CSE 240A Branch Predictor Project Report

https://github.com/Allison-Turner/CSE240A

Allison Turner, James Yuan

1 Introduction

Branch prediction is a category of algorithms that aim to *predict* the outcomes of *branch* instructions as accurately as possible, so that when post-branch instructions are loaded during the later execution stages of the branch, the chance that those post-branch instructions will be flushed is *minimal* [1]. Minimizing the occurrence of anticipated post-branch instruction flushing is desirable because such flushes waste instruction cycles, increasing overall execution time and making programs less efficient.

Accurate branch prediction is difficult because branch resolution patterns are highly dependent on the type of program in question. Some applications' branch resolutions obey similar rules of spatial or temporal locality as cache replacement policies, and others may have different behavior patterns based on interdependent branches. Algorithms that reference *all* recent branch outcomes, or "global history", attempt to leverage any correlation between the most recent branch instruction outcomes and the current, while those that reference the most recent outcomes for branch instructions from a similar address, or "local history", attempt to factor the uniqueness of different program regions into their forecasts. At best, branch prediction algorithms could make the performance of programs with branches equal or near-equal to those without, and at worst, these algorithms perform similarly or worse than a processor that always predicts the same branch resolution; so, the cost-benefit ratio of trying out new methods is pretty positive towards new approaches.

1.1 Gshare

Gshare is a **correlated predictor** that combines insights from global history with instruction address locality by accessing a table of 2-bit predictors with the XOR of global history and the branch instruction's address [2]. In its favor, Gshare has a relatively small implementation overhead. However, its performance suffers when branch resolutions are more accurately correlated with localized patterns.

1.2 Tournament

Tournament branch prediction is a **hybrid** approach, which combines the advantages of correlated predictors, such as Gselect and Gshare, with a local-history-aware approach, such as pattern history tables (PHTs), in cases ill-suited to globally correlated prediction [2]. Its implementation overhead is much larger than the compact needs of Gshare, because tournament *contains* not only a global predictor like Gshare, but also a local predictor like PHT, and a method for choosing between the two. However, it is easily more accurate than correlated predictors or local predictors alone, since it can switch to the most accurate predictor ad hoc. The size of each component predictor used in tournament can also be decreased, since only some cases will end up choosing this predictor's forecasts.

1.3 Perceptron

A *perceptron* is a mathematical model of a neuron that can simulate "learning" from a given set of inputs. The model gets better as it refines the coefficients for a function with binary output. This machine learning concept can be applied to branch prediction by maintaining a list of function coefficients for every address whose lower n bits map to a particular row entry in a table of coefficients [3]. The accuracy of Perceptron alone can be enhanced by increasing from a single-layer perceptron to a multi-layer, however the implementation overhead for a multi-layer is significantly more in comparison with any other predictor aforementioned in this writeup.

1.4 Implementation

Our project focused on simulating branch predictor algorithms on a small unit of program execution traces, to demonstrate and compare their misprediction rates and implementation overheads. We wrote our simulated branch predictors in C, and then tested each predictor on each execution trace with a range of parameters.

1.4.1 Distribution of Work

The distribution of work was as follows:

CSE 240A course staff provided traces and the skeleton of a simulator implemented in C.

Allison Turner implemented the data and control structures necessary for simulating gshare and tournament branch predictor schemes.

James Yuan implemented the data and control structures necessary for simulating single Perceptron and Perceptron-Gshare tournament branch predictor schemes.

1.4.2 Gshare

Our implementation of Gshare uses the following data structures:

```
unsigned int global_history
unsigned int* branch_history_table
```

global_history is an unsigned integer of width ghistoryBits, initialized to all zeros. branch_history_table is an array of unsigned integers of size $2^{ghistoryBits}$, where every entry is a 2-bit predictor state, initialized to the constant unsigned int weak_not_taken.

On a call to make_prediction with argument PC, our implementation xors global_history with PC, then applies a bitwise mask to ensure the exclusion of PC's higher-order bits. The resulting value is used as the index with which to access branch_history_table, where the predictor state contained dictates the prediction.

On a call to train_predictor with arguments PC and outcome, our implementation re-calculates the xor of PC and global_history, so it can update the associated predictor state in branch_history_table, and then shifts the new outcome into global_history.

1.4.3 *Tournament*

Our first tournament predictor combines a local pattern history table approach with Gshare and a "chooser" table.

```
//chooser
unsigned* chooser;
unsigned* total_predictions;
unsigned* local_mispredictions;
unsigned* global_mispredictions;

//local pattern history
unsigned* local_pattern_hist;
unsigned* pattern_hist_predictor_state;

//Gshare
unsigned int global_history
unsigned int* branch_history_table
```

local_pattern_hist is an array of unsigned integers meant to represent the last n outcomes for a given PC value or set of PC values, where n = lhistoryBits. All entries in local_pattern_hist are initialized to 0. The PC values are mapped to an entry in this table by selecting m of their lower bits, where m = pcIndexBits. Once the local history is indexed from this table, that pattern is used to index pattern_hist_predictor_state, where, finally, a 2-bit predictor state is stored. This 2-bit state is initialized to weak_not_taken.

For every entry in local_pattern_hist, there is an entry in the chooser array that contains one of two constant unsigned integers: SIMPLE_BHT or CORRELATED_PREDICTOR. On a call to make_prediction with argument PC, we index the chooser array to determine which predictor we should use, and then return based on the relevant entry of// pattern_hist_predictor_state or branch_history_table.

On a call to train_predictor() with arguments PC and outcome, our implementation shifts the newest outcome into the relevant entry of local_pattern_hist and updates the predictor states in pattern_hist_predictor_state and branch_history_table. Finally, we re-calculate the misprediction rates of PHT and Gshare, and reset the chooser value to whichever scheme has the lowest error rate.

1.4.4 *Perceptron*

For our custom branch prediction scheme, we started out in Python, attempting to implement a Multi-Level Perceptron predictor. However, after many difficulties with both Python usage in conjunction with C, and with the speed of our Python Perceptron's predictions, we shifted to a single-layer Perceptron written in C. Curiously, the choke point on performance was not within our usage of PyTorch, but was within the serialization we performed in order to communicate between Python and C.

The custom brach predictor uses a local pattern history table and a 2D array representing the perceptron function.

```
signed int* func;
unsigned int* local_pattern_hist;
```

local_pattern_hist is an array of unsigned integers meant to represent the last n outcomes for a given PC value or set of PC values, where n = lhistoryBits, just like in the implementation of tournament. Each of the entry is initialized to 0.

func is a 2D array of signed integers. The PC values are mapped to a certain row of the array, then each entry from the row represents the coefficient of the perceptron function, and those are mapped to the corresponded local history bits. Each of the entry is initialized to 10.

On a call to train_predictor() with arguments PC and outcome, our implementation updates the entry in local_history_table corresponding to the PC, by shifting the history bits and adding the newest outcome. Besides that, func is updated in the way that if the new outcome is taken, then we go through each bit in the corresponded local history, and if the current local history bit is 1, then we have a positive feedback and increment the learning rate to the corresponded coefficient, otherwise if the current local history bit is 0, then this is a negative feed back so we decrement the learning rate from the coefficient.

When the perceptron predictor needs to make a branch prediction, it simply takes the PC and finds the local history and the coefficients. We calculate the score which is a sum of product of the local history bit and its coefficient, and compare it to the threshold. if the score passes the threshold, then we predict the branch to be taken, otherwise we predict not taken.

1.4.5 *Perceptron-Gshare Tournament*

Our next trial is implementing a hybrid tournament predictor of single-layer perceptron and Gshare.

```
//chooser
unsigned* chooser;
unsigned* total_predictions;
unsigned* local_mispredictions;
unsigned* global_mispredictions;
unsigned custom_local_prediction;
unsigned custom_global_prediction;

//local pattern history
unsigned* local_pattern_hist;

//Gshare
unsigned int global_history
```

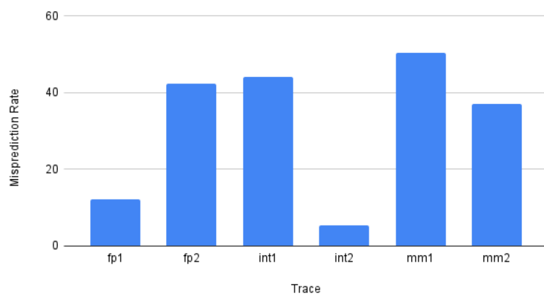
```
unsigned int* branch_history_table

//perceptron
signed int* func;
```

The data structures follow the same descriptions in the section 1.4.3 for tournament and 1.4.4 for perceptron. But now in the training predictor function, we train both gshare and perceptron, and then update the local_mispredictions for the perceptron predictor and the global_mispredictions for the Gshare predictor. In the make predictionfunction, we let both predictors make their own prediction, then based on the misprediction rate, we choose the prediction from a better performed predictor.

2 Observation

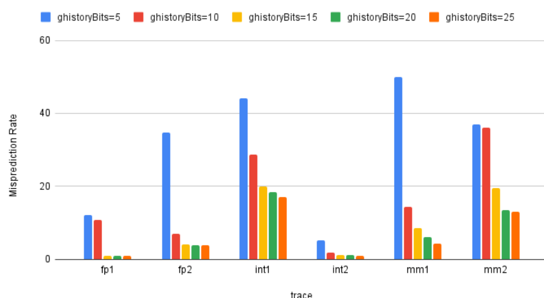
Misprediction Rates with Static

trace	Misprediction Rate
fp1	12.128
fp2	42.350
int1	44.136
int2	5.508
mm1	50.353
mm2	37.045



Misprediction Rates with Gshare (column headers give value of ghistoryBits, cells give misprediction rates for that parameter & execution trace combination)

trace	5	10	15	20	25
fp1	12.127	10.682	0.906	0.912	0.919
fp2	34.755	7.053	4.123	3.917	3.918
int1	44.135	28.641	20.009	18.272	16.994
int2	5.279	1.869	1.114	1.076	0.957
mm1	50.053	14.368	8.505	6.120	4.361
mm2	37.028	36.146	19.471	13.424	13.011

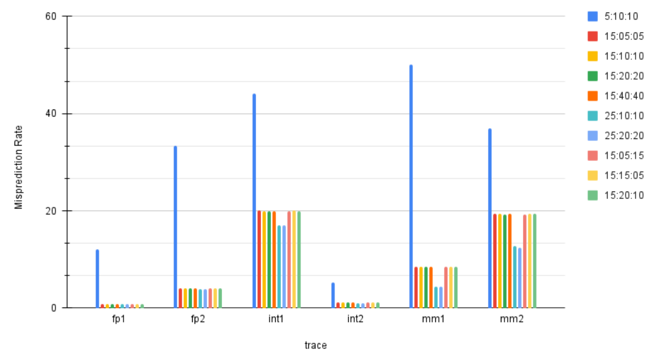


Misprediction Rates with Tournament (column headers give values of ghistoryBits:lhistoryBits:pcIndexBits,cells give misprediction rates for that parameter & execution trace combination)

trace	5:10:10	15:5:5	15:10:10	15:20:20	15:40:40
fp1	12.126	0.905	0.908	0.913	0.903
fp2	33.394	4.124	4.123	4.123	4.123
int1	44.134	20.010	19.843	19.840	19.887
int2	5.273	1.115	1.113	1.113	1.113
mm1	50.045	8.506	8.508	8.508	8.505
mm2	37.027	19.474	19.450	19.311	19.488

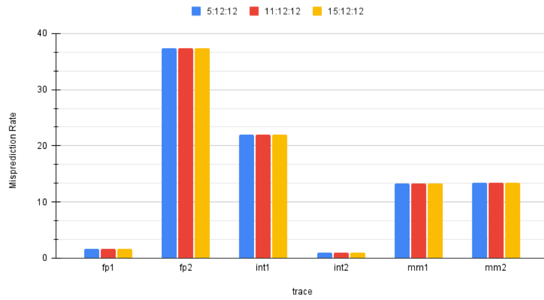
Misprediction Rates with Tournament (cont'd) (column headers give values of ghistoryBits:lhistoryBits:pcIndexBits, cells give misprediction rates for that parameter & execution trace combination)

trace	25:10:10	25:20:20	15:5:15	15:15:5	15:20:10
fp1	0.907	0.912	0.913	0.905	0.908
fp2	3.917	3.917	4.123	4.124	4.123
int1	16.977	16.976	19.840	20.010	19.843
int2	0.954	0.954	1.113	1.115	1.113
mm1	4.367	4.365	8.508	8.506	8.508
mm2	12.821	12.498	19.322	19.474	19.450



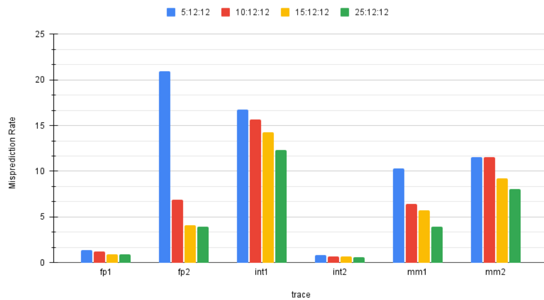
Misprediction Rates with Perceptron Alone (column headers give values of threshold:lhistoryBits:pcIndexBits, cells give misprediction rates for that parameter & execution trace combination)

trace	5:12:12	11:12:12	15:12:12
fp1	1.664	1.664	1.668
fp2	37.338	37.339	37.340
int1	21.922	21.922	21.921
int2	0.950	0.950	0.947
mm1	13.296	13.297	13.300
mm2	13.383	13.379	13.395



Misprediction Rates with Perceptron x Gshare Tournament Prediction (column headers give values of ghistoryBits:lhistoryBits:pcIndexBits, threshold is set to 1, cells give misprediction rates for that parameter & execution trace combination)

trace	5:12:12	10:12:12	15:12:12	25:12:12
fp1	1.407	1.233	0.890	0.891
fp2	20.922	6.889	4.122	3.916
int1	16.772	15.697	14.294	12.338
int2	0.864	0.723	0.682	0.638
mm1	10.304	6.424	5.746	3.954
mm2	11.569	11.521	9.244	8.061



3 Results and Conclusions

When global history is very low for gshare and tournament, performance is more like static. For example, when we are using 5 bits of global history with gshare, the misprediction rate is about the same as static. This is simply because when there are not enough bits to store the global history, temporal locality patterns cannot be captured by the Gshare branch predictor.

Higher global history usually leads to better performance, but it usually plateaus when global history is around 15. After that point, the marginal improvement drops when trying to have more global history bits. Nevertheless, for some tasks it might be higher than 15. For mm_2 we can still see a large increase in performance when global history bits are raised from 15 to 20. This is because the traces have different temporal locality. For simple trace like fp_1, not a lot of bits are required to record the global history pattern, while for relatively complicated trace like mm_2, there is a longer global history pattern, thus more history bits are required to record the pattern.

When local history and PC are very low for tournament, then performance is more like gshare. For instance, when we are using 15 bits of global history and only allocate 5 bits to both local history and PC index with tournament, the misprediction rate is almost the same as just using 15 bits of global history in gshare.

Increasing local history bits does not have an obvious improvement on performance. For example when we are using 15 bits of global history and 5 bits for the PC index, and when we increase local history bits from 5 to 15, no improvement can be observed. The same situation happens when we are suing 15 bits of global history and 10 bits of PC, and increase local history bits from 10 to 20. Moreover, increasing PC does improve performance, but not as obvious as increasing global history. With our tournament implementation for example, changing the parameters from 15:5:5 to 15:5:15 does decreases the misprediction rate for most of the traces, by a slight amount. Noticing allocating same amount of bits to global history gives us better outcome than any of the other parameters. This is suggesting that global history in general gives more information than local history in the given traces, especially in trace fp_2.

In addition, increasing any of the parameters does not necessarily increase performance in all cases. For example, in gshare predictor performance with the trace fp_1, increasing global history bits from 15 actually makes performance worse. In tournament predictor with the trace fp_1, increasing PC index from 5 to 15 with 15 bits of global history and 5 bits of local history also decreases prediction accuracy. This is because fp_1 only has limited temporal locality and spacial locality. When extra global bits are given to the gshare predictor, those bit are not captured by the temporal locality pattern, therefore those become confounding variables for the predictor. Similarly with the tournament predictor, When extra PC index bits are given to the branch predictor, those are not captured by the spacial locality, and thus harm the accuracy.

Our implementation of simple single-layer perceptron does work, however, its performance is underwhelming. Perceptron really shines only when implemented as part of a tournament scheme, in cases where local history is a more reliable forecaster. This is consistent with Jiménez and Lin's original proposal for perceptron branch prediction [3].

The implementation overhead required for the Gshare predictor or PHT and Gshare tournament predictor has to be much higher in order to achieve similar or better performance to our hybrid single layer perceptron and Gshare implementation, in most cases. The one trace where our implementation of Perceptron-Gshare tournament doesn't *significantly* outperform other simulated predictors, the fp2 trace, can likely be attributed to that trace's branches following global patterns more closely, so Gshare wins over any local history predictor for this trace every time, however Gshare is still a suboptimal global history approach; thus simulations on fp2 still fall victim to Gshare's relative underperformance.

In conclusion, of the branch prediction algorithms we simulated, we find that hybrid predictors that prioritize space for global history and articulate a statistics-based approach for local history are optimal. If we were to iterate further on our simulations, our next simulated predictor would compare

Gshare with Gselect and other correlated predictors, both independently and as part of a tournament predictor with Perceptron, and perhaps analyze memory access for a predictor's declared data structures in order to comment on utilization.

4 References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Waltham, MA, USA: Morgan Kaufmann, 5th ed., 2012.

[2] S. McFarling, "Combining branch predictors," tech. rep., Palo Alto, CA, USA, 1993.

[3] D. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, 2001.