is the most accurate predictor available today.

We proposed hybrids of these two schemes, bit counters and perceptrons, believing that this combination would yield a more accurate predictor. Unfortuantely, bit counters hinder the perceptron by taking away hardware bits for the perceptron to use and giving them to the bit counters in the hybrids, and by outvoting the perceptron in majority vote situations. This degrades the performance of the hybrid predictors, and makes them less accurate than a perceptron predictor, as consistantly seen in all of our results.

Although our tests are by no means conclusive that no bit counter and perceptron hybrid

will ever perform as well as the perceptron under a given hardware budget, the evidence is strong that perceptron will outperform most hybrids.

In future work, some areas of improvement include improving the perceptron, evaluating hardware budgets for implementing hybrids that might be more cost effective than the perceptron, and running more tests for completing the optimization over our hybrid predictors.

Figure 13: Miss per 1000 instructions from Championship Branch Prediction trace files.

test	gshare	2bc-gskew	meta perceptron	pweighted	pgskew	ppskew	perceptron
FP-1	3.6780	4.0920	5.5260	3.8810	3.7500	3.5310	2.3930
FP-2	1.1110	1.3940	4.1060	1.1310	1.1430	1.1830	1.0980
FP-3	0.4420	0.4330	0.6610	0.4430	0.4330	0.4340	0.4330
FP-4	0.3170	0.3430	0.8370	0.3060	0.3060	0.2780	0.1910
FP-5	0.9570	1.0900	16.2520	1.3750	0.9400	0.7910	0.7920
INT-1	7.9070	9.4090	15.7910	8.6060	7.8370	8.3980	7.2930
INT-2	8.7080	9.8210	12.4290	9.0690	8.6470	10.1510	9.0640
INT-3	13.3570	18.5630	16.9530	13.4460	12.5370	11.1620	9.4440
INT-4	2.4470	2.6650	7.7810	3.4040	2.6240	3.3340	2.7840
INT-5	0.6140	0.5100	0.7600	0.5060	0.4500	0.4220	0.3820
MM-1	9.1190	10.0030	10.7830	8.4210	8.4870	8.4770	7.4480
MM-2	10.8540	12.5460	12.5630	10.2600	10.2460	10.3570	9.8930
MM-3	4.9330	4.6350	8.9840	4.3420	4.2300	3.2330	1.9070
MM-4	2.1350	2.2880	3.0480	2.0330	2.1170	1.9090	1.5110
MM-5	6.8500	7.4800	8.1610	5.9590	5.7040	6.1120	6.3500
SERV-1	5.6910	7.0140	7.6600	7.0970	6.0100	6.6430	4.8280
SERV-2	5.8970	7.2350	7.4540	7.2710	6.1910	6.8470	5.0560
SERV-3	7.2350	7.1330	9.3110	6.7850	6.6500	7.0700	7.4110
SERV-4	7.0090	7.2160	7.5730	6.9200	6.3540	6.4850	6.3590
SERV-5	6.7670	7.2860	7.6530	6.9550	6.0550	6.6080	5.7210
FP-MEAN	1.3010	1.4704	5.4764	1.4272	1.3144	1.2434	0.9814
INT-MEAN	6.6066	8.1936	10.7428	7.0062	6.4190	6.6934	5.7934
MM-MEAN	6.7782	7.3904	8.7078	6.2030	6.1568	6.0176	5.4218
SERV-MEAN	6.5198	7.1768	7.9302	7.0056	6.2520	6.7306	5.8750
TOTAL MEAN	5.3014	6.0578	8.2143	5.4105	5.0355	5.1712	4.5179

Figure 14: Percentage of mispredictions based on total number of branches from Championship Branch Prediction trace files.

test	gshare	2bc-gskew	meta perceptron	pweighted	pgskew	ppskew	perceptron
FP-1	2.53	2.81	3.80	2.67	2.58	2.43	1.64
FP-2	0.76	0.96	2.82	0.78	0.79	0.81	0.75
FP-3	0.30	0.30	0.45	0.30	0.30	0.30	0.30
FP-4	0.22	0.24	0.58	0.21	0.21	0.19	0.13
FP-5	0.66	0.75	1.17	0.95	0.65	0.54	0.54
INT-1	5.44	6.47	0.85	5.92	5.39	5.77	5.01
INT-2	5.98	6.75	8.54	6.23	5.94	6.98	6.23
INT-3	9.18	2.76	1.65	9.24	8.62	7.67	6.49
INT-4	1.68	1.83	5.35	2.34	1.80	2.29	1.91
INT-5	0.42	0.35	0.52	0.35	0.31	0.29	0.26
MM-1	6.27	6.88	7.41	5.79	5.83	5.83	5.12
MM-2	7.46	8.62	8.63	7.05	7.04	7.12	6.80
MM-3	3.39	3.19	6.18	2.98	2.91	2.22	1.31
MM-4	1.47	1.57	2.10	1.40	1.46	1.31	1.04
MM-5	4.71	5.14	5.61	4.10	3.92	4.20	4.36
SERV-1	3.91	4.82	5.26	4.88	4.13	4.57	3.32
SERV-2	4.05	4.97	5.12	5.00	4.26	4.71	3.47
SERV-3	4.97	4.90	6.40	4.66	4.57	4.86	5.09
SERV-4	4.82	4.96	5.20	4.76	4.37	4.46	4.37
SERV-5	4.65	5.01	5.26	4.78	4.16	4.54	3.93
FP-MEAN	0.89	1.01	3.76	0.98	0.90	0.85	0.67
INT-MEAN	4.54	5.63	7.38	4.82	4.41	4.60	3.98
MM-MEAN	4.66	5.08	5.99	4.26	4.23	4.14	3.73
SERV-MEAN	4.48	4.93	5.45	4.81	4.30	4.63	4.04
TOTAL TOTAL MEAN	3.64	4.16	5.65	3.72	3.46	3.55	3.11

References

[1] M. Evers, S. Patel, R. Chappell, and Y. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. *ISCA*, pages 52–61, 1998. Available from: `http://citeseer.ist.psu.edu/166658.html`.

[2] M. Evers and T.Y. Yeh. Understanding branches and designing branch predictors for high-performance microprocessors. *Proceedings of the IEEE*, 89(11):1610–1620, November 2001. Available from: `http://www.andrew.cmu.edu/user/cfchen/readings/arch/branch-pred.pdf`.

[3] A. Falcón, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet-critic hybrid branch prediction. *31st Annual International Symposium on Computer Architecture*, pages 250–262, June 2004. Available from: `http://citeseer.ist.psu.edu/712585.html`.

[4] J. Hennessy and D. Patterson. *Computer Architecture : A Quantitative Approach; third edition*. Morgan Kaufmann, 2003.

[5] D. Jimenez. Fast path-based neural branch prediction. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003. Available from: `http://citeseer.ist.psu.edu/jimenez03fast.html`.

[6] D. Jimenez. Piecewise linear branch prediction. *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, June 2005. Available from: `http://camino.rutgers.edu/isca05_dist.pdf`.

[7] D. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 197–206, 2001. Available from: `http://camino.rutgers.edu/ut/hpca7_dist.pdf`.

[8] D. Jimenez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, 2002. New York, NY, USA. Available from: `http://doi.acm.org/10.1145/571637.571639`.

[9] G. Loh. The frankenpredictor: stitching together nasty bits of the other branch predictors. *1st Championship Branch Prediction Contest (CBP1)*, pages 1–4, December 2004. Portland, OR, USA. Available from: `http://www-static.cc.gatech.edu/~loh/Papers/cbp-2004.pdf`.

[10] G. Loh and D. Henry. Predicting conditional branches with fusion-based hybrid predictors. *11th Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–176, September 2002. Charlottesville, VA, USA. Available from: `http://www-static.cc.gatech.edu/~loh/Papers/pact2002-pf.pdf`.

[11] M. Monchiero and G. Palermo. The combined perceptron branch predictor. *Proceedings of Euro-Par'05 – Parallel Computer Architecture and ILP Track*, Septmeber 2005. Available from: `http://www.elet.polimi.it/upload/monchier/home_files/monchiero05combined-europar.pdf`.

[12] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approad; second edition*. Prentice Hall, 2002.

[13] A. Seznec. Genesis of the ogehl predictor. *Journal of Instruction Level Parallelism*, April 2005. Available from: `http://www.irisa.fr/caps/people/seznec/JILP-OGEHL.pdf`.

[14] A. Seznec and P. Michaud. Dealiased hypbrid branch predictors. *Technical Report PI-1229, IRISA*, February 1999. Available from: `ftp://ftp.inria.fr/INRIA/tech-reports/publi-pdf/RR/RR-3618.pdf`.

[15] J. Stark and C. Wilkerson. Introduction to jilp's special addition for finalists of championship branch prediction competition. *Journal of Instruction Level Parallelism*, 7:1–4, April 2005. Available from: `http://www.jilp.org/vol7/v7paper5.pdf`.

[16] D. Tarjan and K. Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Transactions on Architecture Code Optimization*, 2(3):280–300, 2005. New York, NY, USA. Available from: `http://doi.acm.org/10.1145/1089008.1089011`.

[17] D. Wall. Limits of instruction level parallelism. *David W. Wall, Limits of Instruction Level Parallelism, Proc. 4th ASPLOS,*, 1991. Available from: `http://citeseer.ifi.unizh.ch/wall90limits.html`.

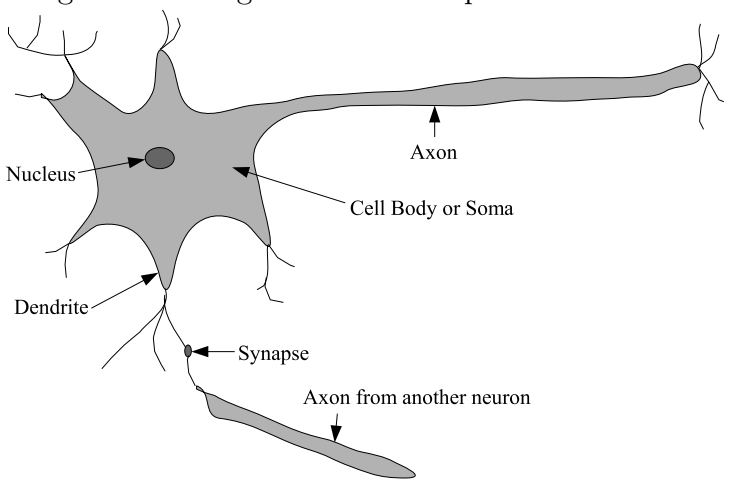
Appendix

A Perceptrons

In 2001, D. Jimenez proposed to using a perceptron to replace the usual two bit counters for branch prediction [8]. The basic idea is to learn the behaviour of branches as a program executes. Perceptrons are a method of learning employed by artificial intelligence.

The perceptron was first proposed by McCullough and Pitts in 1947 to replicate the firing of neurons in the brain [12]. A neuron, or nerve cell, is the principal cell in the human nervous system. Each neuron consists of three basic parts: a cell body or *soma*, *dendrites*, and an *axon*, see Figure 15. Neurons propagate signals from one neuron's axon to another neuron's dentrites. On the end of each dendrite is a synapse, which through an electro-chemical reaction produces a signal that passes through the dendrite into the soma. Upon receiving signals from the denrites, the neuron may or may not send a signal through its axon which is most likely connected to synapses of other neurons. When this signal is sent the neuron is said to be "firing." While it is not clear how the neuron learns when and when

Figure 15: Rough sketch of the parts of a neuron



not to fire, many models have been proposed to replicate this action, the simplest being the perceptron [12].

A.1 Definition of Perceptron

The perceptron model, see Figure 16, assumes that each signal coming into the neuron can only take some small number of values, and that the neuron will fire when a linear combination of these values exceed some threshold. To determine this linear combination, the perceptron keeps a weight, w_i, for each value, v_i, that it receives. The perceptron sums the weight times the value and applies a threshold function, Θ, to determine the output for the perceptron, typically 1 for firing and 0 for not firing. Thus for a perceptron with n inputs we have the output, a as:

$$a = \Theta\left(w_0 + \sum_{i=1}^{n} w_i \cdot v_i\right)$$

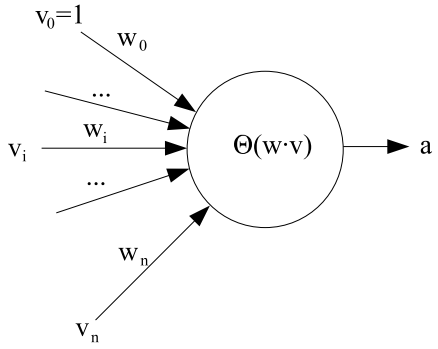
Often w_0 is called the bias weight and included in the dot product by setting $v_0 = 1$. For the purposes of branch prediction, the threshold function is usually:

$$\Theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Where one denotes taking the branch and zero denotes to not take the branch. Thus a perceptron consists solely of a vector of weights, w, and its threshold function, Θ. The input to the perceptron is a vector of values, v.

The learning process of the perceptron takes place as the weights are updated after each output. If the perceptron gives the correct output, then it is not necessary to update the

Figure 16: The perceptron model of a neuron.



weights. If the perceptron is incorrect, then the weight vector is updated as follows:

$$w = \begin{cases} w + v & \text{if } a = 0 \\ w - v & \text{if } a = 1 \end{cases}$$

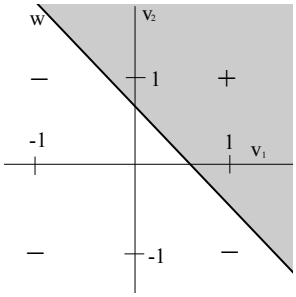
Through this updating process, the perceptron will eventually arrive at the correct weight vector that produces the correct output on every input, if the the input is linearly separable.

A.2 Perceptron Uses

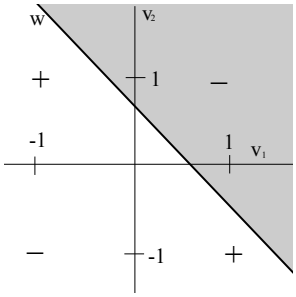
The mathematical model presented for the perceptron allows for the classification of the input into one of two classes. The weight vector defines a hyperplane in n dimensions, and the threshold function of the weight vector dotted with the input values tells which side of the hyperplane the input appears, see Figure 17(a) for an example. In this case the bias weight gives the defined hyperplane the ability to shift away from the origin, thus classifying more types of functions. Whenever the perceptron gives the wrong answer, the update shifts the hyperplane, the shifted hyperplane will then be able to correctly classify the point while maintaining the correct classification from previous points.

This model is cannont learn nonlinearly separable data. The classic case of nonlinearly

Figure 17: Two examples of a perceptron classifying functions with $n = 2$. 17(a) The And function is linearly separable and thus correctly classified; 17(b) the XOR function is not linearly separable and not correctly classified.



(a) Example of AND function.



(b) Example of XOR function.

separable data is the exclusive or function, XOR, see Figure 17(b). The XOR function cannot be classified by a single hyperplane and thus not by a perceptron. The solution is to use a network of perceptrons. A network of perceptrons has a layer of perceptrons whose output feeds into the next layer of perceptrons until the output. Since each output depends on the weights and values of the perceptrons in the previous layer, there is not a good update rule that will guarantee the network is learning the data. The most used heuristic is the Back Propagation Algorithm, as algorithm based on the method of greatest descent. Because of the complexity of this algorithm, it is not suitable for branch prediction since all computations must take place on the chip itself.

B Optimizing Our Predictors

The comparison of the each predictor in our results, section 6, requires that each predictor is optimized to some degree. Our decision to focus our report around the 64Kb predictor, was based only on the comparison of results in the Championship Branch Prediction Competition [15]. Since our predictors should be able to perform at different sizes, this section discusses both optimizing our predictors at the 64Kb sizes and comparing our predictors at different sizes.

B.1 Optimizing each predictor

True optimization of our predictors would require hundreds of tests and the use more than just one benchmark suite. Since our time has been limited, we decided to run five different configurations for each predictor that we implemented, except for the gshare which has only one table, to get a good feel for how an optimized predictor should be configured. In the two-bit counters the table size determines the predictor; whereas, in the perceptrons each table size can be obtained through several combinations of input lengths and number of lines in the weight table.

Here we present the different configurations we tested and their results. For our work, we chose the configuration with the smallest total mean of mispredictions per thousand instructions in the CBP benchmark. While this might not always be the best measure of configuration, especially in cases where a predictor is worse in every category but one, we had to pick a single metric to choose. The graphs of results provided give the mean of each application area, in case the reader would like to apply a different metric for the predictor configuration, e.g. best floating point predictor.

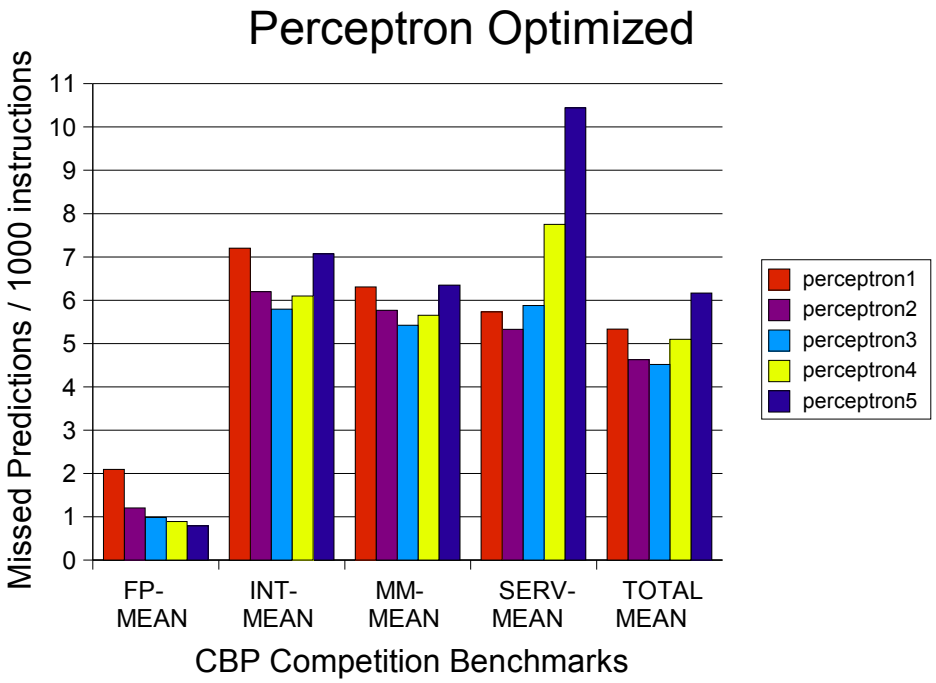
For the perceptron predictor, there was a trade off between number of global history bits used and number of lines in the table. The ppskew predictor did best with a small bi-

mod and meta predictors leaving more space for the two perceptron predictors. The pgskew predictor likewise did better with a small bi-mod and meta predictors leaving more space for the gshare and perceptron predictor. The weighted predictor did best with a smaller bi-mod, larger gshare, and larger perceptron predictors. The meta predictor varied the least with different configurations but did best when one gshare predictor was smaller than the other predictors. Finally the 2bc-gskew predictor did best when each predictor was given about the same amount of space.

Figure 17: Perceptron Predictor optimization

	global hist. bits	lines
perceptron1	8	1000
perceptron2	16	500
perceptron3	32	250
perceptron4	48	125
perceptron5	64	63

(a) Table of configurations

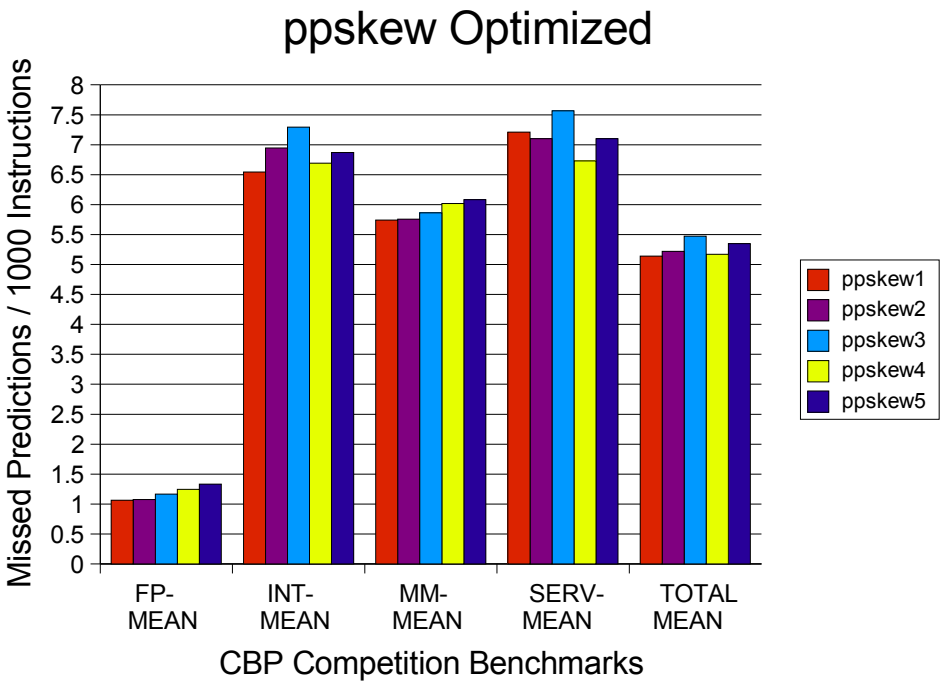


(b) Bar graph of results

Figure 18: ppskew Predictor optimization

	bi-mod size	meta size	global hist. bits	lines
ppskew1	4096	4096	32	93
ppskew2	8192	4096	32	77
ppskew3	8192	8192	32	60
ppskew4	8192	4096	20	124
ppskew5	8192	8192	20	100

(a) Table of configurations

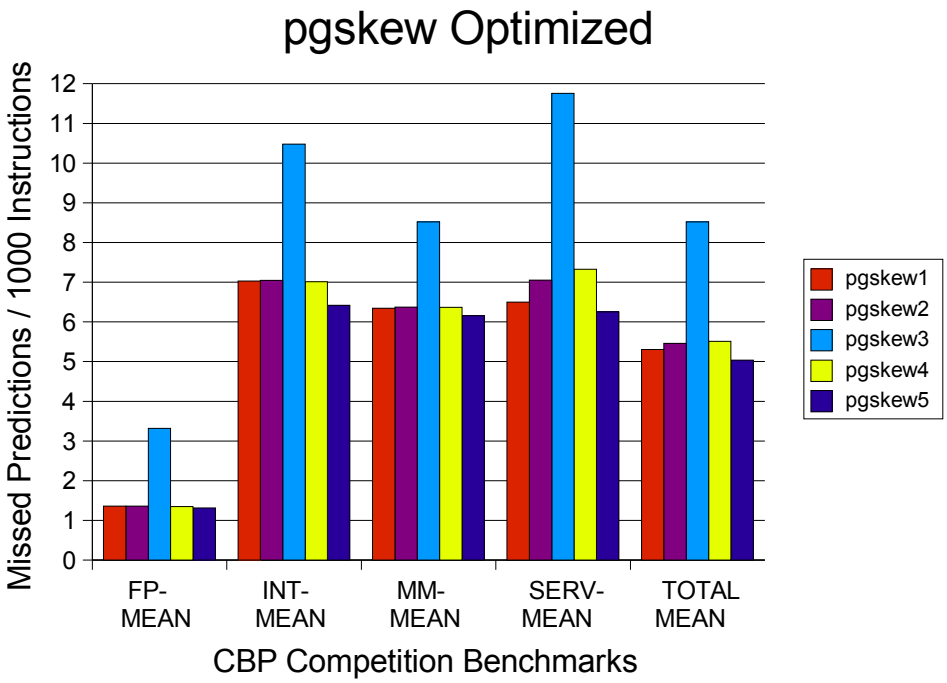


(b) Bar graph of results

Figure 19: pgskew Predictor optimization

	bi-mod size	gshare size	meta size	global hist. bits	lines
pgskew1	8192	4096	4096	27	150
pgskew2	4096	4096	8192	27	150
pgskew3	2048	16392	2048	27	70
pgskew4	4096	4096	4096	50	100
pgskew5	2048	8192	2048	24	300

(a) Table of configurations

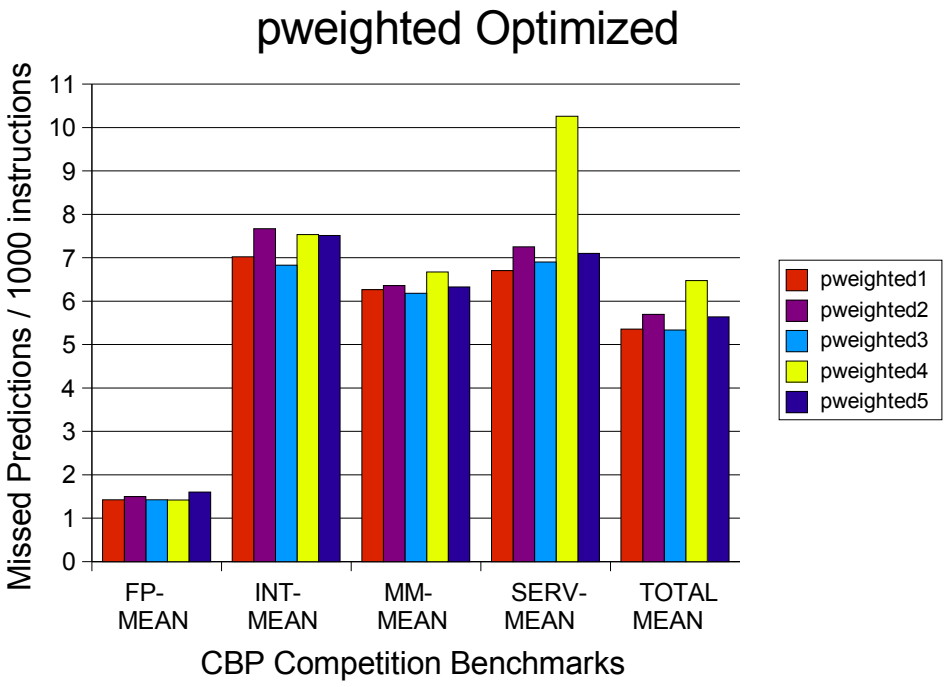


(b) Bar graph of results

Figure 20: pweighted Predictor optimization

	bi-mod size	gshare1 size	gshare2 size	percep. lines
pweighted1	8192	8192	8192	620
pweighted2	8192	2×4096	2×4096	620
pweighted3	4096	8192	2×4096	1000
pweighted4	16392	4096	4096	620
pweighted5	4096	4096	4096	1642

(a) Table of configurations

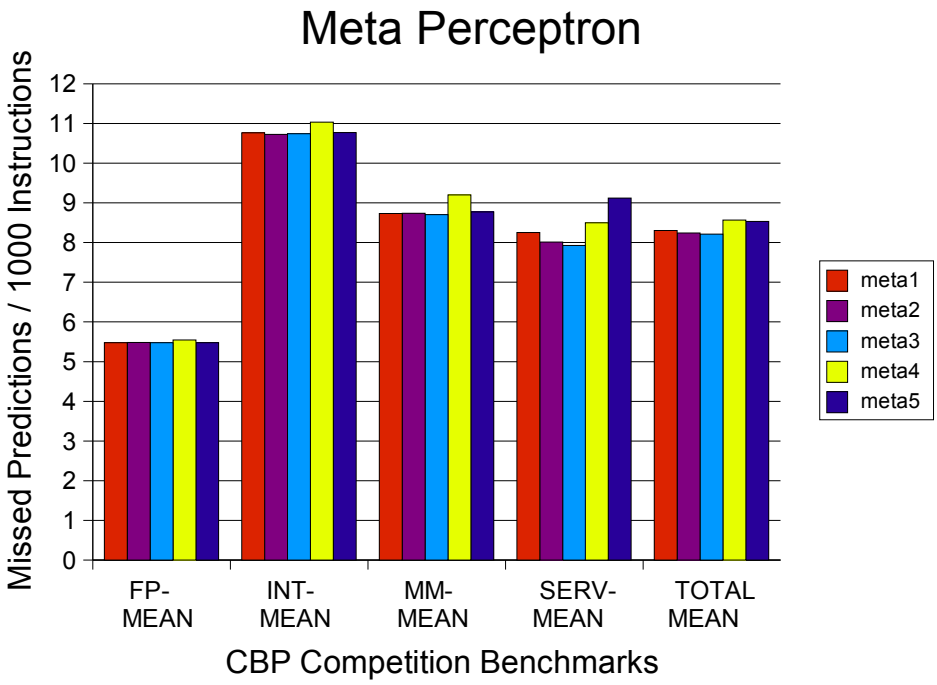


(b) Bar graph of results

Figure 21: meta Predictor optimization

	bi-mod size	gshare1 size	gshare2 size	percep. lines	weight bits
meta1	8192	8192	8192	930	8
meta2	8192	8192	4096	1440	8
meta3	8192	8192	2048	1695	8
meta4	8192	8192	8192	1860	4
meta5	4096	16384	4096	930	8

(a) Table of configurations

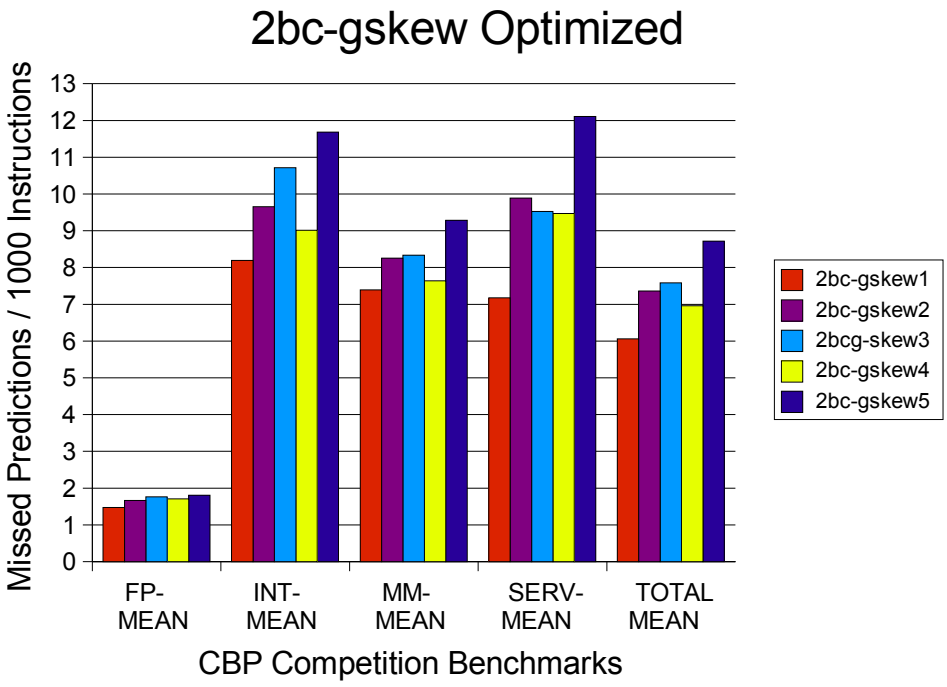


(b) Bar graph of results

Figure 22: 2bcgskew Predictor optimization

	bi-mod size	gshare1 size	gshare2 size	meta size
2bcgskew1	8192	8192	8192	8192
2bcgskew2	4096	16384	8192	4096
2bcgskew3	16384	4096	4096	8192
2bcgskew4	2048	16384	8192	2048
2bcgskew5	4096	4096	8192	16384

(a) Table of configurations



(b) Bar graph of results

B.2 Comparing different sizes of predictors

For each predictor it could certainly be the case that for some sizes the predictor does really well but at other sizes a less complicated predictor will do just as well. In this section, we show some tests of our predictors at different sizes. For each configuration, we took a configuration that had similar ratios to the best optimized 64Kb predictor shown in the previous section. Our graphs show the following sizes: 2, 4, 8, 16, 32, 64, 128, and 256 Kb.

Our results show that the gap between gshare and the perceptron widens as the hardware budget increases. Our pgskew and ppskew predictors fall somewhere inbetween the gshare and perceptron for large budgets but usually does worse for the smaller budgets. The meta and weighted predictor seem to do poorly on any hardware budget. One interesting trend is the misses for the floating point applications are generally the same on any hardware budget.

Figure 23: 2Kb Predictors

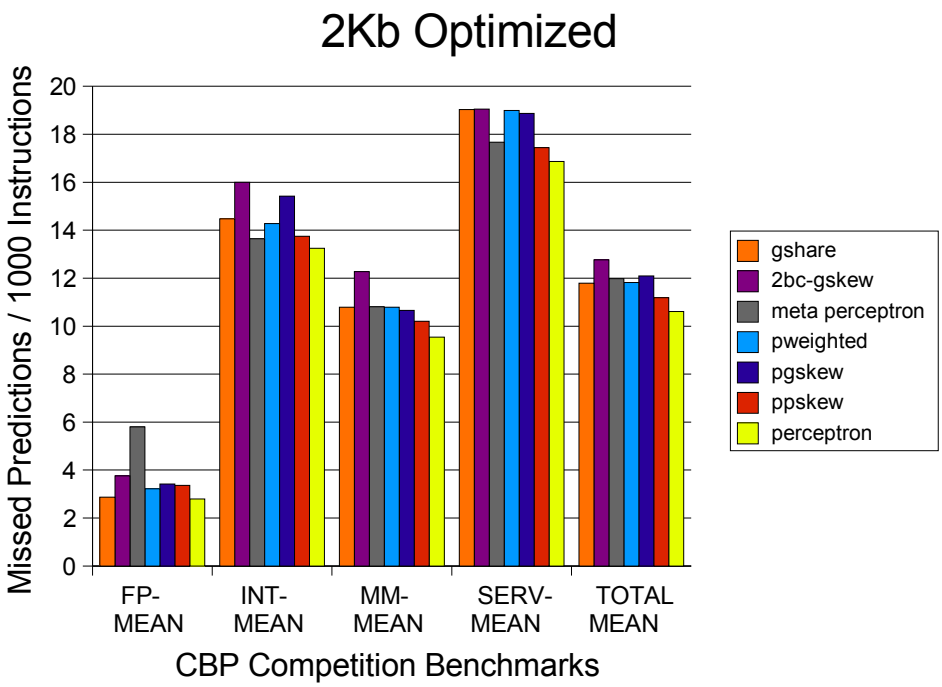


Figure 24: 4Kb Predictors

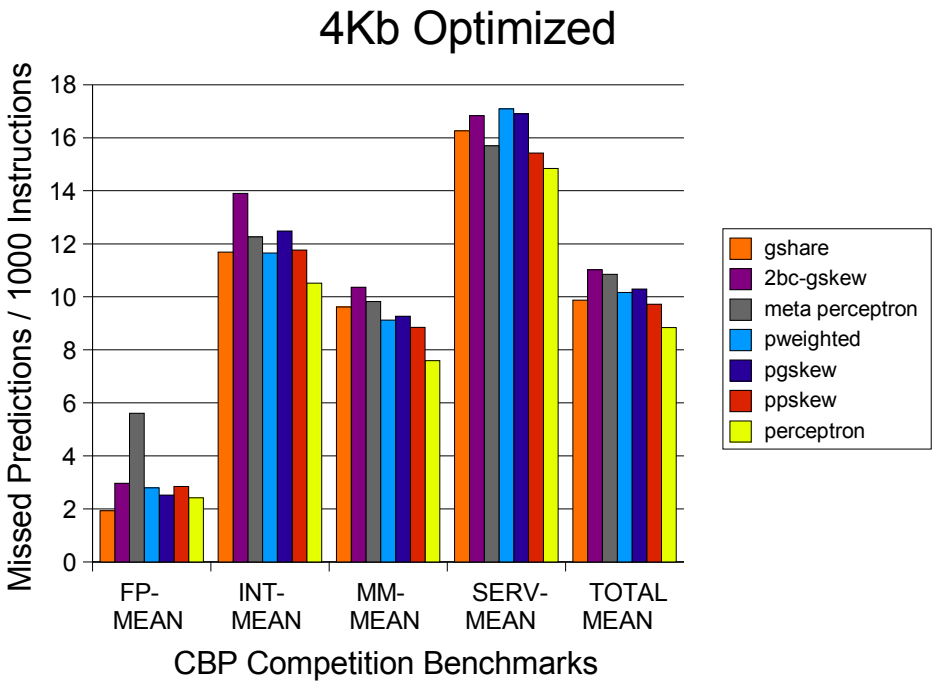


Figure 25: 8Kb Predictors

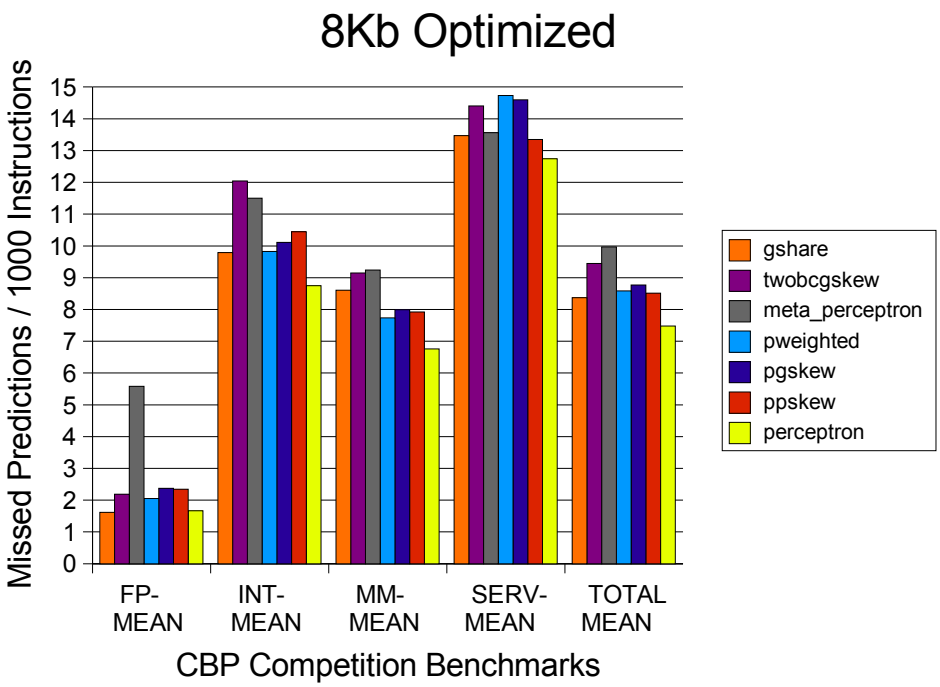


Figure 26: 16Kb Predictors

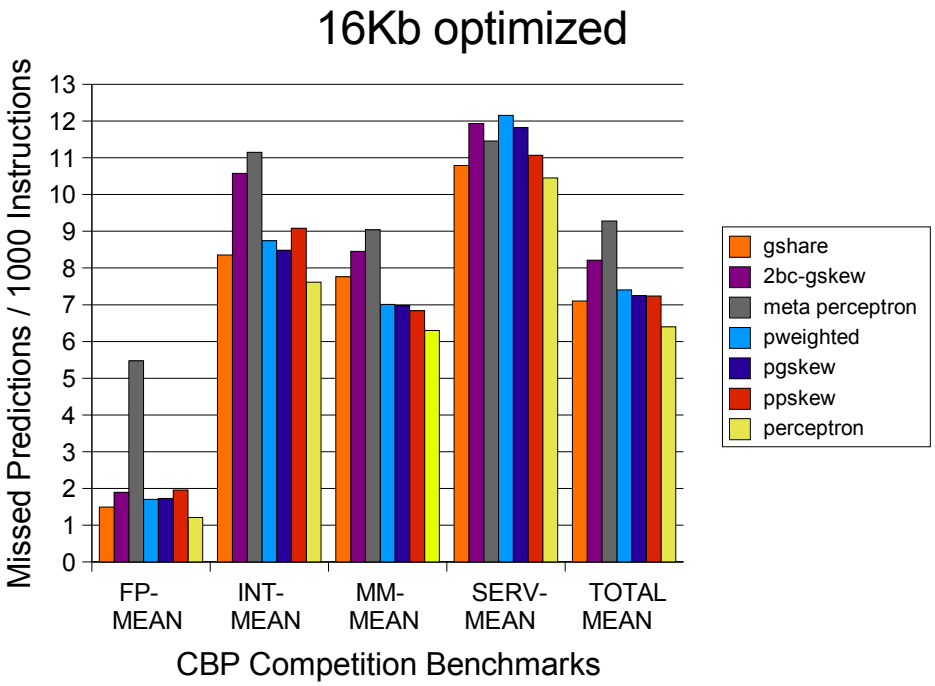


Figure 27: 32Kb Predictors

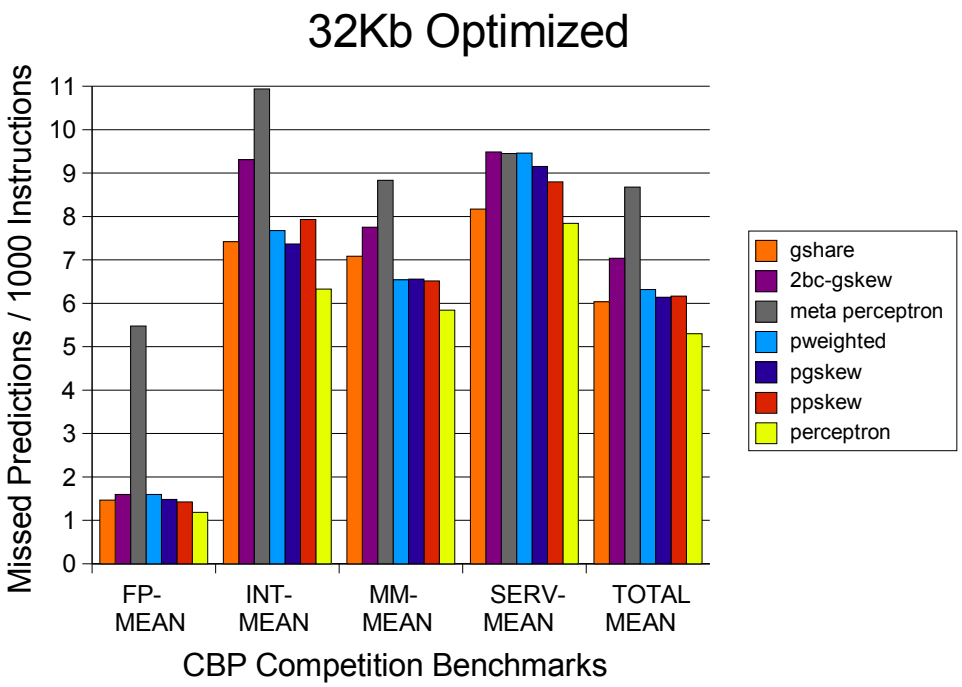


Figure 28: 64Kb Predictors

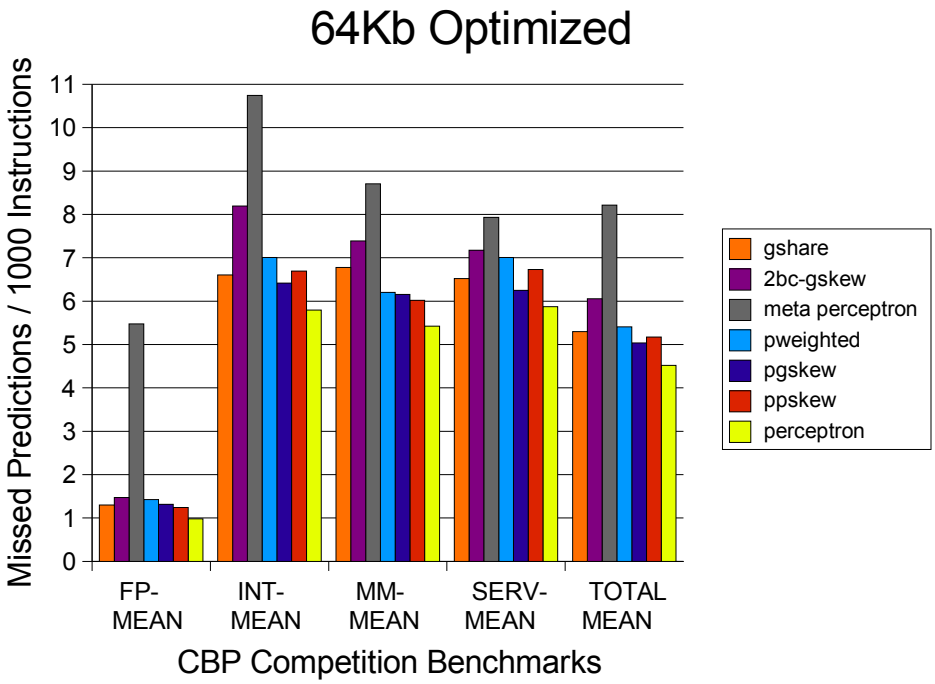


Figure 29: 128Kb Predictors

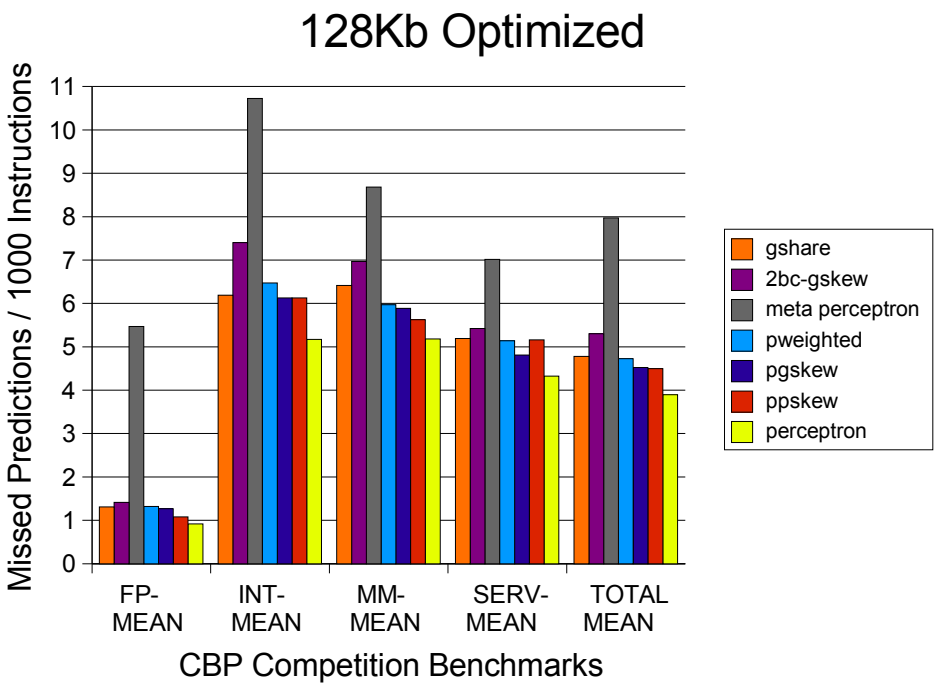


Figure 30: 256Kb Predictors