ECE 511 Assignment 2
Branch Prediction

Vikram Sharma Mailthody

(NetID- vsm2, UIN - 663250535)

Contents

Goals of the Assignment:

1. Understand different types of Branch Predictors – 2bit, Tournament, BiMode, Gshare, Yags and perceptron
2. Analyze various configuration of gshare, yags and perceptron to understand how various parameters effect the branch predictions and IPC.

More details of the assignment can be found here. [10]

Software used:

Gem5 and its dependencies

Tasks:

The assignment requires to submit 3 results (baseline, intermediate and final) and its comparisons. Following are the main steps involved for completing the assignment.

1. Use se.py as the default script for running the system emulation
2. Section 2 - run se.py with default L1,L2 cache sizes.
3. Run 2bit, tournament and BiMode branch predictors with 2 benchmarks – Libquantum and soplex.
4. Write code for gshare, yags and perceptron based predictors
5. Run gshare, yags and perceptron predictors with various configuration of PHT sizes, cache sizes and history lengths.
6. Analyze the data obtained in these configuration and provide comments

Introduction:

Gem5 is a computer system simulator platform initially developed at University of Wisconsin-Madison Square (there are several other collaborators like MIT, UMich, ARM, etc). It's a modular simulator allowing the user to parameterize, extend and rearrange as required. The simulator supports both x86 and ARM ISA allowing the user the flexibility of choice. The simulator is primarily developed using C++ language and scripts to run are typically written in python. There are two main modes of operation when using the simulator – system call emulation mode and full system mode (se and fs respectively). The System call emulation mode is mainly used to simulate a binary file that are linked statically or dynamically. The full system mode runs complete system with choice of operating system to boot in the simulation environment. Further gem5 supports configurable CPU models, pluggable memory systems and device models providing flexibility, availably and enhancing collaboration for the computer system researcher.[1]

For this assignment, we will be using ARM ISA and System Call emulation mode. The primarily goal of this assignment is to understand how branch prediction works, different types of predictors and understand importance of conditional branch predictor accuracy with its relationship with IPC. Two benchmarks, libquantqam and soplex are used to measure the conditional branch predictor accuracy.

Report organization:

Next section describes about existing branch predictor and its results measured. Followed by individual predictors, gshare, yags and perceptron designs are discussed with various configuration. Lastly, best configurations of each of these predictors are analyzed and discussed for possible drop in branch prediction accuracy.

Default Branch Predictors:

Baseline architecture:

In order to measure the conditional branch predictor accuracy, system design shown in figure 1 is used. It houses:

- Single core out of order ARM ISA based system
- L2 cache
- DDR3 1600 memory

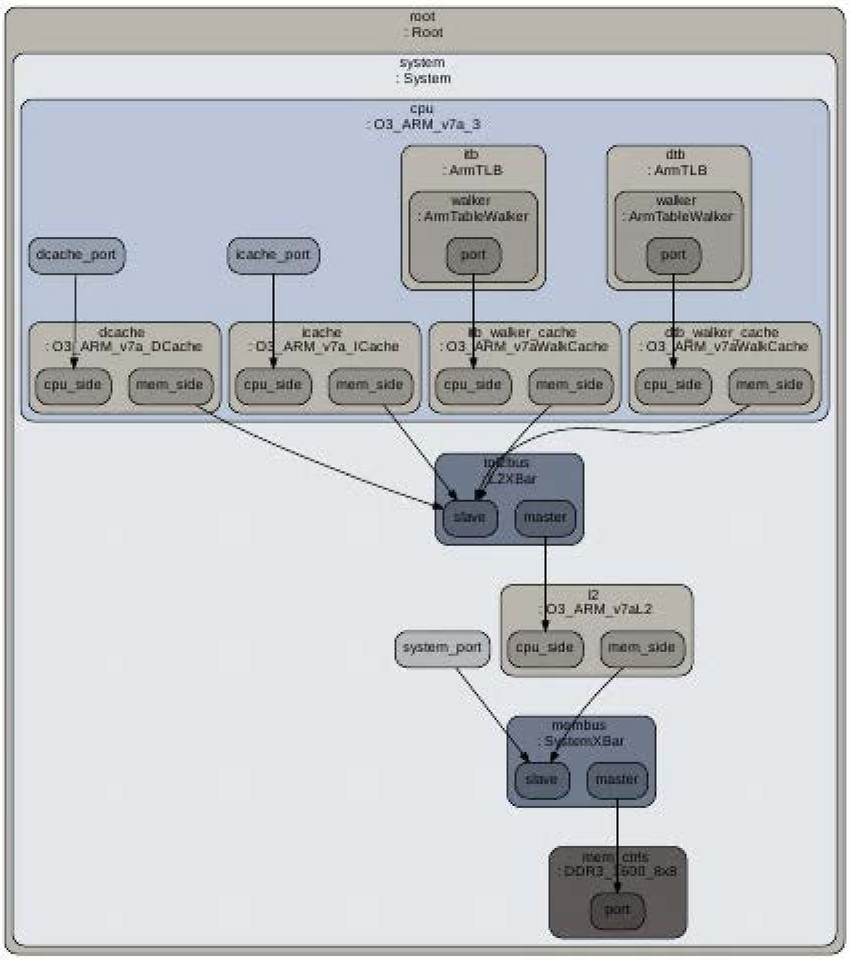

Figure 1: High level baseline architecture for branch prediction

Correlation based or 2bit counter branch predictor:

This is simplest of all the branch predictors discussed in the assignment. Overall architecture of this predictor can be mapped to single diagram as shown in figure 2. As shown in the diagram it has two bit saturating counter acting as PHT. During look up phase, the counter values are taken out from the PHT table with a hashing function, i.e branchAddr>>intshiftAmt && indexMask. [5][9]

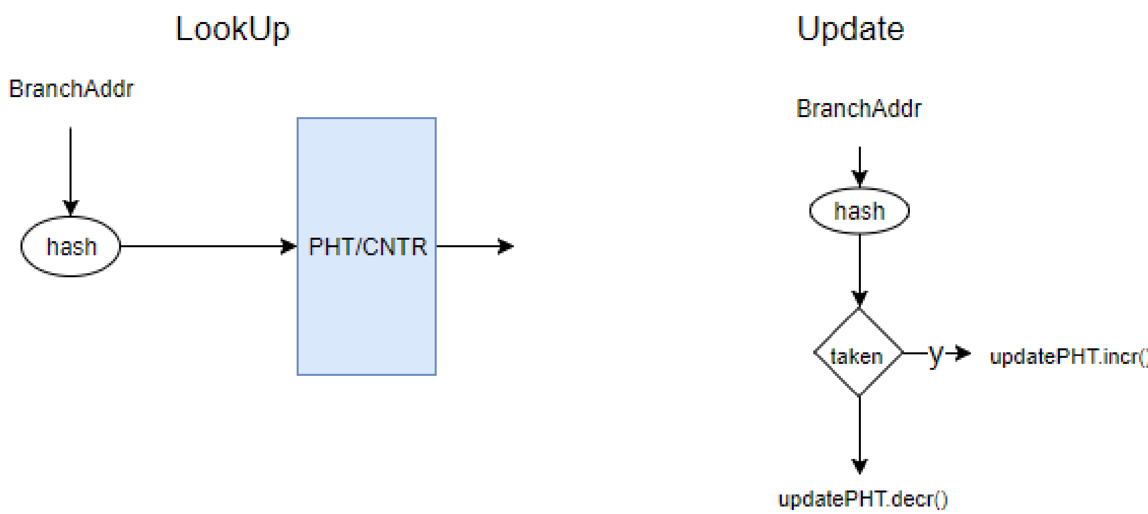

Figure 2: 2bit branch predictor

BiMode Branch Predictor:

BiMode extends the 2 bit counter example with additional two more PHTs each for taken or not taken directions along with choice PHT as shown in figure 3. Further, the hashing function is XORed with the history register to reduce the aliasing effect that was caused with 2 bit counter architecture. [4]

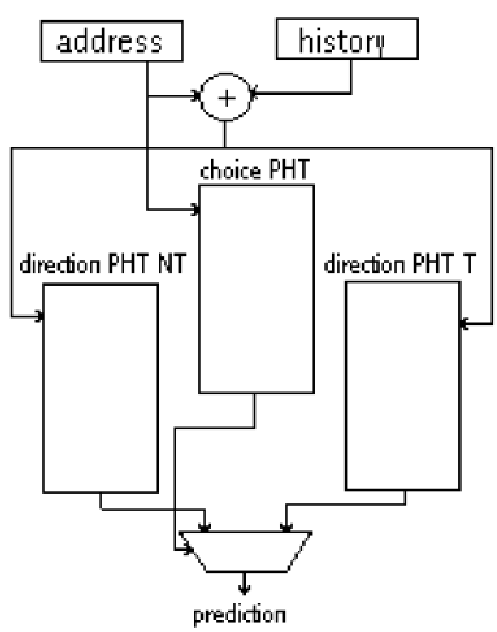

Figure 3: BiMode Branch Predictor

Tournament Branch Predictor:

Tournament branch predictors employees two or more different types of branch predictors (lets take two for example). Some of these branch predictors performs very well for specific set of workloads and some don't. The tournament branch predictor would select one of these predictor's outcomes with the help of meta-predictor (2bits counter). The final outcome is the decision that the meta predictor thinks is a better prediction. The two predictors that are housed are independently trained. The meta predictor is trained based on which how well these two predictors are performing. The over all architecture and working of meta predictor is shown in figure 4. [12]

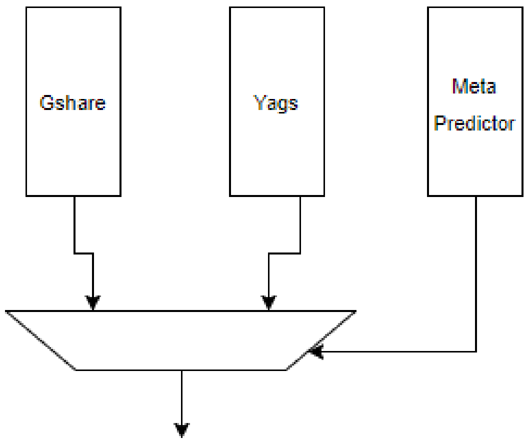

Table 1: Meta Training Heuristics table

Predictor 1	Predictor 2	Meta Training
Taken	Taken	Same Value
Taken	Not taken	Decrement
Not Taken	Taken	Increment
Not Taken	Not Taken	Same value

Figure 4 Tournament branch predictor

Baseline Results:

Two benchmarks named libquantam and soplex are run to measure the performance of the system. Libquantam benchmark has more conditional branch lookups when compared to soplex. However soplex has more indirect branch lookups. Further accuracy of the branch predictor directly govern the IPC values of the cores. This can be clearly been seen from the table 2.

Table 2 and figure 5, 6 provides detailed statistics of respective performance on these benchmarks.

Table 2 IPC and branch prediction accuracy of baseline predictors

	Libquantam	**soplex**	**Libquantam**			**Soplex**		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Acc**	**CondCorrectPred**	**CondIncorrectPred**	**Acc**
LocalBP (2bit)	1.751197	1.234292	51367060	2284506	95.74196	26745999	2432483	91.6634354
BiMode	1.935307	1.307395	45473015	635440	98.62186	24753507	1794049	93.24213122
tournament	1.984506	1.306573	44231850	310574	99.30275	24854449	1824956	93.15968253

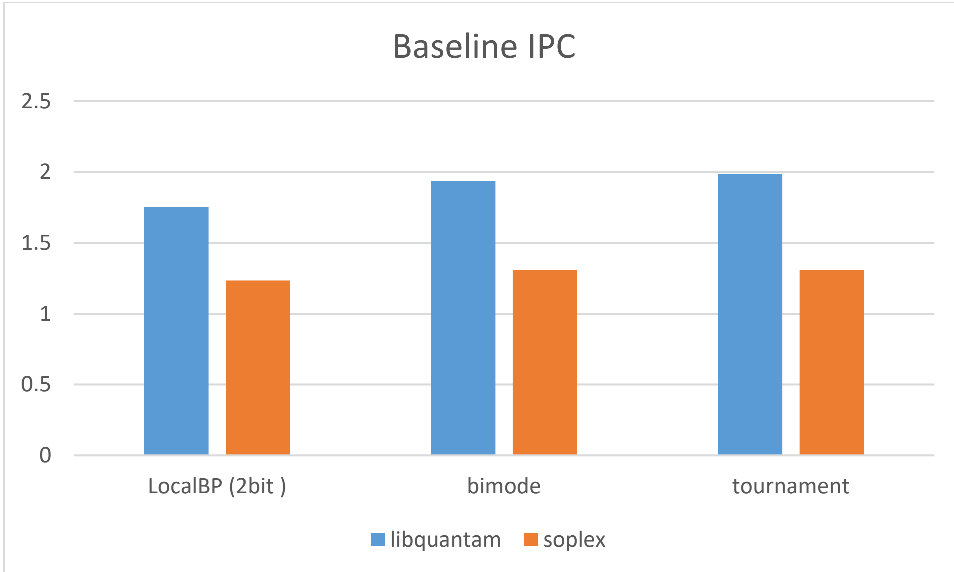

Figure 5: IPC of baseline branch predictors

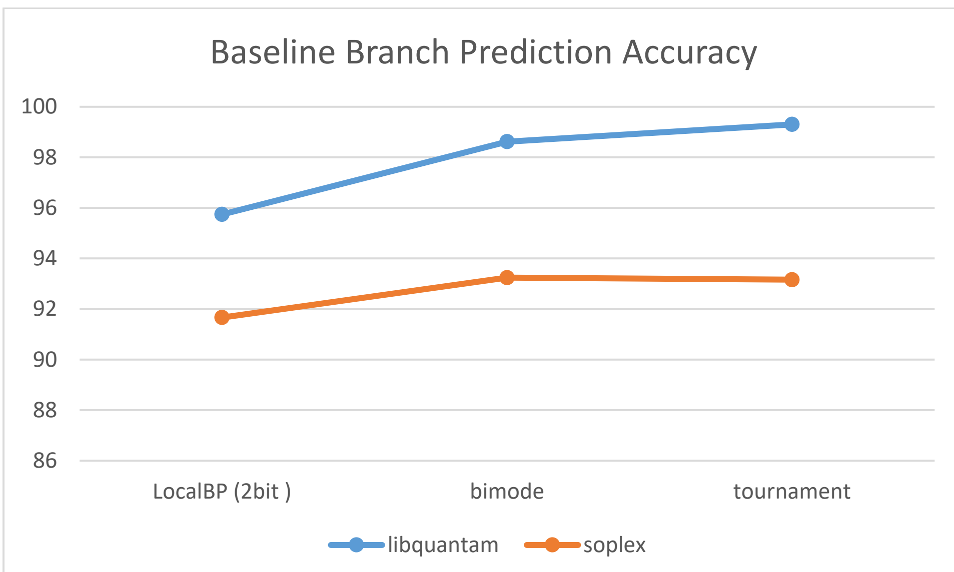

Figure 6: Branch Prediction accuracy of baseline branch predictors

Gshare Branch Predictor:

Ghsare branch predictor is an extension of two level adaptive branch predictor with global shared history buffer and PHT. The hashing function has an XOR of branch address with the global history register. This helps in reducing the aliasing that occurs in the two level branch predictor. Finding a right XOR scheme and size of PHT determines the branch prediction accuracy. The high level architecture of ghare branch predictor is shown in the figure 7. [4]

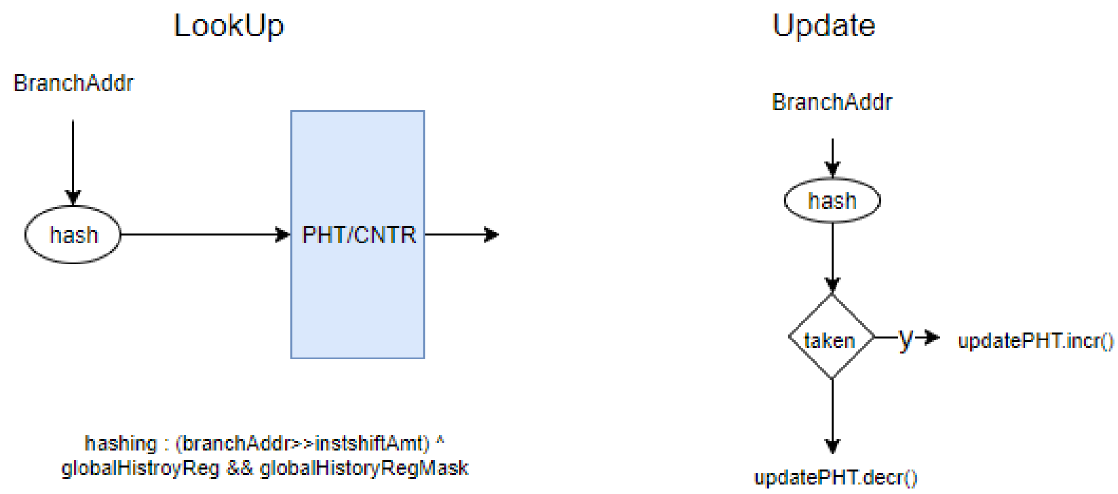

Figure 8: Gshare branch predictor

Modifications in default repository:

1. Two different methodologies were tried to bring up the gshare branch predictor.
   a. Considering BiModeBP as baseline and converting it to gshare branch predictor.
   b. Considering LocalBP as baseline and converting it to gshare branch predictor.
2. Two different PHT sizes where also tried out
3. Two different types of hashing functions are used
   Hashing function 1: `((branch_addr>>instAmt ) ^ globalHistoryReg[tid]) & indexMask; //instAmt was set 2.`
   Hashing function 2: `((branch_addr ) ^ globalHistoryReg[tid]) & indexMask;`

Results:

Two benchmarks named libquantam and soplex are run to measure the performance of the system. Table 3 shows various variations carried out. Figure 9 and figure 10 show the variations of IPC and branch predictor accuracy on various configurations. From the table 3 it is very clear that the hashing function governs the branch predictor accuracy. Identifying a right hashing function is required for better accuracy. With good hashing functions, destructive aliasing can be avoided. Hashing function 2 varies more when compared to hashing function 1 and hence provides reduced aliasing effect. (Hashing function 2 was designed looking into coverage of hashing function 1).

Table 3 IPC and branch prediction accuracy of gshare predictors

	libquantam	soplex	libquantam			Soplex		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**
gshare 2048 hashing func1	1.522101	0.874524	54987369	5165235	91.41311	38846682	7655842	83.53671728
gshare 4096 hashing func1	1.541702	0.873319	54504452	5023860	91.56055	39091352	7667908	83.60130592

gshare 2048 hashing func2	1.589515	0.889025	55365787	4066685	93.15747	37774598	7112332	84.15500459
gshare 4096 hashing func2	1.570493	1.014551	56934084	4416139	92.80176	30513040	4968777	85.9962724
gshare 8192 hashing func2	0.623256	0.847605	55910499	34641989	61.74375	36705748	8097246	81.926998

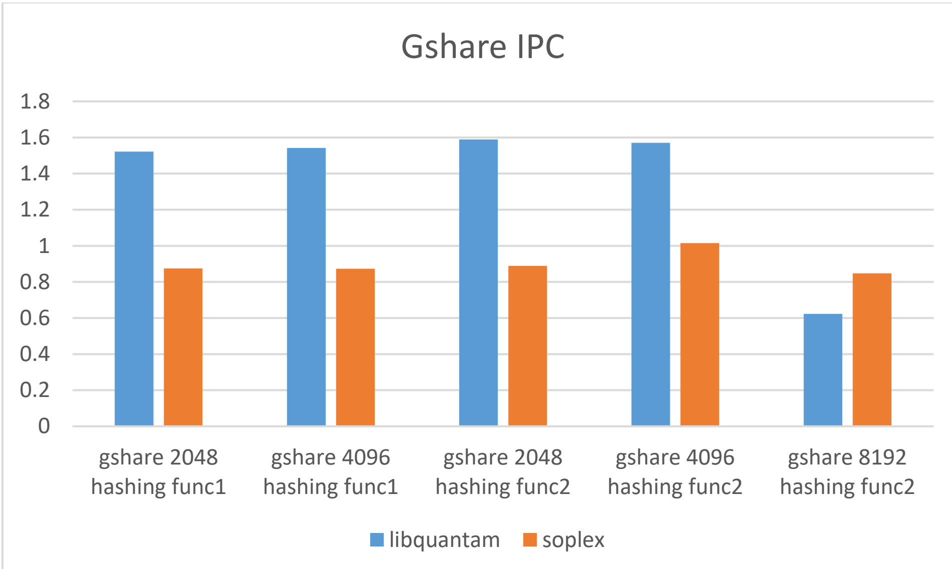

Figure 9: gshare IPC variations

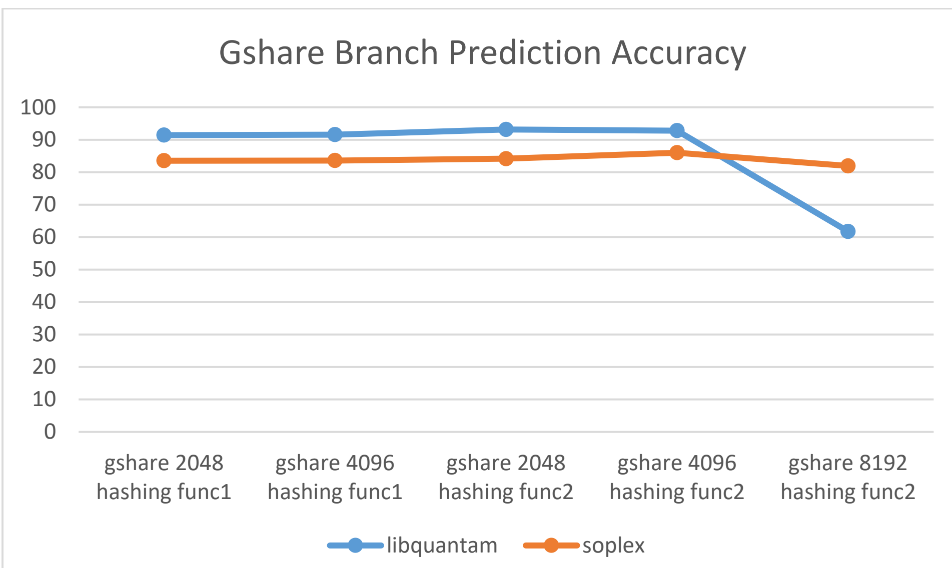

Figure 10: Gshare branch Prediction accuracy

It was quite surprising to note that doubling the PHT size hardly had any effect on branch prediction accuracy. The possible reason is that the PHT table size of 2048 was sufficiently large enough for avoiding the destructive aliasing. A light drop in accuracy was observed with 4096 PHT size. I believe this reduction is primarily because of the constructive interference that is occurring. To confirm my theory, I decided to further increase the PHT size to 8192. This resulted in significant drop of the branch predictor accuracy. Hence choosing right sizes of PHT and hashing are required to get good branch prediction accuracy in gshare branch prediction.

Yags Branch Predictor:

Yags is a short hand representation for Yet another Global Scheme branch prediction. YAGs combines gshare, bimode branch predictors and replaces taken/not taken counters of bimode with taken/not taken caches. Introduction of caches allows the predictor to use tags which helps in reducing aliasing in taken and not taken counters. The overall architecture of yags branch predictor is shown in figure 11.[7]

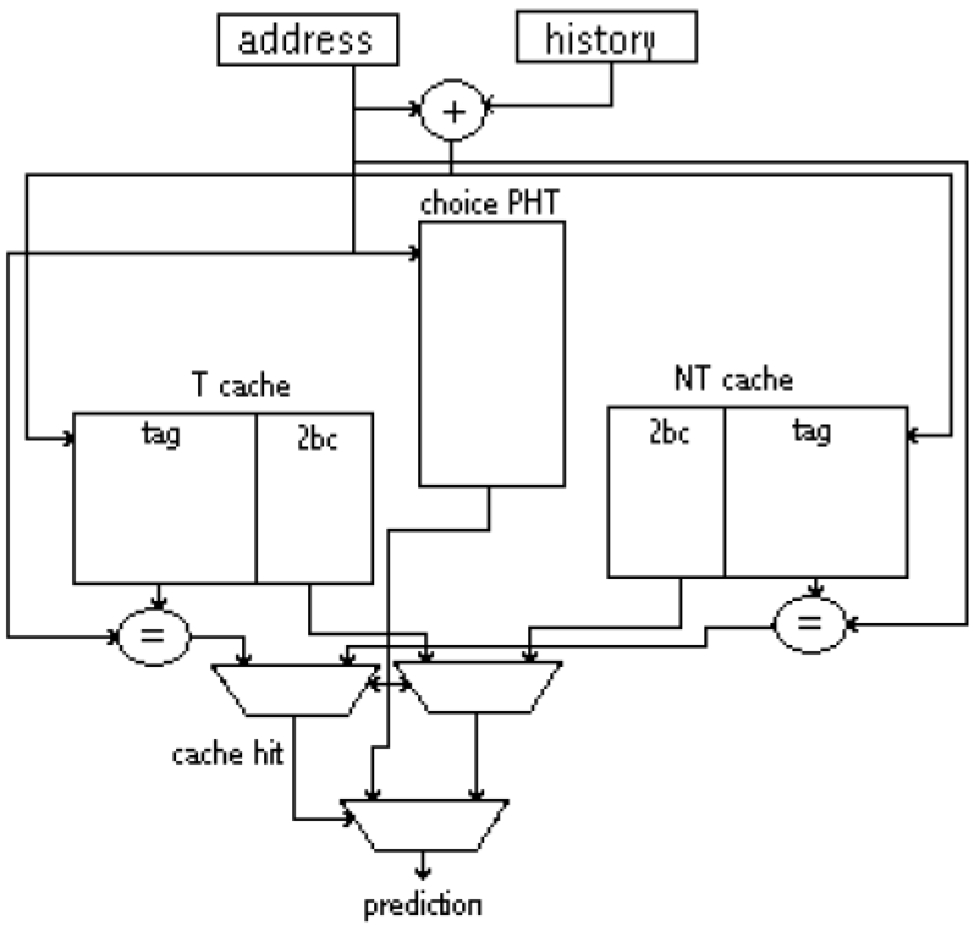

Figure 11: YAGs branch Predictor

Although YAGs does amazing good job for destructive interference, however it may not be able to do well for constructive interference. If the sizes of the PHT is increased, due to constructive interference branch prediction accuracy could be dropped. According to the paper, which implemented both direct mapped (1-way set associative) and 2-way set associative, the performance increases a bit when you go from the former to the latter. This is due to the fact that 2-way set associative array will not thrash mispredict which could be useful in the future. This could be a possible solution to be implemented to reduce the constructive interference. However the implementation that is discussed here has direct mapped caches only.

Modifications in default repository:

BiModeMP is taken as reference to create the YAGs branch predictor. Following modifications where done for handing the yags.

1. Not taken and taken caches are designed with counters along with tags.
2. Tag is generated based on below equation:
   `tag = ((branchAddr >> instShiftAmt) & tagsMask) | ((globalHistoryReg[tid] & globalHistoryUnusedMask) );`
3. A temporary buffer registers are used to track the history and if taken or not taken caches were hit. This is further used to train the predictor during the update phase.
4. In the training phase, counters are incremented or decremented depending on if the cache was hit or not or was it selected from taken/not taken cache or was choice prediction was selected.

Figure 12 and Table 4 shows the flowchart and truth table that govern the high level software architecture.

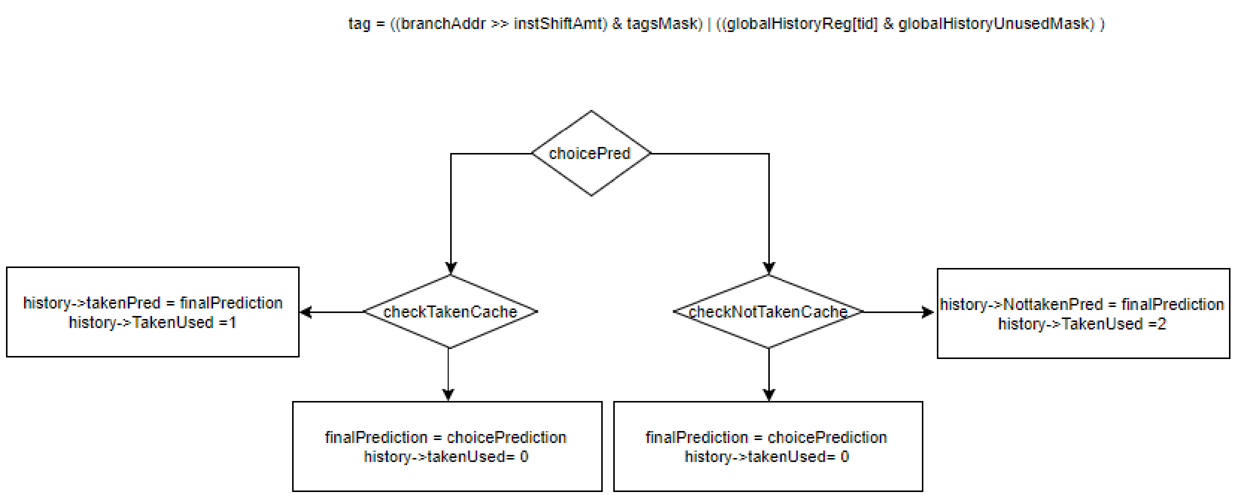

Figure 12: Yags Lookup strategy

Table 4: Yags update strategy

history->takenUsed	History->Predicted	taken	Decision
0 – choice	0	0	PHTcntr--
	0	1	PHTcntr++, updateTakenCache
	1	0	PHTcntr--, updateNotTakenCache
	1	1	cntr++
history->takenUsed	**history->taken**	**taken**	**Decision**
1 – taken	0	0	cntr, updateTakenCache
	0	1	cntr++, updateTakenCache
	1	0	cntr--,updateTakenCache
	1	1	cntr++, updateTakenCache

history->takenUsed	history->Nottaken	taken	Decision
2 – Nottaken	0	0	cntr++, updateNotTakenCache
	0	1	cntr++, updateNotTakenCache
	1	0	cntr--,updateNotTakenCache
	1	1	cntr, updateNotTakenCache

Results:

Similar to other branch predictors, libquantam and soplex benchmarks were run. The results of these benchmarks are in table 5.

Table 5: IPC and branch predictor accuracy for YAGs branch predictor

	libquantam	**soplex**	**libquantam**			**soplex**		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**
yags 1K cache 2048PHT	1.750276	1.237431	51278380	2282297	95.73886	26684750	2395647	91.76198661
yags 1K cache 4096PHT	0.718797	0.817819	55157653	26825262	67.27945	41028332	8814063	82.31613268
yags 2K cache 2048PHT	1.750276	1.237431	51278380	2282297	95.73886	26684750	2395647	91.76198661
yags 2K cache 4096PHT	0.718797	0.817819	55157653	26825262	67.27945	41028332	8814063	82.31613268

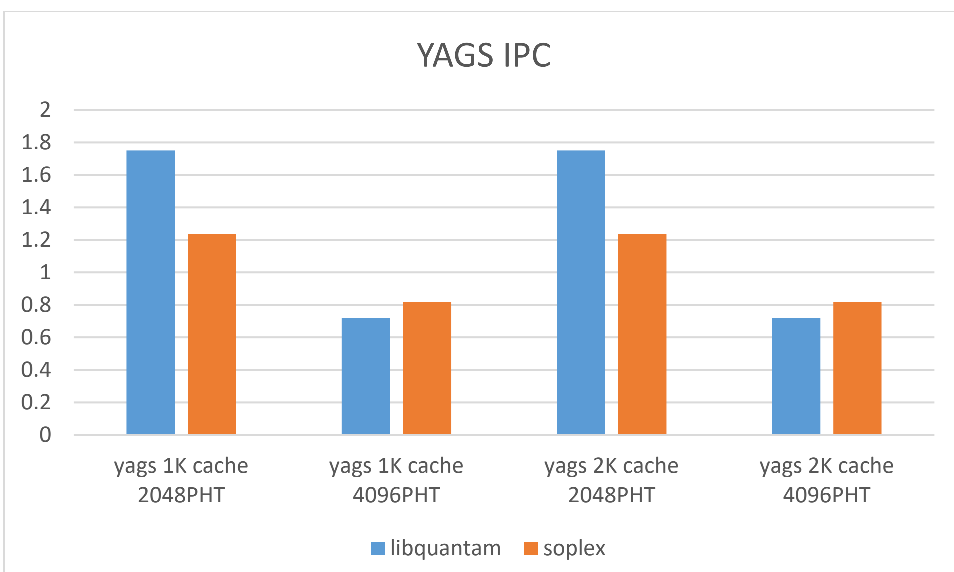

Figure 11: Yags branch predictor IPC

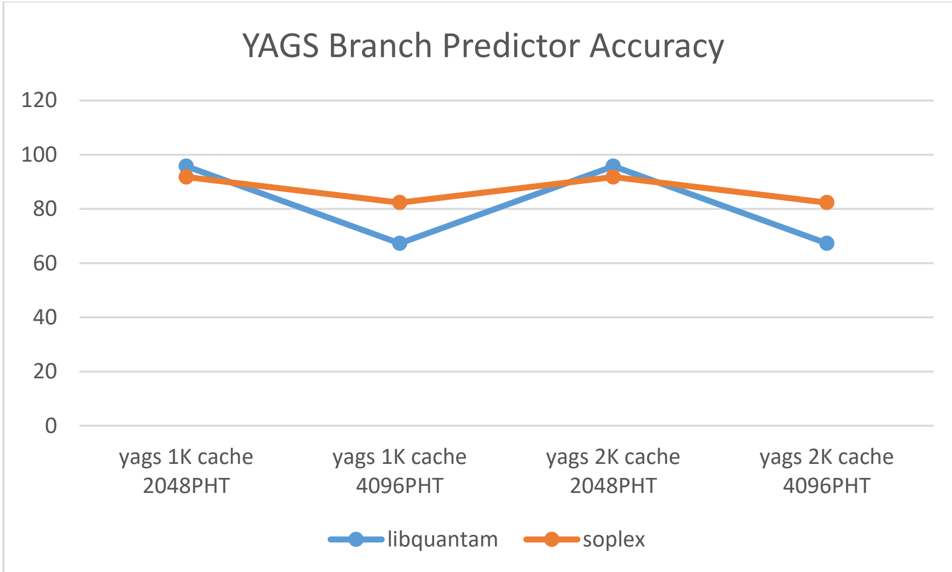

Figure 12: Yags branch predictor accuracy

Discussions:

YAGs provided excellent prediction accuracy when the PHT size was 2048. Variations in cache sizes had hardly any effect in the branch prediction accuracy and IPC. However the accuracy dropped significantly with higher PHT sizes. Double the PHT size keeping the cache size dropped the accuracy by 30%. Increasing the cache size had hardly any effect in branch prediction accuracy. Since the accuracy of branch prediction was low with higher PHT, the IPC values were dropped as the core had to service an instruction longer with more pipeline flush. One of the reasons why this significant drop in accuracy would have occurred is because of constructive interference.

To understand the relationship between cache and PHT size, I decided to reduce the cache size to 512 keeping PHT size of 4096. From table 6, it can be noted that the branch predictor accuracy slightly improved when cache size was reduced. I believe the improvement in accuracy is primarily because of reduction in constructive interference.

Table 6: IPC and Branch prediction accuracy for Yags with 0.5K cache and PHT of 4096

	Libquantam	**soplex**	**libquantam**			**soplex**		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**
yags 0.5K cache 4096PHT	0.747442	0.817064	55427884	24806390	69.08255	40729238	8839944	82.16645173

Perceptron Branch Predictor:

Perceptron predictor is the first dynamic branch predictor that uses neural network. The simple neural network consisting of input/output layer and table of perceptron are used instead of PHT table. In the paper, they claim the perceptron predictor is better for hardware implementation is it is easier to implement over other neural network based branch predictors. Overall architecture of perceptron branch predictor is shown in figure 13. [4][8][11]

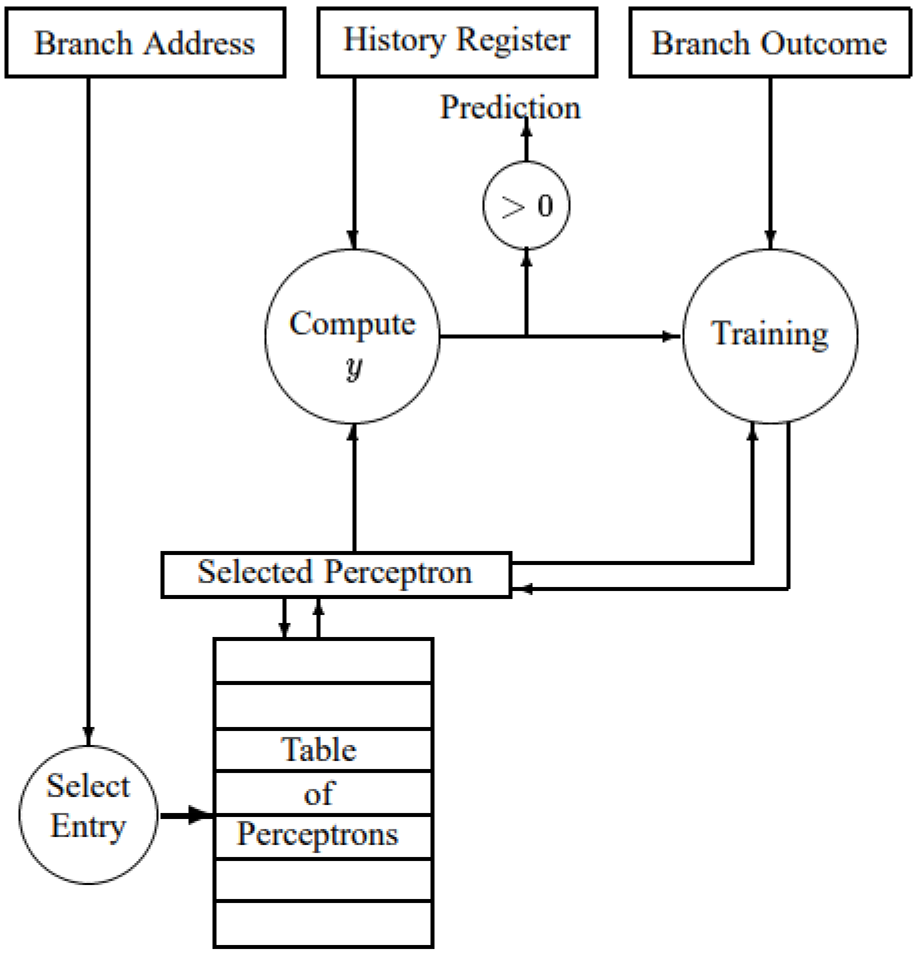

Figure 13: Perceptron based branch predictor architecture

Implementation details:

Following steps are involved when using perceptron branch predictor:

1. The branch address is hashed to produce an index into table of perceptrons.
   A very simple hashing function is used.
   `globalHistoryIdx = ((branchAddr ^ globalHistoryReg[tid]) & globalHistoryMask);`
2. The i^{th} perceptron is fetched from the table into a vector of register
3. In the code, weights acts as the table of perceptrons. The weight is declared a 2 dimensional variable where one of its dimension is historylength and other is history register. This is done to simplify the coding problem that may occur. The history register here can be implemented in

different values. Having an array of integers or in single unsigned 64 bit integer. Both the methods were tried out and hardly any difference in accuracy was observed. Rest of the parameters were referenced from the paper. I also assumed that there is infinite hardware budget.

4. The inference y is calculated as the dot product of weights[index] and global history register bits.
5. If y is negative, the branch is not taken otherwise its is taken.
6. The outcome of y and learning rate are used to train the neural network.
   The equation of learning rate is derived from the paper as follows:
   `theta = (1.93*globalPredictorSize+14)`
7. Weight is calculated with below equation and updated in the table:

```
If sign(yout != t) or |yout| <= theta then
        For i:0 to n do :
                 Weight[i] = weight[i] + taken * x[i]
        End for
End if
```

Results:

Similar to other predictors, libquantam and soplex benchmarks are run with various history length sizes. The table 7 and figure 14 shows the IPC and branch prediction accuracy of various runs. From the table 7, it was quite surprising that no matter what the history length values were, the result hardly changed. I believe this is mainly because of early saturation of learning. In other words the neural network stops learning after certain point in time for a given theta value. To understand in this further, I reduced the learning rate equation to 32 which provided less learning samples to the neural network. It can be clearly seen from the table that reducing the learning rate reduces the branch prediction accuracy. However increasing the same, beyond certain point hardly has any effect on the branch prediction accuracy. This is primarily because of learning saturation which occurs in any neural network.

Table 7: Perceptron Branch predictor results for various lengths and theta

	libquantam	soplex	libquantam			soplex		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**
perceptron 12	1.568471	0.800348	55229432	4432938	92.56996	42340737	9182902	82.17730312
perceptron 24	1.568471	0.800348	55229432	4432938	92.56996	42340737	9182902	82.17730312
perceptron 48	1.568471	0.800348	55229432	4432938	92.56996	42340737	9182902	82.17730312
perceptron 48 theta =32	1.16848	0.819411	55757950	12464025	81.73019	38698286	8745330	81.56689827
perceptron 48 theta = 128	1.568471	0.800348	55229432	4432938	92.56996	42340737	9182902	82.17730312

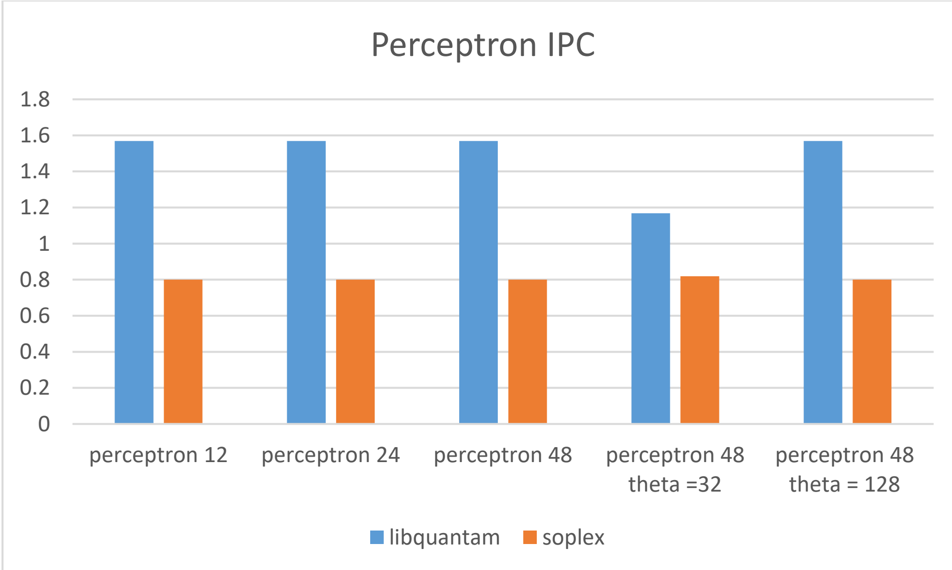

Figure 14: Perceptron IPC variations

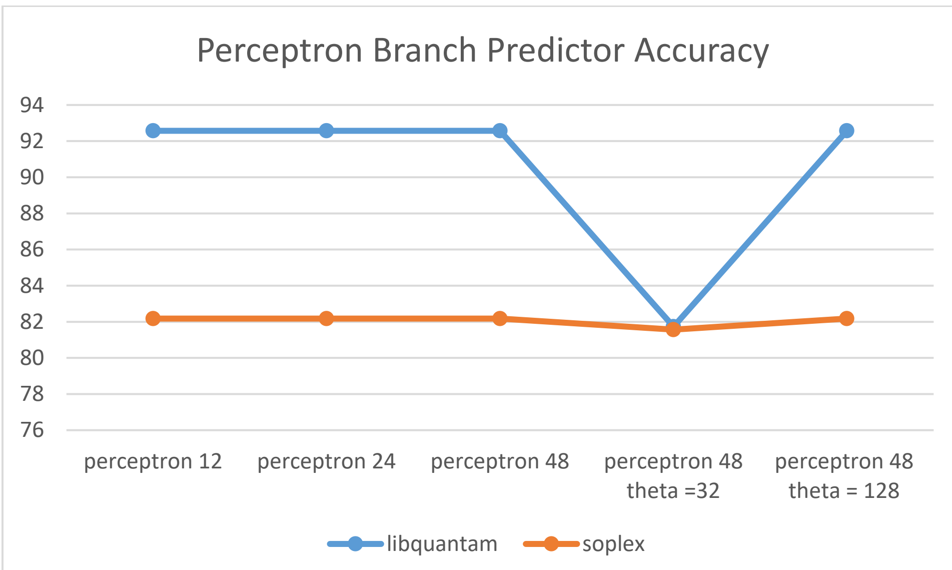

Figure 15: Perceptron Branch Predictor accuracy

Analysis of Results:

Best results from various branch predictor schemes are shown in Table 8.

Table 8: Results of various branch predictors

	libquantam	soplex	libquantam			soplex		
BP Type	**IPC**	**IPC**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**	**CondCorrectPred**	**CondIncorrectPred**	**Accuracy**
LocalBP (2bit)	1.751197	1.234292	51367060	2284506	95.74196	26745999	2432483	91.66344

bimode	1.935307	1.307395	45473015	635440	98.62186	24753507	1794049	93.24213
tournament	1.984506	1.306573	44231850	310574	99.30275	24854449	1824956	93.15968
gshare 2048 hashing func2	1.589515	0.889025	55365787	4066685	93.15747	37774598	7112332	84.155
yags 1K cache 2048PHT	1.750276	1.237431	51278380	2282297	95.73886	26684750	2395647	91.76199
perceptron 48	1.568471	0.800348	55229432	4432938	92.56996	42340737	9182902	82.1773

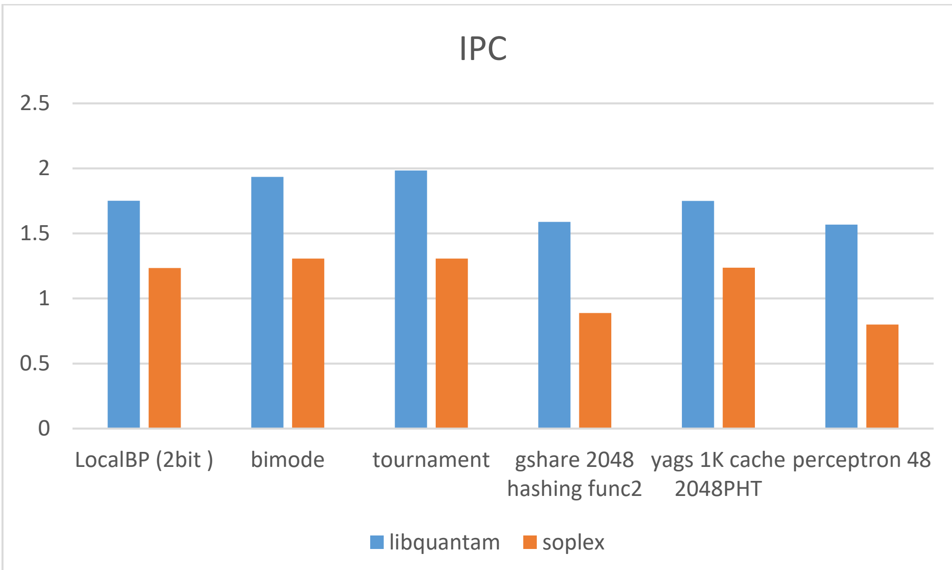

Figure 16: IPC of various branch predictors

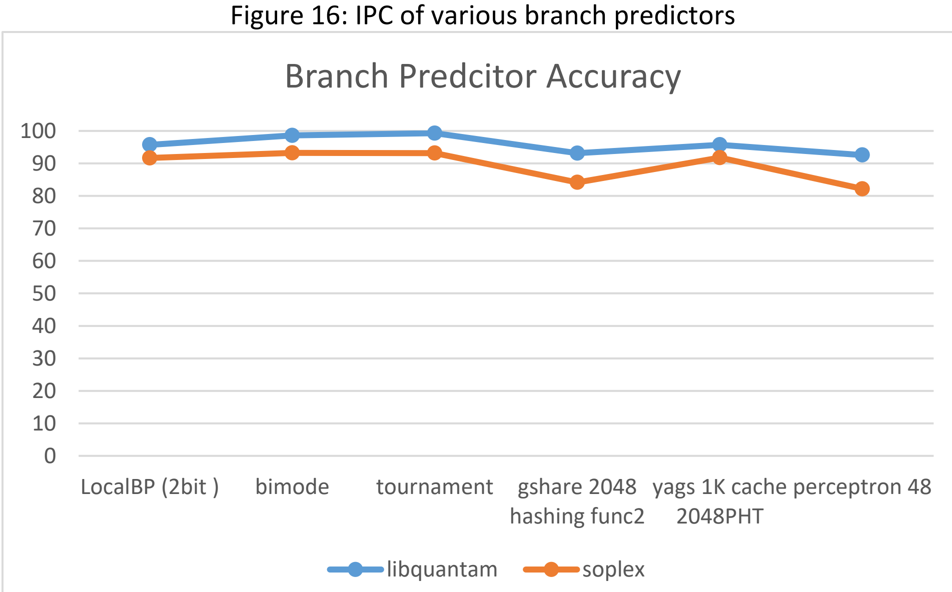

Figure 17: Branch Predictor Accuracy of various branch predictors

Following points can be derived from the table:

1. Among all the predictors, tournament predictor provided best accuracy and IPC values. The reason for this can be attributed to its structure. It combines local and global predictor to evaluate which of the two has better accuracy and for every branch one of them is selected.
2. Lowest among all is the perceptron predictor. There can be multiple causes for this. However one of the main reason is the learning saturation that occur as discussed in Perceptron branch predictor. Choosing appropriate bias and weights govern the outcome of this predictor.
3. Among the three implementation that I have done, YAGs provided the best performance. This can be attributed to reduced aliasing that occur due to cache design and tags. Constructive interference does occur if PHT sizes are changed without changing the cache sizes. The performance can be further improved with associative caches.
4. Ideally speaking gshare should have provided equally good prediction as localBP. However, due to aliasing issues that still occurred the performance is lower than localBP. Identifying right hashing can provide really good results with gshare. This can be seen with second hashing function that was designed over the first one.

Ending thoughts:

1. In current system, context switching is very common. None of the papers that was studied for this assignment speaks about it. All the paper assumed to be of single thread running with full context and no switching. This is hardly a true. For scenario where there are context switching, is it worthwhile to store the least amount of branch prediction state for each thread? (predication, trace cache are good approaches too)
2. Are there any standard methodologies for reducing aliasing? Everyone using a hashing function. In fact even in perceptron paper, a hashing function is used to identify an element from perceptron table. No one has discussed the use hashing methodology or how to identify best hashing policy.
3. Although YAGs and perceptron did not beat the performance of tournament but individually both tries to solve fundamental problems of branch predictor. I wonder how combing YAGs and perceptron predictor like tournament predictor would work. Perhaps a project idea?

References:

1. Gem5.org
2. www.learning.gem5.org/book/index.html
4. http://courses.engr.illinois.edu/ece511/secure/homework/assignment2.pdf
5. New algorithm that improves branch prediction: pdinda.org/icsclass/doc/mpr-branchpredict.pdf
6. I-CK Chen Chih-Chieh Lee and Trevor N. Mudge.
7. A. N. Eden and Trevor Mudge. Yags branch predictor scheme
8. Daniel a. Jimenez and Calvin Lin. Dynamic branch prediction with perceptrons. Hpca, page 197, 2001.
9. Chris Feucht Matt Ramsay and Mikko H. Lipasti. Exploring efficient smt branch predictor design.
10. https://web.njit.edu/~rlopes/Mod5.3.pdf
11. Perceptron understanding – Cpp code – https://github.com/sumitdhoble/Branch-Prediction/blob/master/predictor.c
12. Alpha 21264 - https://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=21264hrm.pdf