CSE560 Final Project

Reasoning on the Fast-Path CPU Branch Predictor

Rahav Dor, Chong Li, Dolvara Gunatilaka

Abstract

We investigate Fast-Path Based Neural (FPBN) branch predictor and compare its performance with GShare, a popular predictor with good characteristics. We assert that GShare has good characteristics because its data structures are compact, it is possible to implement it in hardware, and its mis-prediction rates are low. GShare is used by AMD and on the Intel Pentium M (2003 - 2008), Core and Core Due (2006 - today) processors [01].

Our investigation offers a way of finding the best predictor configuration for a given hardware budget for both GShare and FPBN. We show that in most cases, given equivalent hardware budget, FPBN is a better predictor in terms of yielding lower mis-prediction rates.

We studied the algorithmic properties of those two predictors and offer a way to improve FPBN beyond its original dominance. We then reason on FPBN learning rate, which may be concern for machine learning algorithms; our analysis addresses its initial learning rate and subsequent stability as the application logic changes over time. We end with a discussion on the feasibility of implementing FPBN in hardware and whether it is applicable for all hardware budgets and contexts.

Motivation

Predicting the direction (is it 'taken' or 'not-taken') of a branch is critical in modern processors. Correct predictions allow processors to avoid stalls, fetch and speculatively start to execute instructions at the branched address – yielding faster performance. HW3 three taught us that innovative data structures and simple (after someone else thinks about them) algorithms could achieve surprisingly low mis-prediction rates (e.g. 12.7% for a small 1 KB GShare predictor).

Already during HW3 we were bothered with two limitations in GShare. First there is the use of the saturating counter. The logic inherit in them does not permit for immediate change in decision making. In fact saturating counters can be thought of as smoothing the decisions. More often than not this is exactly the kind of smoothing we want; during lecture we saw examples of predictors that do not use saturating counters rendering undesired, abrupt change in decisions, leading to poor performance. Despite this observation sometimes a quick response is needed. We felt that there is a class of branches which GShare is not able to predict because of this smoothing. In GShare there is no way for the most current state to overwhelm

the information in its saturating counter. Doing away without addressing this class of branches rubs the CPU from some percentage of correct predictions.

The second GShare limitation is the structure of the history register. With only a bit per past branch there is no way to take into consideration how important is each past branch to the decision the predictor needs to render. The third limitation stems from the fact that naturally, in interesting programs, there is more than one way to get to the current branch. We thought that the relative importance of "each prior branch in the path that took us here" could be used to fine-tune predictions.

When we did a prior art search we found that a machine learning algorithm called FPBN might address our concerns. Instead of GShare's single column predictor-table (each entry is a saturating counter) FPBN uses the path information to make decisions. For us Jimenez's insight [02] and what they meant, relative to our HW3 concerns, were sufficient to motivate this study as our final work in Computer Systems Architecture class.

Simulation

Implementation

Our implementation simulates the behavior of the FPBN predictor proposed in [02]. We based our branch predictor simulation on HW3 with additional *FpbnPredictor* class implementing the FPBN algorithm. We modified other helper classes such as NBitSatCtr and NBitHistoryRegister to accommodate FPBN predictor.

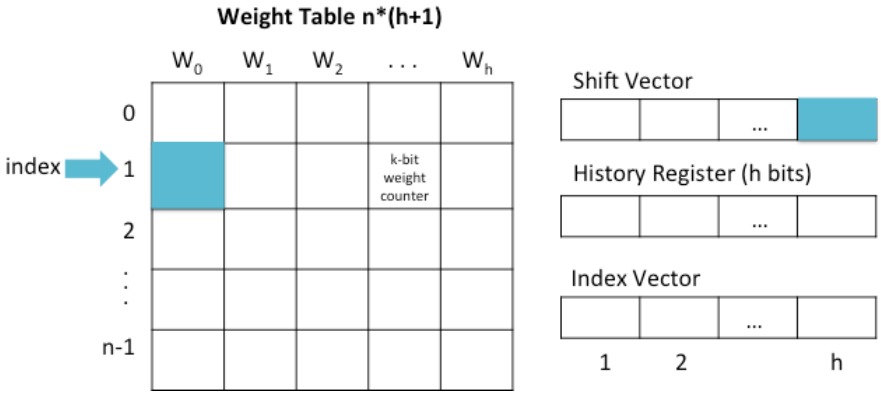

Figure 1: Core structures of FPBN.

Figure 1 shows the data structures we used to implement FPBN predictor. We created a weight table of size $n \times (h+1)$ where n is a number of entries (each entry represents a branch) and h is the history length. Each entry holds a bias-weight W_0 and h path-weights $W_1, W_2, \ldots, W_h$ where each of the h path-weights hold the relative importance of the past h branches in determining the prediction. Each weight W_i is k-bit, signed, saturating counter bounding the maximum and minimum value of the perceptron. A shift vector (SV) holds partial accumulated weights of the past

branches leading up to the current branch. An h-bit global history register and index vector are used to keep track of the outcomes and indexes of the past h branches.

When the simulator encounters a conditional branch, the instruction's PC is hashed into entry i of the table. The hash function is $PC\ mod\ n$. To make a prediction the algorithm calculates a prediction-value from the sum of the bias weight W_0 at entry i and the running total in the shift vector (blue elements in figure 1). If the prediction-value is larger or equal to 0 the prediction is 'taken', otherwise 'not taken'.

We then calculate path information. Each element $SV[j]$ in shift vector is added or subtracted by weight W_j of the current entry if the outcome is 'taken' or 'not-taken' respectively. Therefore, each element in shift vector holds the partial accumulated weight that will be used for future (and fast, within two cycles) branch prediction.

The weight table is updated only when the prediction and the actual outcome do not agree, or when the prediction value is less than or equal to the learning threshold defined in [02]. The learning threshold allows the weight table to be updated and trained more often at the beginning of the learning; otherwise the weight table will be trained only when there is a mis-prediction.

To **learn**, we first update the bias weight W_0 of the current entry i and then adjust weights corresponding to the past h branches based on their correlation to the current outcome. If the correlation is positive, the weights are incremented, otherwise decremented. We see now how this leads to learning, positive correlation reinforce the importance and negative deemphasize it. To do the updates the entry indices and the outcomes of the past h branches are retrieved from index vector and global history register, respectively.

Our branch predictor simulator processes only conditional branches. It takes three command line parameters: 1) Log of weight table entries, 2) The number of weight bits, and 3) History register length in bits. The output is the prediction accuracy.

Sanity Check for Correctness

We ran GShare, our baseline predictor, across all traces to determine the characteristics of each trace. When GShare achieves low mis-prediction rate, then the trace is easy to predict, higher mis-prediction rates suggest the trace is harder to predict. Our sanity check tests FPBN on the same traces and we expect that it would follow the same trend as GShare, getting relatively lower mis-predictions on the easier to predict traces. We observed that FPBN did follow this trend across all traces. For an easy to predict traces such as libquantum-100M and art-100M, FPBN similar to GShare, achieved a very good accuracy, trending relative to GShare. For a more difficult to predict trace i.e. go-100M and mcf-100M, FPBN exhibits the same trend as GShare, where both predictors yielded higher mis-prediction rates.

Study plan

A key goal of our study was to be fair to GShare and ground our result around some common framework for comparison. This guides our experimental plan and thus we prescribe the following steps

- Allocate a set of reasonable hardware budgets (1 to 64) KB for the predictors
- Limit the large design space to areas in which GShare present its best performance
- Limit the large design space to areas in which FPBN present its best performance
- For each hardware budget, first search in GShare's design space for the best configurations
- For the same hardware budget in which the best GShare was found, search in FPBN design space for the best configurations

Summary of goals

The following are one sentence summary of each of our goals.

Goal 1: Be able to identify GShare best configuration for each hardware budget.
Goal 2: For hardware budget no larger than for GShare, determine if there is FPBN configuration that is better.
Goal 3: Articulate a process for hardware designers to configure the best FPBN.
Goal 4: If possible, improve FPBN original algorithm.
Goal 5: Understand FPBN learning rate.

Experiments and Analysis

In this section we describe our experiments and explain the results. Our experiments start with finding the best counter widths for GShare and FPBN. We then find the best GShare for all given hardware budget. For each hardware budget, we conduct experiments to check if FPBN is better than GShare or not. We conclude by evaluating the effect of hash function and learning rate.

Our design spaces consist of 3 parameters: table size (number of entries), history length, and counter width. We use GShare predictor, implemented in HW3 as a baseline for performance comparison since GShare is a well-known two-level predictor that has relatively good prediction accuracy.

Traces

We will use the following five randomly chosen traces: go-100M, sjeng-100M, ART-100M, and sphinx3-100M, and libquantum-100M.

Experiment#1: Exploring Counter Widths

We performed a detailed study to find an appropriate counter width for GShare and FPBN. In each setting (various hardware budgets) we fixed the size of table entry and history length, and varied counter width. We measured the mis-prediction rate

for each width from 2 to 12 bits with a step size of 2. Our setting includes table size of 2^8, 2^{12}, 2^{16}, and 2^{20} and history length of 8 and 16. We observed that both GShare and FPBN require a certain number of counter bits to achieve accurate prediction regardless of table size and history length.

Figure 2 shows sample result from GShare predictor with 2^{12} entries and 16-bit history length. We see that GShare achieves higher accuracy when its number of counter bits is small, 2-4 bits.

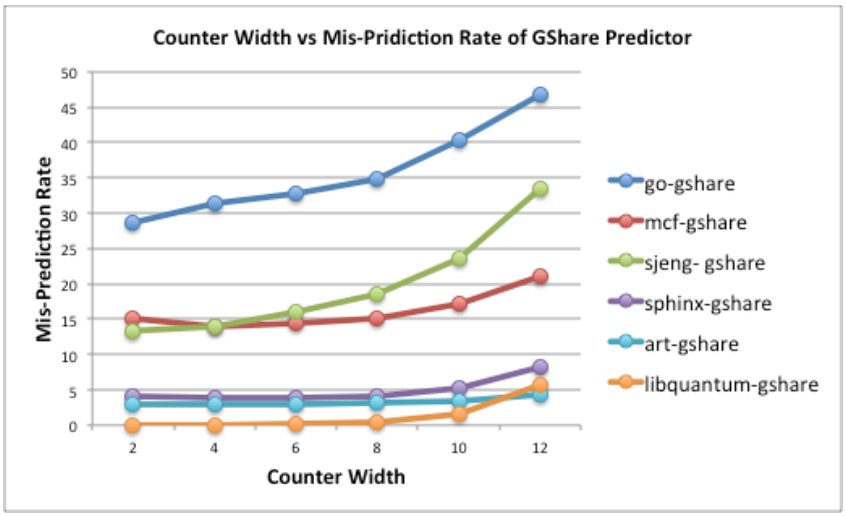

Figure 2: Mis-prediction rates of different counter widths of GShare predictor. The best performance is achieved with width of 2 to 4bits.

Figure 3 presents sample result of FPBN for the same setting. We see that FPBN needs wider weight counter to render good predictions. The optimal value across all settings and traces is 8, which is where the mis-prediction rate becomes stable and is at its lowest (best) point.

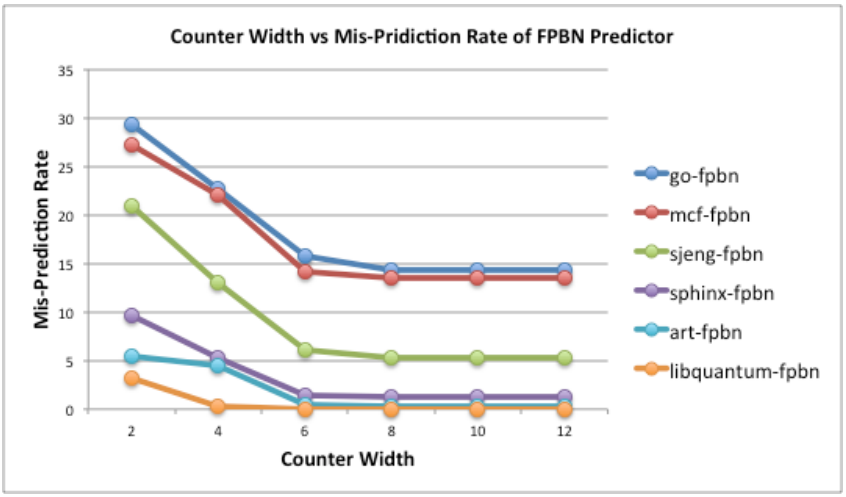

Figure 3: Mis-prediction rates of different counter widths of FPBN predictor. There are no performance improvements beyond a width of 8 bits.

Analysis: Why FPBN requires larger counter width than GShare?

Each entry of FPBN consists of the branch bias and multiple weights associated with its h past branches. It uses the sum of bias and weights of the past h branches to calculate the prediction value. To correctly predict "frequent branches" the value of the bias must be large enough to override sum of the h weights. For example, figure 4 shows a scenario where too few counter bits causes mis-prediction. To play out the example assume that the width of weights is 2 bits. Hence the maximum value of W_i is 1 and the minimum is -2. After executing B1, B2, B3, we want to predict B4, which we name as the "frequent branch". However, since the bias W_{B4} could not be incremented further than 1 its value cannot overwhelm the accumulated value of the past 3 branches (the sum of history counters), and B4 is mis-predicted.

```
B1: for(i=0;i<3;i++) W=-1
B2:     for(j=0;j<3;j++) W=-1
B3:             if(x>0) W=-1
B4:             for(k=0;k<100;k++) W=1
```

Figure 4: Control structure in which too narrow counters causes FPBN to mis-predict.

However, for GShare, each table entry contains one saturating counter shared by all past branches. GShare relies on this single value of the saturating counter for making prediction. As a result, larger counter can cause mis-prediction rate to increase since it would take longer time to change the branch direction.

Experiment#2: FPBN compared to GShare Predictor

The design space for the two predictors GShare and FPBN is too large to explore empirically, it is intractable to simply enumerate and try all possible design combinations. GShare and FPBN designs can employ different number of entries in the predictor table, different widths of the saturating counters, and different history lengths. Both predictors design also need to take into account the hardware budget. With so many options to vary in the design space we had to ground a few design decisions to make this study tractable.

We set hardware budgets (1, 2, 4, 8, 16, 32, and 64) KB and for each fixed budget we will search for the best GShare and FPBN design. Equation (1) and (2) show how we estimate a hardware budget for GShare and FPBN.

$$GShare\ HW\ Budget = nEntries * satCtrBits \quad (1)$$

$$FPBN\ HW\ Budget = nEntries * weightCits * (historyLength + 1) \quad (2)$$

We know from HW3 and our preliminary results that GShare has lower mis-prediction rate when larger prediction tables (more entries) are available. Therefore to find the best GShare design for a fixed hardware budget as much of the hardware should be used to maximize the number of entries. To do so we fix the size of saturating counter to 2 bits since our empirical study shows that GShare works

best with this width. With the maximum entries in hand the search for the best GShare design is done by varying the history length. We have the history length be varied from 0 to 32 in steps of 4. Since history length has negligible effect on GShare's hardware budget we do no need to precisely choose this register size; it is sufficient to choose a length that is beyond the best point. We also know from our preliminary study that loner history does not adversely affect performance for sufficient number of entries (which we maximize).

For FPBN, we use the optimal 8-bit weight counter. Since FPBN hardware budget depends on both table size and history length there is a tradeoff between the two. If we choose to enlarge one, another must be kept low, therefor we considered number of entries to range from 2^4 to 2^{13}, and for each table size the history length is varied from 4 to 32 with the step size of 4.

For each hardware budget and for each of the five traces, once the best GShare design was identified, we contrast it with FPBN that uses the same budget. We then provide analysis of GShare vs. FPBN for the given hardware budget. To the extent that the randomly chosen traces are representative of diverse applications then this analysis should provide general insights into the two predictors behavior.

Analysis: Identify best GShare for each hardware budget.

Hypothesis 1: To lower GShare mis-pediction rate it is sufficient to fix the saturating counter width at 2 to 4 bits, then use all of the rest of the hardware budget to increase the number of entries in the predictor table.

Observation 1.1: *GShare mis-prediction rate gets to its lower point and flattens when its saturating counters width is between 2 to 4 bits (inclusively)*.

This is verified from experiment#1.

Observation 1.2: *Over most of its domain, GShare mis-prediction rate as a function of history length is monotonically none-increasing*.

Programs have some control structure and their level of control nesting determines how long of history would benefit the prediction of a branch. It is reasonable that history shorter than the level of nesting would not be able to capture the program's complete nesting and thus would not achieve the lowest mis-prediction. What was not clear to us by reasoning alone was what occurs when the history is longer than the nesting level. Will incorporating too much history worsen the mis-prediction rate? In our experiments (given only the traces) we actually do not know what is the program's level of nesting, but we believe that we allowed for sufficient history length range to capture this point and beyond.

To study this question we run experiments varying the history length for all budget sizes and traces in our exploration (detailed results are available in SVN and the mean behavior is in figure 5). We see that for all traces excluding *go* the behavior is

monotonic until a certain history length, in which there is an abrupt change in mis-prediction rate. Beyond that point the function is monotonic again, in the same direction as it was before. The only trace that does not strictly follow this observation is *go* which is a trace of a program playing the go game. We do not know if the game is played against a human being, if the game was won or lost, or if all behaviors were rational; furthermore, this is a game that is notoriously hard for computers to play. So some erratic behavior can be expected.

	History length								
Trace	**0**	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**
art	5.84	2.96	2.96	2.97	1.81	1.36	1.36	1.36	1.36
sjeng	15.01	13.77	10.73	8.52	7.69	7.35	7.35	7.35	7.35
go	25.41	21.04	19.92	21.08	21.20	19.62	20.87	20.74	20.61
sphinx	5.20	4.49	4.95	3.13	2.35	2.29	2.29	2.29	2.29
libquantum	10.51	6.17	0.36	0.02	0.02	0.02	0.02	0.02	0.02
Geo.mean	10.40	7.50	4.08	2.02	1.69	1.55	1.59	1.59	1.59

Figure 5: Gshare mis-prediction rate. The rate is generally a none-increasing function of history length.

We also offer figure 6 that plots the geometric mean of mis-prediction for all hardware budgets for each trace. The geometric mean of all hardware budgets and all traces is also offered (orange curve).

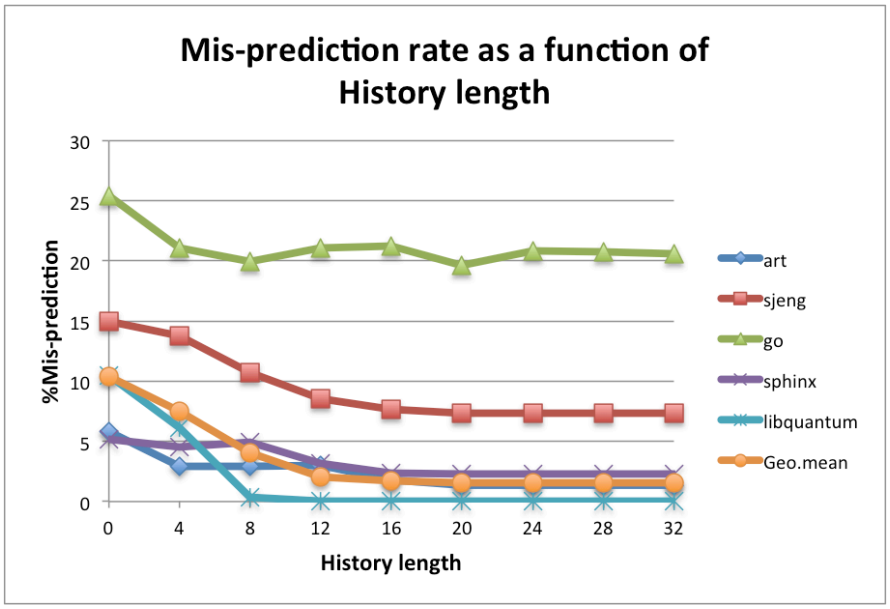

Figure 6: GShare percent mis-prediction as a function of history length. All traces are shown and the geometric mean is plotted in orange. In general the plots are none-increasing function of history length.

We always see that as the number of entries increases GShare's mis-prediction gets better. This should come to light by reasoning on the data structure and algorithm.

With the saturating counter at its best point, what is left to optimize is the number of entries. In GShare, branches map to some table entry for a given PC and history register. Each branch (a certain PC) maps to one or more entries. We see then, that the only implication of having more entries than needed is a resource utilization question and never a mis-prediction question. In fact the more entries we have the less collisions will occur and the mis-prediction rate can only be lower.

Analysis: For given budget no larger than for GShare, find best FPBN.

Hypothesis 2. *For a given hardware budget FPBN will yield better than GShare branch direction prediction (measured by FPBN lower mis-prediction rate).*

Figure 7 shows FPBN mis-prediction rate, summarizing all of our experiments. Each cell in the table shows the **lowest** mis-prediction of GShare and FPBN, selected from all possible designs for that predictor within the given budget (detailed data is available in SVN). In the table each trace has two columns, one for GShare and the other for FPBN with each such pair highlighted green if it has the better mis-prediction of the two, and red otherwise.

	go		sjeng		art		sphinx		libquantum		Arithmetic mean	
HW Size	gshare	fpbn	gshare	fpbn	gshare	fpbn	gshare	fpbn	gshare	fpbn	gshare	fpbn
1KB	18.55	21.84	13.34	10.56	2.96	0.28	4.04	0.95	0.02	0.01	7.78	6.73
2KB	21.67	19.9	10.51	7.62	2.96	0.29	3.17	0.9	0.02	0.01	7.67	5.74
4KB	19.81	18.01	8.45	6.02	2.96	0.29	2.9	0.93	0.02	0.01	6.83	5.05
8KB	18.93	16.54	6.92	5.32	0.86	0.28	2.3	0.91	0.02	0.01	5.81	4.61
16KB	18.67	15.2	5.76	4.81	1.42	0.28	1.66	0.9	0.02	0.01	5.51	4.24
32KB	15.15	14.69	5.11	4.65	0.87	0.28	1.57	0.9	0.02	0.01	4.54	4.11
64KB	14.16	14.16	4.72	4.59	0.31	0.28	1.5	0.89	0.02	0.01	4.14	3.99
mean	18.13	17.19	7.83	6.22	1.76	0.28	2.45	0.91	0.02	0.01	6.04	4.92

Figure 7: FPBN mis-prediction rates. In all cases but 1 and 64 KB for the *go* trace FPBN yields better performance.

We see in the figure 7 that in all cases but two, FPBN yields lower mis-prediction. In one of those cases (64 KB and *go* trace) FPBN and GShare have the same mis-prediction rate. The only case in which FPBN yields a worse rate than GShare in for hardware budget of 1 KB and the *go* trace. GShare mis-prediction is 18.55% while FPBN yields 21.84. Among all of our traces the branches in go were the hardest to predict for both GShare and FP, the highest mis-prediction for all budgets occurred in that trace. By observing the detailed experimental results we see that more entries and longer history were needed to somewhat offset the difficulty of the branches. With limited 1 KB budget however, the maximum number of entries GShare can afford is 2^{12} and FPBN only 2^8, putting FPBN already in a disadvantage. The history length is the second disadvantage for FPBN. The detailed data shows that within the confinement of the 1 KB budget GShare performed its best (18.55%) with a history length of 20. On this trace FPBN also needs longer history but since it uses multiple history registers ($h + 1$ of them on each entry) it could not afford it within the limited budget; the largest history it could afford is 4, which adds to FPBN inferior performance. This was not an issue on easier to predict traces due to FPBN

general superiority, requiring generally much less entries due to the use of the *mod* function instead of the *xor* function to hash into an entry. The *xor* function simply generates many entries for the same branch (PC) address, which makes GShare inherently require higher number of entries.

FPBN is better than GShare because it uses the path leading to the current branch to render decision. Each branch on this path has an importance weight that is taken into consideration. Contrary to that GShare uses only a global view of the program state and each past branch is represent by a single bit. With a single bit holding much less information GShare performance is inferior.

The reason that *go* is so hard to predict, for both GShare and FPBN is that this is a trace of a computer program playing the game *go*. A game that is known to be hard for computers to play, there are just too many options to choose from at each turn. In an interesting twist we may be able to draw some conclusions about the game from the results of this branch prediction study. The detailed results show that at most 16 past branches was needed to achieve the best prediction. In determining the direction of the branch FPBN takes into account the path (past branches) that led to the branch that is about to be taken or not. We suggest then, that for a computer to play *go* well at most 16 past turns need to be taken into account. It would be worth to explore in another study if this observation can lead to a computer algorithm that can run in reasonable time and play *go* well, not being plagued by longer histories. The most recent result published for a computer program playing go is for Zen19D. In March 2012 the program reached the rank of dan 6 (the highest is dan 9) but to do that it was running on 28 cores [04].

After demonstrating that FPBN is almost always better than GShare, we ask how much better? Figure 8 shows a graphical answer to this question.

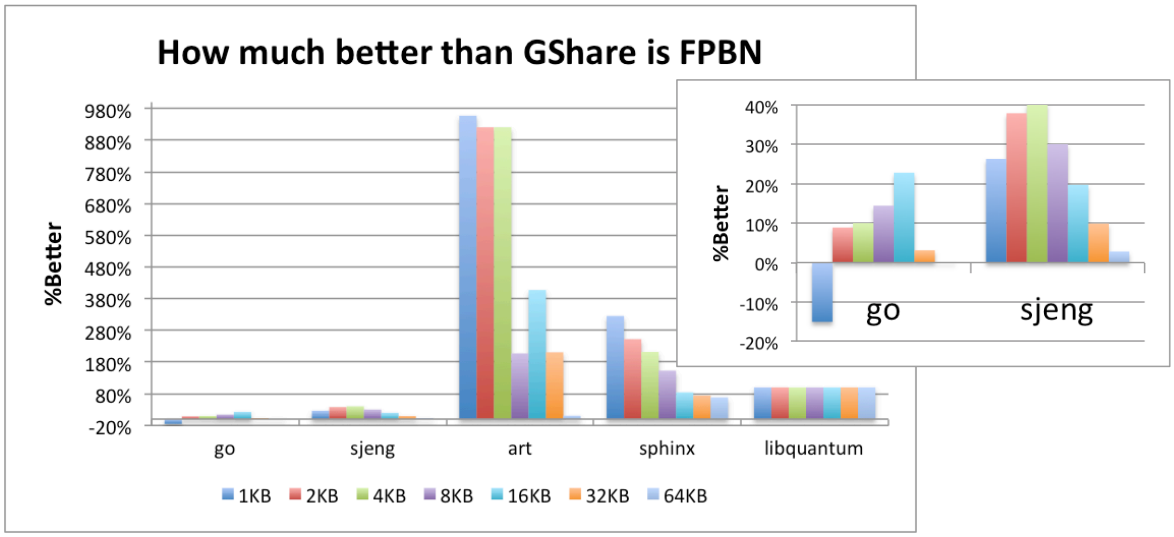

Figure 8: Percent that FPBN is better than GShare. The go and sjeng traces are zoomed on in the top right corner.

Analysis: A process for hardware designers to configure the best FPBN.

We do not presume that the five traces we used represent all typical today's programs, but assuming that this is the universe we know of, we proceed as follows. To find the best FPBN configuration we considered all possible combinations of number of entries and history lengths. We can see in the detail experiments (available in SVN) and in figure 9 that shows the geometric mean across all traces that FPBN lowest mis-prediction is truly a function of both the number of entries and history length. Unlike GShare, for FPBN to use all available hardware is not only a question of utilization, but also a question of worse performance in terms of higher mis-prediction. For example, in figure 9 we see that for hardware budget 4 KB (2^{10} entries) the best configuration is history length of 24, achieving mis-prediction of 0.47%. But if we use longer history, say 28, the mis-prediction is 0.72% so the configuration of 24 is $\frac{0.72}{0.47} - 1 = 53.2\%$ better.

log(entries)	his4	his8	his12	his16	his20	his24	his28	his32
4	11.21	3.03	2.72	1.60	1.52	1.45	1.41	1.47
5	9.21	2.30	2.05	1.13	1.10	1.09	1.05	1.08
6	7.52	2.00	1.85	1.00	0.98	0.98	0.94	0.97
7	6.94	1.83	1.68	0.91	0.88	0.88	0.85	0.88
8	6.63	1.72	1.54	0.84	0.82	0.81	0.78	0.81
9	6.44	1.66	1.48	0.80	0.78	0.49	0.74	0.76
10	6.35	1.63	1.45	0.78	0.76	0.47	0.72	0.74
11	6.30	1.62	1.44	0.77	0.76	0.47	0.72	0.73
12	6.31	1.61	1.43	0.77				
13	6.26	1.61						

HW budget
1KB
2KB
4KB
8KB
16KB
32KB
64KB

Figure 9: FPBN mean mis-prediction rate across all traces. Each cell is the percent mis-prediction. The best configurations are highlighted in red font. Hardware budget legend is displayed on the right.

Lastly, we note that we believe that the five traces do not necessarily represent the universe of programs. Therefore, a more general and less sensitive statement about FPBN is in order. In Figure 9 we count which history lengths participated in the highest number of best configuration and we see that both history length of 16 and 24 stand out, each with three best configurations. Therefore, if hardware budget is of no concern and a better mis-prediction are necessary, choosing history of length 24 can often achieve about 50% better performance (lower mis-predictions). If hardware budget is a concern, while delivering 50% better is important, the absolute value of FPBN is low enough that it might not matter. In that case choosing a history length of 20 is a good design tradeoff.

Experiment#3: The Effect of Hash Function

In this section we study how hash function affects the accuracy of the predictor. The original FPBN always index the same branch to the same entry by using

$PC\ mod\ nEntries$, while GShare takes into account the global history for indexing ($PC\ xor\ historyRegister$). Although FPBN learns the path history, what is learned is the local history that leads to the current branch. We thought that by adding more information to the indexing process, in terms of the global history, we can explore an improvement for FPBN. We implemented a new version of FPBN with GShare's hash function and conducted experiments on five different traces. For each experiment, the number of entries was varied from 2^6 to 2^{15} while we fixed the history length to 16 bits and counter width to 8.

Figure 11 shows the impact of different hash functions on FPBN predictor (we only present the results from 3 example traces for the ease of visualizing, the other results do not contradict our conclusions). We observe that FPBN with xor hash suffers from high mis-prediction when table size is small, but outperforms the original FPBN when the table size gets larger.

Analysis: Why FPBN with GShare hash is better than the original FPBN?

The advantage of FPBN with xor hash is that it takes advantage of the path information while at the same time xoring builds more context (here context means the results of the last global h branches). There are multiple paths leading to the current branch. So, xoring the global history with PC allows each branch to have different context to its decision. While the original $PC\ mod\ nEntries$ will always map a branch to the same entry regardless of its context, the xor separates branches by their context. The major cost of FPBN with xor is that it requires more entries to achieve the best accuracy. But if the table size becomes big enough this is not a concern anymore and we earn better mis-prediction rates.

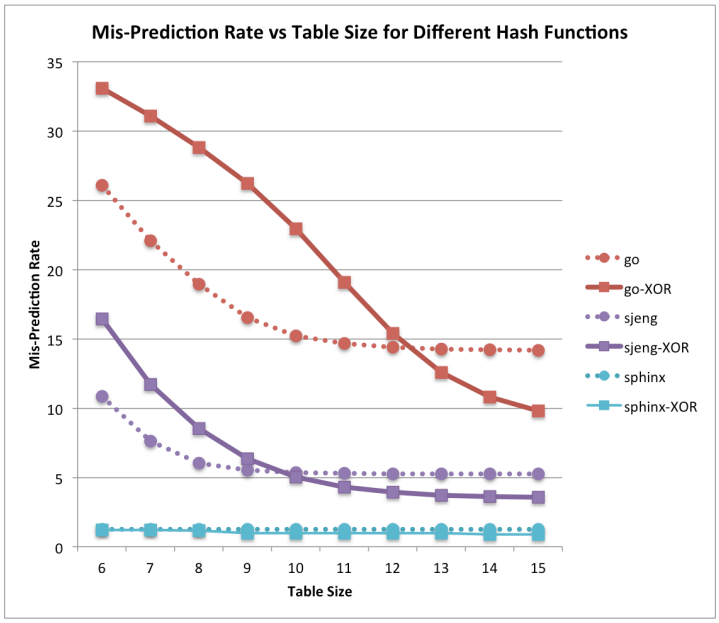

Figure 11: Mis-prediction rate as a function of log of table size of FPBN with original hash function (dot line) and the improved FPBN (solid line)

Experiment#4: Learning Rate

With FPBN being a machine learning algorithm we want to study its temporal behavior (how fast it can learn) compare to GShare. In this experiment, we count the number of mis-predictions made by the predictor (the y-axis) over the training lifecycle in the beginning of the trace. The setup for this experiment is as follows:

	Gshare		**FPBN**	
HW Budget	Table entries	History Len	Table entries	History Len
8KB	2^{15}	16	2^{9}	16

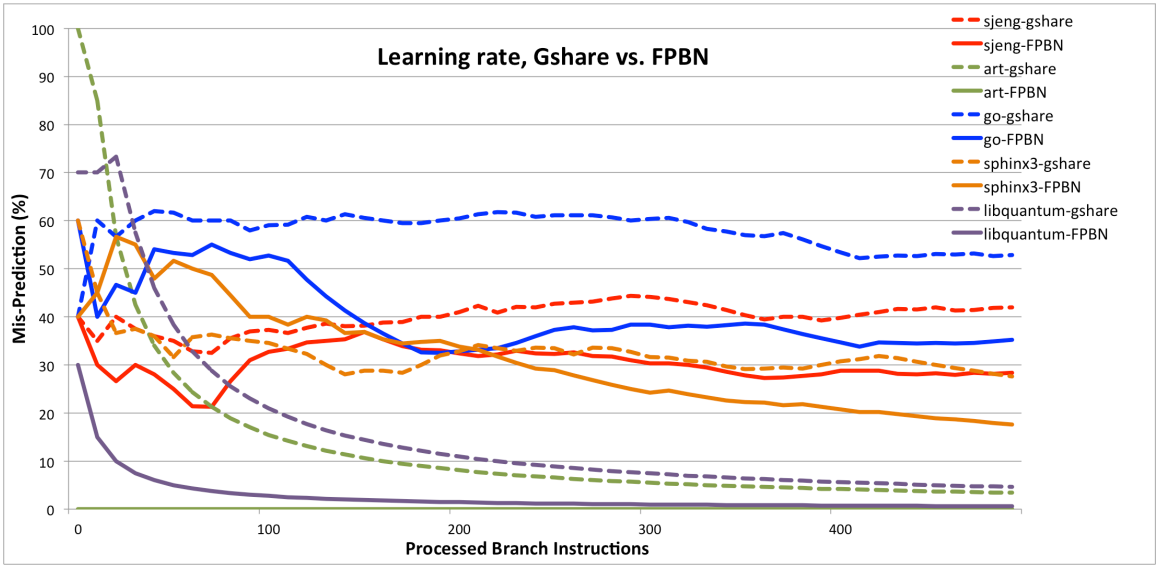

Figure 12: Learning rate of Gshare and FPBN. Each trace is depicted in its own color with GShare in dashed lines and FPBN using solid.

In general FPBN achieves lower mis-prediction rate faster than GShare. The reason is that FPBN have immediate information about each branch after it sees it for the first time. While this may be albeit not perfect information until more is learned, it is more than what GShare has because each branch is mapped to multiple entries.

Actual Implementation/Cost Discussion

From the research paper [02] we also learned that FPBN is implementable in hardware. Unlike many other machine learning algorithms that require too much resources to be feasible for a processor, or too many cycles before a decision can be made, FPBN is calculating intermediate results on each step. Assuming that we are using a simple scalar machine, FPBN is always two cycles away from rendering a prediction; it needs to calculate one prediction value and make a prediction based on that value. Prior work [03] by the same author on the Perceptron branch predictor required no less than *h* cycles since each prediction requires a dot product of h past branches and their corresponding weights. FPBN still suffers from a high computational latency when there is a mis-prediction; updating FPBN weight table

incurs at least h cycles to adjust all weights related to the past h branches. In term of hardware budget, our experimental results show that FPBN does not require more space than the traditional GShare predictor to achieve good performance. Also, it has shorter training overhead compared to GShare predictor.

Conclusion

We simulated FPBN Predictor and compared its performance with GShare predictor. We conducted empirical studies to limit our design spaces, and found that GShare works better with small counter width, while FPBN achieves high accuracy with a larger counter width. The best GShare can be achieved by allocating the maximum table size. However, for FPBN, we need to consider the combination of both table size and history length. Our results show that with a given budget, FPBN in most cases, outperforms GShare. This is because FPBN considers both path information ("how I got to this branch") and history pattern. We concluded by studying FPBN learning rate and showed that FPBN in general has better learning rate than GShare. A reasonable behavior considering that GShare needs larger table size than FPBN, hence it takes longer time to train the predictor. In addition, we also studied the effect of 2 different hash functions. We observed that FPBN with GShare hash function can help us yields a better prediction than the original FPBN when the table size becomes large enough.

Contributions

Chong Li: GShare experiments, learning rate experiment and analysis, hash function analysis.
Dolvara Gunatilaka: Simulation implementation, counter width experiments and analysis, FPBN experiments.
Rahav Dor: Research plan and motivation, GShare experiments and analysis, FPBN experiments and analysis, and Comparison insights and analysis.

The FPBN is not our algorithm. It was invented by Dr. Jimenez, the author of [02]. Our ideas were to bound the design space to be able to find the best configurations and the incorporation of the xor hash function.

References

[01] Branch predictor. (2014, April 11). In *Wikipedia, The Free Encyclopedia*. Retrieved 13:34, April 26, 2014, from
http://en.wikipedia.org/w/index.php?title=Branch_predictor&oldid=603789711

[02] Daniel A. Jimenez. 2003. Fast Path-Based Neural Branch Prediction. In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36). IEEE Computer Society, Washington, DC, USA, 243-.
http://www.microarch.org/micro36/html/pdf/jimenez-FastPath.pdf

[03] Daniel A. Jimenez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01). IEEE Computer Society, Washington, DC, USA, 197-.
https://www.cs.utexas.edu/~lin/papers/hpca01.pdf

[04] Computer Go. (2014, April 20). In *Wikipedia, The Free Encyclopedia*. Retrieved 03:07, April 26, 2014, from
http://en.wikipedia.org/w/index.php?title=Computer_Go&oldid=605060303