Module code: CS4202
Matriculation number: 200017941

Practical 2: Branch Prediction

Overview

The aim of this practical is to simulate branch prediction strategies and to conduct a scientific experiment analyzing the results produced. As part of the objectives, I have implemented four strategies: always taken, 2-bit predictor, gshare and profiled. Then I compared the results produced from running each strategy using a subset of the provided trace files to determine how the performance of each strategy ranked against each other.

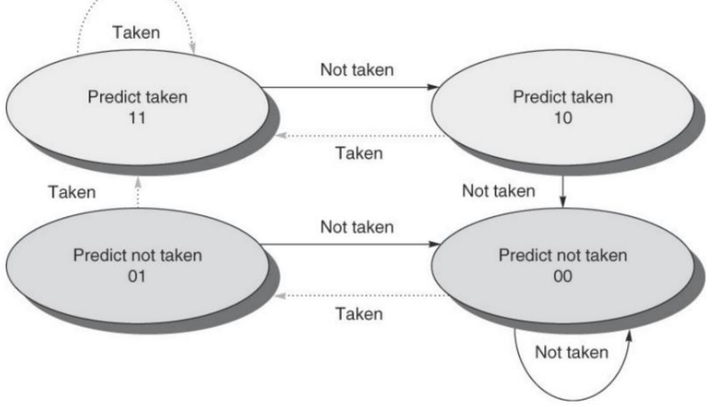
Implementation

Always taken

This branch prediction strategy involves always predicting that a branch will be taken. The implementation for is in the *AlwaysTaken* class and is very straightforward. For each line read in the trace file, the taken field is extracted and is checked to see if it is equal to one. If so then the count for the correct prediction is incremented, otherwise the count for the incorrect prediction is incremented.

2-bit predictor

This branch prediction strategy involves a prediction buffer with a fixed number of entries and is indexed by the lower n bits of each branch address. Each entry stores a 2-bit predictor which is initially the prediction is set to strongly not taken (0). The prediction is only changed to a strongly taken (2) if two consecutive incorrect predictions are made. The behavior of the two-bit prediction is summarized by this state transition diagram:



The implementation for this is in the *TwoBit* class. For each line read, the branch address and taken fields are extracted. The lower n bits are sliced from the address and are converted to an integer which is used to index into the branch prediction buffer. The prediction value is then extracted and compared against the taken value to see whether the prediction is correct or incorrect.

Gshare

This prediction strategy is a more refined version of branch prediction as it is a combination of global and local prediction. It involves a pattern history table which is identical to a branch prediction buffer from the 2-bit predictor strategy. This is indexed using a XOR value of a global history register and the lower n bits of the branch address. The global history register keeps track of the global history of branch outcomes as each bit in the register corresponds to the outcome of a recent branch, the right most bit being the most recent branch outcome. I chose to have the number of bits in the global history register to be equal to the number of bits slicked from the branch address.

The implementation for this is in the *GShare* class and is mostly the same as the implementation for the 2-bit predictor strategy. The only difference is that before indexing into the buffer, after slicing the lower n bits from the branch address, the integer value of the sliced address and the value in the global history register are combined using XOR. Additionally, after each prediction, the register is updated with the taken value of the branch that was just read. This involves shifting all the bits to the left by one and adding the taken value at the start.

Profiiled

This prediction strategy has two phases where the first phase involves a profile run of the code to gather information and the second phase involves an actual run of the code where predictions are made for each branch. During the profile run, for each branch, the number of times it was taken and the number of times it wasn’t taken are recorded. This is stored in a buffer. Then during the actual run, this information is used to calculate the probability that a branch will be taken after indexing into the buffer using the lower n bits of the branch address. A probability greater than 0.5 results in a ‘taken’ prediction being made, a probability less than 0.5 results in a ‘not taken’ prediction being made and a probability of exactly 0.5 results in a guess. The calculation of the probability is straightforward:

$$\text{Probability}(\text{Taken}) = \frac{\text{Number of Times Taken}}{\text{Total Number of Encounters}}$$

The implementation for this is in the *Profiled* class and is generally the same as the other two implementations. The only main difference is that it goes through the trace file twice and stores the counts for the number of correct and incorrect predictions in the buffer. This is used to calculate the probability after indexing into the buffer, before making a prediction.

Experiment Methodology

This experiment aims to demonstrate how each of the implemented branch prediction algorithms perform and how their performances compare against each other in terms of a set of metrics. The first metric is the total execution time of the program with branch prediction. The second and arguably most important metric is the prediction accuracy which is the percentage of correct predictions. These two metrics will be the focus of the analysis.

$$\text{Prediction Accuracy} = \left(\frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}\right) \times 100$$

The third metric is the total cycles with branch prediction which is the total number of cycles it took to execute the program with branch prediction. The calculation for this has two parts. The first part is the total misprediction cycles which is the penalty added from all the mispredictions. The misprediction penalty is the number of cycles the CPU wastes due to the branch misprediction. Assuming a CPU pipeline of 5 stages, 5 cycles are wasted because once a misprediction is detected the 5 stages in the pipeline need to be flushed out which is equivalent to 5 cycles. 2 cycles because after the pipeline is flushed, it takes two additional cycles to refill the pipeline with the correct instruction. Therefore, this is a total of 7 cycles. The second part is the total correct prediction cycles which is the total number of cycles it took to execute all the instructions when the prediction was correct. The correct prediction cost is 1 cycle since a 5-stage pipeline is assumed.

$$\begin{aligned}&\text{Total Cycles with Prediction} = (\text{Number of Correct Predictions} \times \\ &\text{Correct Prediction Cost}) + (\text{Number of Incorrect Predictions} \times \\ &\text{Misprediction Penalty})\end{aligned}$$

The fourth and final metric is the number of cycles per instruction. This uses the total cycles with prediction from the previous metric.

$$\text{CPI} = \frac{\text{Total Number of Cycles for Execution}}{\text{Total Number of Instructions Executed}}$$

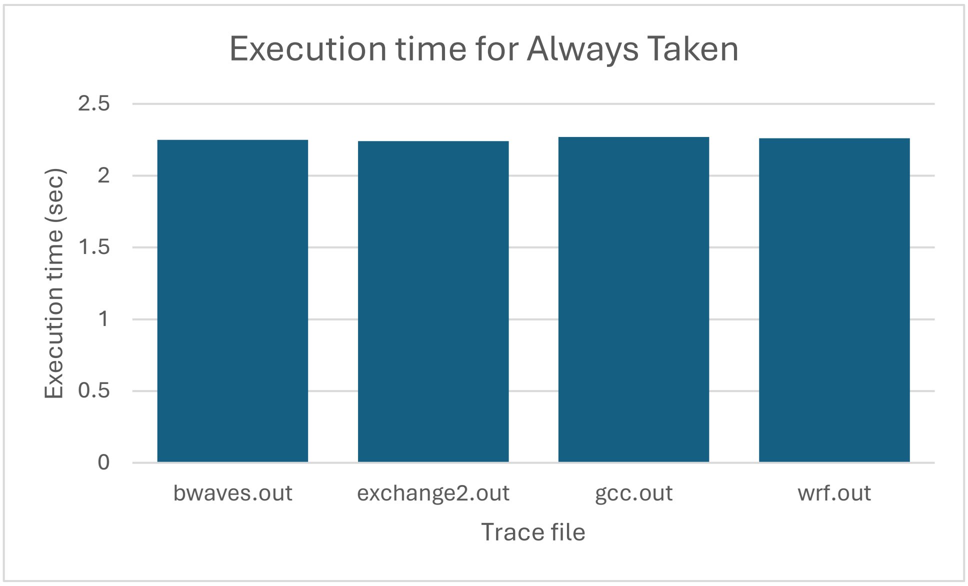
Additionally, for each strategy (except for always taken), the buffer sizes used are 512, 1024, 2048 and 4096 entries. The purpose of this is to assess how changing the size of the buffer affects the performance of the branch prediction strategy.

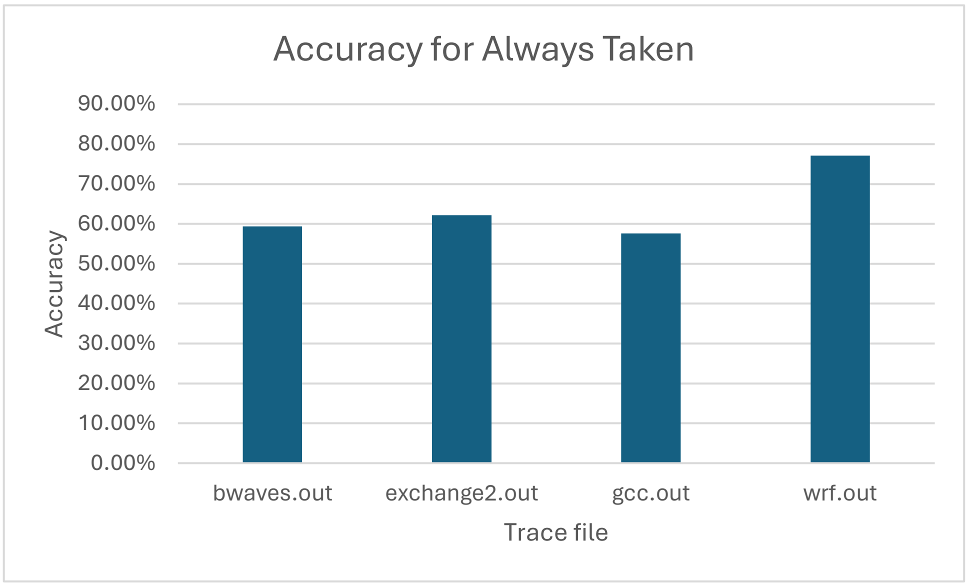
The experimental procedure involved running all four branch prediction algorithms with a subset of the given trace files, and this is repeated for each of the buffer sizes listed above. I randomly chose four trace files to use, and the chosen ones are *bwaves.out*, *exchange2.out*, *gcc.out* and *wrf.out*. For each trace file, I chose to only use the first 5 million branch instructions.

Results

<u>Always taken</u>

Trace file	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	2.25	59.40%	17190452	3.44
exchange2.out	2.24	62.20%	16350110	3.27
gcc.out	2.27	57.60%	17709344	3.54
wrf.out	2.26	77.10%	11868566	2.37





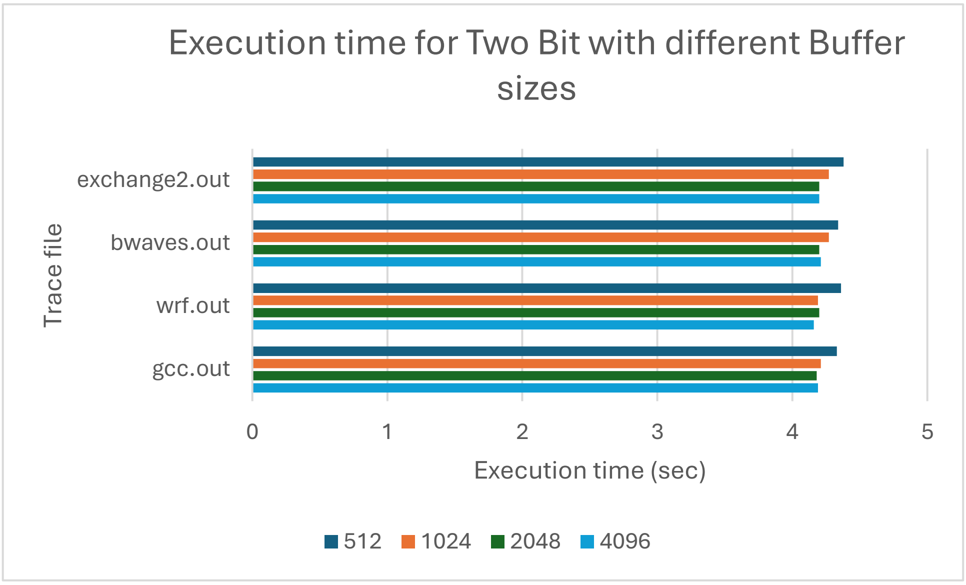
2-bit Predictor

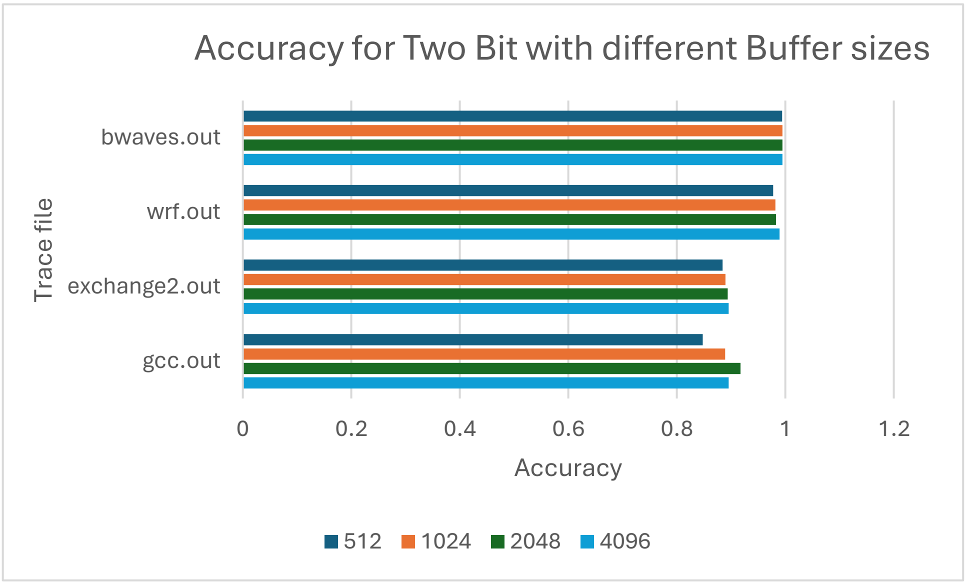
Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	512	4.34	99.42%	5173424	1.03
exchange2.out	512	4.38	88.45%	8463866	1.69
gcc.out	512	4.33	84.76%	9572054	1.91
wrf.out	512	4.36	97.74%	5677922	1.14

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	1024	4.27	99.45%	5164436	1.03
exchange2.out	1024	4.27	89.00%	8301440	1.66
gcc.out	1024	4.21	88.92%	8323898	1.66
wrf.out	1024	4.19	98.17%	5548718	1.11

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	2048	4.2	99.47%	5159426	1.03
exchange2.out	2048	4.2	89.40%	8179166	1.64
gcc.out	2048	4.18	91.72%	7484570	1.5
wrf.out	2048	4.2	98.28%	5516240	1.1

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	4096	4.21	99.48%	5156732	1.03
exchange2.out	4096	4.2	89.55%	8134994	1.63
gcc.out	4096	4.19	89.55%	8134994	1.63
wrf.out	4096	4.16	98.97%	5310200	1.06





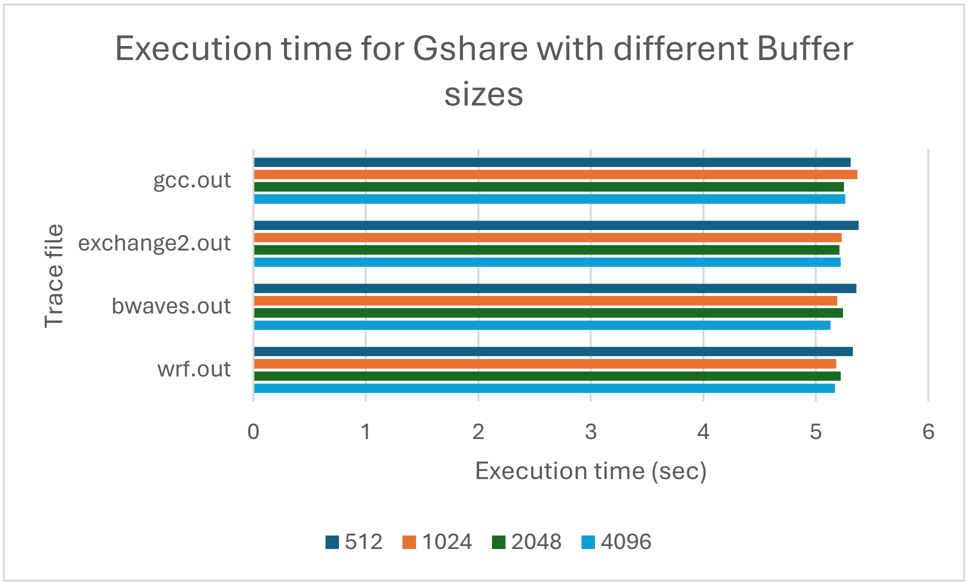
Gshare

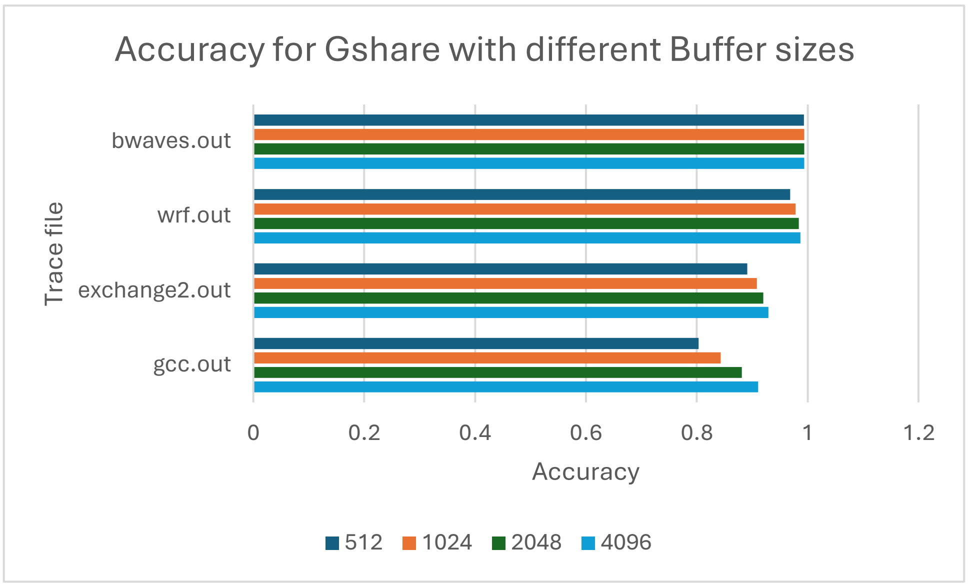
Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	512	5.36	99.33%	5200346	1.04
exchange2.out	512	5.38	89.10%	8268512	1.65
gcc.out	512	5.31	80.34%	10898216	2.18
wrf.out	512	5.33	96.86%	5941874	1.19

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	1024	5.19	99.36%	5191136	1.04
exchange2.out	1024	5.23	90.85%	7746440	1.55
gcc.out	1024	5.37	84.32%	9704366	1.94
wrf.out	1024	5.18	97.83%	5651384	1.13

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	2048	5.24	99.39%	5184338	1.04
exchange2.out	2048	5.21	91.98%	7404884	1.48
gcc.out	2048	5.25	88.10%	8568872	1.71
wrf.out	2048	5.22	98.42%	5473286	1.09

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	4096	5.13	99.39%	5183690	1.04
exchange2.out	4096	5.22	92.94%	7118240	1.42
gcc.out	4096	5.26	91.05%	7685900	1.54
wrf.out	4096	5.17	98.69%	5392850	1.08





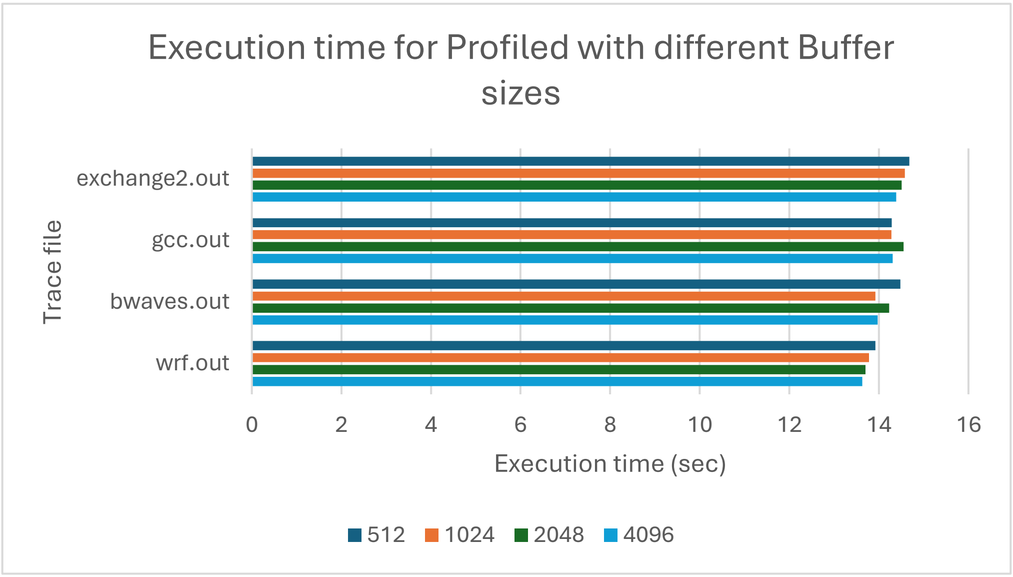
<u>Profiled</u>

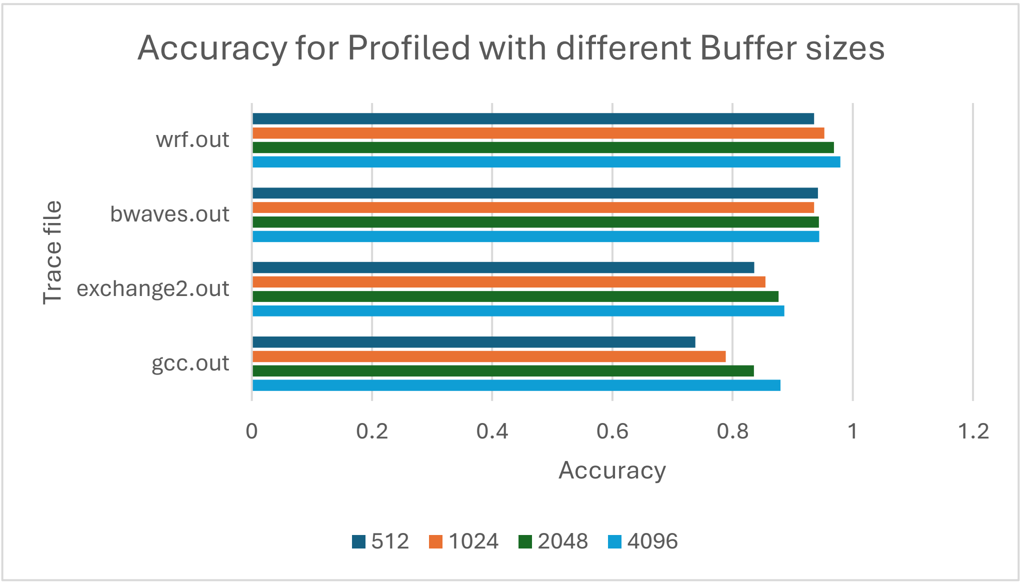
Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	512	14.48	94.21%	6738230	1.35
exchange2.out	512	14.68	83.59%	9924332	1.98
gcc.out	512	14.29	73.82%	12854534	2.57
wrf.out	512	13.92	93.55%	6933878	1.39

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	1024	13.92	93.55%	6715820	1.34
exchange2.out	1024	14.58	85.47%	9357800	1.87
gcc.out	1024	14.28	78.84%	11347286	2.27
wrf.out	1024	13.78	95.28%	6417368	1.28

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	CPI
bwaves.out	2048	14.23	94.34%	6698384	1.34
exchange2.out	2048	14.51	87.67%	8698550	1.74
gcc.out	2048	14.55	83.55%	9934424	1.99
wrf.out	2048	13.7	96.85%	5943860	1.19

Trace file	Buffer size	Execution time (sec)	Accuracy	Total Cycles	PCI
bwaves.out	4096	13.97	94.43%	6672134	1.33
exchange2.out	4096	14.39	88.58%	8426132	1.69
gcc.out	4096	14.31	87.97%	8608298	1.72
wrf.out	4096	13.63	97.92%	5624696	1.12





Analysis

Within each branch prediction algorithm, for almost all the trace files, the results show that increasing the buffer size decreases the execution time and increases the prediction accuracy. This is as expected since a larger buffer means more entries can be stored which would result in more of branches being tracked simultaneously and less conflicts between different branch addresses.

Between the different algorithms, always taken has the worst prediction accuracy across all the trace files, but it also has the fastest execution time by far. This is because of how overly simple the strategy is. It doesn't keep track of how the branch outcomes and always makes the same prediction regardless of the behavior of the program. 2-bit prediction and gshare both produced reasonably high prediction scores which are very similar which makes sense considering how similar both strategies are. Both implementations involve a local predictor to keep track of the history of each branch instruction. However, the only difference is that gshare also maintains a global predictor as well which is a more comprehensive approach. For this reason, gshare is slower since it requires more processing as it maintains more information. And since the prediction accuracy isn't significantly different, there wasn't much of a performance gain from using ghsare compared to two-bit prediction. Additionally, the characteristics of the trace files used in the experiment might also be another indicator as to why gshare didn't consistently perform better. Since gshare is a more comprehensive strategy which combines two types of predictors, it would perform better in more complex scenarios. However, it could be that the trace files used don't have said complex scenarios, at least as indicated by the results produced. Lastly, profiled has by far the worst execution time due to its implementation involving 2 separate runs of the entire trace file. Moreover, it also isn't consistently producing high accuracy scores compared to two-bit prediction and gshare, especially for smaller prediction buffer sizes. This issue arises because profiled is a more static strategy which relies entirely on the observed pattern of each branch. And so, if the buffer isn't large enough to store most of the branch instructions, it would result in a prediction table that doesn't accurately capture the behavior of the program. Every time there is an overlap in the index, the counts for the number of times the branch was taken and the number of times the branch was not taken are combined for all the conflicting branch instructions which makes the calculated probability meaningless.