1. Hardware Simulated

# Software flow

All 4 code files were generated using a similar structure. To begin, the main function opens a file pointer to the first trace, and passes it to the function readTrace, as well as the number of bits for the size of the predictor (if applicable) the function is called 10 times/trace, for 2 bits, 4 bits, …, 20 bits predictors.

Once in the readTrace function, the first step is initialization and formatting of the arrays. We also calculate 2 boundary variables named “Nmask” and “threshold”. Nmask = 2^n -1 and serves as a maximum boundary for the predictor (the minimum boundary is always 0). Treshold = (Nmask + 1)/2 and serves as a predictor decision threshold. As a convention, low value predictors favor branch not taken. For example, on an 8 bit test, Nmask = 255 which is the maximum value that the predictor can take, and threshold is 128, meaning that if the predictor < 128, it predicts that the branch will be taken.

Afterwards, we fetch one line of data at a time from the file (hence the huge processing time due to massive I/O wait). The address is extracted and a lookup is made on the “addresses” array to see if it is the first time it is encountered. The array is scanned from index k = 0, until the system finds either a previous entry of that branch, or an empty spot on the array. If the entry is found, k is recorded and used as an index for selecting predictors. If the system encounters an empty spot on the array before finding a match, the address is recorded at that spot and the index k will now be associated with this branch for future occurrences.

Note: due to the memory address limitations of my 32 bit PC, I was not able to run the correlating predictor with the server trace file (required at least 10300 x 256 addresses) and constantly got segmentation fault. To counter this, a Boolean variable called “tooBig” was introduces to inform the predictor that it needed to “compress” the data if it was analyzing the server trace. In order to do so, the first 2 decimal bits of the addresses were ignored, which generated a decrease in performance for the server traces, since multiple addresses might have been sharing the same LSBs. The same scheme was used for every types of predictors on the server trace, therefore the data collected is still usable (we can adequately comment on the performance of different predictors, but we cannot compare server traces with INT traces, since the INT traces were fully indexed, where as the server traces were indexed using a subset of address bits).

The data was then fed to the predictors and statistical data was gathered based on weather a good prediction was made or not. At the same time, the predictors were adjusted to reflect the new branch outcome.

The predictors were implemented using different algorithms (described below). In order to interpret the prediction of a branch, if the predictor < threshold, the prediction is not taken, and if predictor > threshold, then we predict branch taken. Weather the prediction was right or wrong, the predictor got modified with accordance to the branch result (decremented for not taken and incremented for taken). The predictors were never decremented below 0 or above “Nmask. Whenever a threshold was passed, the predictors jumped at the other edge of the prediction (see book p.83). For example, on a 3 bit predictor (Nmask = 7, threshold = 4, therefore 0-3 predict not taken and 4-7 predict taken), if the predictor was at 4 and the branch was not taken, the predictor got set to 0.

2. Predictors Algorithms

# 2.1 Static branch prediction policies

Code file: /code/static.c   
Data collected: See annex 2-a)

In order to analyze the result of a simple static prediction policy, the trace files were read and a basic statistical analysis was performed on the number of branched taken and not taken. This information is sufficient to discuss the implementation of a static always predict taken or not taken algorithm, since their performance are complementary.

For example, by knowing that 18% of branches were taken on the FP traces, we know that a “predict taken” mechanism would have a failure rate of 82%, whereas a “Predict not taken” algorithm would result in a failure rate of 18%.

# 2.2 Branch prediction buffer (n-bit predictor)

Code file: /code/buffer.c   
Data collected: See annex 2-B)

A branch prediction buffer was implemented and results were gathered for 2, 4, 8, …, 20 bit buffers. Due to the non dynamic nature of this project, we were able to generate a unique predictor for every single branch addresses present on the trace files (see section 1.1 Software flow). After figuring out the branch index value (k), this value as well as the branch result was passed on to the predictor.

The predictor consists of an array of 10 000 integers called “predictorLocal”. The index k is used to select which predictor to use and the integer present is used as a predictor. In order to minimize memory use, the predictors mimicked the behavior of binary strings, but were manipulated using integers. To demonstrate the methodology used we will used in this report (h) so imply hardware implementation and (s) for software implementation.

Here is the explanation of an 8 bit predictor. Here, Nmask = 255 and threshold = 128.

Range: (h) [0000 0000, 1111 1111] (s) [0, 255]   
Range for predict not taken: (h) [0000 0000, 0000 1111] (s) [0, 127]   
Range for predict taken: (h) [0001 0000, 1111 1111] (s) [128, 255]

After interpreting the predictor, the system looks at the branch result and “totalGoodPredictions” (an integer keeping track of the number of good predictions) got incremented in the event of a good prediction. Also, the local predictor gets incremented in the event of a branch taken and decremented in the event of a branch not taken. As described in the book, whenever a threshold is reached, the predictor jumps to the other end of the scale.

For example:

Predictor value: (h) [0000 1111] (s) [127]   
New value if branch Taken: (h) [1111 1111] (s) [255]   
New value if branch not Taken (h) [0000 1110] (s) [126]

After these updates, the system fetches the next line of the trace and continues until the trace is done. All for traces are analyzed in this manner.

# 2.2 Correlating Predictor

Code file: /code/correlator.c   
Data collected: See annex 2-B)

The correlating predictor uses similar predictors with one major modification. Every branch address has 256 local predictors. As above, once the branch address is known, the addresses array is scanned to see if there already exists a history for this branch. A slight modification had to be made for the server trace. Because of the large number of different branch addresses, the amount of memory allocated at run time to account for every single address caused a Heap/Stack collision and forced the OS to terminate the process. To counter this, the variable “tooBig” was implemented to notify that the server address space was too big. In which case, the 2 most significant digits of the branch address were ignored, which resulted in a reduced accuracy of predictions when evaluating the server traces.

The branch enters the predictor with an index k to select which predictor to choose. Once predictor[k] is selected, the globalPredictor is used to select among 256 local predictors for a single branch. If a good prediction is made, a counter is incremented to keep statistical data. Within the same thread, both the local predictor selected as well as the global predictors get updated (in the same manner described in section 2.1 Branch prediction buffer) to reflect the outcome of the branch.

This provides an extra advantage to the predictor, because the global history is used to reflect similar loops and counters encountered within a program.

# 2.3 Tournament predictor

Code file: /code/tournament.c   
Data collected: See annex 2-B)

The tournament predictor implements both the buffer predictor and the correlating predictor. For every branch encountered, both predictors make a prediction based on their state in the same manner described above. In addition, a 2 bit history table is kept as a record of which predictor was the most accurate at the last iteration. The 2 bits of history is stored in an integer array called “chooser” and every branch address has its own history table. If in the last few iterations the buffer predictor was more accurate, it is selected to make this prediction, and vice versa for the correlating predictor.

Once the outcome of the branch is known, the buffer predictor, the local predictors and the global predictor present within the correlating predictor get updated. Based on weather one of the 2 predictor was more accurate than the other, the history table associated with this particular branch also got updated to reflect this information.

For every address of every traces, statistical data was gathered based on the performance of the tournament predictor.

3. Results

The following table was built using the result gathered while running static.c (see Appendix II for raw data).

**Figure 3.1: Success rate of a static Branch predictor. In Red, predict Taken, and predict not taken in blue.**

We can easily observe that for all 4 trace files, 68% to 82% of branches end up not being taken. This is coherent with what could be expected, since the purpose of loops is usually to the same task many times until it is fully accomplished. This implies that branch conditions, when evaluated, will more often be not taken such that the loop’s body can be executed many times.

Based on this, the most efficient static branch prediction bias would be to predict that branches will not be taken. This would end up being right roughly ¾ of the time for general purpose processors. As further results demonstrate, there are better prediction schemes.

The following data was gathered to compare all 3 “intelligent” branch prediction algorithms.

**Figure 3.2: Comparing the different branch prediction algorithms using 4 trace files**

From the data gathered, we can see that the optimal size for prediction buffer is 2 bits. This is due to the fact that for larger bit size predictors, there needs to be too many consecutive mistakes before the predictor changes it’s bias. By doing such, those predictors loose sensitivity and their performance decrease. This phenomenon is a good thing for actual implementation, because a good performance from low bit predictors reduces the quantity of hardware required.

Another thing we can observe is that in general, the correlating predictors are better than the buffer predictors. This result was also predictable since correlating predictors take advantage of global branch information. This information provides a way to pick within an array of local predictors, so different predictors can be selected to reflect different situations, for every branch. The performance gap between buffer and correlating predictors was most accentuated in the FP trace.

We can also observe that for FP trace, the tournament prediction was not more accurate than the correlating predictor. This is most probably due to the fact that operations were repeated many times and this was captured by the tournament choosing table. The floating point operation generally require many clock cycles, so the global predictor became a good way of capturing this regularity.

In the INT trace, the best prediction algorithm was the tournament predictor. This can be attributed to the fact that the tournament makes intelligent selections of predictors. Since there are a relatively small number of integer operations, the choosing table was efficient in allocating an adequate predictor for the branch operation at hand.

A final thing worth pointing out is that the performance of the tournament predictor in the server trace was lower. This is not necessarily due to a lack in performance of the predictor. As mentioned earlier, because of the chosen architecture, the branch address array was too large to be malloced for the server trace which forced the implementation of a predictor less sensitive to changes in addresses. By conglomerating different addresses within the same predictors, the performance of the predictors was greatly decreased.

# .

Annex I: Code included

# I-A) Overview

All the results were generated using c code on a Linux platform. The scripts were compiled using gcc and debugged using ddd.

Notes:

* All code files are located in the subdirectory ‘code/’
* The code will probably not compile and will definitely not work properly on a Windows platform

Instructions:

* To get gcc, open open a terminal window in Linux and type $ sudo apt-get install gcc
* In order for the trace files to be accessed, they need to be placed in a directory called ‘traces/’, to which the directory containing the program is root (code/traces/).
* To compile a code file, type $ gcc filename.c –w
* To execute the binary produced, type $ ./a.out

Code files included:  
**static.c** : Gives out statistics on the number of branches taken/not taken without any prediction algorithm.

**buffer.c**: Implements a buffer prediction algorithm. Every trace will be tested for 2,4,…,20 bytes buffers

**correlating.c**: Implements a correlating prediction algorithm, which uses an 8 bit global predictor to select from 256 local predictors/branch address. Every trace will be tested for 2,4,…,20 bytes buffers

**tournament.c**: Every branch is passed through a buffer as well as a correlating predictors. Based on previous accuracies of both predictors, a 2 bit controller selects which predictor to implement.

Annex II: Data collected

# II-a) static.c

**FP.trace**

Total BR: 2213673 Taken: 412899 Ratio: 18%

**INT.trace**

Total BR: 4184792 Taken: 1068045 Ratio: 25%

**MM.trace**

Total BR: 2229289 Taken: 713587 Ratio: 32%

**SERV.trace**

Total BR: 3660616 Taken: 1111595 Ratio: 30%

# II-B) Prediction schemes

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **biffer.c** | | **correlating.c** | | **tournament.c** | |
|  | **bits** | **TotBR** | **GoodPr** | **% good** | **GoodPr** | **% good** | **GoodPr** | **% good** |
| **FP.trace** | 2 | 2213673 | 2041014 | **92** | 2070849 | **93** | 2040070 | **92** |
| 4 | 2213673 | 2045204 | **92** | 2070828 | **93** | 2044077 | **92** |
| 6 | 2213673 | 2037944 | **92** | 2061395 | **93** | 2036417 | **91** |
| 8 | 2213673 | 2034160 | **91** | 2056295 | **92** | 2030878 | **91** |
| 10 | 2213673 | 2024693 | **91** | 2044533 | **92** | 2021957 | **91** |
| 12 | 2213673 | 1990585 | **89** | 2006191 | **90** | 1990710 | **89** |
| 14 | 2213673 | 1918330 | **86** | 1933428 | **87** | 1918330 | **86** |
| 16 | 2213673 | 1869871 | **84** | 1881298 | **84** | 1869870 | **84** |
| 18 | 2213673 | 1800774 | **81** | 1800774 | **81** | 1800774 | **81** |
| 20 | 2213673 | 1800774 | **81** | 1800774 | **81** | 1800774 | **81** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **biffer.c** | | **correlating.c** | | **tournament.c** | |
|  | **bits** | **TotBR** | **GoodPr** | **% good** | **GoodPr** | **% good** | **GoodPr** | **% good** |
| **INT.trace** | 2 | 4184792 | 3761918 | **89** | 3766125 | **89** | 3749958 | **89** |
| 4 | 4184792 | 3658473 | **87** | 3716688 | **88** | 3702496 | **88** |
| 6 | 4184792 | 3564452 | **85** | 3646954 | **87** | 3643838 | **87** |
| 8 | 4184792 | 3573771 | **85** | 3656075 | **87** | 3655361 | **87** |
| 10 | 4184792 | 3557480 | **85** | 3632158 | **86** | 3607969 | **86** |
| 12 | 4184792 | 3507075 | **83** | 3562238 | **85** | 3532909 | **84** |
| 14 | 4184792 | 3372952 | **80** | 3395253 | **81** | 3376535 | **80** |
| 16 | 4184792 | 3126537 | **74** | 3126532 | **74** | 3126536 | **74** |
| 18 | 4184792 | 3116747 | **74** | 3116747 | **74** | 3116747 | **74** |
| 20 | 4184792 | 3116747 | **74** | 3116747 | **74** | 3116747 | **74** |
| **MM.trace** | 2 | 2229289 | 1905842 | **85** | 1918628 | **86** | 1895885 | **85** |
| 4 | 2229289 | 1908492 | **85** | 1920977 | **86** | 1912730 | **85** |
| 6 | 2229289 | 1905796 | **85** | 1922498 | **86** | 1910134 | **85** |
| 8 | 2229289 | 1898798 | **85** | 1923038 | **86** | 1901594 | **85** |
| 10 | 2229289 | 1876778 | **84** | 1902297 | **85** | 1877087 | **84** |
| 12 | 2229289 | 1812288 | **81** | 1868053 | **83** | 1813427 | **81** |
| 14 | 2229289 | 1703586 | **76** | 1799523 | **80** | 1703504 | **76** |
| 16 | 2229289 | 1537199 | **68** | 1699376 | **76** | 1537199 | **68** |
| 18 | 2229289 | 1515702 | **67** | 1515702 | **67** | 1515702 | **67** |
| 20 | 2229289 | 1515702 | **67** | 1515702 | **67** | 1515702 | **67** |
| **SERV.trace** | 2 | 3660616 | 3325242 | **90** | 3134406 | **85** | 2667523 | **72** |
| 4 | 3660616 | 3297983 | **90** | 2940661 | **80** | 2651695 | **72** |
| 6 | 3660616 | 3261423 | **89** | 2846581 | **77** | 2616696 | **71** |
| 8 | 3660616 | 3149829 | **86** | 2775663 | **75** | 2520838 | **68** |
| 10 | 3660616 | 3006560 | **82** | 2728744 | **74** | 2382900 | **65** |
| 12 | 3660616 | 2895137 | **79** | 2709956 | **74** | 2263377 | **61** |
| 14 | 3660616 | 2751657 | **75** | 2667985 | **72** | 2113811 | **57** |
| 16 | 3660616 | 2655985 | **72** | 2613431 | **71** | 2311200 | **63** |
| 18 | 3660616 | 2557681 | **69** | 2549021 | **69** | 2213557 | **60** |
| 20 | 3660616 | 2549021 | **69** | 2549021 | **69** | 2210291 | **60** |

Annex III: Source c code

III-A) static.c