University of St Andrews

CS4202 - Branch Prediction

140015554

October 18, 2017

1. Introduction

Branch prediction is used in computer architecture to minimise the performance losses originated from conditional branch instructions. By guessing if the branch instruction is going to branch or not, the CPU can speculatively execute consecutive instructions and improve the flow in the instruction pipeline. However, in case of a misprediction, a delay is introduced, as the speculatively executed instructions have to be discarded and the pipeline needs to fetch the correct branch and start over. Therefore, various branch prediction strategies have been introduced [2] to reduce the misprediction rate and are currently implemented in the hardware of the modern CPUs.

This report will explore the Always Taken, Always Not Taken, 1-bit, 2-bit, Correlating and Gshare branch prediction strategies and their simulations and provide an analysis and insights into the behaviour of those predictors when combined with various benchmark programs. Furthermore, a profiled approach will be developed and discussed that will use information gathered from a *profile* run of the code to make the predictions. Also, suggestions will be made on how such approach then could be represented in the hardware.

2. Implementation

In order to analyse the different branch prediction strategies, a simulator was implemented in Python which, given a trace file and a particular branch predictor, would simulate the performance of the predictor in terms of the misprediction rate. Python was chosen due to its high level language constructs, rich libraries and the ability to write short scripts for different experiments.

The trace file is acquired using a tool provided by the Intel called *PIN* version 3.4 [1]. The tool generates a footprint of all branches in the program and provides an output file containing two columns with memory addresses of each branch instruction and 1 or 0 indicating if the branch was taken or not.

2.1. Simulator

The code was structured in a modular way. An abstraction over the predictor object was provided and two methods: *predict* and *update* are used by the simulator to control the predictor independently of its underlying branch prediction strategy. Here is the code of the *simulate* function:

[1] https://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads

```
def simulate(self, predictor):
    miss_rate = 0

    for trace in self.traces:
        result = predictor.predict(trace.address)

        if (result != trace.taken):
            miss_rate += 1

            predictor.update(trace.address, trace.taken)

    return (miss_rate / len(self.traces)) * 100
```

The simulator iterates through the traces and for each branch instruction it compares the predicted result with the actual choice made by the program. If the prediction is wrong, the miss rate is incremented and after all the branches have been processed in a similar manner, a misprediction value is calculated and returned.

2.2. Always Take (Not Take) Predictor

Always Take and Always Not Take strategies were implemented under a single predictor class. The object is initialised by specifying if the branches should be taken or not. As the name implies, this predictor is simple and always predicts the same outcome. It does not have any understanding of the global or local states.

2.3. 1-Bit Predictor

One bit predictor simply records the last outcome of a particular branch and predicts the same behaviour until a change in the outcome occurs. It does have a small understanding of the local state, however that is usually not enough for an insightful prediction. It is implemented using a table of entries that contain the previous outcome and are indexed by the n least significant bits of the branch address, where n is a power of two.

2.4. 2-Bit Predictor

Two bit predictor introduces another bit that makes it more resilient to singular anomalies in the branching pattern. A counter is used again to store the past behaviour of a particular branch, however this time two bits are available, enabling the counter to have four different states:

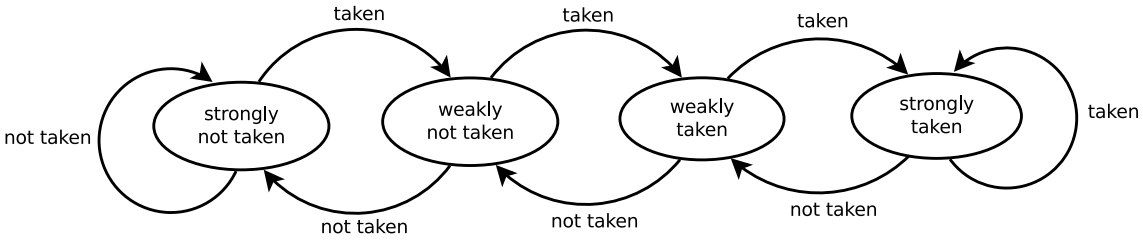

Figure 1. 2-bit saturated counter [2]

[2]Image source Wikipedia: https://en.wikipedia.org/wiki/Branch_predictor

The additional states allows the predictor to have hysteresis and enables it to make a stronger prediction rather than changing its mind every time a different outcome happens. Moreover, such predictor has more information about the local state and is able to achieve a better prediction accuracy, however is also more expensive in terms of the hardware cost required to implement it.

In the implementation, a separate class is used to describe the 2-bit saturated counter. Furthermore, the predictor class has a table to provide a counter for each unique branch address using the same principal described in the 1-bit predictor.

2.5. Correlating Predictor

Correlating predictor includes the global history of the recent branch behaviours into the prediction strategy and combines that with the local state of a targeted branch to make the prediction. This is achieved by using a shift register which contains the history of the outcomes that the last n number of branches produced.

Initially, the branch address is used to index the table, as in the previous predictors. However, this time, the entries in the table contain a pointer to another table (*Counter Pool)* that is indexed using the current value of the shift register:

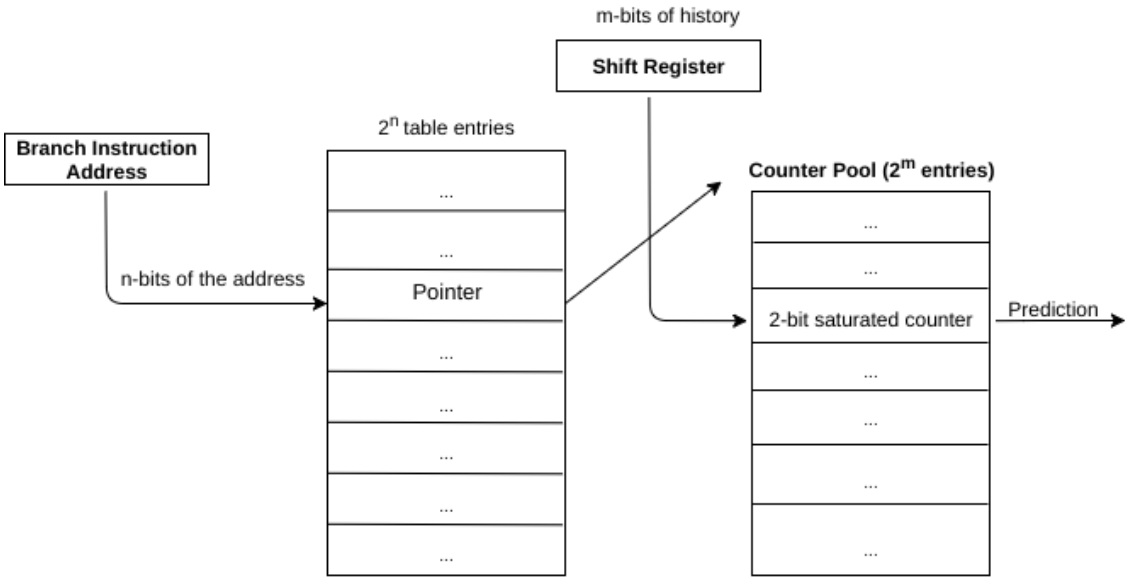

Figure 2. Correlated Predictor

The implementation introduces a new class for the second table called *CounterPool* and uses it to store and retrieve the 2-bit counter for each combination of the shift register's values. Similarly, as with the last n-bits of the branch address indicating the size of the first level table, the shift register size is directly proportional to the number of entries in the *Counter Pool*. Both the first level and the second level table sizes can be specified and fed into the simulator.

2.6. GShare Predictor

GShare predictor reduces the amount of storage space required to make a prediction, while still incorporating the global and local states into the decision. Instead of using two tables

to find the appropriate counter for a branch, it performs an exclusive OR operation on the branch address and the global history stored in the shift register to index the counter table and make a prediction:

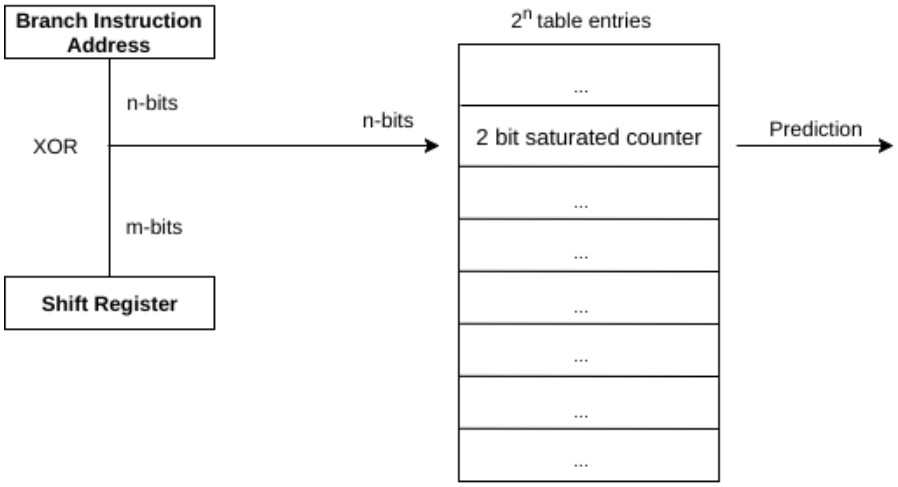

Figure 3. GShare Predictor

In the implementation, lower order n-bits from the branch instruction address and the higher order m-bits from the global shift register are used to perform the XOR operation and produce the index key.

3. Experiments and Analysis

To analyse the different branch prediction strategies and find helpful insights, experiments were conducted using the implemented simulator solution and dedicated Python scripts. Different set-ups were tried, where the variables such as the number of entries in a counter table or the size of the shift register were tweaked, in order to find more interesting correlations. All data produced from the experimental scripts was stored in a CSV file format and then later used to plot the graphs.

3.1. Benchmarks

Branch prediction is a non-trivial task that requires a dynamic analysis of the branch instruction stream. The main difficulty is to predict how a particular branch will act without knowing its previous behaviour. It might be possible to build an optimal predictor for a particular program or branching pattern. However, it is much harder to expand such solution to a larger scale and make it optimal for all general computing tasks. Therefore, to perform a more comprehensive analysis of the different branch prediction strategies, various benchmark programs were gathered and are described in the table (Figure 4) below.

Benchmark	Language	Description
hello	C	Prints ”Hello” to the console
fibonacci	C	Generates Fibonacci numbers recursively
oddeven	C	Jumps to a different branch if the number is even or odd
insersort	C	Sorts a list of numbers using insertion sort algorithm
fft	C	Performs Fast Fourier Transformations
linpack	C	Solves simple linear algebra problems

Figure 4. Benchmark programs

Benchmarks were written and selected to cover a wide range of computational needs. *Hello* is designed to check the behaviour of branch predictors with I/O operations. *Fibonacci* introduces a recursive pattern, while *oddeven* contains a loop and has a branch that behaves differently every iteration. Furthermore, *insersort*[3] performs sorting of an array of numbers. While, *fft*[4] and *linpack*[5] are used to analyse the predictors with one of the most common computations performed: Fast Fourier Transforms and solving linear algebra problems.

To provide analysis of the branch predictors' performance when combined with benchmark programs, trace files were generated for each benchmark and simulations were made to determine the misprediction rate. The results are plotted in Figure 5 below.

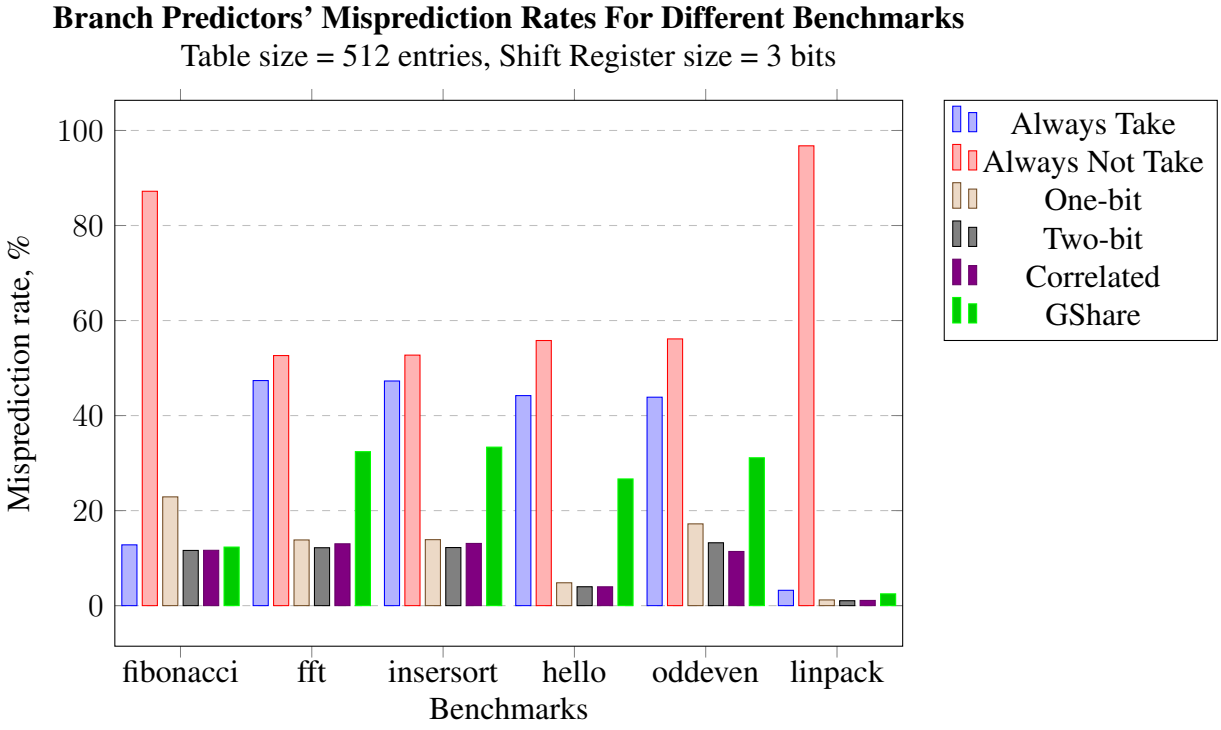

Figure 5.

[3] http://www.c4learn.com/c-programs/program-to-implement-insertion-sort-in-c-programming.html

[4] http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html

[5] http://web.mit.edu/~emin/www.old/source_code/fft/fft.c

We can see from the plotted data that four out of six benchmarks exhibit a similar misprediction rate pattern. In those cases, branches are taken and not taken similar amount of times with taken leading the way. Therefore, The Always Take and Always Not Take predictors are not very useful in predicting the patterns that the branches take. However, their performance might be improved with the *profile* approach, which will be discussed in the section 3.2.

One could also notice that while One-Bit, Two-Bit and Correlated predictors seem to be performing reasonably well, gshare predictor is doing consistently poorly in those four cases. This can probably be explained if we consider that the table, where the 2-bit predictors are stored for each branch, has 512 entries and only three bits of history are used in the XOR operation to generate the index key for a particular branch. Therefore, a significant number of index collisions most likely happen and reduce the performance.

Finally, the other two benchmarks (*fibonacci* and *linpack*) are dominated by always taken branches, as visible from the performance of the Always Take Predictor. The recursive structures in the *fibonacci* program are most likely converted to loop instructions in the assembly code, hence making the program loop until all the Fibonacci sequence numbers, with a given length, are found. The unconditional branch instruction (jump statement) is always taken during the execution of the loop, therefore explaining the good performance of the Always Taken predictor. A similar loop pattern must be occurring in the linpack benchmark as well.

3.2. Investigation of Two-Bit, Correlated and GShared Predictors

To investigate further how Two-Bit, Correlated, GShare predictors work, an approach was tried, similar to the one taken by Su and Zhou [3]. The predictors in question can be customised by specifying different sizes of the counter table and the global shift register. Therefore, an attempt was made to find the optimal settings for these parameters.

Two-Bi Predictor Analysis

Initially, the change in performance, as the size of the counter table increases, was analysed for the Two-Bit predictor and can be seen in the Figure 6.

Two Bit Predictor's Benchmark Performance With Varying Table Size

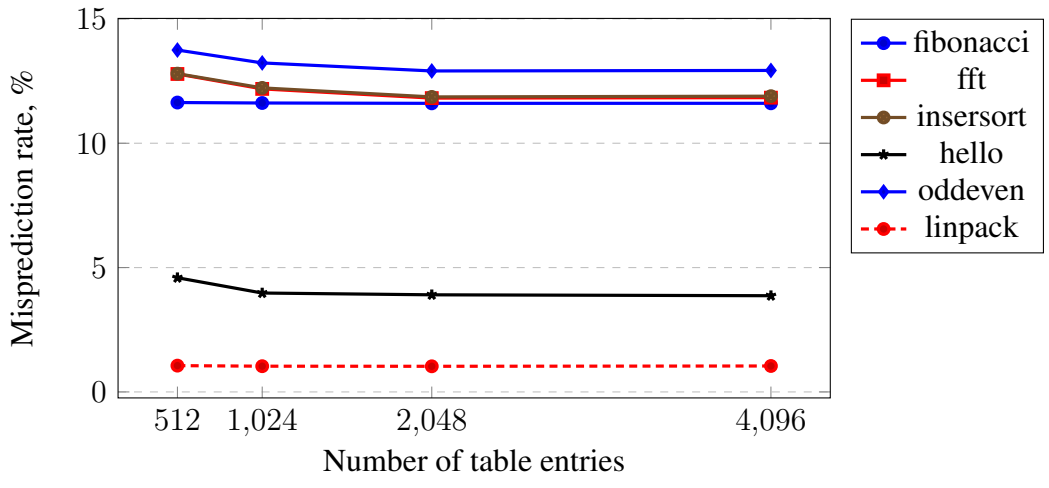

Figure 6.

The Two-Bit predictor uses a table to store the counters for each unique branch instruction. However, due to limited table size available only a small portion of the branch instruction's address is used to index the table. Therefore, if the table size is not big enough to accommodate all possible addresses, collisions will happen and the misprediction rate will increase. We can see the evidence of that in Figure 6. As the number of table entries increase, the misprediction rate goes down, implying that the number of collisions is decreasing as well.

The optimal counter table size for the Two-Bit predictor with the existing benchmarks seems to be reached at 2048 and no more performance is gained going beyond that point. This implies that the used benchmarks have at maximum around 2048 unique branches. Therefore, in general it would make sense to have a bigger counter table size, however the benefits gained have to be traded against cost of space in the hardware.

The prediction accuracy of the Two-Bit predictor is varied across the benchmarks. Each 2-bit saturated counter tries to determine the pattern of a particular branch, however the behaviour of some branches might be dependant on others or the overall state of the program, hence requiring an understanding of the global state to make an accurate prediction, which Two-Bit predictor does not have. Nonetheless, It still performs well in some cases where the local behaviour of a particular branch can be described using 2-bits, as seen in Figure 6 with *linpack* benchmark.

Correlated Predictor Analysis

Furthermore, similar measurements were made for the Correlating predictor to determine its most optimal properties. This time, however, another variable had to be introduced, which is the shift register size that contains global history state of the recent branches. To be able to see how the number of history bits influence the performance, multiple experiments were performed (Figure 7) with fixed table sizes of 512, 8192 and varying number of history bits stored.

Correlated predictor's benchmark performance with varying parameters

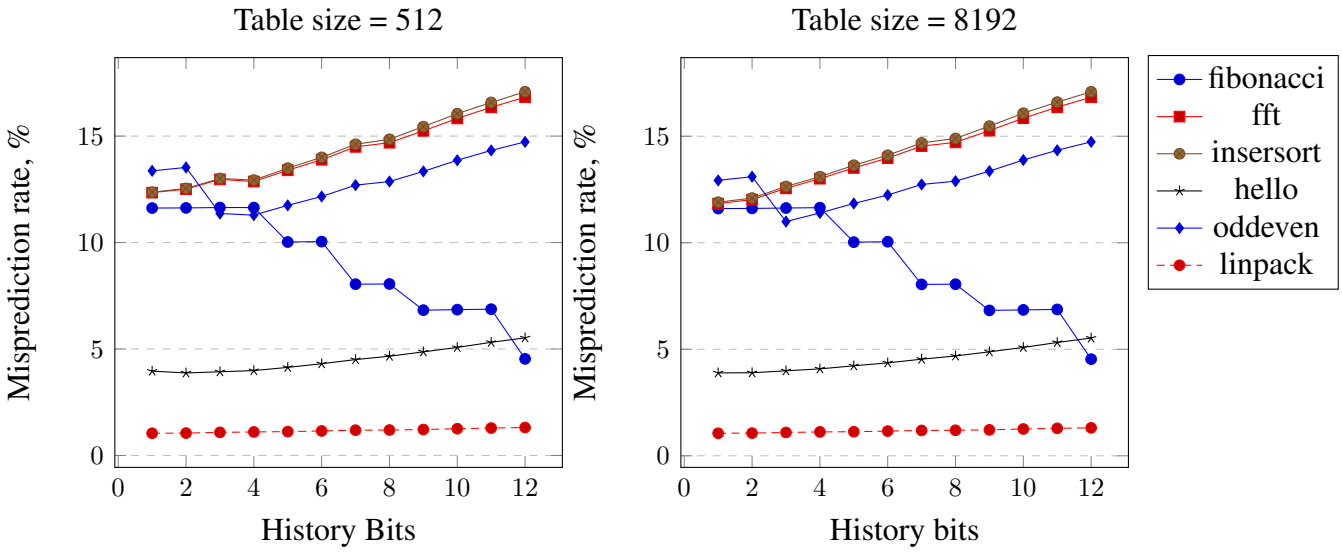

Figure 7.

Increasing the table size doesn't seem to have any effect on the performance of the Correlating predictor. However, if we look more closely at the plotted data, we can see a small increase in performance as the table size rises. This could be explained by considering again that bigger table size reduces the probability of collisions.

A more interesting property is the shift register size. As the number of history bits increases in the register, the misprediction rate for some of the benchmarks goes down, but for overs, steadily goes up or almost doesn't change at all. Each branch in the Correlating predictor has it's own pool of counters that are indexed using the global state which allow a better prediction rate if the branch is influenced a lot by the behaviour of other branches before it. However, if there are no patterns to be found for a branch in the global state's context, the performance might be reduced, due to history bits indexing a new entry in the counter pool table almost every time and thus failing to encode the patterns of individual branches.

GShare Predictor Analysis

Finally, the GShare predictor was analysed in a similar way by having the tables of size 512, 8192 and varying the number of history bits to determine the change in the performance rate with different benchmarks. The results can be seen in Figure 8 below.

GShare predictor's benchmark performance with varying parameters

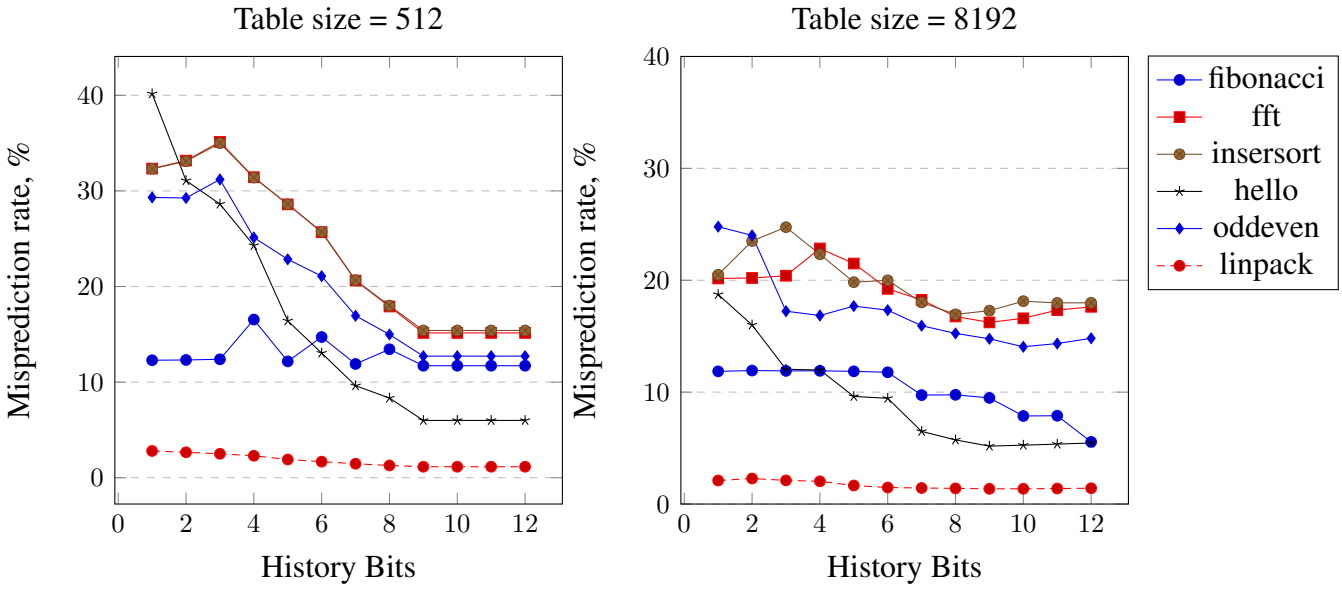

Figure 8.

GShare uses the global state stored in the shift register and combines it with the branch address by XOR operation, to index the counter table and make the prediction [1]. This allow to take into account the behaviour of an individual branch and its correlations with the global state, while achieving a similar performance to that of a Correlating predictor, but with much smaller storage overhead.

The plotted data seem to have varying changes in the improvement of the performance and it's hard to tell from the current measurements what could be the cause of this. In

order to understand this behaviour better, benchmark program code could be analysed in more detail and might provide some more insight. Nonetheless, there is clear shift of performance increase for all benchmarks, then the table size or the shift register size is made bigger. This might be explained, if we consider that GShare relies on one table to store the counters for each branch and thus by increasing the table size and the number of history bits used in the calculation of the index, we provide more amount of entries available to different branching patterns.

3.3. Profile Based Branch Prediction Approach

A different approach to branch prediction was also explored, where a profile is generated during the first run of the program and is later used to identify certain types of branches and adjust the predictions according to those types. Figure 9 illustrates the design of such predictor.

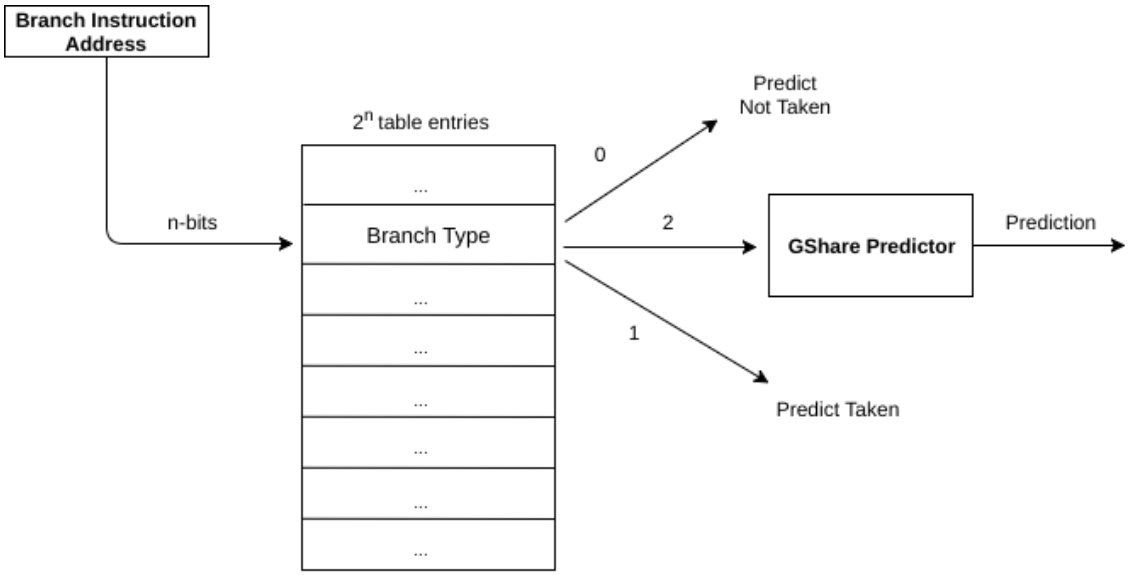

Figure 9. Profile Predictor

Initially, during the first run of the program a profile is generated that contains the information on how many times each branch was taken and not taken during the execution. Then, the branch classifier groups the branches into three groups. If the branch is taken a percentage of the times higher or equal to the threshold value, then it will be classified to group 1 and the predictor will always predict taken. Similarly, if it the branch was not taken most of the time it will be classified to group 0 and predicted to do so in the future. Finally, if the behaviour of a branch cannot be defined easily by placing it in the first two groups, it is assigned to group 2 and a GShare predictor is used instead.

This approach allows to combine knowledge from a single run of the program and identify and predict the well behaving branches. At the same time, it has a fallback mechanism, where, if the behaviour can not be easily identified, a different more suited predictor is used.

In hardware, the profiles of the recently ran programs could be stored in a special cache. Therefore, after the first run of the program a profile would be generated and stored. Then,

if the same program is being routinely executed in the CPU, the profile could be retrieved from the cache and be used to improve the performance.

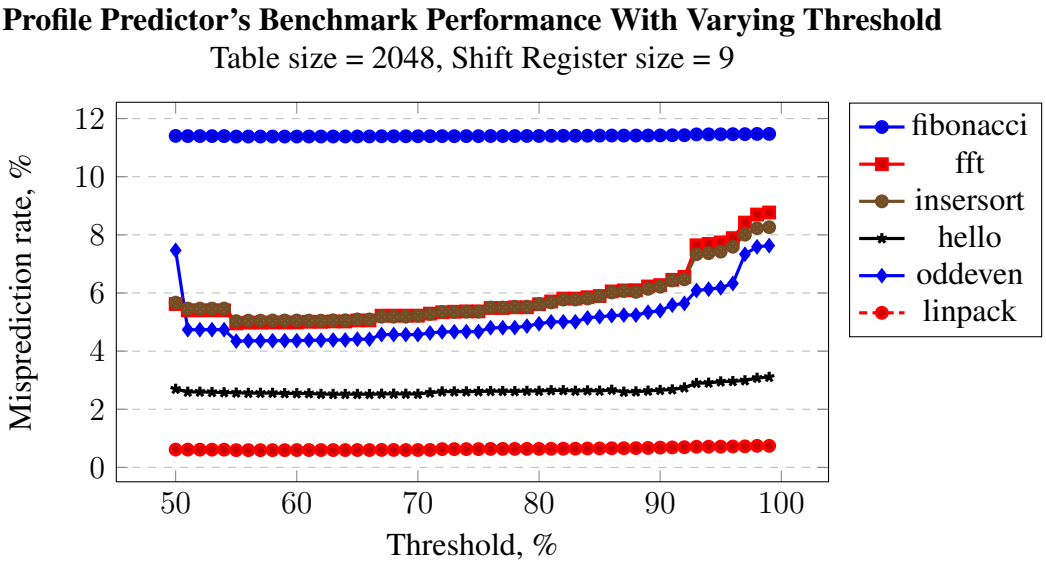

Figure 10.

In order to determine the most optimal threshold value, experiments were performed with a table of size 2048 and the shift register of size 9. The results, plotted in the Figure 10, seem to indicate that the most optical performance is achieved with a threshold value of around 60%.

3.4. Final Evaluation

After exploring and analysing the Two-Bit, Correlating, GShare and Profile-based branch predictors, an attempt was made to compare their performance using more optimal parameters (Figure 11). It was hard to determine the best settings for the Correlated and GShare predictors, therefore, a decision was made to use an approximately optimal shift register size of 9 for all.

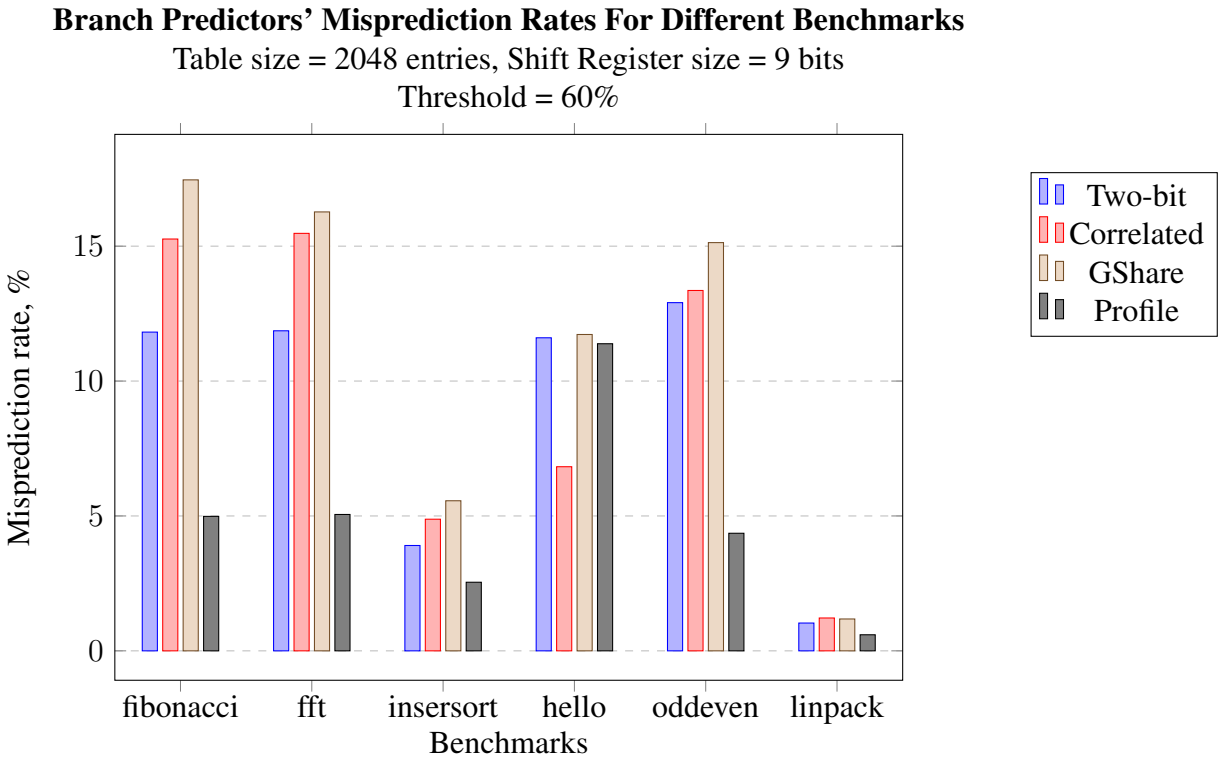

Figure 11.

We can see, from the plotted results, that the profile-based approach almost always performs better, however requires an initial run of the code. Furthermore, Two-Bit predictor has a smaller misprediction rate than Correlated one in most of the benchmarks, however is not optimal all the time. Finally, the GShare predictor manages to almost keep up with the Correlated one in most of the cases, while requiring a much smaller storage overhead.

In the end, it seems that certain predictors are better for predicting behaviour of branches in specific kinds of computations. To achieve a great performance overall, an approach combining multiple predictors and using them in parallel, depending on the branch pattern's type, might be the most optimal.

4. Conclusions

Branch prediction is used in modern computer architecture to increase the utilisation of the instruction pipeline and improve the performance of the CPU. Many branch prediction techniques have been proposed and researched over the years. This report explored and analysed the Always Taken, Always Not Taken, One-Bit, Two-Bit, Correlated, GShared branch prediction strategies by simulating them with different benchmark programs and provided an in depth analysis of the behaviour of such predictors with different parameters of the running environment. Furthermore, it introduced a profile-based approach and tried to find the most optimal branch prediction strategy.

References

[1] S. McFarling. Combining branch predictors. `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf`.

[2] J. E. Smith. A study of branch prediction strategies. https://courses.cs.washington.edu/courses/cse590g/04sp/Smith-1981-A-Study-of-Branch-Prediction-Strategies.pdf.

[3] Z. Su and M. Zhou. A comparative analysis of branch prediction schemes. http://web.cs.ucdavis.edu/~su/Berkeley/cs252/project.html.