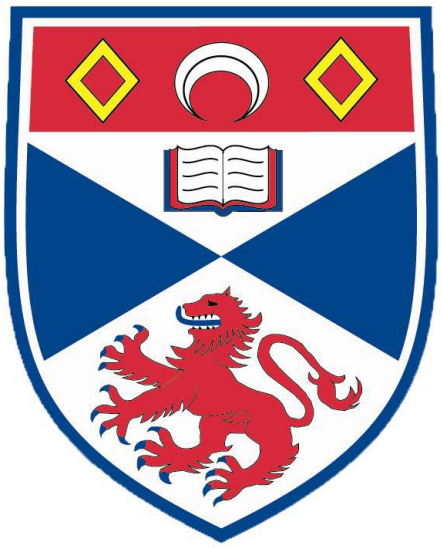


University
of
St Andrews

CS4202 Computer Architecture:

Practical 1 – Branch Predictor

Word Count: 3572

Execution Help

When running the branch prediction strategies, first ensure the files have been compiled using:

*javac *.java*

To run a chosen branch prediction strategy, use the following:

java <FILE NAME> <BRANCH TRACE FILE OR DIRECTORY> <TABLE SIZE (OPTIONAL)>

The four accepted file names are: *AlwaysTaken*, *TwoBit*, *GShare*, *Profiled*

A specific table size can be entered in the command line for those specific results. The four accepted table sizes are: *512*, *1024*, *2048*, *4096*

Introduction

We have been asked to create a series of four branch prediction strategies: the always taken method, standard 2-bit predictor, gshare and a custom profiled approach. For each of these strategies, a final misprediction rate will be produced from a set of branch trace files provided to us. Analysis of the four strategies will then be carried out.

Design & Implementation

I chose to implement my simulators in Java due to my confidence with the language. I was aware that for file management and data handling that a language like Python would have been better suited, but I believed implementing the theory of the prediction strategies would have been slightly more difficult for myself in Python due to my lack of familiarity.

The four strategies are run separately. The entire BranchTraces directory can be run at once, but individual branch trace files can also be run. When multiple files are run, a total average is displayed at the end of the output. This is a weighted average, comprised of the total of every file average acquired, divided by the total number of files looked at. This was chosen over the other possible implementation of total correct predictions divided by the total number of branches encountered as I believe file size should have no impact on the overall average of a system. For example, if a strategy had an accuracy of 100% with a 100-branch long file, but an accuracy of 50% with a 10,000-branch long file, the weighted average would be 75%, whilst the running average would only be 50.5%. The average of a strategy, for me, should be based on a strategies performance across all files equally and due to the sheer size of some of the branch files provided, I feared that certain files would contribute towards the average more than many others if the second method was implemented. By using this weighted average, all files are treated equally, achieving the definition of what I believe the representing average should be. It should be noted that only files containing data that can be acted on will go towards the weighted average. For example, index.txt is not counted in the total file count as it does not contain any branch addresses.

My implementation also allows for every possible table size to be run at once for the three strategies that make use of it.

To aid in the repeatability of certain experiments, the other values for variables can be found commented out beside the initial value. To test each aspect of the experiment, one need only to

comment out the original line and uncomment the line containing an experimental value. I touch on my experiments in the Evaluation section.

For every strategy, the branch addresses and taken/not taken choices are acquired by reading through a file, storing each line's values in respective ArrayLists. These ArrayLists are then what is acted upon within the strategy methods.

The final aspect to note that is applicable to all strategies is that the output for each run is the misprediction rate (% of incorrect predictions). However, if one would like to flip the output to see the correct prediction rate, then they can simply uncomment the line below as this calculates the flipped percentage.

Always Taken

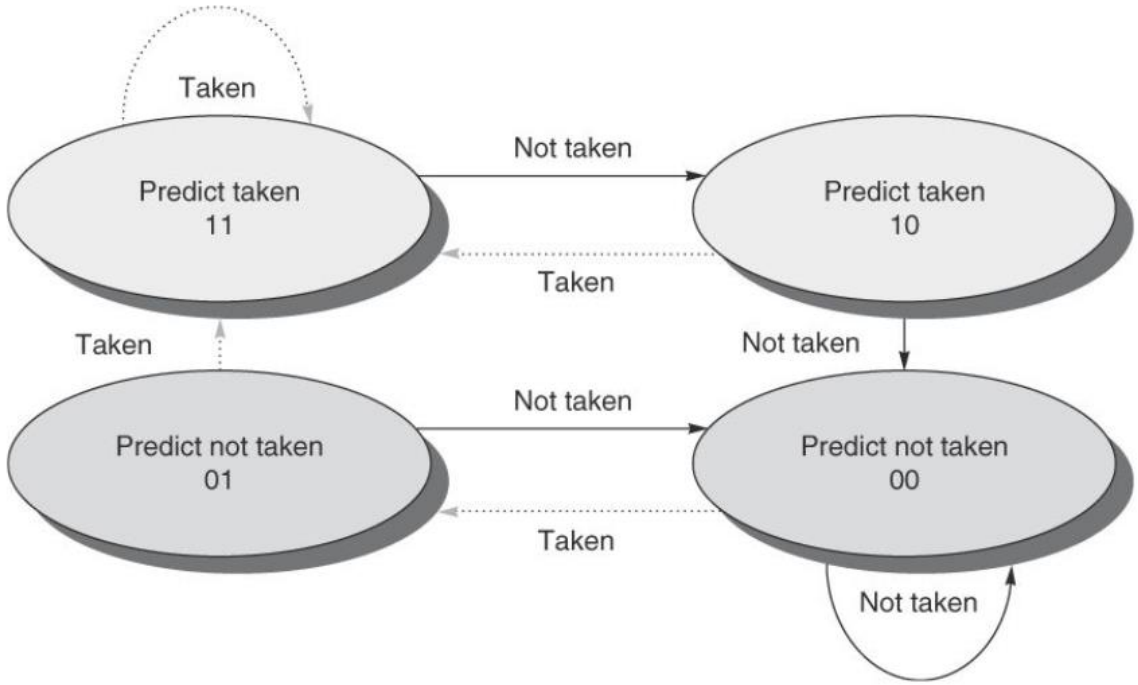
The simplest of the four strategies, the strategy method for this merely consists of a for loop that iterates through each branch within the ArrayList. A check is carried out to see if that branch was taken (== 1), updating the correct counter if the case is true.

Two-Bit w/ Buffer

As this is the first of the strategies where table size comes into play, checks are required for the command line input to see if a table size has been specified and that it is of an acceptable value. If a table size has not been specified, then the strategy is looped for the four possible table sizes.

Once in the strategy method, an array of states the size of the pre-determined table size is initialised as is the buffer size from said table size ($\log_2$(table size)).

For every branch address in the file, the binary equivalent of the branch is acquired before it is reduced to the buffer size. This is then turned back into an integer to get the index of the state array which will be acted on. An initial check is carried out to see if the current state of this index has been defined. If that is not the case, then it is set to the starting state (default 00). The state of the index then goes through a state machine akin to the one found in lectures:



If the branch is taken and the current state is 11 or 10, then the prediction is correct, and the counter is incremented. The same can be said for states 00 and 01 when the branch is not taken.

GShare

GShare follows much of the same structure as the TwoBit strategy, with the only real difference being the implementation of the global buffer. The global buffer is initialised at 0 at the start of the strategy method. The binary for the branch address is acquired like in the TwoBit strategy. However, this time before the binary is turned back into an index for the state array, it is combined with the global buffer using XOR. This produces a different index from TwoBit, and this is used to get the state that will be acted upon, going through the same state machine as with TwoBit. Whether the branch address was taken or not is then incremented onto the global buffer for the next branch address in the file to utilise. It is worth noting that the global is limited to the size of significant bits from the table size, however this size can be reduced further and is the focus of one of the experiments in the Evaluation section.

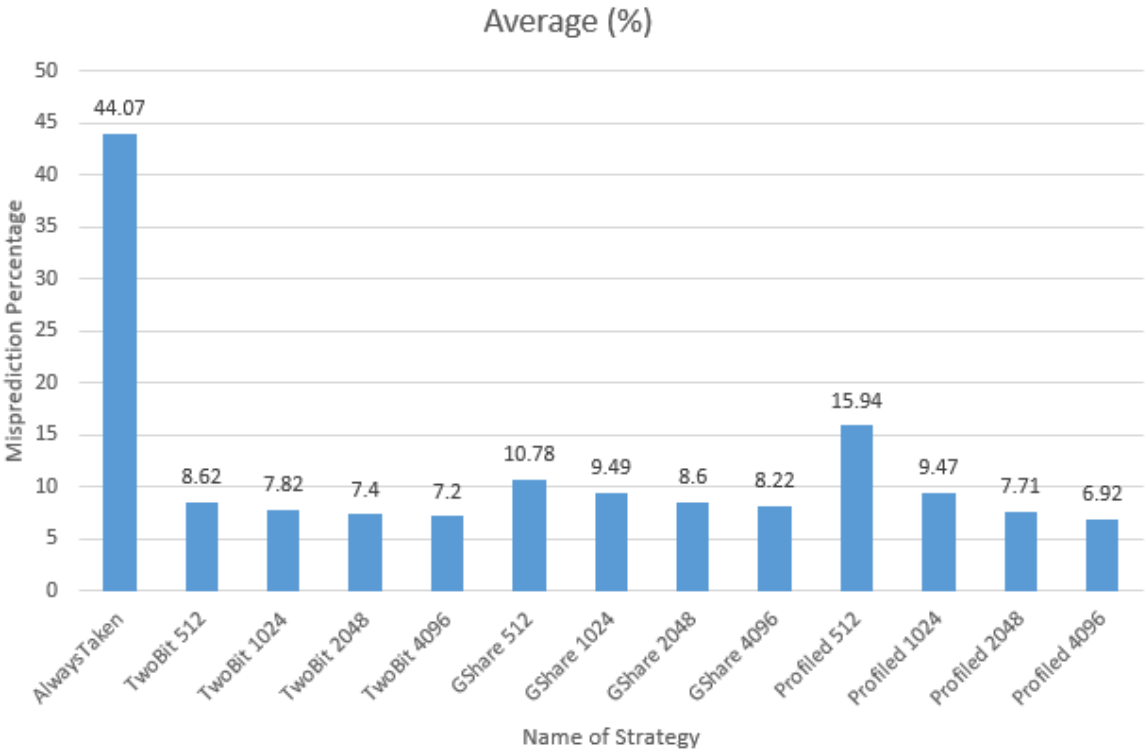
Profiled

For my profiled strategy, after struggling to implement some more difficult theorised strategies, I decided to simply see if specific addresses are more often taken or not. If an address is profiled as being more often taken, then during the full run after the profiling stage, it will be predicted as taken. The opposite is true for branches that are more often not taken.

In terms of the actual implementation, a HashMap is used where the keys are the masked branch addresses. Like the previous two strategies, the addresses are once again masked using the significant bits from the table size. The value for each key is a counter used to track how often the branch is taken compared to not taken, with the value being incremented with every 1 encountered and the value being reduced with every 0 encountered for the branch address. The profiling stage lasts until the HashMap reaches the size of the pre-determined table size. At this point, the profiling stage ends via the use of a 'break' and the predicting begins. If a branch address is not found within the HashMap, then the predictor just assumes taken. This is another experiment that can be found in the Evaluation section. The varying HashMap size from the table size lends itself to the idea of seeing if allowing more time and resources for the profiling stage helps reduce the misprediction rate and thus, I believe my implementation for a profiled branch predictor is more than suitable and opens itself up to multiple possible experiments.

To implement this strategy into a real-world architecture, a register or some other form of storage would be required to hold the binary representations of addresses already encountered during the profiling stage. When this register of pre-defined size is then full, the profiling stage ends and the 'real' run can begin

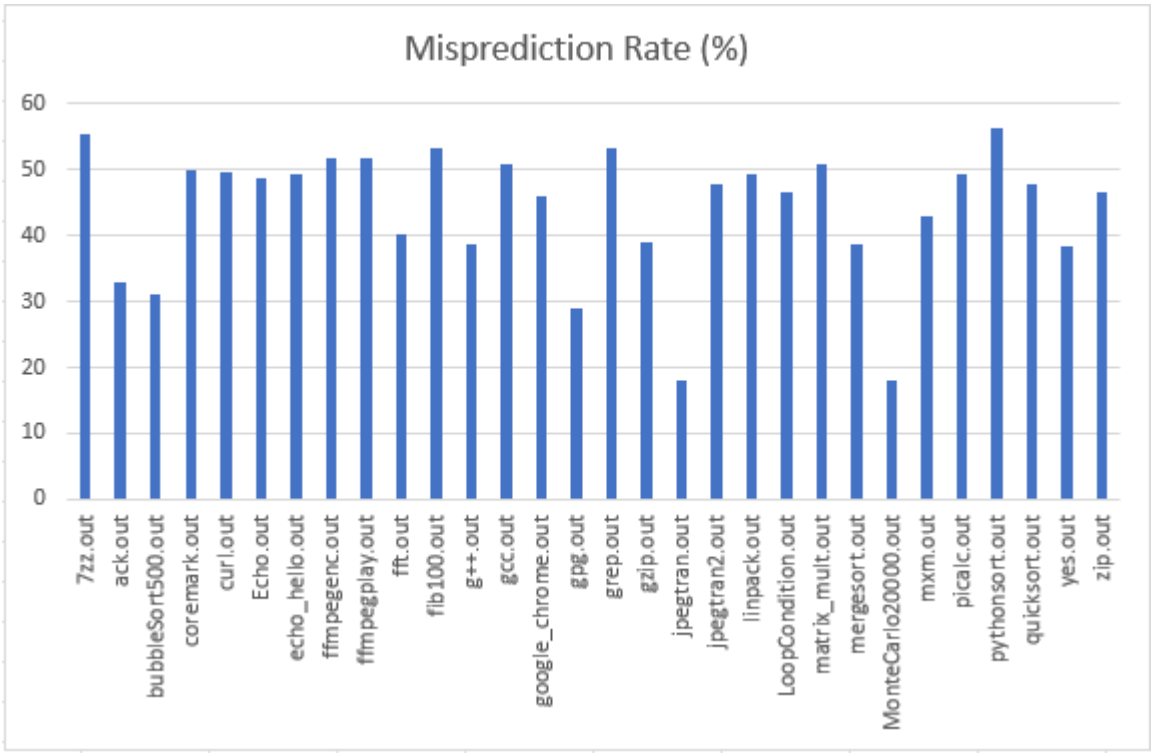
Evaluation



Graph 1: Average Misprediction Rate for Strategies w/ Default Values

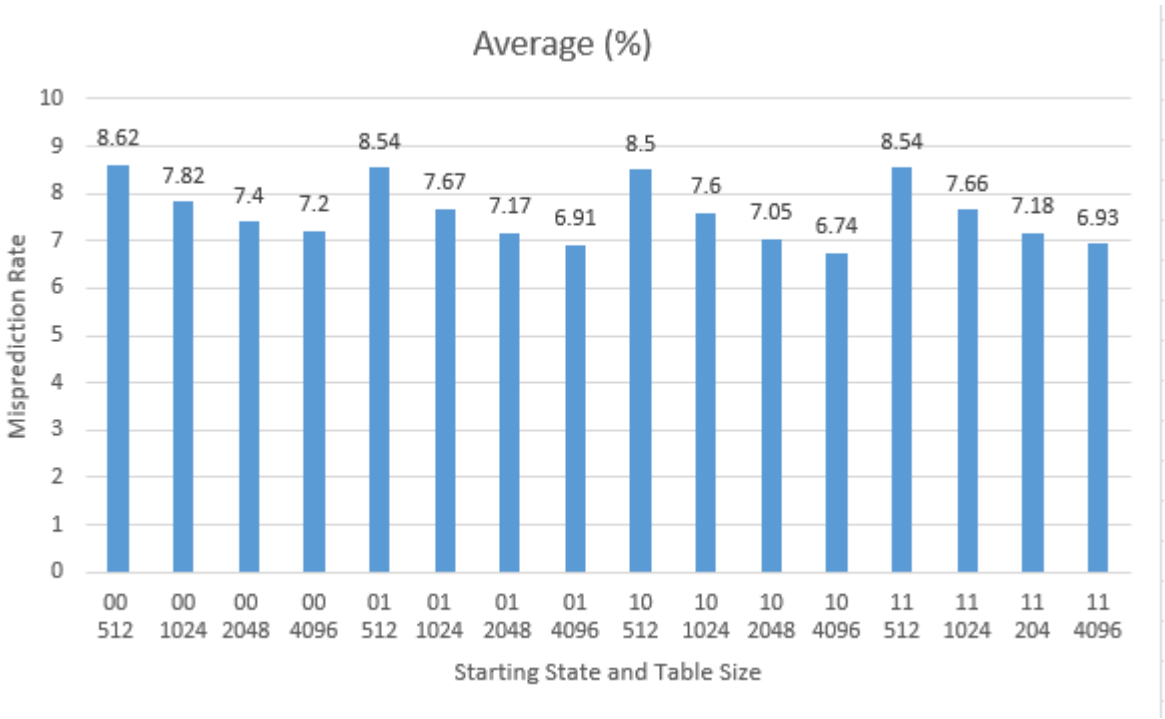
From Graph 1, one can see that Always Taken is massively inaccurate when compared to any other strategy at any of the allowed table sizes. This makes sense due to the almost 50/50 nature of the strategy. I thought this strategy completely nonsensical as, logically, all averages produced by it should be within the range of 50% . Therefore, I decided to try and find some uses for the strategy to justify its existence. I learned about SPARC (Scalable Processor Architecture) and MIPS (Microporcessor without Interlocked Pipelined Stages) and how both follow a single-direction static branch prediction method, forever predicting branches will not be taken, fetching the next instruction. This means that only when the branch is evaluated and found to be taken will the instruction pointer not be set to the next address. Both architectures were released almost 40 years ago however, and thus I believe Always Taken to be an obsolete branch prediction strategy in today's architectural systems; only being used as a fallback method by companies such as Motorola and Intel.

Something that should also be taken note of from this graph is the trend of the other three strategies reducing their misprediction rate as their table size increases. This will be touched on further in the strategies' own graphs.



Graph 2: Always Taken Misprediction Rate for All Files

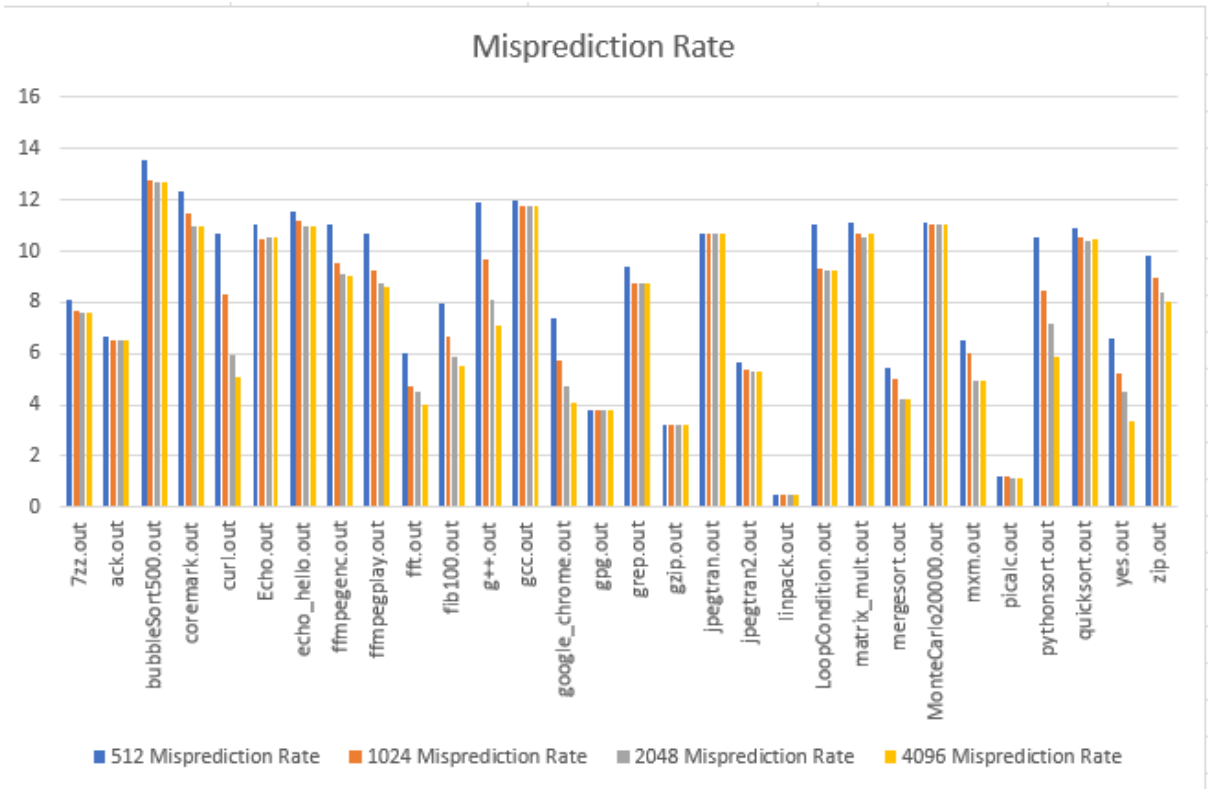
I believe Graph 2 merely shows the random and widely inaccurate nature of the Always Taken strategy, with the two slightly lower misprediction rates being able to be explained away by simply having a much larger share of 1s than 0s. From Graph 2, one can also surmise that there was no file in the BranchTraces folder to have an overwhelming ratio of not 0s to 1s.



Graph 3: TwoBit misprediction rate at each buffer size for the four possible starting states

For this experiment, the starting state that an index is set to would be changed between the four possible states (00, 01, 10 and 11) before the code was recompiled and run.

From Graph 3, one can see that there is a negligible difference between the four possible starting states. I believe this shows that no matter what the starting state is, the TwoBit strategy quickly works towards the accurate states and thus, the starting state has no real impact after the branch address has been handled for the first couple of times. A slightly more notable aspect of Graph 3 is that one can see the continual decrease in the misprediction rates as the table size increases for every starting state. This makes sense as with the increasing table size, a larger number of significant bits are allowed. This increases the size of the mask that addresses are cut to, thus less addresses use the same index within the state table and therefore more accurate predictions can be made.



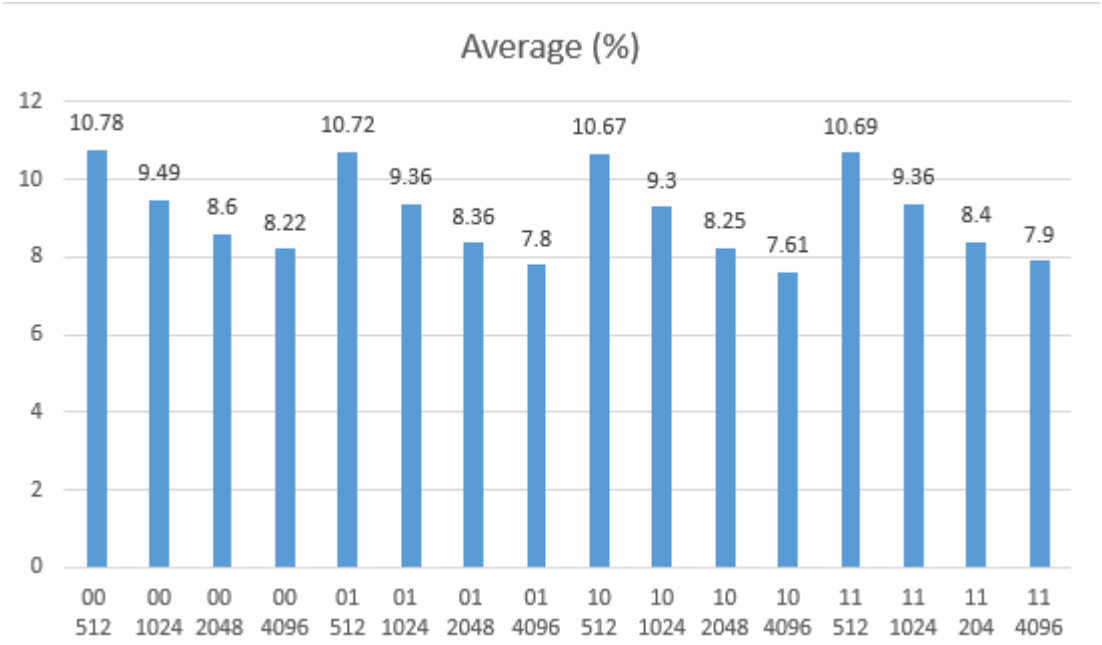
Graph 4: Two Bit Misprediction Rate for All Files at Default Starting State (00)

Graph 4 shows the impact of increasing the table size on the misprediction rate for all files. There is an interesting mix of results here as for some files (see zip.out and google_chrome.out) there is a clear decrease as the table size increases, whilst for others they appear almost perfectly flat for all table sizes (see gpg.out and gzip.out). I had a theory about why this was the case and had this theory confirmed when I tested to see just how many unique branch addresses occurred in these branch trace files:

File Name	Number of Unique Branches
zip.out	5054
google_chrome.out	10486
gpg.out	37
gzip.out	102

I theorised, that an increase in table size would have less of an impact on files with a smaller number of unique addresses as the lower the number of unique addresses, the less addresses were being cut down to the same mask and index in the state array. Clearly from this, two of the flattest results in Graph 3 have two incredibly small numbers of unique branches when compared

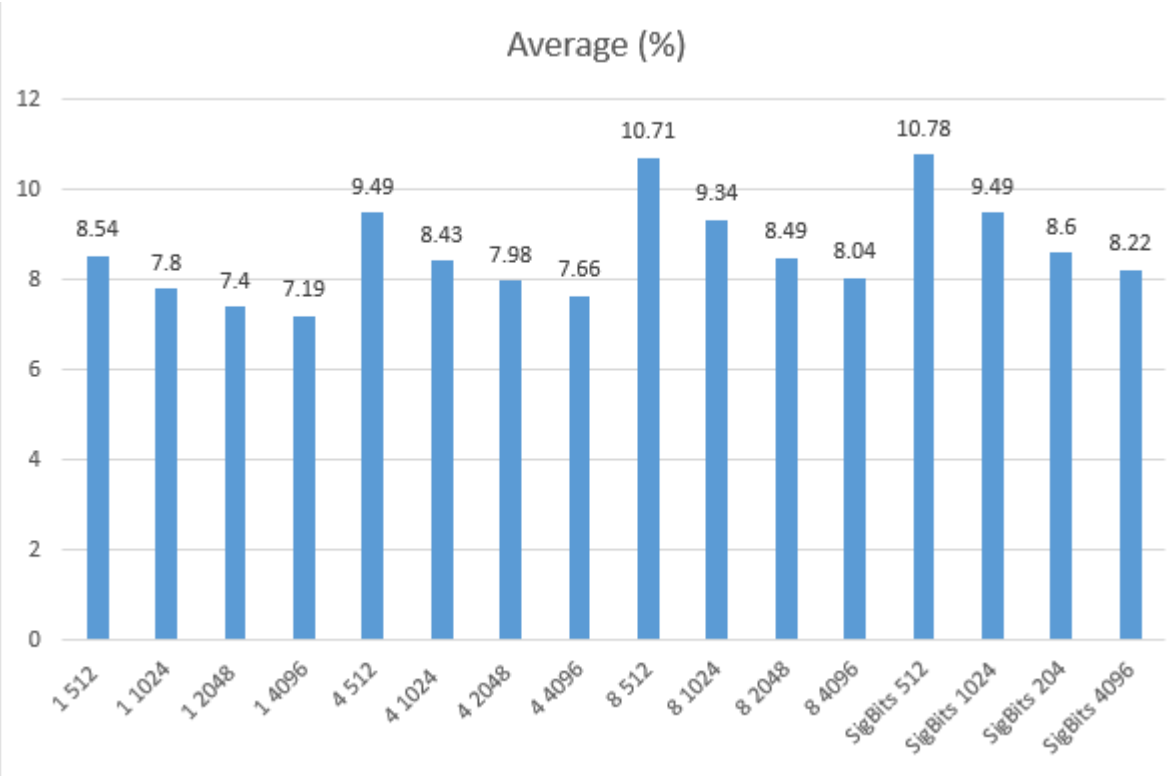
to other addresses. Therefore, I believe I have found the reason for their lack of change as the table size increases.



Graph 5: GShare misprediction rate at each buffer size for the four possible starting states

For this experiment, the starting state that an index is set to would be changed between the four possible states (00, 01, 10 and 11) before the code was recompiled and run.

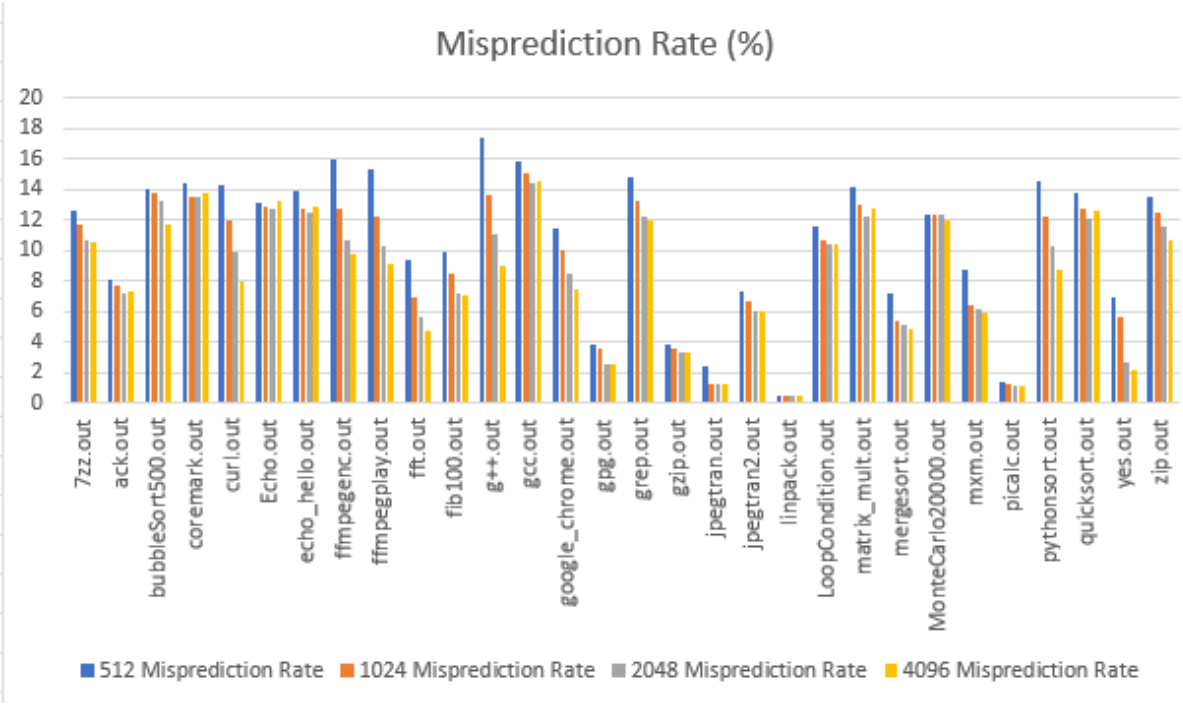
Graph 5 furthers the analysis of Graph 3, that initial starting state has a negligible impact on the overall misprediction rate of the strategy. However, again like Graph 3, this graph shows the trend of how increasing table size decreases the misprediction rate. This shows that GShare is much like Two Bit in that regard. This makes sense as the two strategies are very similar. I thing I would like to note in regards to starting states is that 10 as the starting state for both strategies appears to produce the lowest mispredictions rates. I simply believe this to be the case as 10 can reach a not taken state within one wrong prediction, whilst the default state (00) and 11 must have two incorrect predictions before they reach a state that will predict the opposite. Therefore, as much as I believe the Two Bit strategy to be massively superior to its One Bit counterpart, perhaps utilising a starting state that can be only one mistake away from switching state (10 or even 01) will produce better results than the alternatives.



Graph 6: GShare misprediction rate at different global buffer sizes

For this experiment, the max size that the global history can be would be changed between the four sizes (1, 4, 8 and SigBits ($\log_2$(table size)) before the code was recompiled and run.

I believe the real analysis of GShare occurs with Graph 6 as we see the impact of the global history – the only thing differentiating the GShare implementation from Two Bit. As the size of the global history increases, it has a greater impact on the final index of the state array, reducing the full impact of the actual branch address being read in. From Graph 6, one can see that as the size of the global history increases, the misprediction rate of the strategy increases at every possible table size. This was very interesting to me as I envisioned GShare as a more effective strategy than Two Bit due to the extra steps taken in the process. Since the increase in global history size means an increase in the impact of GShare over standard Two Bit, and since an increase in the global history size also coincides with an increase in the misprediction rate, I have concluded that Two Bit is a more effective strategy than GShare for most cases (see Graph 7).



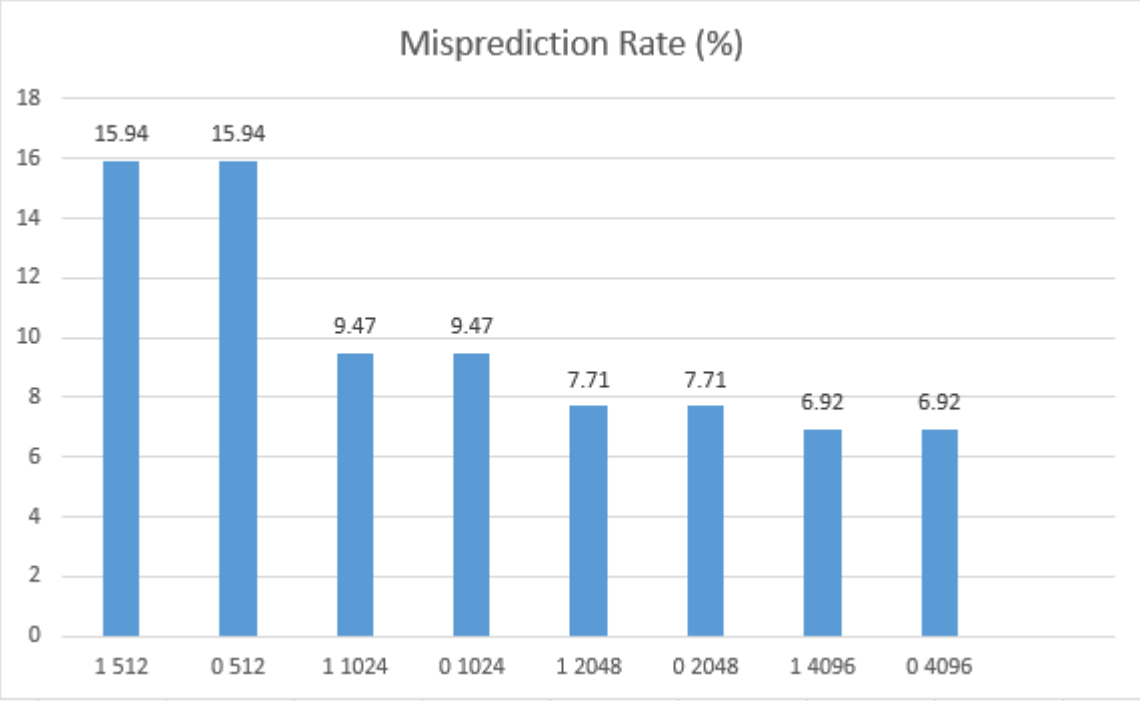
Graph 7: GShare Misprediction Rate for All Files at Default Starting State (00) at Default Global Size (SigBits)

Whilst Graph 7 shows much of the same results as Graph 4 – that increasing the table size tends to reduce the misprediction rate for every file – one thing of note is that we do not see as many 'flat' results as we did in Graph 4. I believe this is due to the impact of the global history, changing what index in the state array was being accessed, thus creating a continual difference in indexes for the same address.

Since there was no 'flat' results for the same files as mentioned in the analysis for Graph 4, I had to look into the impact of the global history (and GShare in general) for these files.

File Name	TwoBit 512	TwoBit 1024	TwoBit 2048	TwoBit 4096	GShare 512	GShare 1024	GShare 2048	GShare 4096
gpg.out	3.80	3.80	3.80	3.80	3.84	3.63	2.60	2.61
gzip.out	3.22	3.22	3.22	3.22	3.86	3.55	3.39	3.29

Once again, I was surprised by the results as the files showed results in favour of different strategies. At table size 1024 onwards, GShare vastly improved on the constant misprediction rate of Two Bit for gpg.out, whilst no GShare misprediction rate for gzip.out ever reaches the constant rate of TwoBit. This made me believe that there were uses for both strategies and despite TwoBit producing the better results overall for the provided files, GShare was not to be disregarded. Due to how similar both implementations are, I would recommend implementing both strategies and seeing which produces better results for any given data set, before proceeding with that strategy.

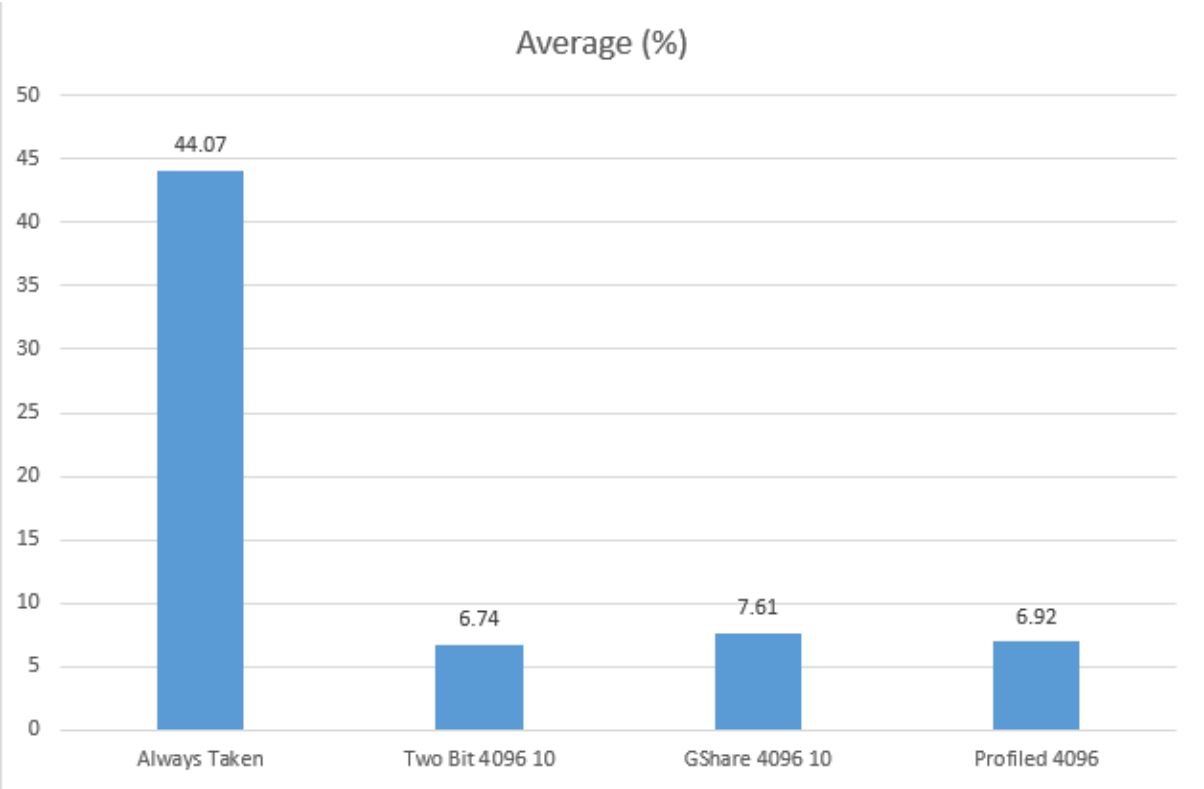


Graph 8: Misprediction Rate for Profiled Strategy with Always Taken/ Always Not Taken for Addresses Not Encountered in Profiling

For this experiment, the check during the 'real' run to see if an address was encountered during the profiling stage had its outcome changed between deeming all non-encountered addresses as Always Taken or Always Not Taken.

Easily the most interesting experiment result. Graph 8 shows that predicting always taken or always not taken for branch addresses not encountered during profiling has no impact on the final misprediction rate. This means that every address must have been encountered in profiling. More accurately, every binary representation that an address could be masked to was encountered in profiling. Therefore, both results are the exact same for always taken/not taken. Since there is a clear limit on the number of binary representations that an address can be masked to, this experiment also explains why as the table size increases, the misprediction rate decreases as this limited number is made bigger, allowing for a more accurate treatment of branch addresses as less unique addresses are cut to the same binary representations.

The increasing table size results also coincide with the question of how much profiling is required for a profiled branch structure, as the more time is spent profiling (bigger table size) the more accurate the predictions are. The question then comes with finding the correct balance between time and accuracy, as any time spent profiling is time not spent in the real 'run' of the branch files.



Graph 9: The Average Misprediction Rate for Each Strategy Under Optimal Values

Ultimately, from Graph 9, one can see that the profiled approach does not have the lowest misprediction rate and thus cannot be deemed the out and out optimal strategy of the four. This is also without considering the fact that to implement the profiled strategy in an architectural sense, more time and space would be taken up as the profiling stage requires both before any 'real' run can commence. Therefore, to conclude my evaluation, I would recommend a pairing of Two Bit and GShare as they are almost inseparable, with TwoBit having the better results on average, but GShare having its moments with the lower misprediction rates at certain files. Both strategies also take up less time and space than the profiled strategy implemented, which is a very important feature of architectural methods. Graph 9 also makes sense to me as Two Bit and GShare are branch prediction strategies implemented in the real world of architecture, whilst the profiled approach was one thought up by a university student in his room, so one would think the tried and tested methods would prevail, as they have. Perhaps there are better profiling strategies out there that produce better results than the two suggested strategies, but again, one must also consider if the results produced by the profiled strategy are so much better that they also make the extra space and time required a worthwhile tradeoff.

Conclusion

I am very happy with what I have been able to produce for this practical. I feel that I have gained a fair understanding of the four branch prediction methods just from the implementation alone. However, the experimentation and analysis that followed the implementation has only helped to further solidify this knowledge. I believe this practical has set me up well for the rest of the module and I am looking forward to being able to apply this knowledge wherever I can; be it in the exam or out with this module entirely.

Something that I was not best pleased with during this practical was that before I settled on my implementation of the profiled strategy, I looked at multiple theorised methods from academic papers such as the Spotlight approach or the Elastic History Buffer. Sadly, I was not quite able to

comprehend how to translate these methods into their code equivalents for my program. This was a shame due to the papers listing how to implement these methods in an architectural sense, which was perfect for this practical's report. Fortunately, however, I feel I learned a fair bit just from reading the papers alone, so all was not lost.

One thing I would have liked to have implemented if I had more time for this practical would have been the ability to write to files at execution time. This was something I attempted to implement at multiple stages during the creation of the strategies, but I was never quite happy with how it worked, or it threw up strange errors regarding results. In the end, I believe the terminal output of my programs is enough for one to run and see that my results are both valid and reproduceable.

References

Help to read multiple files at once –

https://stackoverflow.com/questions/25536845/how-to-run-a-java-program-on-all-files-in-a-directory-from-command-line

Help to split the address of a file containing '\' –

https://stackoverflow.com/questions/23751618/how-to-split-a-java-string-at-backslash

Spotlight Branch Predictor Paper –

https://ieeexplore.ieee.org/document/5403813

Elastic History Buffer Paper –

https://ieeexplore.ieee.org/document/628853

SPARC –

https://en.wikipedia.org/wiki/SPARC

MIPS –

https://en.wikipedia.org/wiki/MIPS_architecture

Always Taken's Uses Today –

https://en.wikipedia.org/wiki/Branch_predictor#Static_branch_prediction

Appendix

All results can be found in the same folder as this report within their respectively named Excel files. Any experiments where variable values are changed can be found in the Experiments Excel file.