CS4202 Practical 2: Branch Predictor

190018035

April 5, 2024

Contents

1 Introduction

Branch prediction is a essential component of modern processors, allowing for optimized execution of programs by predicting the outcome of conditional branches. This practical aims to implement various branch prediction strategies and evaluate their performance on a set of benchmarks. The main goals of this practical are to gain practical experience with branch prediction strategies, understand their importance for particular programs, and to experiment and evaluate the performance of these strategies on a set of benchmarks. The implementation of this practical achieves all of these goals, providing a comprehensive understanding of branch prediction strategies.

2 Methodology

This section will provide details on the key design decisions and assumptions made during the implementation of the branch prediction simulator. The simulator was implemented in Golang, and is capable of simulating the execution of a program using various branch prediction algorithms. The simulator reads in a dynamic trace file, containing information about each branch in the program, and simulates the execution of the program using the specified branch prediction algorithm.

2.1 Key Assumptions

This section will outline the key assumptions made during the implementation of the simulator. These assumptions were made to simplify the implementation and to focus on the core functionality of the simulator.

1. **Sequential Trace Files:** The trace files provided represent the execution of a program in a sequential manner. This means that the head of the trace file represent the initialization phase of the program, and the tail of the trace file represents the cleanup phase of the program.

2. **Available Trace Information:** The additional information provided in the trace files, such as the branch kind and whether the branch was conditional, direct and taken, are all available for the simulator to use.

3. **Conditional Branches Only:** Only the conditional branches in the trace file are considered for branch prediction, since these are the branches that require prediction.

4. **Limited Profile Data:** To simulate a realistic scenario, where the entire trace file may not be available, the default amount of data used for profiling for the profiled algorithm is limited to 8% of the total trace file size.

2.2 Configuration Files

In order to experiment with different branch prediction strategies, the simulator reads in a configuration file that specifies the parameters for the simulation. This is the main entry point for a user wishing to reproduce the results of the experiments. A configuration file was chosen as a way to specify the parameters for the simulation, as it allows for a user-friendly, flexible and efficient way to experiment with multiple configurations of an algorithm at one time.

The configuration file specifies a few global parameters that apply to all algorithms and algorithm-specific parameters. The configuration file is in JSON format. An example configuration file for the gshare algorithm is shown in Listing 1.

```
{
    "algorithm": "gshare",
    "max_lines": [1000, 10000, 100000, -1],
    "configs": [
      {
        "initial_state": "StronglyNotTaken",
        "table_size": 2048,
      },
```

```
        {
          "initial_state": "StronglyTaken",
          "table_size": 4096,
        },
    ]
}
```

Listing 1: Example configuration file for the gshare algorithm

Sample configuration files can be found in the `configs/` folder in the project root directory. The following sections will describe the key parameters in the configuration file.

2.2.1 Global Parameters

The global parameters specify the parameters that are shared across all algorithms. They are as follows:

- **algorithm**: The name of the branch prediction algorithm to use. This can be one of `always-taken`, `two-bit`, `gshare`, or `two-bit-profiled`.
- **max_lines**: An array of integers specifying the maximum number of lines to read from the *middle* of the trace file. This was a key design decision to allow for faster experimentation with different trace file sizes. The middle of the trace file was chosen in accordance with the key assumption that the trace file is sequential, indicating that the middle would represent the most a "stable" part of the program execution, avoiding the initialization and cleanup phases where the branching behavior may be different from the main loop of the program. The simulator will stop reading lines once it reaches the specified number of lines. If a value of -1 is specified, the simulator will read the entire trace file. The simulator will run the simulation for each value in the array.

2.2.2 Algorithm-Specific Parameters

The algorithm-specific parameters specify the parameters that are specific to each algorithm. They are contained within the `configs` array in the configuration file. Each element in the array corresponds to a configuration object for the algorithm, allowing the user to test multiple configurations of the same algorithm at once. The parameters for each of the algorithms are as follows:

Always Taken This algorithm requires no additional parameters, as it always predicts that a branch will be taken.

Standard 2-bit Predictor & gshare These algorithms requires the following parameters:

- **initial_state**: The initial state of the predictor table. This can be one of `StronglyTaken`, `WeaklyTaken`, `WeaklyNotTaken`, or `StronglyNotTaken`.
- **table_size**: The size of the predictor table in bytes. This can be one of 512, 1024, 2048 or 4096.

Two-Bit Profiled To profile the branches in the trace file, the simulator uses a strata-based profiling approach, where stratified sampling of the trace file is used to profile the branches. By dividing the portion of the trace file (specified by the `max_lines` parameter) into different strata and collecting a proportion of that data for profiling, the simulator is able to profile the branches in a more representative manner, as opposed to just taking the first few branches of the middle of the trace file.

This algorithm requires the parameters for the standard 2-bit predictor, as well as a few additional parameters.

- **strata_size**: The number of strata to use for profiling.
- **strata_proportion**: The proportion of the strata to use for profiling. This must be a float value between 0 and 1. The simulator will take the first `strata_proportion` of each strata for profiling to avoid disrupting the sequential nature of the trace file.

3 Branch Prediction Strategies

This section will describe the branch prediction algorithms implemented in the simulator. This includes the Always Taken, Standard 2-bit Predictor, gshare, and Profiled algorithms. We will briefly cover the key concepts behind each algorithm, and how they were implemented in the simulator.

3.1 Always Taken

The Always Taken algorithm is the simplest branch prediction algorithm, where the predictor always predicts that a branch will be taken. This is a useful baseline to compare the performance of other branch prediction algorithms. The implementation of this algorithm is straightforward, as it predicts a branch whenever it encounters one.

3.2 Standard 2-bit Predictor

The Standard 2-bit Predictor is a branch prediction algorithm that uses a 2-bit saturating counter to predict the outcome of a branch. The algorithm is a type of local branch predictor, where the prediction is based on the history of the branch itself. Each branch has an associated 2-bit saturating counter that is used to predict the outcome of the branch. Figure 1 demonstrates the state diagram of the 2-bit predictor, which shows how the predictor transitions between different states based on the actual outcome of the branch.

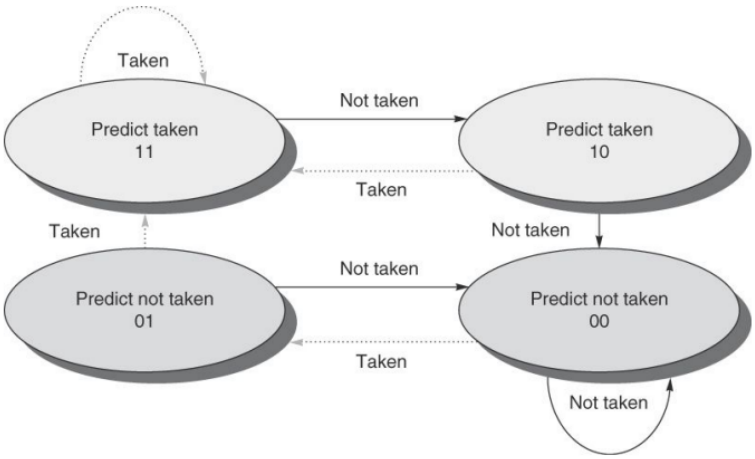

Figure 1: 2-bit Predictor State Diagram [1]

A `State` enum was created to represent the different states of the 2-bit predictor. The predictor table is implemented as a hash map, where the key is the branch address (masked to the table size) and the value is the 2-bit saturating counter.

3.3 gshare

The gshare algorithm is a combination of a local branch predictor and a global branch predictor. The local branch predictor portion of the algorithm is identical to the Standard 2-bit Predictor, where each branch has an associated 2-bit saturating counter that is used to track the local history of the branch. The state mappings are stored in a *pattern history table*, similar to the 2-bit predictor. The global branch predictor portion of the algorithm uses a *global history register* to track the global history of the branches. When a branch is encountered, the global history register is shifted left by one bit, and the outcome of the branch determines the least significant bit of the global history register. Finally, the global history register is XORed with the branch address to generate an index into the pattern history table. The prediction is made based on the state of the saturating counter at the index. This process is illustrated in Figure 2 below.

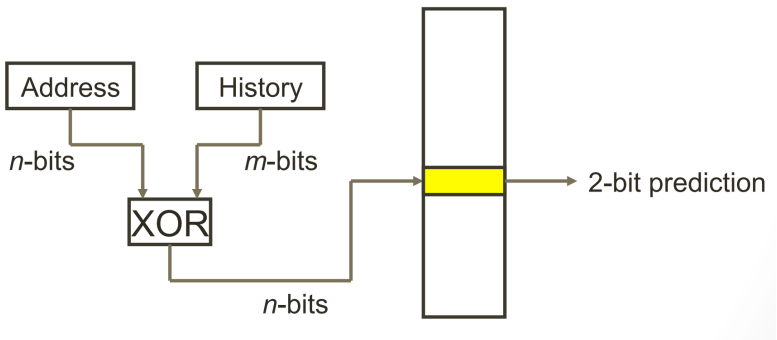

Figure 2: gshare Predictor [2]

The implementation of the gshare algorithm in the simulator combines the local and global history components to make predictions. The pattern history table table is

implemented as a hash map, where the key is the XOR of the global history register and the branch address (masked to the table size) and the value is the 2-bit saturating counter. The global history register is masked to the size of the table to ensure that it does not exceed the size of the pattern history table.

3.4 Profiled

The Profiled algorithm is an extension of the Standard 2-bit Predictor algorithm that uses profiling to improve the prediction accuracy. This type of algorithm was chosen to clearly demonstrate the impact of profiling on the prediction accuracy - by profiling 2-bit, we can show how the prediction accuracy could improve when the branch history is known.

As mentioned previously, the algorithm uses stratified sampling of the trace file to gather data for the profiled run. The implementation of the Profiled algorithm in the simulator uses the same 2-bit saturating counter as the Standard 2-bit Predictor, along with the prediction table mapping. However, this algorithm utilizes the `MetaData` struct, which stores the profiling data for each branch. The algorithm first runs a profiling phase on the stratified sample of the trace file to collect the branch outcomes. Each time a branch is encountered and processed, the current state of the branch is stored in a `StateRecord`, a list within the `MetaData` struct that stores the history of states for the branch. Once the profiled run is over, the most frequent state in the `StateRecord` is used as the initial state for the prediction phase for each branch encountered during the profiling phase. By doing this, the algorithm saves the warm-up time required for the 2-bit predictor to learn the branch behavior, as it uses the most frequent state as the initial state for the prediction phase. For branches not encountered during the profiling phase, the state is set to the default initial state specified in the configuration file.

4 Experimental Setup

Experiments were performed on a system equipped with a 12th Gen Intel Core i7-12700H (20 threads, 2.30 GHz base frequency) processor on system running Fedora 39. The Golang version used for the experiments was 1.20.12. Configuration files for each algorithm were specified. For the algorithm-specific parameters, every combination of the parameters was tested. For instance, for the gshare algorithm, the initial state was varied between `StronglyTaken` and `StronglyNotTaken`, and the table size was varied between 512 and 4096. This ensured that data was collected for every possible combination. The only exception for this was the profiled algorithm, where the strata size and strata proportion were kept constant at 10 and 0.2, respectively. This was done to ensure that the profiling is consistent across all experiments, and using the same proportion of the trace file (8%) for profiling. For the global parameters, the list of the maximum number of lines to read from the trace file was specified as [1000, 10000, 100000, 1000000, 10000000, -1]. This gave an appropriate range of entry sizes to test the algorithms on. Data was gathered for each of the trace files provided to get a comprehensive view of the performance of the algorithms under different configurations and trace file sizes. Running each configuration for each algorithm across all the trace files can be done by running the `run.sh` script in the project root folder. Results of the experiments can be found in the `results/` folder in the project

root directory.

5 Results & Discussion

This section will present the results of the experiments conducted on the branch prediction algorithms described above. An analysis of the results will be provided, detailing the performance of each algorithm under different configurations and trace file sizes and including interesting observations from the experiments.

5.1 General Observations

This section will present the general observations made when comparing the performance of the branch prediction algorithms. The key metrics used to evaluate the performance of the algorithms were the misprediction rate. Taking an average of the misprediction rate across all trace files produces the following results, presented in Figure 3 below:

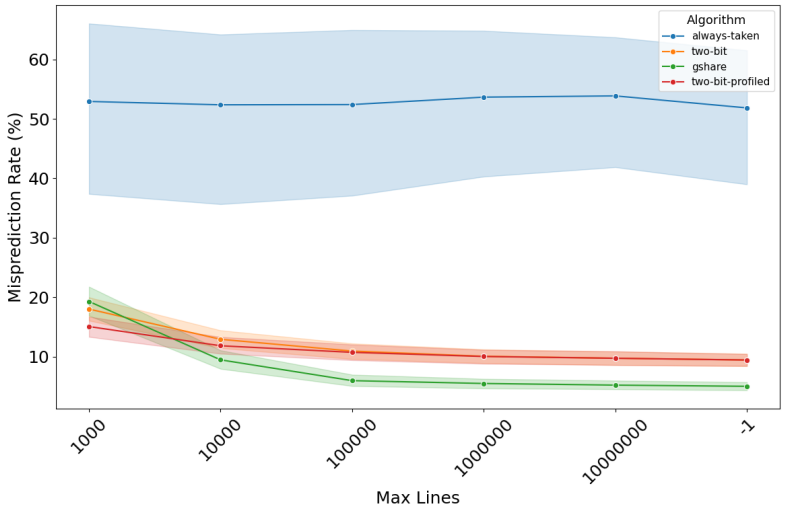

Figure 3: Misprediction Rate vs Max Lines of All Algorithms

The Always Taken algorithm consistently had the highest misprediction rate across all trace files. This is expected, as the algorithm always predicts that a branch will be taken, regardless of the actual outcome. The Standard 2-bit Predictor and profiled 2-bit predictors demonstrated similar misprediction rates, with the profiled algorithm having a lower misprediction rate for smaller trace files. This phenomenon will be detailed further in Section 5.3. Finally, the gshare algorithm had the lowest misprediction rate across all trace files. This is as expected, as the gshare algorithm combines the local and global history of the branches to make predictions, providing a more accurate prediction compared to the other algorithms.

One interesting phenomenon to note is that in in some trace files, notably the GCC and Cactusbssn trace files, the misprediction rate of gshare is higher than both the profiled algorithm and the standard two-bit algorithm for smaller trace files. This can be seen in Figure 4 and Figure 5 below:

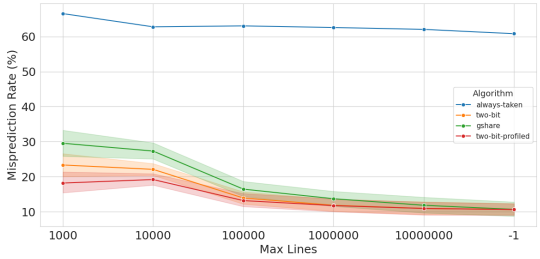

Figure 4: Misprediction Rate vs Max Lines of Profiled vs Non-Profiled Algorithm Using GCC Trace File

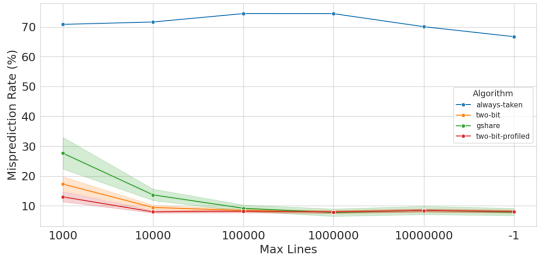

Figure 5: Misprediction Rate vs Max Lines of Profiled vs Non-Profiled Algorithm Using Cactusbssn Trace File

This observation may be attributed to the fact that gshare makes use of past behavior from different parts of the program to make predictions. In the case of smaller portions of the trace file being read, there is fewer opportunities and less repetition for the global history register to accurately reflect the behavior of branches. This may make it perform worse than the local predictors, since they can react to branch behavior without the interference of global history register. Additionally, this observation may also show that these trace files in particular have a branching pattern that do not correlate well with past global patterns, making it more unpredictable in the short term.

5.2 Parameter Analysis

This section will explore the impact of the different parameters of the branch prediction algorithms on the misprediction rate.

5.2.1 Table Size

A trend observed across all trace files was that the misprediction rate of the branch prediction algorithms decreased as the table size increased, shown in Figure 6. This is as expected - having a larger table size allows the algorithms to store more information about the history of each branch, which can help improve the prediction accuracy. This is particularly evident in the case of the gshare algorithm, which has a sharper decrease in misprediction rate as the table size increases.

5.2.2 Initial State

The initial state of the branch prediction algorithms can also have a significant impact on the misprediction rate. The initial state determines the state of the predictor table when the simulation begins. Initializing the branch with a arbitrary state can lead to a higher misprediction rate, as the predictor table may not be in an optimal state to make accurate predictions. The impact of the initial state can be seen in Figure 7 below. Each algorithm that uses initial state as a parameter shows a noteworthy difference amongst the different initial states. This is quite evident with the gshare algorithm, where on average, the initial state of `StronglyNotTaken` has a nearly 2% lower misprediction rate compared to the initial state of `StronglyTaken`.

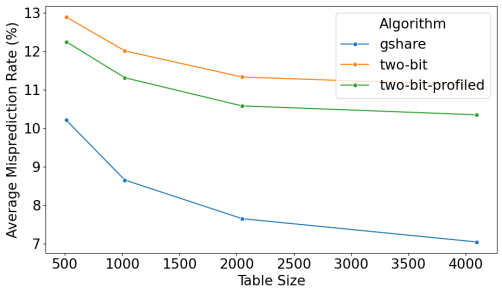

Figure 6: Average Misprediction Rate vs Table Size of All Algorithms

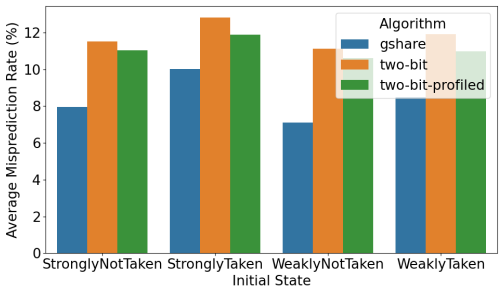

Figure 7: Average Misprediction Rate vs Initial State All Algorithms

5.3 Profiling vs Non-Profiling

Another interesting observation is the impact of profiling on the prediction accuracy of the branch prediction algorithms. When analyzing the data, two key observations were made. Firstly, the profiled algorithm outperforms the standard 2-bit predictor algorithm in terms of prediction accuracy when the number of entries read from the trace file is limited. However, as the number of entries grows, the benefit of profiling seems to diminish. This can be seen in Figure 8 and Figure 9 below:

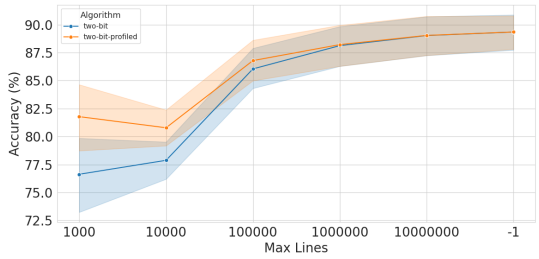

Figure 8: Accuracy vs Max Lines of Profiled vs Non-Profiled Algorithm Using GCC Trace File

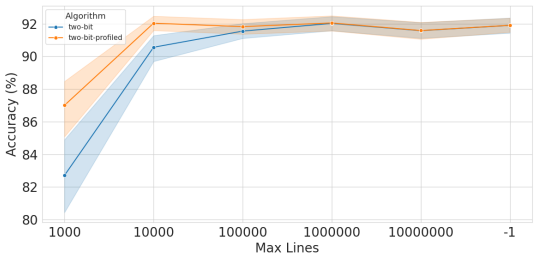

Figure 9: Accuracy of Profiled vs Non-Profiled Algorithm Using Cactusbssn Trace File

One possible explanation for this can be that as the number of entries increases, the amount of dynamic execution information available to the standard two-bit algorithm allows it to effectively adapt, diminishing the initial advantage provided by profiling. This intuitively make sense, as a larger number of entries read from the trace file would provide more opportunities for the standard algorithm to 'learn' and update its predictions.

Secondly, the profiled algorithm tends to outperform the standard 2-bit predictor algorithm in terms of misprediction rate as the table size increases. This can be seen in Figure 10 and Figure 11 below:

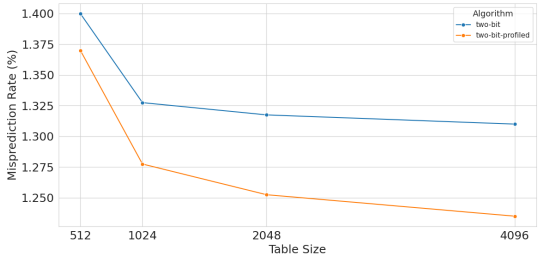

Figure 10: Accuracy vs Table Size of Profiled vs Non-Profiled Algorithm Using WRF Trace File

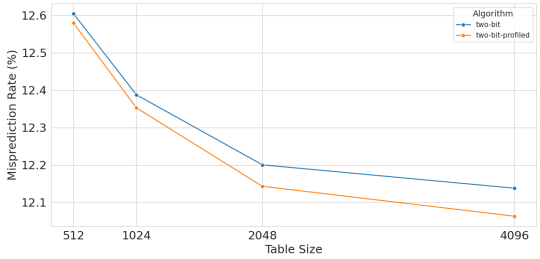

Figure 11: Accuracy vs Table Size of Profiled vs Non-Profiled Algorithm Using Exchange2 Trace File

This observation can be attributed to the fact that larger table sizes provide more context and history for each individual branch, since the table size determines the number of entries that can be stored in the predictor table. By having a larger table size, the profiled algorithm is able to store more information the states of each branch in the past, helping it stay more accurate in its predictions.

6 Conclusion

To conclude, the branch prediction simulator implemented for this practical, along with the experimentation of different branch prediction strategies, achieves all the objectives set out in the project specification. The experimental results show the impact of different branch prediction strategies on the prediction accuracy and the number of mispredictions. The results also demonstrate the importance of profiling in improving the prediction accuracy of the branch prediction algorithms. Overall, this practical has provided valuable insights into the design and implementation of branch prediction algorithms. Future work could involve experimenting with more complex branch prediction algorithms, such as tournament predictors or a profiled gshare predictor, to further improve the prediction accuracy and reduce the number of mispredictions. Additionally, the relationship between the strata size and strata proportion can be further explored to determine the optimal values for profiling the branches in the trace file, without using too much data for profiling.

References

[1] John L. Hennessy, David A. Patterson, and Krste Asanovic. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, 2019.

[2] *Branch Prediction*. URL: `https://people.cs.pitt.edu/~childers/CS2410/slides/lect-branch-prediction.pdf`.