CSE 240A Branch Predictor Project

Subhashree Navaneethan
snavaneethan@ucsd.edu

Suhas Pai
sspai@ucsd.edu

I. Introduction

Branch prediction is the process of guessing or speculating the direction of a conditional branch instruction before the correct branch outcome is known. Branch speculation is often performed as correct speculations lead to improved processor performance and also avoids control hazards in a pipelined processor. However, incorrect speculations require flushing out incorrect instructions from the pipeline and undoing any changes made to processor state. Hence, branch speculation is a risky transaction with chances of profit.

Branch prediction can be static or dynamic. Static branch prediction is performed when the processor decides to either always take the branch (and continue program execution from the branch target) or always not take the branch (and continue program execution as usual). This is often a naïve strategy and can lead to several mispredictions (and ensuing penalties in the pipeline), especially when the program has loops. In dynamic branch prediction, the hardware tries to guess the branch outcome with a branch predictor that uses branching history and other available information.

With increasing pipeline depths and complexity of programs, the misprediction penalty increases. Moreover, with super scalar processors the number of branch instructions in a program also tremendously increase. Hence, there is need to improve upon existing state-of-the-art predictors. One of the most common trends are AI based predictors that look at past prediction data and predict the outcome based on patterns. [1] Predictions based on perceptrons, convolutional neural networks, belief networks and reinforcement learning are some of the techniques that are yielding promising results.

In our project, we first simulate the working of G-share and tournament predictors with the assumption that there is no prior history before the program runs. We compare and analyze the results to see how our predictor works for the given traces (we use commands and options provided to us in the starter code that tests the simulation). Figures 1 and 2 show the circuit structures we simulate. Thereafter, we attempt to implement a predictor that performs better than both the former. Figure 3 shows the simulation structure of our custom predictor. We benchmark and report the results.

II. Implementation

A. G-Share

We implemented the the G-share branch predictor [2] that predicts the outcome of the branch using the branch address and the global history of branch `TAKEN` (represented by 1) or `NOTTAKEN` (represented by 0). Here, global history is stored in n-bit registers, where n is the number of history bits we consider for determination of the branch direction. The branch history table is indexed by the program counter (`PC`).

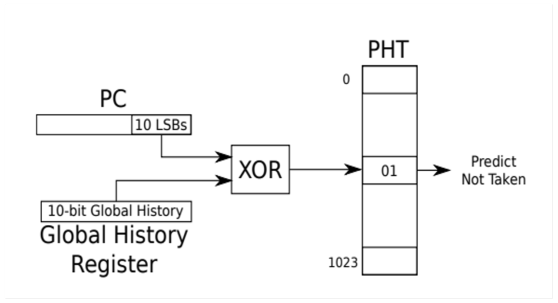

Fig. 1. G-share Predictor

Figure 1 shows the circuit diagram of a G-share predictor where the number of history bits $n = 10$. An index into a pattern history table (PHT) of 2-bit counters is made using both global branch history and the address of a branch instruction. Each of these 2-bit counters represent a state of the branch direction as given in Table I. The states in the PHT are updated based on the actual outcome of the branch.

From an implementation perspective, we perform the following:

1) Initialization: To initialize our predictor we set the global history to 0 as we assume there is no prior history before running our code and initialize a global history register of size 2^n where n is the number of history bits we get from the terminal from the user. Now, we set each of the values in this pointer array to `WN`.

2) Prediction: As the circuit in figure 1 suggests, we perform the `XOR` of the `PC` value and the current global history to obtain the index we need to access from the PHT. We predict based on the value in the PHT according to Table I.

3) Training the predictor: In order to tune the predictor we need to adjust the values of the PHT based on the correctness of the branch outcome. To do this, we compare the values we get from (2) above and the actual outcome of the branch. We adjust the values of the global history table by incrementing it if we predict correctly and decrementing it if we predict incorrectly. This is corresponding to the values in Table I. For example, if we predict `WT` and outcome is `TAKEN`, we update the corresponding global history counter to SN. However, if we predict WT and outcome is NOTTAKEN, we update the corresponding global history counter to WN.

Counter	Token	Meaning	Prediction
00	SN	Strong not-taken	NOTTAKEN
01	WN	Weakly not-taken	NOTTAKEN
10	WT	Weakly taken	TAKEN
11	ST	Strongly taken	TAKEN

TABLE I
COUNTER STATES

Advantages: In G-share, a global branch history table does not maintain a separate history record for each conditional jump. Instead, a shared history of *all* conditional jumps is maintained. A shared history has the advantage of including any correlation between several conditional jumps in the prediction process. This makes determining patterns between branches in a program much easier than earlier implementations of branch predictors.

Disadvantages: As mentioned before, a shared history is maintained for all conditional branches. The disadvantage is that the history is polluted by irrelevant data if many conditional jumps are uncorrelated, and if there are numerous other branches between, the history buffer might not contain any bits from the same branch that is relevant for accurate prediction.

Evolution: Initially, *bimodal branch prediction* was the most efficient form of predicting which simply predicted the branch based on the direction the branch took in the last few instructions. This was followed by mimicking patterns by considering the history of each branch independently [2] for a more accurate result (known as *local branch predictor).* This was followed by the *gselect branch predictor* that uses both the branch address and the global history. G-share [2] is very similar to gselect, differing only in the way that it computes the XOR of the branch address and the global history.

B. Tournament

We developed a tournament predictor that selects the predictor that produces the most accurate results for a given branch instruction out of a global predictor and a local predictor. The outcome states are represented as TAKEN (represented by 1) or NOTTAKEN (represented by 0). The states of the global predictor and the local predictor is same as that of G-share as defined in Table I. However we also define *additional* states for the selector to specify the predictor (which of the two predictors, global or local, to choose) as per Table II.

Figure 2 shows the circuit diagram from the tournament predictor with the number of history bits $n = 10$. An index into the PHT is made using the branch address in the PC. It contains 2^n entries of 2-bit counters that each represent the predictor to choose with respect to the states in Table II. The selected predictor provides the final prediction for branch TAKEN or NOTTAKEN. As shown, the global predictor is the G-share predictor which is implemented as described in Section II-A. The local predictor is defined as a pointer of predictors that are specific to a branch instruction. Each of these local predictors contains a *local prediction table* which is a 2-bit counter for the particular branch prediction and has states as in Table I.

Counter	Token	Meaning	Prediction
00	TSN	Strong not-taken	Chooses the local predictor
01	TWN	Weakly not-taken	Weakly chooses the local predictor
10	TWT	Weakly taken	Weakly chooses the global predictor
11	TST	Strongly taken	Chooses the global predictor

TABLE II
TOURNAMENT: SELECTOR COUNTER STATES

From an implementation perspective, we perform the following:

1) Initialization: To initialize our predictor, we set the global history to 0 and define a 2^m bit local history table for the local predictor where m is the number of local history bits we consider. Similarly, we also define a pattern history table or the selector with 2^n entries that specifies which of the two predictors to use. We also define a global and local prediction table that stores the predicted values.

Each element of the local history table is initialized to 0 indicating there is no history prior to program execution. Both global prediction table and the local prediction table are initialized to WN (similar to G-share). The selector is initialized to weakly select the global predictor by default (TWT).

2) Prediction: As the circuit in Figure 2 shows, for the local predictor we choose the value given by the local history table indexed by PC and the global predictor obtains the results as described in G-share. Now, we determine the final prediction by indexing into the PHT using the conventions defined in Table II.

3) Training the predictor: In order to train the predictor, we need to adjust the values of the selector to choose the better option of the two predictors. At the same time, we also need to train the individual predictors to predict better. We train the global predictor as described in G-share. As the local predictor also follows the same counter states as G-share, we compare the results in the local prediction table with the actual outcome, and then increment or decrement the counter states accordingly. For example, if we predict WT and outcome is TAKEN, we update the local prediction counter to ST. However, if we predict WT and the outcome is `NOTTAKEN` we update the corresponding local counter to `WN`.

To train the selector, we compare the outcome to the predictions made by both the predictors and we update the selector states accordingly. For example, if the outcome is `TAKEN` and the predictions of local and global predictors are `WT` and `WN`; and the current selector is at state `TWT` (select global predictor weakly) we decrement it to state `TWN` as the local predictor has a more accurate prediction.

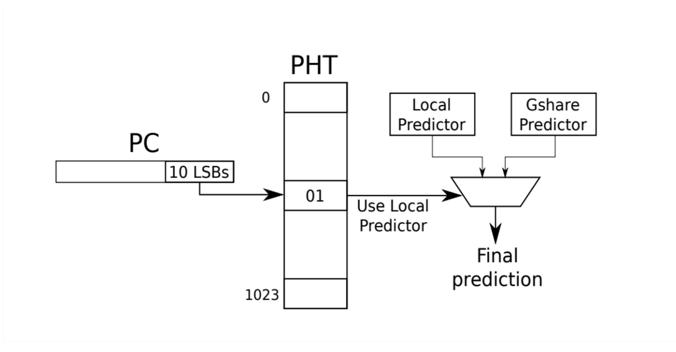

Fig. 2. Tournament Predictor

Advantages: The optimal number of history bits to consider for a branch instruction may be dependent on its inherent nature. Some branches work well while considering small length of histories while some work well over long histories. Since we have two predictors in Tournament, we can choose the predictor that supplements a particular branch's nature.

Disadvantages: Tournament predictor contains two extra (local prediction and selection) predicting structures relative to G-share. Therefore, larger the number of history bits we consider, the amount of hardware requirements increases significantly. With increase in complexity of the prediction workflow, tournament might predict slower for larger values of history bits (it can take more cycles to predict relative to G-share).

Evolution: In addition to the advantages presented above, and the initial G-share implementation with experiments it was found that that explicitly selecting between the *local branch predictor* (that existed and was used extensively already) and G-share [2] provided much better results as this enabled us to configure the predictor by considering the individual branch natures independently. This led to the implementation of the tournament [2] predictor.

C. Custom Predictor - TAGE

We implemented the TAGE [7] (TAgged GEometric length predictor) portion of the 64 KBits ISL-TAGE predictor [5]. It correlates long branch histories of increasing geometric lengths based on the O-GEHL predictor [4] with (partially) tagged components as the PPM-like predictor [3].

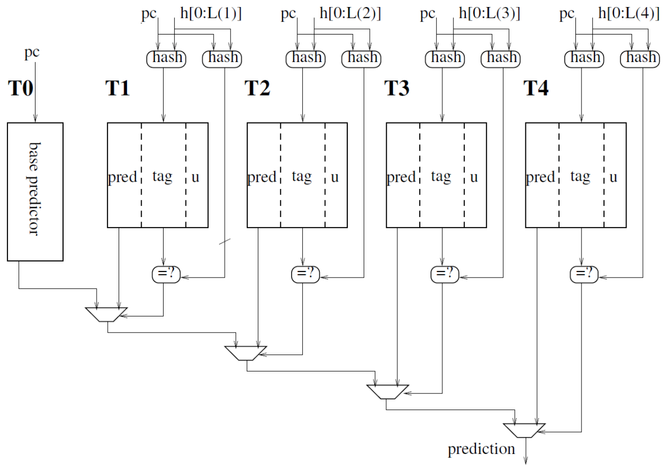

Fig. 3. A 5-component TAGE predictor (image taken from [7])

1) Description of TAGE: Figure 3 shows a TAGE predictor with a base predictor $T0$ and 4 (partially) tagged predictors $Ti, 1 \leq i \leq 4$. The base predictor $T0$ provides a basic prediction and consists of a 2-bit bimodal counter indexed by the program counter pc. The tagged predictors Ti are indexed using different history lengths that form a geometric series. The (global) history length $L(i)$ for a tagged predictor Ti is given by:

$$L(i) = (int)(\alpha^{i-1} \times L(1) + 0.5) \tag{1}$$

The tagged predictors consist of entries with the following components:

1) *ctr* - signed 3-bit counter which provides the prediction
2) *tag* - a (partial) tag
3) *u* - unsigned 1-bit useful counter

Some definitions: The *provider component* is defined as the predictor component with longest history that provides a prediction. In the case that the provider component was not producing a useful prediction (with useful bit $u = 0$), then the alternate prediction is called *altpred*. If none of the tagged predictors provide a useful prediction, then *altpred* (from the base predictor) is the default prediction.

2) Prediction computation: At prediction time, all the predictor components (base and tagged) are accessed simultaneously. The base predictor gives a default prediction, and the tagged components give a prediction when there is a tag match. The prediction from the provider component (longest history) is considered when multiple tag matches occur. If no tag matches occur in any tagged predictor, the base predictor's (default) prediction is used. However, if the prediction provided by the provider component is weak, the confidence in prediction is low ($< 60\%$) [6]. In such a situation, the default prediction is used instead. The use of the default prediction *altpred* is counted using a 4-bit signed counter *USE_ALT_ON_NA* (i.e., it is incremented when *altpred* is used and decremented when the prediction from a tagged component is used).

Now, the prediction is computed using the following steps:

1) Find the matching tagged predictor component with longest history.
2) On finding a tag match, if either the prediction is not weak or *USE_ALT_ON_NA* is negative then the sign of the prediction counter is used. If neither holds, then the sign of *altpred* is used.

3) Predictor Update/Training: The prediction counter of the used prediction (from either the base or one of the tagged predictors) is updated based on the actual outcome. If the prediction comes from a tagged predictor, is correct and *altpred* is wrong, then the useful bit u is set to $u = 1$. As usual, the actual branch outcome is appended to the global history.

In case of a misprediction, if the provider component Ti does not use the longest history (i.e., $i < M$, the number of tagged predictors), an entry of a tagged component Tk using a longer history than Ti (i.e., $i < k < M$) is allocated. If there exists some k such that the u bit of Tk is 0, then the indexed entry is allocated and initialized with the prediction counter set to the weak correct prediction value. The useful bit stays $u = 0$.

The above scenario highlighted is a successful entry allocation in a tagged predictor. When a prediction is useful, the useful bit is set to $u = 1$. To avoid this staying forever set, the u bits are reset when more failures than successes occur in entry allocation. This is monitored using an 8-bit counter *TICK* which is decremented on successful allocation, and incremented on failure in allocation. All u bits of the tagged predictors are reset when *TICK* saturates. This usually occurs when every other entry in the used portion of the predictor has its useful bit set to 1 [5].

4) Predictor Indexing/Matching: The base predictor $T0$ is indexed using program counter pc. The sign of the 2-bit counter gives the base prediction. The tagged predictor has a more complex index calculation process, as defined in [3]. For a given tagged predictor Ti and its corresponding global history length $L(i)$, the global history is "folded" using bit-wise XOR of groups of consecutive history bits when $L(i)$ exceeds the number of index bits. For example, if we have 10 index bits and we consider 20 bits of global history h, then the index is calculated as $pc[9:0] \oplus h[9:0] \oplus h[19:10]$ where $\oplus$ denotes the bit-wise XOR operation.

After calculating the index for a tagged predictor, we have to match the entry's tag value. The tag to be matched is also computed using global history folding, as described in [3]. For each tagged predictor we use two circular shift registers (CSR), $CSR1$ and $CSR2$ (as a single CSR is sensitive to periodic patterns in global history). Each CSR is populated by folding the global history using bit-wise XOR operations, and the final tag is computed as $pc \oplus CSR1 \oplus (CSR2 \ll 1)$ where $\ll$ denotes bit-wise left shifting. With an 8-bit tag, $CSR1$ and $CSR2$ have sizes of 8 and 7 bits respectively and the tag is calculated as $pc[7:0] \oplus CSR1 \oplus (CSR2 \ll 1)$. This computed tag is matched with the tag in the indexed entry, and on a match, the counter is used to determine the prediction provided by the tagged predictor Ti.

5) Predictor Initialization: Assume no global history (initialized to 0, which means assuming that branches are `NOTTAKEN` in global history) and that all predictors (base and tagged) are empty i.e., all entries are 0. The two counters *USE_ALT_ON_NA* and *TICK* are set to 0 as well.

6) Implementation Details: We use up to 64 bits of global history h. The base predictor $T0$ has a 12-bit index, and hence we use the last 12 bits of the program counter i.e., $pc[11:0]$. Each tagged predictor has a 10-bit index, and we use the pc and fold the global history as described in Section II-C4. Its entries contain an 8-bit tag, which is computed as described in Section II-C4 again.

After performing experiments with different global history lengths, using $M = 4$ tagged predictors with global history lengths of {1, 4, 12, 43} bits were good enough to beat (with lower misprediction rates) both G-share and Tournament implementations. The history lengths are in a geometric series that satisfy $\alpha = 3.5$ according to Equation 1.

7) Predictor Size: Based on the above implementation details, we have:

- Base predictor $T0$ has 4K entries of size 2 bits each, giving a total of 8K bits.
- Each tagged predictor Ti has 1K entries, each of size 8+3+1 = 12 bits, giving a total of 12K bits for each Ti and a total of 48K Bits for all tagged predictors.
- Each tagged predictor also has 2 CSRs of 15 bits, giving a total of 60 bits for all CSRs.
- We also have 2 counters *USE_ALT_ON_NA* and *TICK* of 4 and 8 bits each.
- Finally, there are 64 bits of global history h.
- Total size of predictor = 8K + 48K + 60 + 4 + 8 + 64 **= 56K + 136 bits**

Advantages: Hashing of pc with global history bits h for indexing and tag matching allows for correlating branch outcomes by considering very long histories. The use of an alternate prediction instead of the prediction from the provider component when the counter is weak allows for high confidence predictions to be made. Resetting of the useful bit u in tagged components allows for a "refresh" of table entries to ensure the branch predictor doesn't "saturate" and refuse to learn new (or changing) branching patterns in programs. Finally, the use of multiple predictors with varying history lengths in a geometric series allows for a choice of the most appropriate (and potentially accurate) prediction of branch direction to take.

Disadvantages: TAGE is often bulky with so many components, that it requires complex hardware to implement and this added complexity can make prediction computation very slow. TAGE does not store local history like tournament does, and sometimes fails to predict loops with constant number of iterations [5]. TAGE also fails to predict statistically biased branches (with very small bias towards a direction, but not correlated to history) [5].

Evolution: TAGE is derived from Michaud's tagged PPM-like predictor [3]. At the time, the O-GEHL predictor [4] was the most storage-effective reasonably implementable conditional branch predictor presented. In addition to using an adder tree as the final computation function, its main characteristic is the use of a geometric series as the list of history lengths. This allows the predictor to exploit very long history lengths and also correlate recent branch outcomes. The authors of [7] use geometric history length series in PPM-like predictors and a new and efficient predictor update algorithm to implement a predictor that outperforms O-GEHL at equal storage budgets and equivalent predictor complexity.

III. Observation

A. Comparing G-share and Tournament Predictor

Table III shows the comparison between G-share and Tournament for all six traces, using 9 bits of global history.

Trace	G-share	Tournament
int_1	26.114	12.622
int_2	0.968	0.426
fp_1	1.466	0.991
fp_2	10.733	3.246
mm_1	16.646	2.581
mm_2	15.157	8.483

TABLE III
Comparing misprediction rates of G-share and Tournament

As observed in Table III, for the same number of history bits n, the Tournament predictor performs much better than G-share for all six traces.

B. Comparing G-share, Tournament and Custom Predictor

The results in Table IV compare the misprediction rates of G-share (with 13 bits of global history), Tournament (with 9 bits of global history, 10 bits of local history and 10 PC bits) and our custom predictor (TAGE) that uses $\{1, 4, 12, 43\}$ bits of global history in its tagged predictor components. The plot in Figure 4 also visually highlights that our custom TAGE predictor outperforms (with lower misprediction rate) both G-share and Tournament implementations for all six traces.

Trace	Gshare	Tournament	Custom
int_1	13.839	12.622	8.911
int_2	0.420	0.426	0.272
fp_1	0.825	0.991	0.812
fp_2	1.678	3.246	1.53
mm_1	6.696	2.581	1.613
mm_2	10.138	8.483	8.180

TABLE IV
Misprediction rates (in %) of all 3 Branch Predictors

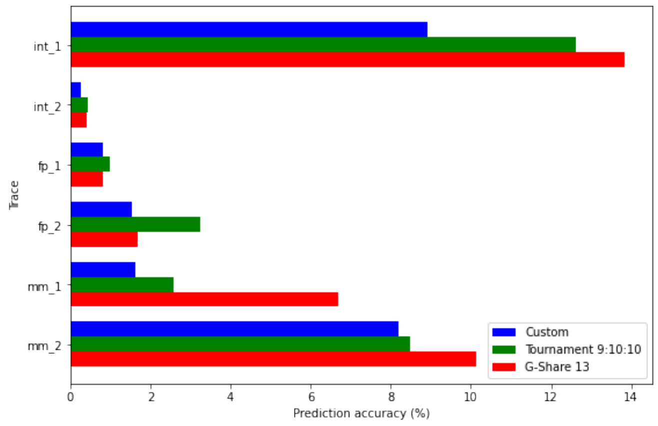

Fig. 4. Plot of misprediction rates of all 3 predictors

We expect this behavior as our custom predictor considers longer global history of up to 43 bits, which is more than the 13 bits that G-share considers and the 9 bits that Tournament accounts for. Also, local branch patterns are embedded into the counters in both base and tagged predictors in TAGE. The unique hashing method to calculate both index and tags may also capture branching patterns and history better than G-share and Tournament. Finally, low confidence predictions (when the counter is weak) are not made, and instead alternate predictions (from tagged predictors with lesser history bits or the base predictor) are made which can be a reason for lower misprediction rate of TAGE.

IV. Results and Conclusion

It can be seen that our custom implementation of the TAGE predictor using $\{1, 4, 12, 43\}$ bits of global history and a few optimizations outperforms both G-share (using 13 bits of global history) and Tournament (using 9 bits of global history and 10 bits of local history) predictors for the six chosen traces. From this, it can be concluded that the use of varying lengths of global histories and hashing it with the pc can produce a good branch predictor in TAGE.

In summary, we implemented three branch predictors in this project: G-share, Tournament and TAGE, and compared their performances on a predefined set of branch traces.

G-share and TAGE are global history predictors, while Tournament is an ensemble that uses both global and local history predictors and chooses the best prediction from both. TAGE outperformed G-share and Tournament, given the constraints of the problem. This report compiles the background, implementation and results of each predictor in detail.

Individual Contributions

The starter code was obtained from the GitHub repository provided on the course website. We divided the project into two parts and each worked on one.

- Subhashree Navaneethan implemented the code for initialization, prediction and training for simulating G-share the tournament predictor. The implementation and workflow are described in Sections II-A and II-B, and the results are reported in Section III-A.

- Suhas Pai implemented the code for initialization, prediction and training for simulating the custom TAGE predictor. The implementation and workflow are described in Section II-C, and the results are reported in Section III-B.

Each person tested their implementation for different number of histories for correctness and efficiency and reported the results in the respective sections of the report.

Acknowledgements

This document is derived from previous conferences, in particular ISCA 2019 and MICRO 2019. We also used resources at the website *https://ssine.ink/en/posts/tage-predictor/* to understand some implementation details of TAGE.

References

[1] R. Joseph, "A survey of deep learning techniques for dynamic branch prediction," *CoRR*, vol. abs/2112.14911, 2021.
[2] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.
[3] P. Michaud, "A ppm-like, tag-based branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 7, p. 10, 2005.
[4] A. Seznec, "The o-gehl branch predictor," *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.
[5] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.
[6] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 117–127.
[7] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, p. 23, 2006.