Multi-Level Branch Predictor

Alex Cho, Dylan Bing-Manners, Abay Kulamkadyr

December 2022

Abstract

This project attempts to implement a multi-level-branch-predictor proposed by Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993). We benchmark this multi-level branch predictor against a basic single-level bimodal branch predictor and measure their performance differences.

1 Introduction

For our project, we have attempted to implement multi-level branch predictors proposed in the paper Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993), on the traced-based simulator, ChampSim. ChampSim is an open-source CPU Simulator that tracks the performance of the implemented architecture. By default, ChampSim only allows for one width worth of instructions to be fetched in a single cycle thus, limiting possible branch prediction to a single-level. We will discuss the ways in which we changed the simulator source code to work with multi-level branch predictors, how the branch predictors work in theory and how they performed in practice. In the end, we will also show the performance compared to the single-level bimodal branch predictor.

2 Theory

The multi-level predictors described in the paper by Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993) base their implementation on a two-level adaptive branch prediction (Figure 1), and expanded it to a multi-level version.

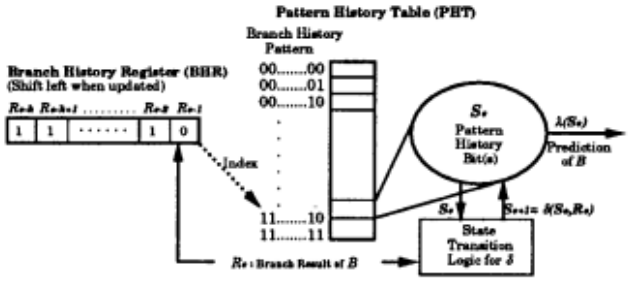

Figure 1: Basic structure of Two-level Adaptive Branch Prediction.

Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993) Figure 1.

MGAg and MGAs has the CPU do multiple actions in a single cycle. First, the CPU would fetch the next set of instructions from the I-Cache, and send them to the instruction buffer. The CPU would then find the first branch instruction within the I-buffer. Then, by referring to the branch address cache, the CPU would find the set of instructions pointed to by the branch depending on whether or not the branch was predicted to be taken or not. Within that set of instructions pointed to by the branch, the CPU would again find the first branch and fetch another block of instructions. This continues until the maximum levels of prediction has been reached, or, a block with no branches is encountered. Figure 2 shows the fetched blocks with the emboldened line showing the path of predicted branches.

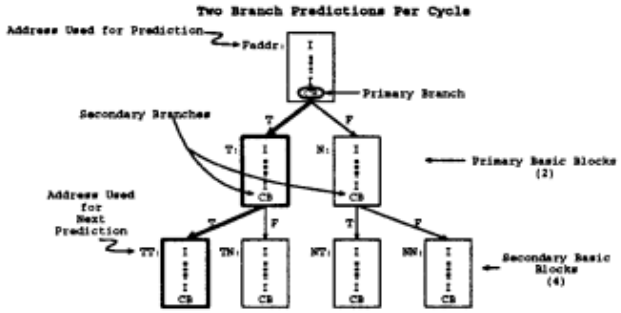

Figure 2: Identification of the primary and secondary branches, and the primary and secondary basic blocks.

Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993) Figure 2.

How the predictions are made is dependant on the multi-level branch predictor. MGAg and MGAs both use a global history register (GHR) which index into the pattern history table (PHT) which holds 2-bit saturating counters. How the pattern history table is indexed varies depending on the level of prediction.

The secondary, tertiary, and beyond predictions are indexed by taking one bit off of the GHR for every level, and use the result to index into the PHT. To get the prediction of all branches in the kth level, the predictor takes the last bits taken off of the GHR and uses the permutations of 2^k bit strings, to index into neighboring 2-bit saturating counters (Figure 3).

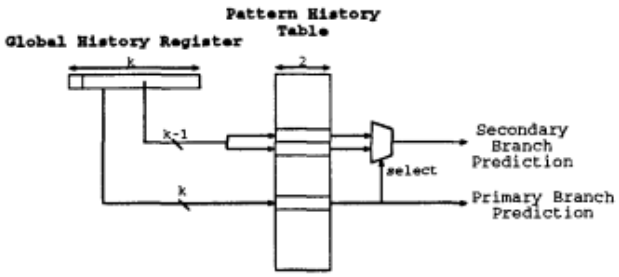

Figure 3: Algorithm to make 2 branch predictions from a single branch history register.

Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt (1993) Figure 3.

3 Implementation

The first requirement to implement multi-level prediction was to modify ChampSim to fetch multiple instruction blocks from I-Cache per cycle. This was done by modifying the `read_from_trace()` function. By default, ChampSim iterates through the trace and fetches `FETCH_WIDTH` instructions per cycle. Our modification performs this iteration `LEVELS_TO_PREDICT` times to simulate multiple instruction block fetches in a single cycle. If ChampSim encounters a branch instruction, the block is considered fetched and a new block is started. If `LEVELS_TO_PREDICT` basic blocks have been read or a block doesn't contain a branch instruction at all, ChampSim finishes fetching instruction for that cycle. Whenever a prediction needs to be made, `predict_branch()` is called and passed all the previous predictions from the current cycle in the `last_branch_predictions[]` array, as well as `blocks_predicted`, which is the size of the array minus one. Additionally, to maintain compatibility with the bimodal branch predictor, the current program counter is passed in as the final argument.
After ChampSim is done fetching instructions, `last_branch_result()` is called and passed `last_branch_results[]` and its size, `blocks_predicted`, which contains the actual results of each branch in the current cycle. As with `predict_branch()`, the current program counter is passed as the final argument. In case of a misprediction, an array of file pointers to the trace file are stored to `snapshot_file[]`. After `last_branch_result()` is finished, the file pointer is adjusted to one instruction after where the branch misprediction occured.

The second requirement was to implement *Multiple Branch Global Two-Level Adaptive Branch Prediction using a Global Pattern History Table* (MGAg). We store the previously predicted branches along with the correct branch results in an array (up to `LEVELS_TO_PREDICT`). At the end of a cycle, the algorithm iterates over the predicted branches and passes the correct branch results to the branch predictor to update its structures. The branch predictor corrects its pattern history table and its global history register. If the algorithm detects a misprediction, the file pointer for the trace is updated to where the misprediction occurred and cycle penalties are added.

The `predict_branch()` function was modified to take in the information about previously predicted branches and the `branch_predicted` parameter represents the current predicting level. The algorithm uses the full bits of the global history register (BHR) to index the pattern history table to obtain the primary prediction.
If the current predicting level is 0, the algorithm returns the primary prediction. Otherwise, the branch predictor uses k-1 bits of the global history register (BHR) and indexes the pattern history table with the following bit patterns:

1) $\text{BHR}_{k-1}0$ if the primary branch prediction was not taken.

2) $\text{BHR}_{k-1}1$ if the primary prediction was taken.

If the current predicting level is 1, the algorithm returns the secondary prediction. Otherwise, the branch predictor uses k-2 of the global history register(BHR) and indexes the pattern history table with the following bit patterns:

1) $\text{BHR}_{k-2}00$ if the primary prediction was not taken and the secondary prediction was not taken.

2) $\text{BHR}_{k-2}01$ if the primary prediction was not taken and the secondary prediction was taken.

3) $\text{BHR}_{k-2}10$ if the primary prediction was taken and the secondary prediction was not taken.

4) $\text{BHR}_{k-2}11$ if the primary prediction was taken and the secondary prediction was taken.

Moreover, the branch predictor saves the states of the global history register when the predictions were made (up to `LEVELS_TO_PREDICT`). This information is used to correct the pattern history table when `last_branch_result(uint8_t actual_branches_taken[],` is called at the end of cycle.

The third requirement was to implement another variant of multiple branch prediction, *Multiple Branch Global Two-Level Adaptive Branch Prediction us-*

ing Per-set Pattern History Tables (MGAs). The difference from MGAg is that MGAs uses multiple pattern history tables (PHT). Current fetch address and current predicting level are passed as arguments to `predict_branch()`. If the level of prediction is the primary branch, the branch predictor saves its fetch address to choose a particular PHT for all of the subsequent predictions using a hash function. Once a PHT is determined, the algorithm behaves exactly like MGAg: the branch predictor uses full bits of the global history register (BHR) to index the table to make a primary prediction, k-1 bits to make secondary prediction, and k-2 to make the tertiary prediction.
As suggested in the paper, we kept the total number of bits used by the GHR and PTHs at a constant (16384 bits) to fix the hardware cost.
The bullet point below represents the configurations we used in our experiments.

- GHR size: 14 bits. Number of PHTs: 1
- GHR size: 13 bits. Number of PHTs: 2
- GHR size: 12 bits. Number of PHTs: 4
- GHR size: 11 bits. Number of PHTs: 8
- GHR size: 10 bits. Number of PHTs: 16
- GHR size: 9 bits. Number of PHTs: 32
- GHR size: 8 bits. Number of PHTs: 64
- GHR size: 7 bits. Number of PHTs: 128
- GHR size: 6 bits. Number of PHTs: 256
- GHR size: 5 bits. Number of PHTs: 512

4 Benchmarking

As expected, the accuracy and performance results from our implementation have similar trends to those found by Tse-Yu et al. As shown in figure 4, the IPC generally increases as the number of levels of prediction increases when the global history size is high. This makes sense because, as shown in figure 5, there is a strong positive correlation between global history size and accuracy. This higher accuracy leads to a high fetch bandwidth and thus, a high IPC. For both IPC and accuracy we do see some results that don't perfectly line up with with the upward trend. This is most likely due to the variations in the benchmarks– each having different proportions of branch type or simply a differing number of branches overall.
Unlike the results in the paper, in our experiments, accuracies tend to increase with the number of pattern history tables, while their accuracies would drop as they increase the number of PHTs. When we increase the number of PHTs, we are decreasing the possibility of two branches mapping to the same table,

and in extreme cases, each branch will receive its own PHT. At the same time, there will be significantly fewer entries in the tables since we fixed the hardware cost, and the history of branches in the GHR is much shorter. Therefore, there will be less chance to capture the relationships between primary, secondary, and tertiary branches within a single table because there will not be enough entries to reflect them (the relationships). This reasoning suggests the branches in our experiments benefit more from keeping their individual histories and lack correlating branches.

In figure 7 we can see the comparison of accuracy between the multi-level MGAg, MGAs and the single-level bimodal predictor. The accuracy for each predictor is quite similar, with MGAg being on average superior by a small amount. In figure 8 we see the comparison of IPCs for both multi-level predictors and the bimodal predictor. MGAg and bimodal perform similarly, whereas MGAs is far superior to both. Both figure 7 and 8 show that accuracy and performance of the multi-level predictors is comparable to that of the bimodal predictor when their global history size is 8 bits and have 2 levels of prediction and the bimodal has a 16 bit table. Given the results shown in figure 4 and 5, it is reasonable to conclude that both multi-level predictors would vastly out-perform the single-level predictor if their global history size and levels of prediction were increased.

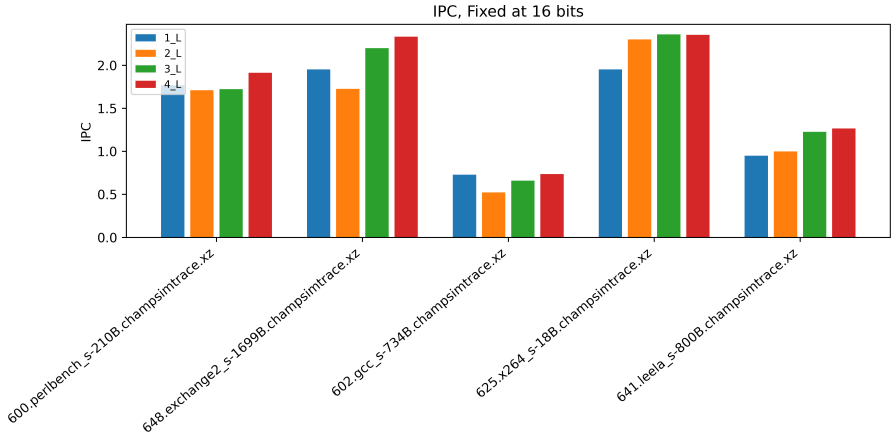

Figure 4: IPC of MGAg with a global history size of 16 bits and varying levels of prediction.

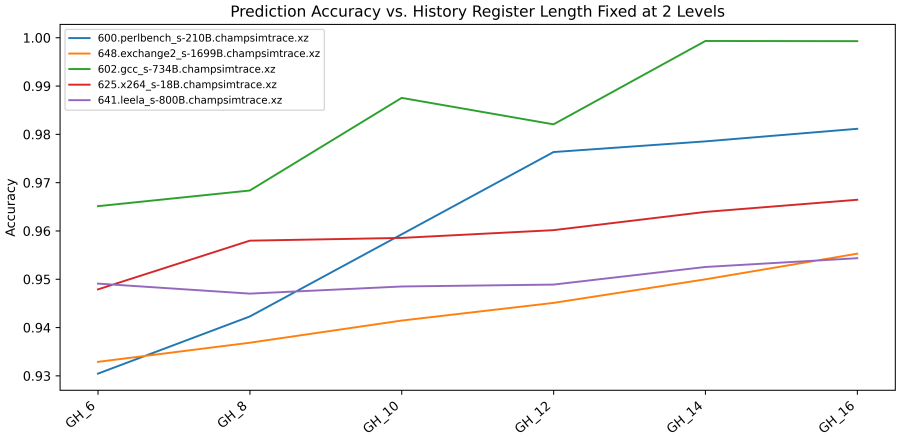

Figure 5: Accuracy of MGAg with 2 levels of prediction and varying global history size.

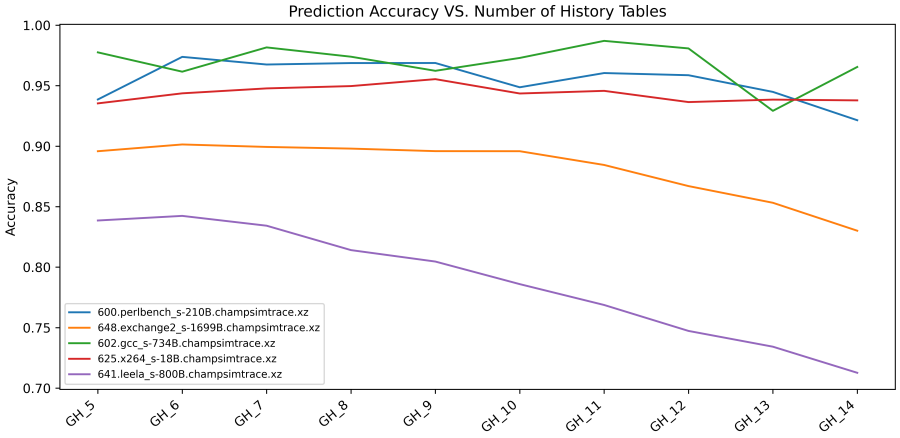

Figure 6: Accuracy of MGAs with 2 levels of prediction, varying global history size and pattern history table size with a combined total size of 14 bits.

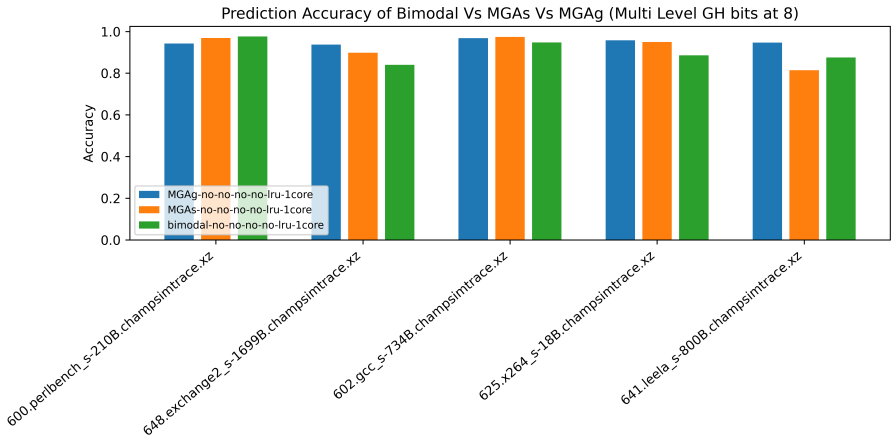

Figure 7: Comparison of accuracy between bimodal with a 16 bit table size and MGAg/MGAs with 2 levels of prediction and a global history size of 8 bits.

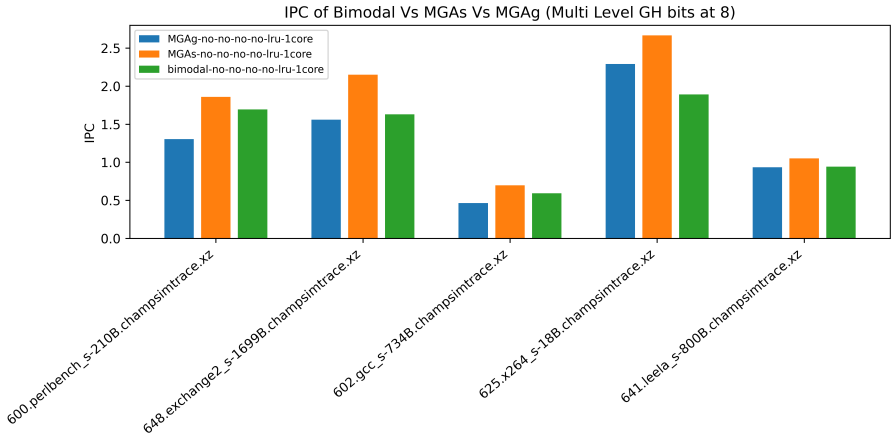

Figure 8: Comparison of IPC between bimodal with a 16 bit table size and MGAg/MGAs with 2 levels of prediction and a global history size of 8 bits.

5 Conclusion

Multi-level predictors are great performance boosters for IPC assuming high accuracy. Although the performance may be good, there are still hardware costs to be concerned about. Additionally, increasing instruction fetch width may increase pipeline sizes which may not scale well. Adding buffers to handle misprediction flushes may impact performance depending on the accuracy of the multi-level branch predictor. Increasing the levels of predictions using both MGAg and MGAs multi-branch predictors proved to significantly increase the IPCs of sequential programs and achieved similar (and for some benchmarks,

better) results when compared with similar accuracies to the bimodal predictor.

6 References

[1] Yeh, Tse-Yu et al. *Increasing the instruction fetch rate via multiple branch prediction and a branch address cache.* ICS '93 (1993).

7 Appendix

Contributions

Alex Cho - Contributed to ChampSim modification, Plot generation, Theory section of report, Benchmark, Key findings and Conclusion of Slides.

Dylan Bing-Manners - In the code, contributed to the ChampSim modification and MGAg implementation. In the report, contributed to the implementation and benchmarking sections. In the presentation, contributed to the theory and modifying ChampSim sections.

Abay Kulamkadyr - Contributed to Champsim modification, MGAg, and MGAs implementaions. In the report, contributed to the implementation and benchmarking sections. In the presentation, contributed to the theory of the MGAs implementation and described implementation challenges.

Code Access

Please find our code repository here (https://github.com/CMPT-450-GROUP/Multi-Level-Branch-Predictor)
A README is included explaining how to compile and run our code.