Investigating Branch Prediction Strategies

CS4202 170002815

10th October 2020

1 Introduction

The objective of this practical is to investigate the success rate of branch prediction strategies in relation to computer processor design.

The prediction strategies required to be investigated are an always taken strategy, a 2-bit predictor with a variety of branch prediction table sizes, gshare, and a profiled approach.

The design of an experiment to investigate the success rate of these strategies is outlined and analysed in Sections 3 and 5.1 of this report, and the data collected during the perfomance of this experiment is discussed and analysed in Sections 4 and 5.2.

A short exploration of experiment improvements and further investigations is conducted in Section 6, and Section 7 provides a summary of findings as well as a reflection on how successful I have been in relation to the practical specification.

Information on the submission directory and source code can be found in Section 2, and the bibliography for this report can be found on the final page.

2 Submission Information

2.1 Directory Structure

```
.
├── CS4202_Practical_1_Report.pdf
├── README.md
├── branch_traces
│   ├── *.out
│   └── unused
│       └── *.out
├── output
│   ├── all_data.xlsx
│   ├── always-taken.csv
│   ├── 2-bit.csv
│   ├── gshare.csv
│   ├── profiled.csv
│   ├── chrome_cut_and_uncut.csv
│   └── jpegtran_both.csv
└── src
    ├── main.go
    ├── AlwaysTakenPredictor.go
    ├── TwoBitPredictor.go
    ├── GSharePredictor.go
    └── ProfiledPredictor.go
```

Alongside this report are three directories and a README file. The `branch_traces` directory contains the `.out` tracefiles used in this report, and the `branch_traces/unused` directory contains all other `.out` tracefiles included with the submission.

The `output` directory contains the `.csv` files for each predictor, two extra `.csv` files to store data used in further investigations, and a *Microsoft Excel* spreadsheet (`.xlsx`) which contains all data processing for this report. The `src` directory contains the source code for the simulator used in this report, written in Go, the design and implementation of which is discussed in Section 2.3.

The README file alongside this report contains similar information to Sections 2.1 and 2.2 of this report.

2.2 Program Usage

To run the source code for this practical, execute the following steps.

1. Ensure *Go* is installed on the machine you are using. *Go* can be installed via this link `https://golang.org/dl/`, and is available on the School of Computer Science lab machines.
2. Change into the `src` directory as described in Section 2.1
3. Execute `go build` to compile the code and create an executable.
4. Execute the created executable in step 3 with the following command line arguments.
   (a) *'always-taken'*, *'2-bit'*, *'gshare'* or *'profiled'* to use an always-taken predictor, two-bit predictor, gshare predictor or profiled predictor respectively.
   (b) The directory path to the branch trace files.
   (c) A list of branch trace files to be experimented on, separated by spaces.

   An example of this would be executing

   ```
   .\src.exe 2-bit ..\branch_traces\choices coremark.out g++.out picalc.out
   ```

   to run an experiment using a 2-bit predictor to evaluate the coremark, g++ and picalc files in the `../branch_traces/choices` directory in Windows Powershell.
5. Interpret the output of the program manually from `stdout`, or pipe into a CSV file and open using a spreadsheet application.

2.3 Simulator Design & Implementation

General Information

To implement the simulator required for this experiment, *Go* was used as the programming language as it is efficient and effective to write in.

The main method for the source code can be found in `main.go`, which also contains an interface for a predictor (`Predictor`) with `Setup`, `Predict` and `Update` functions. Implementations of the `Predictor` interface can be found in `AlwaysTakenPredictor.go`, `TwoBitPredictor.go`, `GSharePredictor.go` and `ProfiledPredictor.go`.

The `Setup` function is used to setup the predictor and takes parameters for the table or history size and the filepath for the tracefile, the `Predict` function takes in the branch address and uses the predictor's state to form a prediction for whether the branch will be taken, and the `Update` function is used to update the state of the predictor after a branch is known to be taken or not taken.

Input & Output

The main method handles command line arguments to take in a predictor selection and a list of one or more tracefiles to be simulated. The simulation is completed for each tracefile, outputting information about the simulation in CSV format to `stdout`, complete with headers and a separation indicator to facilitate data manipulation and analysis.

The information outputted includes the name of the predictor, the name of the tracefile, the number of successful predictions, the number of unsuccessful predictions, and the misprediction rate as a ratio.

Always Taken Predictor

The always taken predictor, `AlwaysTakenPredictor.go`, has stub `Setup` and `Update` functions as no setup or update is required. The `Predict` function always predicts taken, making this predictor very simple.

Two-Bit Predictor

The two-bit predictor, `TwoBitPredictor.go`, uses the `Setup` function to instantiate a branch prediction table with a size dependent on parameters passed to the function. The starting state for all entries is `01`, relating to "soft not taken" to reduce the number of initial mispredictions.

The `Predict` function masks the address provided to reduce it to a binary number of length n, where 2^n is the size of the branch prediction table. This string is then used to access the branch prediction table and form a prediction according to the matching state in Figure 1.

The `Update` function adjusts the accessed table entry for the `Predict` method according to the state transitions in Figure 1.

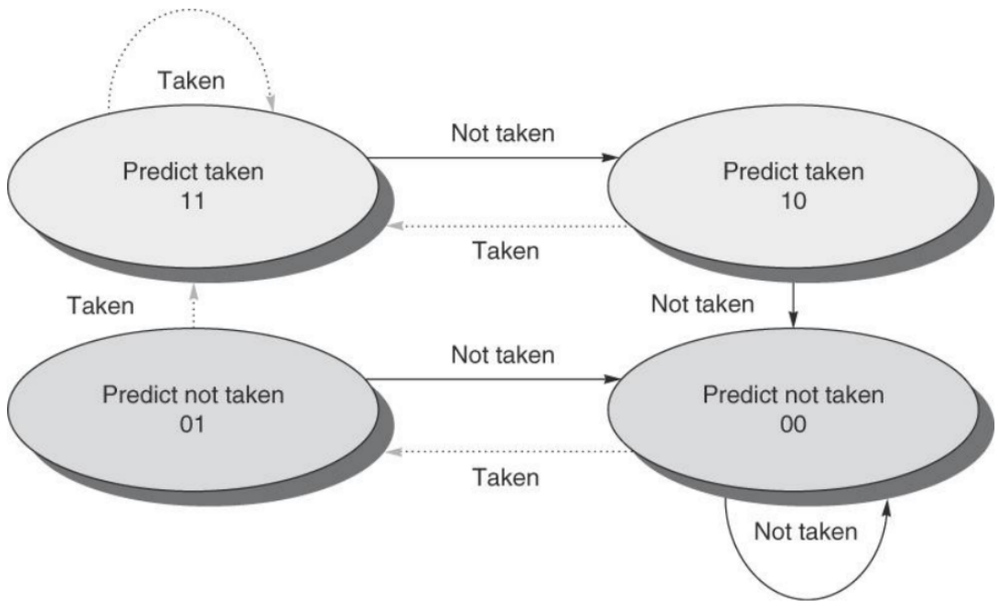

Figure 1: Two-bit predictor prediction states. Sourced from University of St Andrews School of Computer Science's lecture slides for Computer Architecture, CS4202 [1].

gshare Predictor

The gshare predictor, `GsharePredictor.go`, is implemented according to the original gshare paper [2]. The `Setup`, `Predict` and `Update` functions for the predictor behave very similar to those described for the two-bit predictor, but there some subtle differences.

The `Setup` function creates a binary history of specified length, n as well as a branch prediction table with 2^n entries. The `Predict` function masks the provided branch address to length n and then proceeds to XOR the masked address with the history to calculate the index for the branch prediction table. The `Update` function and the remainder of the `Predict` function behave identical to those of the two-bit predictor.

Profiled Predictor

A profiled predictor is able to analyse the branches of a program and whether they are taken before executing the program, with the ability to make a "profile" to improve branch prediction. To implement this in the simulator, the tracefile is passed to the profiled predictor's `Setup` function, allowing for state changes to be made accordingly.

The profiled predictor used in this simulator separates the program into equal-sized "segments", with the number of segments dependent on the value passed to the `Setup` function. Each of these segments is then analysed in the tracefile to determine whether an always taken or always not taken prediction strategy would be most beneficial for each segment, saving the result to the predictor's state.

Upon having the `Predict` function called, the predictor checks against a line counter saved in the it's state to find the current segment. The strategy determined for this segment in the `Setup` function is then used to make a prediction.

The `Update` function simply increments the line counter for this predictor.

Predictor Variants

The two-bit, gshare and profiled predictors simulate more than one predictor per tracefile, as the program tests each predictor with different parameters. The parameters used for each predictor are as follows.

- **Two-Bit Predictor**: Outputs simulation results for branch prediction tables with 512, 1024, 2048 and 4096 entries to conform with the practical specification.
- **gshare Predictor**: Outputs simulation results for shared history of length 8, 9, 10, 11 and 12 bits. 8 bits matches the length described in the original gshare paper [2], and 9, 10, 11 and 12 bits are used to simulate table sizes matching those used in the two-bit predictor.
- **Profiled Predictor**: Outputs simulation results for 1, 2, 8, 64, 128 and 2048 segments.

3 Experiment

3.1 Choice of Variables

3.1.1 Independent & Dependent Variables

The specification for this practical required experiments to be conducted to investigate the effect of using different branch prediction strategies on branch misprediction rates, giving reasoning for the branch prediction strategy and misprediction rate to be used as the independent and dependant variables respectively.

The independent variable of the branch prediction strategy is manually set by the experimenter and will be changed throughout the duration of the experiment. The dependent variable of misprediction rate is measured for each set strategy, and acts as a metric of performance.

Although other metrics for branch prediction strategy performance are available, misprediction rate offers a normalised value in comparison to other metrics such as the number of mispredictions, which would vary drastically between tracefiles. Although prediction rate is also a normalised metric, misprediction rate provides visualisations with finer-grained detail, as the average misprediction rate for branch prediction strategies is less than that of the prediction rate, meaning that changes in values can be represented by greater visual differences. Another reason why misprediction rate is used is because it is required by the practical specification.

A secondary independent variable used throughout this experiment is the choice of tracefile. Although the choice of tracefile will likely make less of an impact on results than the choice of branch prediction strategy, misprediction rates are still affected by the choice and therefore it needs to be considered as an independent variable.

3.1.2 Control Variables

To ensure the dependent variable is solely affected by the manipulation of the independent variables, extraneous variables must be controlled.

The most important variables to control in this experiment are those relating to the environment in which the tracefiles are generated. The input use for programs should be kept constant (e.g. the file used for a compression program), and executing background processes should be avoided where possible (e.g. a web browser clearing cache). The tracefile generation method should also be kept constant between repeats.

Other variables to control include the initial state of each predictor, and the contents of the tracefile between the simulation with different predictors.

As the tracefiles have been pre-generated for this pracitcal, the only control measures applied are to the initial state of the branch predictors and the tracefile between uses for different predictors.

3.2 Method

1. Implement or acquire a simulator for various branch prediction strategies. The simulator should accept input matching the form discussed in step 2, and output the misprediction rate of simulators run against the provided input. An example implementation of such simulator is discussed in Section 2.3

2. Generate or acquire tracefiles for branch addresses and whether they were taken for a variety of programs. Each line of each tracefile should be in the following format, where <BRANCH_ADDRESS> is the address of the branch and <TAKEN> is either 1 to represent that the branch was taken or 0 to represent that the branch was not taken.

   <BRANCH_ADDRESS> <TAKEN>

   An example snippet of a tracefile is as follows.

   ```
   173628492058 1
   129576290203 1
   028362819233 0
   371929471782 0
   173628492058 0
   ```

   The selection of programs used to generate tracefiles should avoid any programs which are known to be IO-bound, such as `cat` on Linux, and should ideally be programs which are open source to facilitate the verification of results by other experiments.

   If subsections of large tracefiles are to be used for timing considerations, the subsection should be taken from the middle portion of the tracefile to mimic the majority of the tracefile, rather than the startup or teardown.

3. Each branch prediction strategy implemented in the simulator from step 1 should be run using all tracefiles from step 2. The tracefiles should not be altered between simulations using different strategies, and predictors should be returned to their initial state after each simulation. The output from each simulation-tracefile combination should be saved appropriately.

4. *[OPTIONAL]* Repeat step 3 using a different simulator from step 1.

5. Repeat steps 1 to 4 at least three times.

4 Results

4.1 Choice of Tracefiles

Location of Files

As discussed in Section 2, the choice of tracefiles used for this experiment can be found in the `branch_traces` directory alongside the report, while the unused tracefiles can be found in the `branch_traces/unused` directory.

Tracefile Table

The chosen tracefiles used in this experiment are displayed in Table 1, where each file is listed along with its size and an example real-world use case. These files have been chosen as they represent a wide range of tasks a modern processor may have to execute, including daily tasks such as web browsing (*google_chrome*) or sorting in an email client (*pythonsort*), as well as more mathematical tasks such as *coremark* and *picalc*.

Filename	**Size**	**Real-World Use Case**
coremark.out	1,028 kB	CPU benchmarking
g++.out	9,314 kB	Code compilation
*google_chrome.out	47,108 kB	Web browsing
google_chrome_cut.out	8,740 kB	Web browsing
*jpegtran.out	8,348 kB	Image transformation
*jpegtran2.out	15,248 kB	Image transformation
picalc.out	6,782 kB	Mathematical computation
*pythonsort.out	134,526 kB	Sorting (e.g. email client)
pythonsort_cut.out	9,000 kB	Sorting (e.g. email client)

Table 1: Chosen tracefiles for experiment. Asterisked (*) files are not included when calculating the average misprediction rate.

Variety

The decision to incorporate a wide range of programs used to generate tracefiles is to promote the external validity of the experiment, as most processors in the modern world are required to perform well for a wide-variety of programs.

Although there are many unused tracefiles which could be used in this experiment to increase the variety of programs represented by the experiment, such as `zip.out`, the set in Table 1 has been used for conciseness.

IO-Bound vs. CPU-Bound

Tracefiles generated from IO-bound programs have purposely been avoided, as the branch prediction performance for CPU-bound programs has a much greater effect on a processor's overall performance. This is because the CPU execution time of IO-bound programs constitutes to much less of the total execution time than CPU-bound programs, meaning CPU-bound gains from branch prediction are more important to consider when deciding upon a branch prediction strategy for a CPU. Such tracefiles include `curl.out`, which is from an IO-bound program which predominantly involves sending data over a network, and `ffmpegplay.out`, which is from an IO-bound program which predominantly involves reading video data from a disc.

The `google_chrome.out` tracefile could be considered to be generated using an IO-bound program, as *Google Chrome* is a web browser and therefore involves sending and receiving data over a network. However, the fact that *Google Chrome* does perform CPU-bound processes such as rendering and pre-rendering [3], as well as it being an extremely popular program [4] suggest that the branch prediction performance for *Google Chrome* is an important consideration for a general purpose processor.

The tracefiles of `jpegtran.out` and `jpegtran2.out` have potential to be IO-bound, as if large images are being transformed, fetches of data from lower levels of the memory hierarchy likely take up a large percentage of execution time. However, *jpegtran* was the only program used to generate multiple tracefiles, and the tracefiles are therefore used to investigate internal validity in Section 5.1.

Larger Files

For the experiment performed for this report, the larger files of `pythonsort.out` and `google_chrome.out` were shortened to reduce computation time. As described in step 2 of the experiment, the middle of the tracefiles were used to avoid startup and teardown branches - the validity of doing so is explored in Section 5.1.

The shortened files for `pythonsort.out` and `google_chrome.out` are named `pythonsort_cut.out` and `google_chrome_cut.out`, and were shortened to around 9,000 kB using the following command on a Linux machine.

```
tail -n +<START_LINE> <IN_FILE> | head -<CUT_LENGTH> > <OUT_FILE>
```

4.2 Data Collection

As discussed in Section 2.3, the simulator used for this experiment outputs data as in CSV format, facilitating the storage, manipulation and analysis of the results. The files where the results are saved for this experiment can be found in the `output` directory alongside this report, as discussed in Section 2.1.

The simulation was completed on *Windows 10* using an AMD Ryzen 7 1700x processor, allowing for all results to be generated in a short time frame.

4.3 Data Processing

To process the data, *Microsoft Excel* was used as the availability of statistics and visualisation tools is appropriate for this experiment.

The Excel spreadsheet, `all_data.xlsx`, can be found in the `output` directory alongside this report, as discussed in Section 2.1. The spreadsheet consists of several tables, the largest of which contains the raw data from the simulation, while the others are used to create charts.

To compare different strategies, an average prediction rate is calculated for each. The average used is the mean, as the median value would likely fall to the misprediction rate for the same tracefile, and the mode would require a compromise to precision to achieve as no misprediction values are identical.

The averages calculated for each prediction strategy avoid the asterisked (*) files in Table 1, as the `google_chrome.out` and `pythonsort.out` are much larger than other files, and `jpegtran.out` and `jpegtran2.out` are highly likely to be IO-bound - excluding these files increases the experiment's validity. Section 4.1 explains this reasoning further.

4.4 Data Visualisation

As the predictors and tracefiles are discrete variables, the majority of data visualisations in this report are column charts.

Figure 2 displays a column chart containing data on the averages of all prediction strategies simulated for this report. The hue and position of columns has been used to indicate data associativity within predictor families (e.g. 2-bit vs. gshare), while saturation and lightness are used to differentiate between predictors within the same family (e.g. 2-bit, table size of 512 vs. 2-bit, table size of 1024).

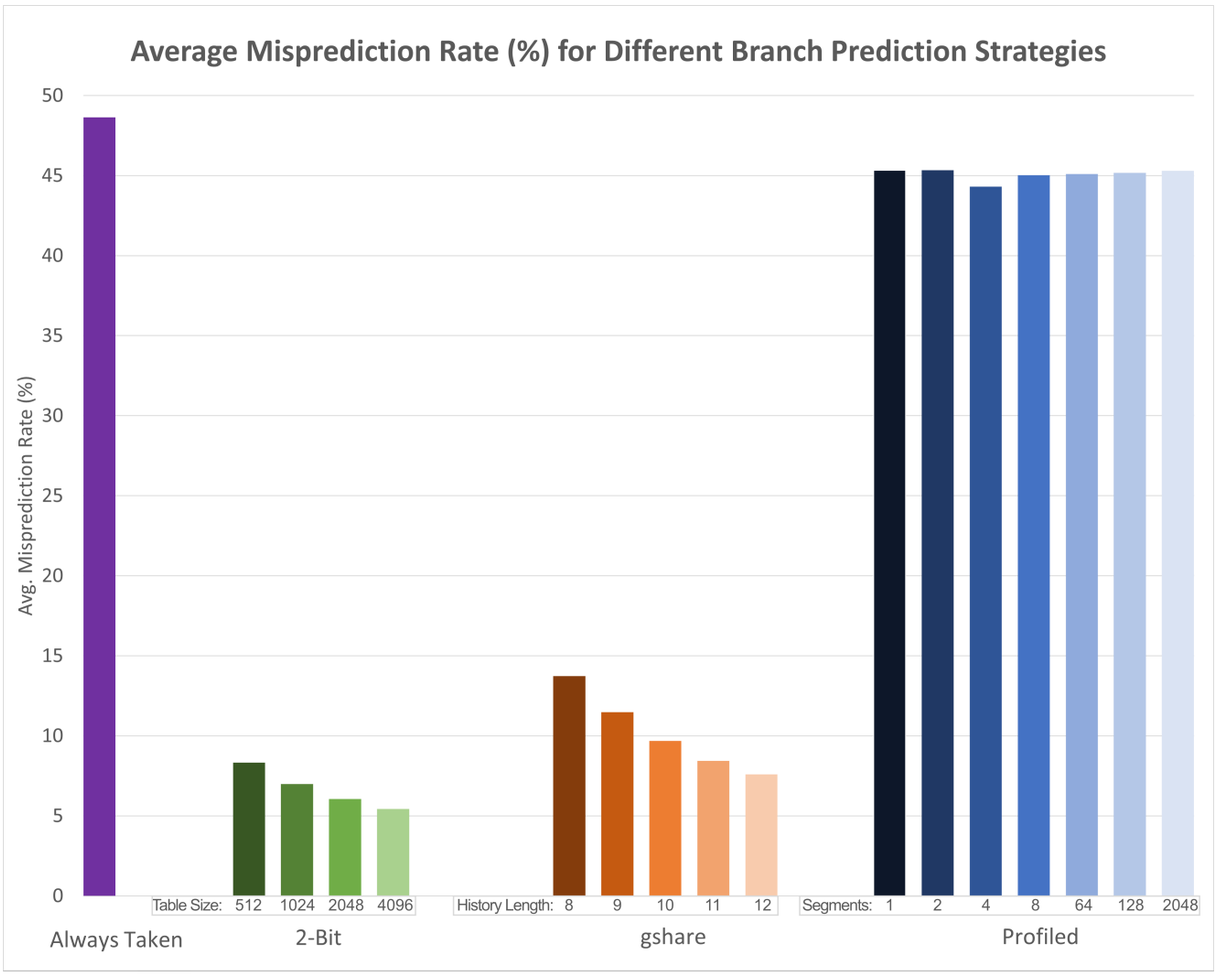

Figure 2: Percentage misprediction for always taken, 2-bit, gshare and profiled prediction schemes.

5 Analysis

5.1 Experiment

Accuracy

The accuracy of an experiment is determined by how close the results are to the true value, which for this experiment is very close. Provided the branch prediction simulator is implemented correctly and the tracefiles are generated correctly, the results will be near-identical to the misprediction rate of each predictor-program combination when executed on a real processing unit.

Although a small difference may be caused by the initial state of each simulator being set differently to the real machine due to previous use of the branch prediction unit, most tracefiles will contain information on over 100,000 branches, meaning the change in results caused by a difference in initial state are insignificant in relation to the final result.

Although some simple tests to verify the fundamental functionality of the simulator were completed during its development, testing every execution path is a theoretically difficult concept and the short time frame given for this practical did not allow for this to take place. A less-than 100% test coverage for the simulator used potentially reduces the experiment's accuracy as the simulated predictor may contain an error which causes it to behave differently than a correctly-implemented predictor in a real processing unit.

Precision

The precision of an experiment is determined by how close repeats of the same experiment are to one-another, which for this experiment may vary.

Provided the simulator used in the experiment is implemented correctly, overcoming the challenges discussed in the *Accuracy* section of this analysis, simulations for each predictor-tracefile combination should have 100% precision. This is because step 3 of the experiment outlined in Section 3.2 specifies that predictors should be returned to their initial state, meaning identical results should be generated for each simulation. An exception would be any predictor which uses a stochastic prediction mechanism, where the results may vary due to statistical variation.

Although simulations for each predictor-tracefile combination are likely to return identical results, the precision for each predictor-program combination is likely much less. Each program executed to generate tracefiles in step 2 of the experiment could potentially take different branches each time. Reasons for this include background processes, such as a web browser clearing its cache when loading the same web page, or a different program input, such as a different image for an image compression program. The potential for difference in tracefiles generated for the same programs gives rise to a potential for different misprediction rates for each predictor-program combination, giving a lower precision.

To investigate the differences between tracefiles for the same programs, simulations for `jpegtran.out` and `jpegtran2.out` were run for three different branch prediction strategies - always taken, two-bit with a table of 4096 entries, and gshare with a table of 4096 entries. The reason why these two files were chosen is because they are the only two files out of the selection provided for this practical which are presumably generated using the same program.

The results from this investigation are visualised in Figure 3, where it can be seen that the misprediction rate differs significantly between the two files for every predictor used. This suggests that differences in tracefiles during the generation step of this experiment could significantly reduce its precision. However, `jpegtran` provides multiple different functionalities which could have been used during tracefile generation, making this investigation inconclusive.

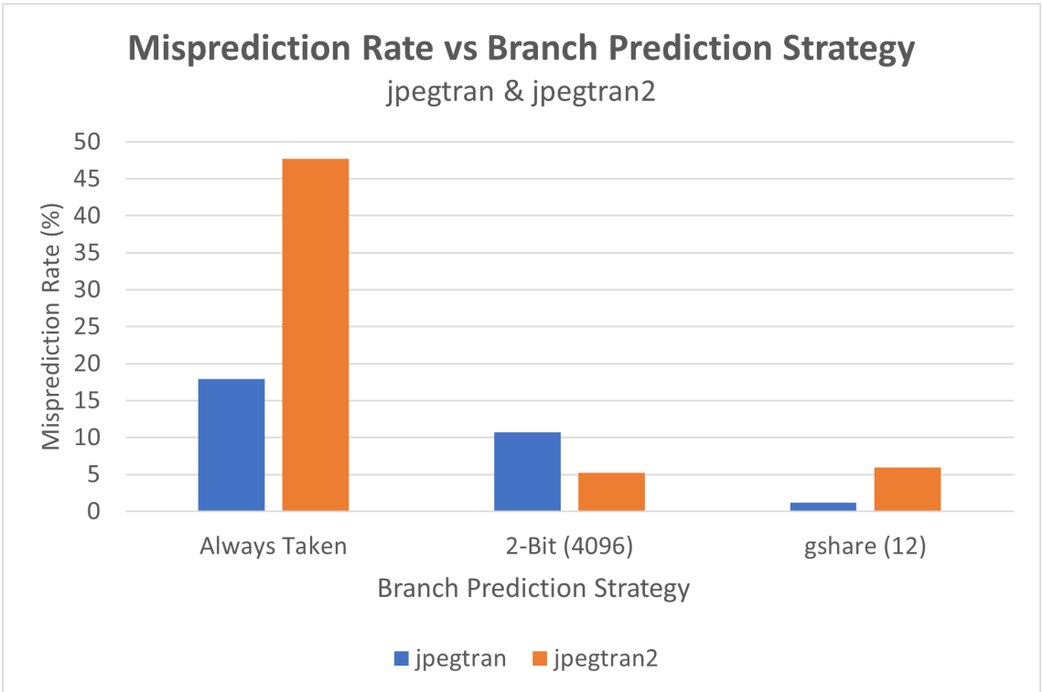

Figure 3: Misprediction rates for `jpegtran.out` and `jpegtran2.out` when simulated with different predictors

Internal Validity

As step 3 in the experiment specifies the same, unmodified tracefile should be used for each simulation in the same repeat, as well as the reversion of each predictor to its initial state after a simulation, each simulation is independent from one-another, promoting internal validity for the experiment.

Other factors which increase internal validity for the experiment are steps 4 and 5. Step 4 increases validity as the results gathered can be cross-checked with other results using a different simulator, increasing the chance of discovery for mistakes in simulator implementations. Step 5 increases reliability in the results and therefore internal validity, as outliers and implementation mistakes are more likely to be discovered.

Unfortunately neither of step 4 nor step 5 were completed when executing the exeperiment for this report, as the time frame set did not allow for multiple simulators to be implemented, and the provided tracefiles did not contain repeats for any program except for `jpegtran`.

External Validity

The external validity is how well the experiment relates to other similar experiments and real-world use cases, which for this experiment is dependent on the selection of programs used to generate tracefiles in step 2.

As discussed in the *Accuracy* section of this analysis, the simulations for this experiment are likely to be near-identical to real processors, promoting external validity. However, the choice of programs used to generate tracefiles in step 2 has the ability to greatly reduce external validity.

Generating tracefiles from programs which are considered in other branch prediction research and processor development promotes external validity, as this research and development can be more easily compared to the data and analysis associated with this report. An example of this is choosing CPU-bound programs over IO-bound programs, as discussed in Section 4.1.

The execution of the experiment outlined in Section 3.2 for this report focuses on general purpose processors, but this could reduce external validity in a setting where the branch prediction strategies of processors used in supercomputers are considered.

Another consideration for external validity is the file trimming which may be performed in step 2 of the experiment outlined in Section 3.2. By reducing the file size manually, there is potential for the tracefiles to no longer be representative of real programs.

To investigate this further, simulations for `google_chrome.out` and `google_chrome_cut.out` were run using three different branch prediction strategies - always taken, two-bit with a table of 4096 entries, and gshare with a table of 4096 entries. The results of this investigation are displayed in Figure 4, where it can be seen that the misprediction rate of the cut and uncut files were very similar for all three prediction strategies, suggesting that cutting the files does not decrease the external validity for the experiment.

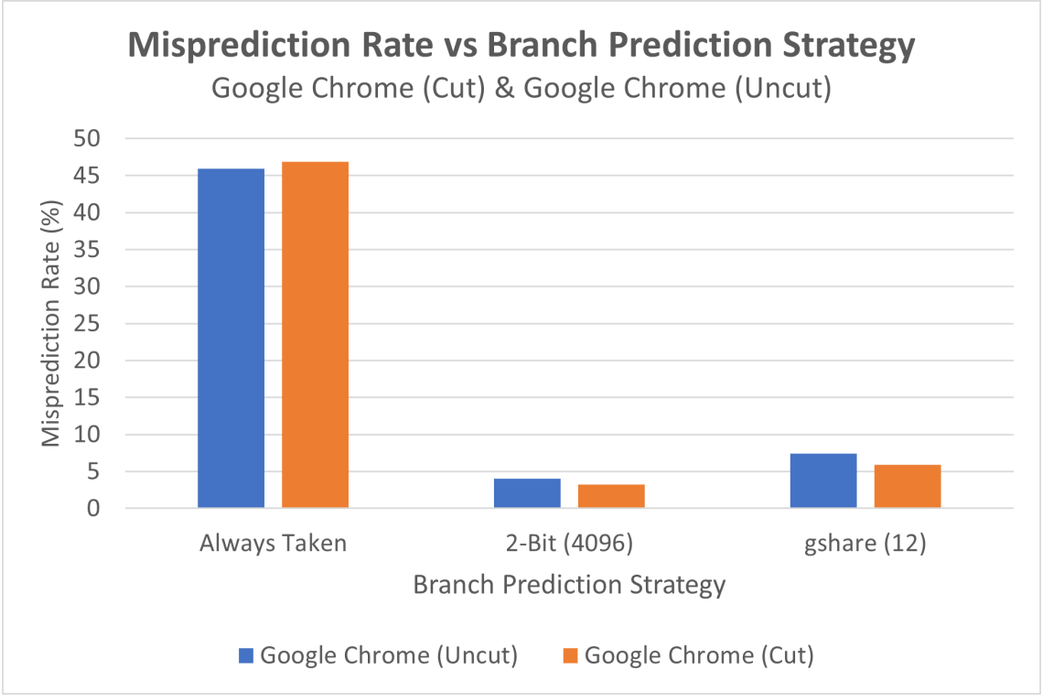

Figure 4: Misprediction rates for `google_chrome.out` and `google_chrome_cut.out` when simulated with different predictors

Reproducibility

Reproducing the experiment in the same conditions as performed in this report is near-impossible. Accessing the identical hardware and software used to generate the tracefiles would be very difficult to achieve, and even then the tracefiles for the same program are unlikely to be identical. Different branch addresses may be used, leading to different behaviour in predictors, and programs may be passed different parameters depending on the current system state or supplied input.

However, although exact replication may be an issue, I believe reproducing the experiment outlined in Section 3.2 to achieve the general trend represented in the results of another experiment is relatively straight forward, as the specifics of software and hardware are unlikely to affect the results substantially.

Another consideration for the reproducibility of this experiment is the challenges which must be overcome in relation the the generation or acquisition and the implementation or acquisition of tracefiles and a simulator respectively, particularly in relation to ensuring the simulator is correctly implemented, as discussed in the *Accuracy* section of this analysis.

5.2 Results

Always Taken Predictor

The purple column in Figure 2 represents the always taken predictor, where it can be seen that this strategy achieves a 48.6% average misprediction rate.

Although it may seem counter-intuitive that the misprediction rate of an always taken predictor is less than 50%, there are usually more taken branches than not-taken branches in a program [5], making an always taken prediction scheme superior to an always not taken prediction scheme in terms of prediction success rate.

Although Figure 2 demonstrates that an always taken predictor performs worse than all other predictors used in this experiment, the simplicity of an always taken predictor could make it a viable choice for cheaper or power-conservative processors. An always taken predictor requires no read-write memory, reducing the cost of a processor implementing this strategy in comparison to others, and the reduction in complexity will also reduce power consumption in comparison to other strategies.

However, although the reduction in complexity will likely reduce power consumption in a processor implementing an always taken strategy, the difference in the number of mispredicted branches will increase the total power and energy consumption. A processor implementing a two-bit strategy may use slightly more power computing the prediction, but will often require less power overall than a processor implementing an always taken strategy because of the superior prediction success rate.

Two-Bit Predictor

As visualised in Figure 2 by the green columns, the two-bit branch prediction strategy performed the best in terms of prediction success rate out of all of the strategies used in this experiment.

Table 2 displays the average misprediction rates measured for the simulation of two-bit predictors with various table sizes. It can be seen from this table and Figure 2 that the larger the branch prediction table used, the lower the misprediction rate. However, increasing the branch prediction table size also increases the required amount of read-write memory, leading to an increase in cost to develop a processor using the strategy.

Table Size (Number of Entries)	**Average Misprediction Rate**
512	8.32%
1024	6.99%
2048	6.05%
4096	5.42%

Table 2: Chosen tracefiles for experiment. Asterisked (*) files are not included when calculating the average misprediction rate.

Although the cost of a two-bit predictor may be more than an always taken predictor due to the requirement of read-write memory, the performance gain by using even a simple two-bit predictor over an always taken predictor leads to two-bit predictors being preferred overall. Two-bit predictors are also much simpler than many other predictors which perform similarly (e.g. gshare), therefore making two-bit predictors the most popular branch prediction strategy in real processors [6].

gshare Predictor

As visualised in Figure 2 by the orange columns, the gshare predictor significantly out-performs the always taken predictor, but it does not match the two-bit predictor in terms of prediction success rate.

Figure 2 represents different gshare implementations by their history length, but by instead representing gshare implementations by the size of their branch prediction table, the two-bit and gshare strategies can be directly compared. The table size for a gshare predictor is 2^n where n is the length of the history when represented as binary.

Figure 5 displays a direct comparison of the misprediction rates generated by the simulation of two-bit and gshare predictors when using the same sized branch history tables. It can be seen that for all table sizes, the two-bit predictor (blue) outperforms the gshare predictor (orange) in terms of average misprediction rate.

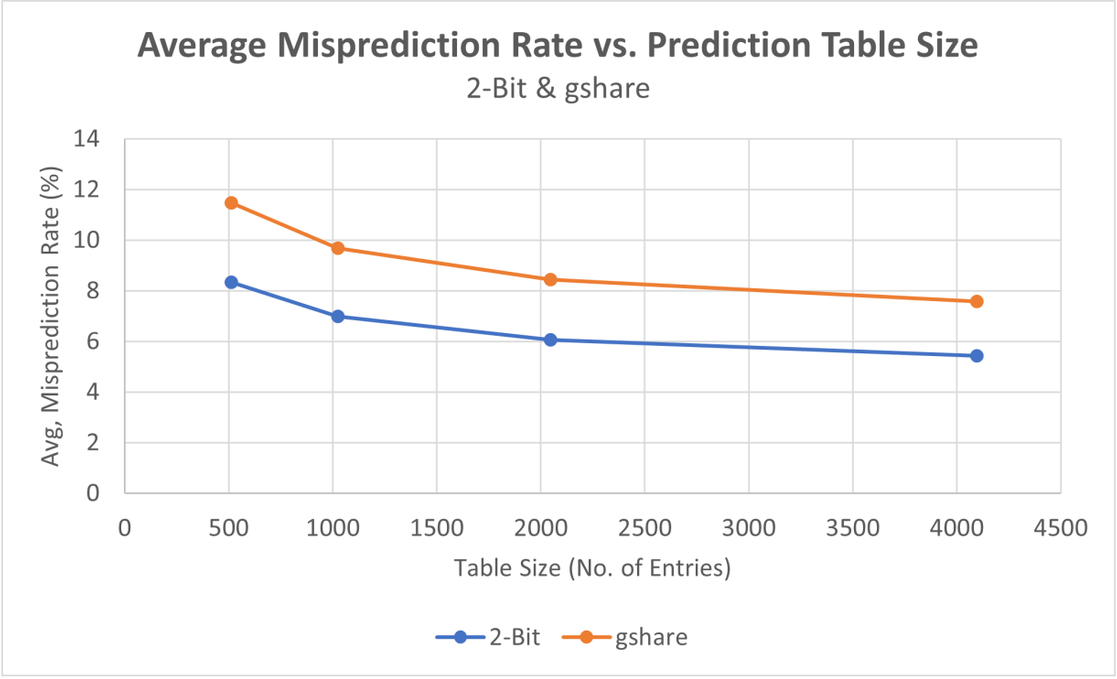

Figure 5: Average misprediction rates for two-bit and gshare predictors using different branch prediction table sizes.

The results seen in Figure 5 seem counter intuitive, as gshare is implemented in modern computer processors [7] which would be a waste of resources if it is consistently out-performed by two-bit predictors. Investigating this further, the original gshare paper provides data that suggests two-bit predictors perform significantly worse than gshare predictors [2]. This data can be seen in Figure 6 of this report, where "bimodal" is used in place for "two-bit".

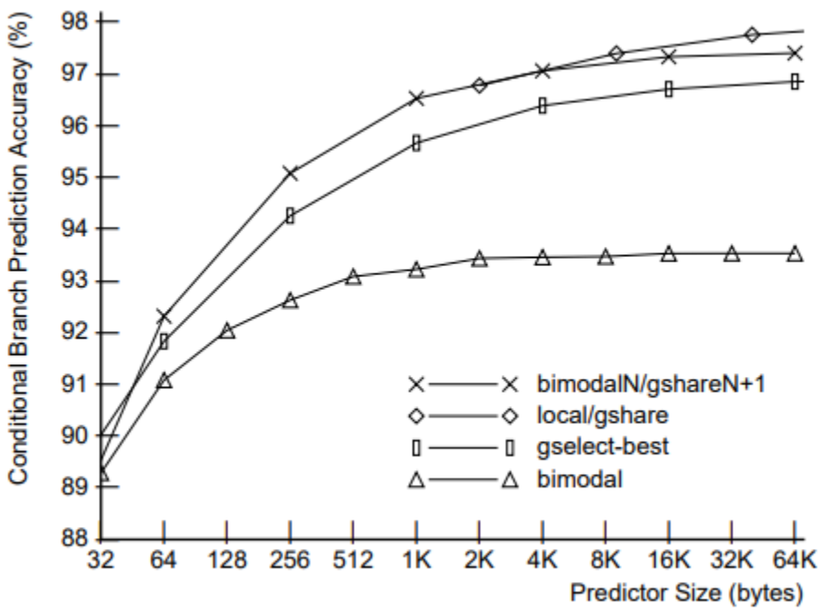

Figure 6: Conditional branch prediction accuracy vs. predictor size. Sourced from Figure 16, Combining Branch Predictors, Scott McFarling [2]

A major difference in the experiment used in McFarling's paper in comparison to the experiment completed for this report is the choice of programs used. Figure 2 in McFarling's paper lists the programs used for simulation, all of which are very mathematically focused, which contrasts with the wide variety of general purpose programs used for this report.

However, Figure 7 visualises the misprediction rates of the two-bit and gshare predictors for all files individually, where both predictors are using the same table size, and it can be seen that the two-bit predictor outperforms gshare even in the most mathematically oriented programs used in the experiment for this report. This suggests that this experiment has flaws with respects to accuracy and validity, which could improved by taking repeats of tracefile generation, and using a different selection of programs.

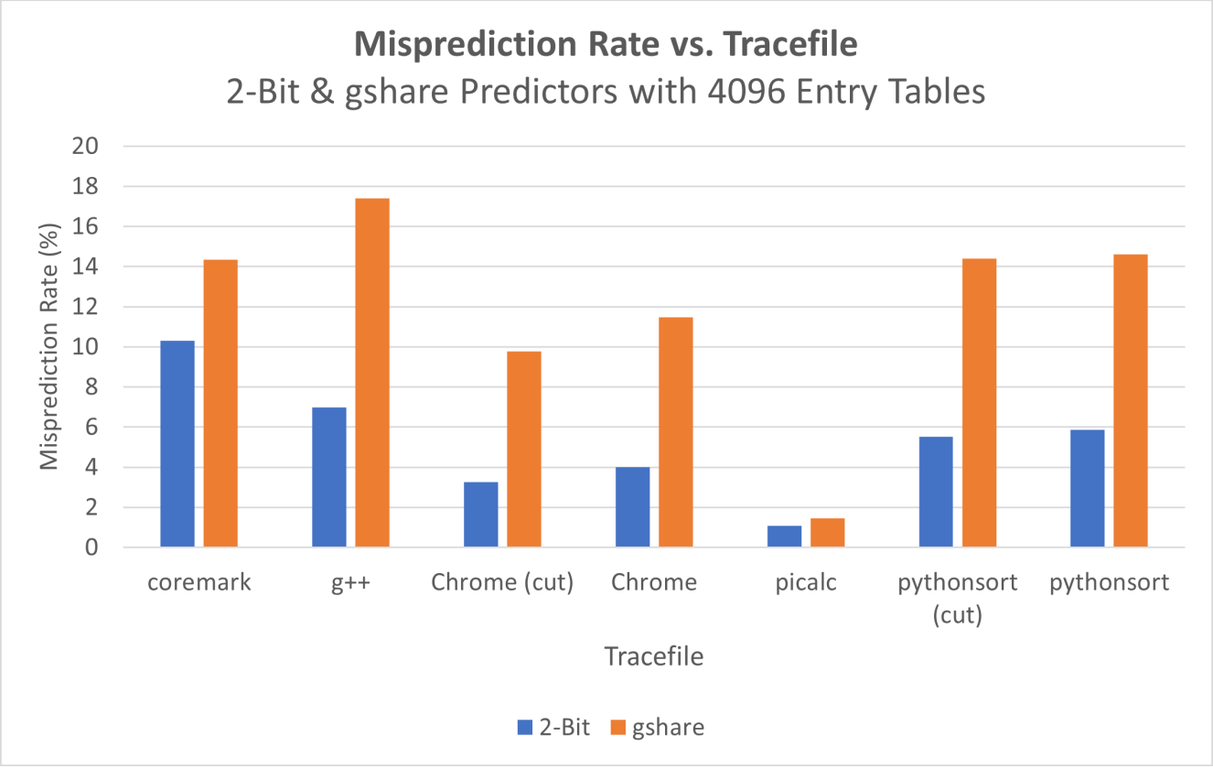

Figure 7: Misprediction rates for two-bit and gshare predictors with tables of 4096 entries using a variety of files.

Profiled Predictor

The blue columns in Figure 2 represent the average misprediction rates of the profiled predictor outlined in Section 2.3 when using various numbers of segments.

It can be noted from Figure 2 that all simulations of the profiled predictor outperform the always taken predictor in terms of the prediction success rate, suggesting that the profiling performed affects the success rate of the predictor positively.

Although the single-segment profiled predictor acts like an always-taken predictor for programs which have more taken branches than not taken, the single-segment profiled predictor will use an always not taken strategy for files which have more not taken branches, explaining why it outperforms the always taken predictor on average in terms of prediction success rate.

Figure 8 visualises the misprediction rate of the profiled predictor when using different numbers of segments. The intuitive trend for this chart would be the more segments which are profiled, the finer grained the prediction, and therefore the more successful predictions made. However, this is not the case, as increasing the number of segments past 4 increases the misprediction rate, meaning there have been fewer mispredictions made.

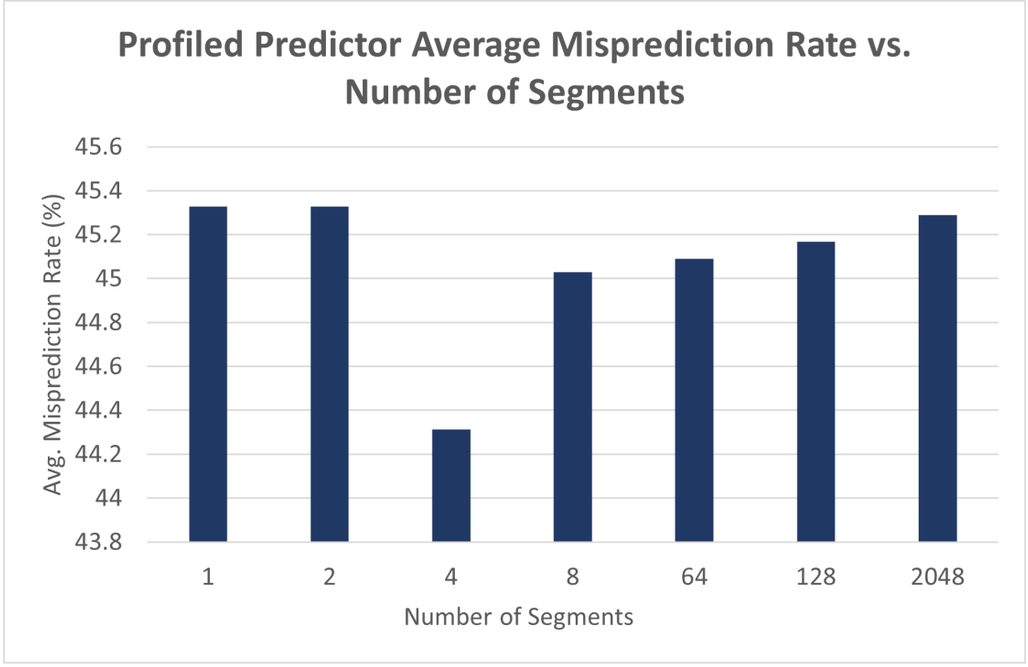

Figure 8: Average misprediction rate for the profiled prediction strategy with various numbers of segments.

One explanation for the trend shown in Figure 8 is that there are outlier results which are influencing the averages. To explore this idea, Figure 9 displays the averages excluding `coremark.out`. It can be seen that the trend has completely changed, suggesting that the results from the `coremark.out` simulations were in fact outliers. This leads to further questions about other tracefiles causing outliers, and promoting the idea that repeats should have been taken.

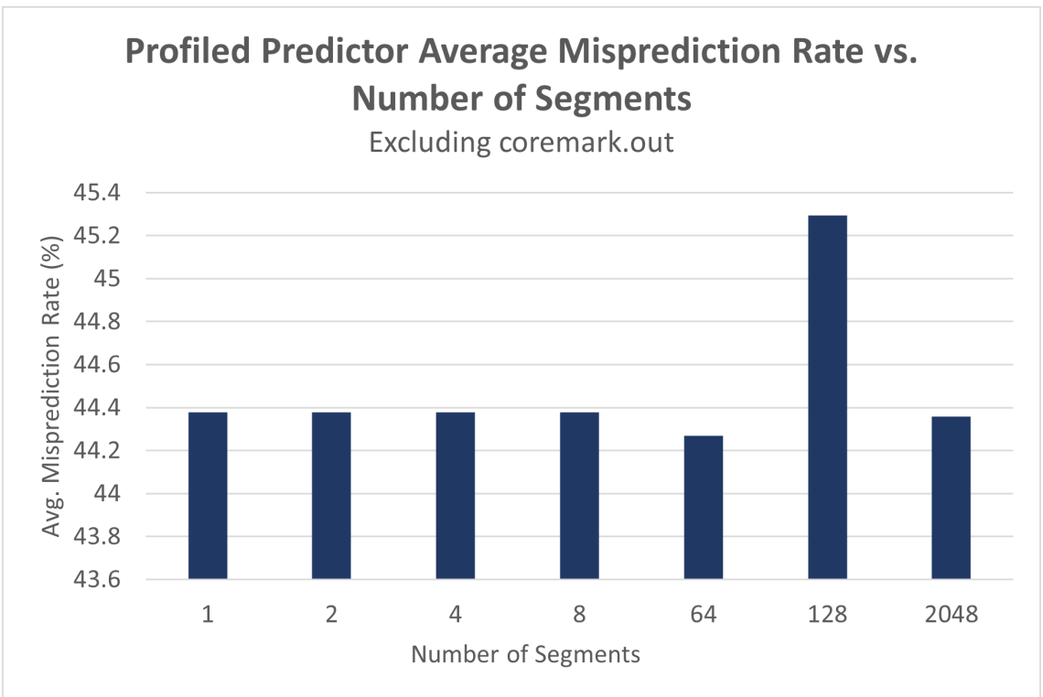

Figure 9: Average misprediction rate, excluding results from `coremark.out`, for the profiled prediction strategy with various numbers of segments.

It can be seen from Figure 2 that the profiled predictor used in this report is inferior to both a two-bit predictor and gshare. Although this is the case, in the unique circumstance where read-only memory is in abundance and no read-write memory is available the profiled predictor outlined in this report would be the preferred predictor, as the profiled predictor is superior to an always taken predictor without the need for any read-write memory.

6 Experiment Improvements & Further Investigations

The most important improvement which could be made to the experiment performed for this report is to take repeats. Section 5.1 outlined issues relating to precision and validity of this report, many of which are caused by the lack of repeats in generating and simulating tracefiles.

Another change which could improve the external validity of the exeperiment is the choice of programs used to generate tracefiles. As noted in Section 5.2, the choice of programs used in McFarling's paper is significantly different than the selection used in this report, which may be a reason for the unpredictable results. By aligning the program choices more closely, this report could be cross-checked with academic papers, increasing the external validity of the results.

Further investigations for this experiment could involve analysing the variance in prediction rate between tracefiles generated from the same program, and simulating more branch prediction strategies, such as those which use stochastic or machine learning techniques.

7 Evaluation & Conclusion

Overall I believe I have been successful with this practical. I have designed and conducted a thorough investigation into the misprediction rates of different branch prediction strategies.

Reflecting on the learning objectives for this practical listed in the specification, I have learned a significant amount about simulating aspects of computer architecture, and the importance of branch prediction for processor performance.

To summarise the discoveries in this report, gshare and two-bit prediction strategies are significantly superior to an always taken strategy, and repeats are very important when designing a practical, as they significantly affect the accuracy and validity of an experiment.

References

[1] "Week 5 Lecture Slides - Branch Prediction, CS4202 Computer Architecture, University of St Andrews School of Computer Science. Accessed 10 Oct 2020." [Online]. Available: https://studres.cs.st-andrews.ac.uk/CS4202/Lectures/05%20-%20BranchPrediction.pdf

[2] S. McFarling, "Combining branch predictors," 1993.

[3] "Chrome Prerendering, The Chromium Projects. Accessed 15 Oct 2020." [Online]. Available: https://www.chromium.org/developers/design-documents/prerender

[4] "Browser Market Share Worldwide, StatCounter Global Stats. Accessed 15 Oct 2020." [Online]. Available: https://gs.statcounter.com/browser-market-share#monthly-202008-202008-bar

[5] "Lecture 5 - Handling Hazards, INF3 Computer Architecture Lecture Slides, University of Edinburgh, School of Informatics. Accessed 16 Oct 2020." [Online]. Available: http://www.inf.ed.ac.uk/teaching/courses/car/Notes/2017-18/lecture05-handling_hazards.pdf

[6] "Dynamic Branch Prediction, ECE 570 High Performance Computer Architecture, Electrical and Computer Engineering, Oregon State University. Accessed 16 Oct 2020." [Online]. Available: http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/

[7] "Intel's Low Power Architecture, Silvermont. Accessed 16 Oct 2020." [Online]. Available: https://www.realworldtech.com/silvermont/2/