Report on Branch Prediction Techniques

Prashant Kumar, Pranav Dipesh Bhole, Kartikey Sahu

March 26, 2024

GitHub Repository

https://github.com/Prashant370/RISC-V-Assembler/tree/main

1 Introduction

Branch prediction refers to a set of algorithms designed to forecast the outcomes of branch instructions with precision. The primary goal is to minimize the likelihood of flushing post-branch instructions during later stages of execution, which would otherwise waste valuable instruction cycles, leading to inefficiencies in program execution. Achieving accurate branch prediction is challenging due to the diverse nature of branch resolution patterns across different types of programs.

Some applications exhibit branch resolution patterns that align with spatial or temporal locality principles similar to cache replacement policies, while others demonstrate unique behavior patterns influenced by interdependent branches. Algorithms leveraging global history analyze recent branch outcomes to identify correlations with current instructions, whereas those utilizing local history focus on recent outcomes specific to branches from similar addresses. Ideally, branch prediction algorithms aim to achieve performance levels comparable to programs without branches, or at least perform no worse than a processor that consistently predicts the same branch resolution.

Despite the complexity involved, there is a positive cost-benefit ratio in exploring new branch prediction methods, as these approaches have the potential to significantly enhance program efficiency.

1.1 Always Not Taken

A simple implementation of branch prediction that has an increase in performance over just not predicting branches at all is the prediction that all branches are not taken. If no predictions are made, ever, and instructions are only fetched after previous instructions are decoded, branches will always take 3 cycles in our example 5 stage pipeline and regular instructions will take 2 cycles.

If we always assume the branch is not taken, sometimes we'll be right and the branch will only use 1 cycle, while sometimes we'll be wrong and the branch will use 3 cycles - this is still better than making no predictions at all. In this model, regular instructions only take 1 cycle, as well.

The predict not-taken predictor is the simplest predictor we can have, all it does is increment the program counter. No extra hardware is required to support the predict not-taken predictor - we should always have the ability to increment the program counter.

A rule of thumb in computer architecture is that 20 percent of all instructions are branches. For branches, about 60 percent of all branches are taken. Thus, the predict not-taken predictor is correct 80 percent of the time (because of non-branch instructions), plus another 8 percent of the time (because of branch instructions). Overall, the predict not-taken predictor is incorrect 12 percent of the time.

1.2 Always Taken

In the "Always Taken" branch prediction model, all branches are predicted to be taken. This approach assumes that branches are frequently taken, optimizing performance by reducing the number of cycles needed for branch execution. While this strategy may occasionally result in incorrect predictions,

leading to longer branch execution times, overall performance improves compared to not predicting branches at all. Regular instructions also benefit, taking only one cycle for execution. This simplistic approach, although not perfect, demonstrates the potential for performance enhancement through basic branch prediction techniques.

1.2.1 Why do we need better prediction?

Better branch prediction is crucial for improving the performance of modern processors. Branches are common in programs and occur when conditional statements, loops, or function calls are encountered. Predicting the outcome of branches allows processors to speculatively execute instructions ahead of time, filling the pipeline with potentially useful instructions and avoiding pipeline stalls. With accurate prediction, processors can achieve higher instruction throughput and better resource utilization, ultimately leading to improved performance.

While "taken" and "not taken" predictors offer simple solutions to branch prediction, they have limitations in accurately predicting branches in modern processor architectures. A single-bit predictor, for example, only maintains one bit of history for each branch, resulting in frequent mispredictions, especially for loops or conditional statements with alternating outcomes. Two-bit predictors improve accuracy by incorporating additional history, allowing for more nuanced predictions based on recent branch behavior. These predictors utilize state machines to track branch history, adjusting prediction behaviors dynamically. By employing two-bit or multi-bit predictors, processors can achieve higher prediction accuracy and reduce the performance impact of mispredictions, ultimately improving overall system performance and efficiency in handling complex branch patterns.

1.3 One Bit Predictor

A one-bit predictor uses a single bit to predict whether a branch will be taken or not taken. If the bit is set, it predicts taken; otherwise, not taken. While simple, it struggles with dynamic or alternating branch patterns, leading to frequent mispredictions. Modern processors often employ more sophisticated predictors, like two-bit or dynamic predictors, to improve accuracy by considering additional history and adapting to changing execution patterns

1.3.1 Problems with One bit prediction

The 1-bit predictor predicts certain branch patterns well, such as branches that are always taken, always not taken, or taken significantly more often than not taken. However, it fails to provide accurate predictions in several scenarios:

- Branches that are taken slightly more often than not, or not taken slightly more often than taken, leading to frequent mispredictions.
- Encountering short loops where the branch behavior is not accurately captured by the predictor.
- Situations where the number of taken and not taken branches are approximately equal, resulting in unreliable predictions.

These limitations necessitate the exploration of more advanced branch prediction techniques, such as pattern predictors, to improve prediction accuracy and overall processor performance.

1.4 Two Bit Predictor

The two-bit predictor, also known as the two-bit counter, enhances branch prediction by introducing an additional bit to the Branch History Table (BHT) to implement a state machine. This additional bit, known as the hysteresis or conviction bit, complements the existing prediction bit in the 2BP. The 2BP operates with the following states:

- **NT,NT - Strong not-taken state:** Indicates a strong prediction that the branch will not be taken.
- **NT,T - Weak not-taken state:** Signifies a weak prediction that the branch will not be taken.

- **T,NT - Weak taken state:** Represents a weak prediction that the branch will be taken.
- **T,T - Strong taken state:** Indicates a strong prediction that the branch will be taken.

The 2BP starts in the strong not-taken state for a particular program counter. When a misprediction occurs, indicating that a branch is not taken when it should be, the hysteresis bit is incremented, transitioning the predictor to the weak not-taken state. The branch is still not taken until further evidence proves otherwise. Subsequent mispredictions may transition the predictor to the weak taken state.

Unlike the 1-bit predictor, the 2BP exhibits less susceptibility to rapid state changes in branch prediction, providing more stable predictions. This addresses the issue encountered with the 1-bit predictor, where anomalies in branch behavior could lead to multiple mispredictions and performance penalties.

1.5 Implementation

Our project focused on simulating branch predictor algorithms on both small unit of program execution traces such as Binary search , Bubble sort , Matrix Multiplication and large such as wikisort, Factorial, Sqrt, Recursion etc , to demonstrate and compare their Accuracy 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.

2 Observation

Below are the observations for the accuracy of different branch prediction techniques across various code segments:

Factorial

- **Always Taken Predictor:** Accuracy: 63.75%, Mispredicted: 27379/75530
- **Always Not Taken Predictor:** Accuracy: 36.25%, Mispredicted: 48151/75530
- **One Bit Predictor:** Accuracy: 67.27%, Mispredicted: 8769/75530
- **Two Bit Predictor:** Accuracy: 74.55%, Mispredicted: 7359/75530

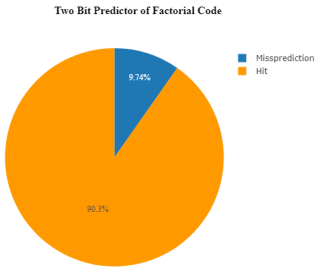

Figure 1: Factorial Accuracy and Misprediction rate

Quick Sort

- **Always Taken Predictor:** Accuracy: 68.23%, Mispredicted: 26381/83038
- **Always Not Taken Predictor:** Accuracy: 31.77%, Mispredicted: 56657/83038

- **One Bit Predictor:** Accuracy: 93.60%, Mispredicted: 5316/83038
- **Two Bit Predictor:** Accuracy: 94.74%, Mispredicted: 4366/83038

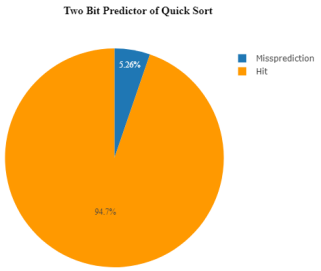

Figure 2: Quick Sort Accuracy and Misprediction rate

Recursion

- **Always Taken Predictor:** Accuracy: 72.03%, Mispredicted: 23347/83478
- **Always Not Taken Predictor:** Accuracy: 27.97%, Mispredicted: 60131/83478
- **One Bit Predictor:** Accuracy: 96.20%, Mispredicted: 3172/83478
- **Two Bit Predictor:** Accuracy: %, Mispredicted: 2795/83478

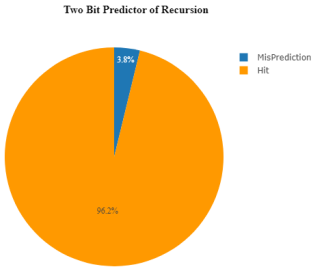

Figure 3: Recursion Accuracy and Misprediction rate

Binary Search Code

- **Always Taken Predictor:** Accuracy: 47.27%, Mispredicted: 29/55
- **Always Not Taken Predictor:** Accuracy: 52.73%, Mispredicted: 26/55
- **One Bit Predictor:** Accuracy: 67.27%, Mispredicted: 18/55
- **Two Bit Predictor:** Accuracy: 74.55%, Mispredicted: 14/55

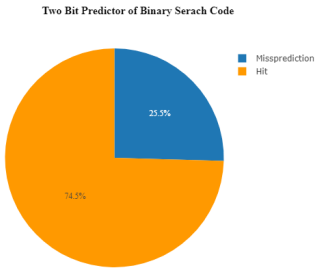

Figure 4: Binary Search Accuracy and Misprediction rate

Bubble Sort

- **Always Taken Predictor:** Accuracy: 41.48%, Mispredicted: 454666/776965
- **Always Not Taken Predictor:** Accuracy: 58.52%, Mispredicted: 322299/776965
- **One Bit Predictor:** Accuracy: 97.11%, Mispredicted: 22471/776965
- **Two Bit Predictor:** Accuracy: 98.40%, Mispredicted: 12444/776965

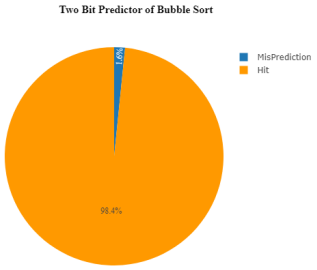

Figure 5: Bubble Sort Accuracy and Misprediction rate

Matrix Multiplication

- **Always Taken Predictor:** Accuracy: 0.77%, Mispredicted: 258/260
- **Always Not Taken Predictor:** Accuracy: 99.23%, Mispredicted: 2/260
- **One Bit Predictor:** Accuracy: 99.23%, Mispredicted: 2/260
- **Two Bit Predictor:** Accuracy: 99.23%, Mispredicted: 2/260

wiki Sort

- **Always Taken Predictor:** Accuracy: 51.68%, Mispredicted: 1056199/2186027
- **Always Not Taken Predictor:** Accuracy: 48.32%, Mispredicted: 1129828/2186027
- **One Bit Predictor:** Accuracy: 95.14%, Mispredicted: 106153/2186027
- **Two Bit Predictor:** Accuracy: 96.43%, Mispredicted: 78015/2186027

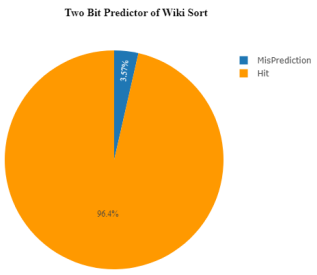

Figure 6: Wiki Sort Accuracy and Misprediction rate

3 Result and Conclusion

For Small number of Traces accuracy is less compared to large number of traces so keep in mind that a normal program must have large number instructions that leads to more acccurate result if we predict the output so it is always used So we've seen that the 2 bit predictor is better than the 1 bit predictor at handling anomalous outcomes. Does this get better if we add 3 bits to the predictor? Well, maybe. Another bit will just increase the number of states, making it more difficult to transition to another prediction - this is useful if the anomalous behavior comes in streaks. This behavior is not common, however , in programs. Adding another bit imposes more cost, as it requires more space to store the information per program counter.

2-bit predictor is as good as k-bit predictor and this is proven by our Obersvation too that for large number of traces like 10M+ we have accuracy of 98