Branch Prediction Exploration Project

Connor Aksama

CSE 470

May 2023

University of Washington

0 Abstract

Branch prediction is a long-established topic with an ever-growing importance that has the potential to greatly improve processor performance with more advanced techniques. This project takes an intermediate view into the design, implementation, testing, and evaluation of branch prediction in a 5-stage, pipelined RISC-V CPU. The main study of this project aims to measure the misprediction rate of several branch prediction techniques across different program types of varying time complexity at different magnitudes of input sizes. Experimental results are then analyzed to gauge differences in accuracy between prediction strategies based on these parameters.

1 Background

Developing good branch prediction techniques has been and continues to be an important aspect in improving the performance of processors. As the complexity of superscalar processors and the number of pipeline stages in processors grows, so does the penalty associated with the misprediction branches [1]. Additionally, with the widening prevalence and importance of ASICs for high-performance/throughput applications such as machine learning [2], pushing a processor's misprediction rate to 0 becomes ever more important.

2 Project Goals

Given the time constraints of the course and current resources and abilities, but also given the ongoing potential for significant improvements [3] in branch prediction, the goals of this project are centered around gaining intermediate experience in the implementation and testing of branch prediction in a CPU.

The first goal of this project is to implement several different branch prediction strategies for a previously implemented RISC-V CPU designed in CSE 469. This will involve creating a generalized branch prediction module that can switch between different prediction strategies.

The second goal is to develop an experimentation process that can take many diverse programs and output data on the misprediction rate for runs of each of those programs. This will involve implementing a program loader for the CPU, testbenches to run programs under different conditions and collect data, as well as finding test program sources.

Finally, the third goal of the project is to use the collected data from the experimentation to develop more of an understanding of how the performance of a branch prediction strategy depends on the structure of the program being run and the data input into the program.

3 Design and Implementation

3.1 Design Rationale

Three main considerations were made to determine the final design for the CPU's branch prediction unit: the predictor evaluation technique, the simulation environment, and test program size [efficiency -> earlier better].

3.1.1 Evaluation Technique

The goal of this study is to examine the performance of different branch prediction strategies in isolation, i.e. in isolation of other factors of the architecture of the rest of the CPU design. As such, the BPU design and experimental design is focused around measuring the misprediction rate of the run of a single program. Variables such as misprediction penalty, overall instruction throughput, or impacts on pipelining are left for future studies.

3.1.2 Simulation Environment

To prioritize testing efficiency for large programs, experimentation was chosen to take place exclusively in a simulated environment. At the cost of evaluation speed, experimenting in simulation

enabled the BPU design to be agnostic of physical considerations like clock synchronization or gate delay between pipeline stages.

3.2 Implementation

3.2.1 BPU Design

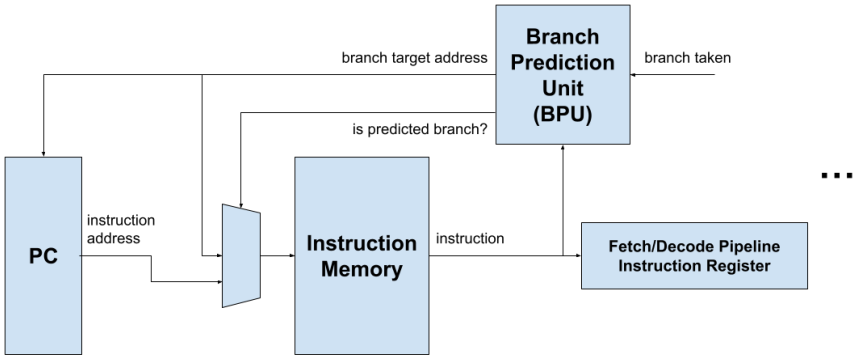

Figure 3.2.1. Block Diagram of BPU Design in Fetch Stage.

The processor that this project's BPU was built for utilized 5 pipelined stages corresponding to "Fetch", "Decode", "Execute", "Memory" and "Writeback". Each of these pipeline stages takes two clock cycles to complete. Given the rationale for the design, the BPU was placed in the first pipeline stage ("Fetch") of the processor directly alongside the instruction memory.

At the start first cycle of this stage, the current instruction is present on the output bus from the instruction memory and is routed into the BPU. During this cycle, the BPU produces a prediction for the current instruction, as well as the target branch address if the branch was predicted to be taken, which becomes available on the start of the second cycle. The current branch instruction is also marked as "predicted", which is used to correct mispredictions at the branch control unit (BCU) in the "Memory" stage.

This new target address is routed into both the program counter, which saves this new address; and the instruction memory, which fetches the instruction at the target address.

Once the branch instruction reaches the "Memory" stage, the BCU uses the results from the "Execute" stage to determine whether the branch should have been taken. If the result agrees with the original prediction, the pipeline proceeds as normal. If a misprediction is determined, the earlier pipeline stages are flushed, and a corrected program address is fed from the BCU to the PC using either the address of the branch instruction, or the target address of the instruction.

3.3 Prediction Strategies

The following is a description of the branch prediction strategies implemented for this project (see [4]):

3.3.1 Static Predictors

Static prediction strategies consider only the data encoded in the current branch instruction.

Always/Never Taken

These prediction strategies will make the same prediction for every branch instruction encountered, either always predicting taken, or always predicting not taken.

Only Forward/Only Backward

These prediction strategies examine the offset encoded in the current instruction. In *Only Forward* prediction, taken is predicted if and only if the offset is positive. In *Only Backward* prediction, taken is predicted if and only if the offset is negative.

3.3.2 Dynamic Predictors

Dynamic prediction strategies maintain some kind of "history" of previous branch results that influence the prediction taken at the current branch instruction.

One/Two-Bit Saturating Counter

In these strategies, the BPU maintains a saturating counter of N bits. When the BCU evaluates a branch instruction as taken (regardless of whether the original prediction was correct or not), this increments the counter by 1. When a branch instruction is not taken, the counter is decremented by 1. The prediction made by the BPU is the most significant bit of the counter (1 corresponds to predicting taken, 0 predicts not taken).

Correlated Predictor

The correlated branch predictor consists of two parts. The first is a global history of the last *k* branch evaluations. Each time a branch is evaluated, a new bit is shifted into a register representing this history representing whether the branch was taken or not. The second part is a table of 2^k 2-bit saturating counters.

When a branch instruction is evaluated by the BPU, the current global history is used as an index into the counter table, and that counter is used to make the prediction. When a branch is evaluated by the BCU, the global history is updated, then the associated counter is updated.

4 Experimental Design

The performance evaluation experiments were focused around three program types: Quicksort, Dijkstra's Algorithm, and Gauss-Jordan Elimination. For each of these program types, 3-5 common implementations were sourced (see Appendix A) to be run on the CPU. Each program implementation is run using each branch prediction strategy, with a series of different input sizes (see Results and Analysis).

To control for the input data into each program run, different program implementations running with the same parameters are given the same input data for each trial. In other words, the first trial Program Type T, Implementation X, using Strategy S with input size N receives the same input data as the first trial for Implementation Y of the same type, strategy, and input size, for all pairs X, Y and for all T, S, and N.

Data is collected using the following:

$$r_{T,S,N} = \frac{\sum_x \sum_i C_{T,x,S,N,i}}{\sum_x \sum_i B_{T,x,S,N,i}}$$

r := weighted average of ratio correct branches to total branches for Program Type T using Strategy S with input size N

x := Current Program Implementation

i := Current Trial Number

5 Results and Analysis

The following graphs show the measured correct prediction ratio of each branch prediction strategy for each of the three program types.

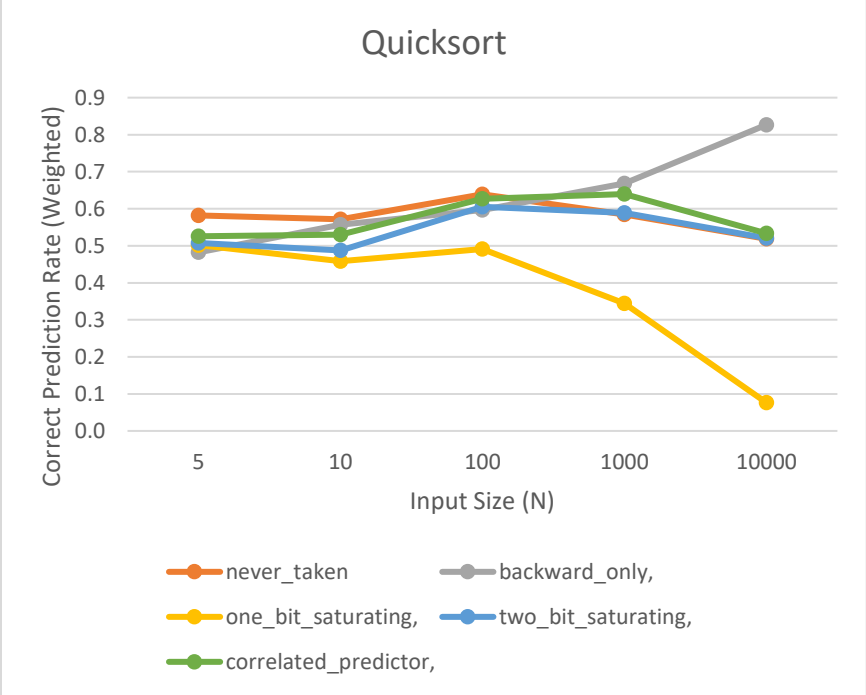

Figure 5.1.1. Weighted average of correct prediction rate for Quicksort programs at different array sizes using different prediction strategies.

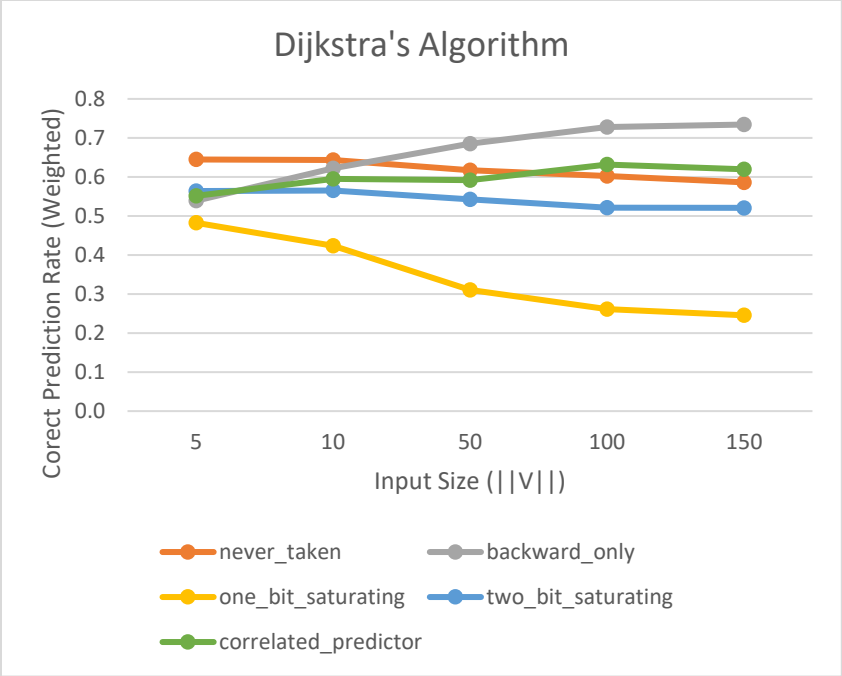

Figure 5.1.2. Weighted average of correct prediction rate for Dijkstra's Algorithm programs at different graph sizes using different prediction strategies.

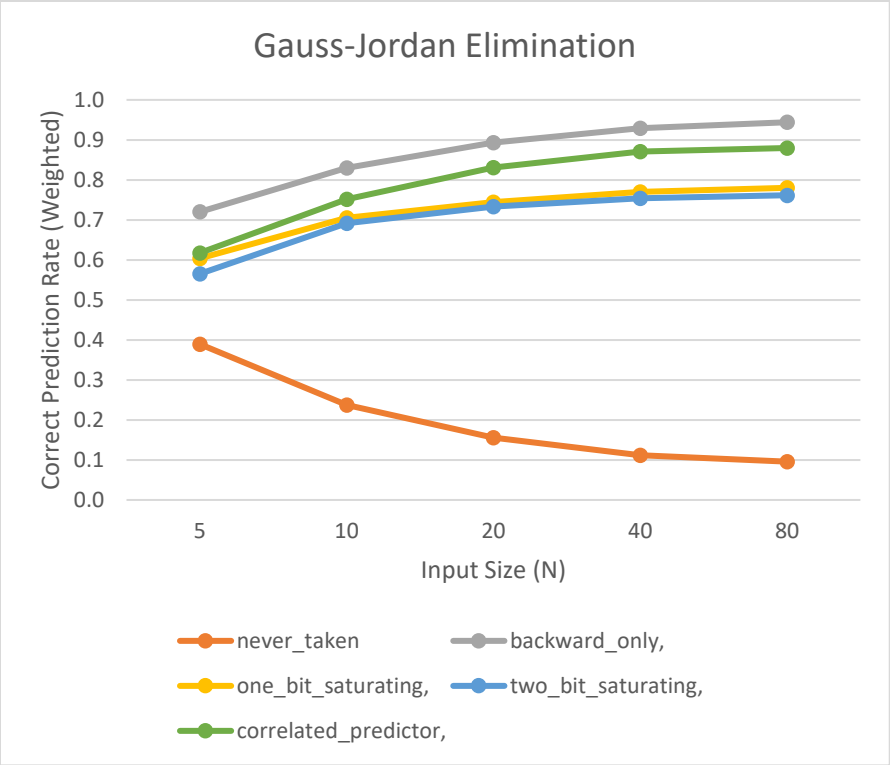

Figure 5.1.3. Weighted average of correct prediction rate for Gauss-Jordan Elimination programs at different matrix sizes using different prediction strategies.

6 Conclusion

The results of this project first show that the input size of data into programs can have a dramatic impact on the accuracy of the prediction strategy being used. This can be seen with the accuracy of the one-bit saturating counter strategy whose performance quickly tanked with increases in input size for both Quicksort and Dijkstra's algorithm programs. Conversely, the accuracy of the backward only predictor approaches >95% for an input size of 80 in the Gauss-Jordan Elimination programs.

Differences in the overall accuracy between program types, and the different rates of change in accuracy for input sizes also highlights the idea that granular testing of branch prediction strategies based on program type can discriminate potential performance drawbacks.

7 Further Study

This project has spawned many different veins for further study that could be done to expand the breadth of this project, including:

- Implementing and testing more complex and more modern branch prediction strategies, such as branch target predictors, neural branch predictors [5], and TAGE predictors [6].

- Increasing the amount of data collected by testing more program types, with a larger spectrum of input sizes.
- Implementing this project's testing pipeline into hardware to increase the efficiency of data collection.
- Measuring other branch prediction performance characteristics such as overall instruction throughput, and also experimenting with different implementations of the BPU at different pipeline stages.

8 References

[1] "Branch Prediction Review," Paul G. Allen School of Computer Science and Engineering, https://courses.cs.washington.edu/courses/csep548/06au/lectures/branchPred.pdf.

[2] H. Li *et al.*, "An architecture-level analysis on deep learning models for low-impact computations," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 1971–2010, 2022. doi:10.1007/s10462-022-10221-5

[3] C.-K. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *2019 IEEE International Symposium on Workload Characterization (IISWC)*, Jun. 2019. doi:10.1109/iiswc47752.2019.9042108

[4] O. Mutlu, "Computer Architecture: Branch Prediction," Carnegie Mellon University Electrical and Computer Engineering, https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module7.4.1-branch-prediction.pdf.

[5] L. N. Vintan and M. Iridon, "Towards a high performance neural branch predictor," *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, Jul. 1999. doi:10.1109/ijcnn.1999.831066

[6] A. Seznec, "A new case for the Tage Branch Predictor," *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011. doi:10.1145/2155620.2155635

Appendix A

The following sources were used for the implementations for the respective program type:

Quicksort

https://www.geeksforgeeks.org/quick-sort/

https://hackr.io/blog/quick-sort-in-c

https://www.javatpoint.com/quick-sort

https://sites.google.com/a/rdaemon.com/rdaemon/home/algorithms/quick-sort

https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#C

Dijkstra's Algorithm

https://www.codewithc.com/dijkstras-algorithm-in-c/

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

https://www.programiz.com/dsa/dijkstra-algorithm

https://www.thecrazyprogrammer.com/2014/03/dijkstra-algorithm-for-finding-shortest-path-of-a-graph.html

Gauss-Jordan Elimination

https://github.com/adityathebe/row_echelon

https://www.codesansar.com/numerical-methods/gauss-elimination-method-python-program.htm

https://www.codespeedy.com/gaussian-elimination-in-cpp/