Analysis of Branch Prediction Strategies

Md Mehedi Hassan Galib, Maloy Kumar Devnath, and Ehsan Ahmed Dhrubo
Computer Engineering Department
University of Maryland, Baltimore, County
Maryland-21250, USA
mgalib1@umbc.edu, maloyd1@umbc.edu, and edhrubo1@umbc.edu

***Abstract*—In this project, we compare eight different types of branch prediction strategies on two famous instruction set architecture in order to find the branch miss-prediction rate, since the penalty associated with miss-prediction is very high for the modern μ processor. To carry out this comparative study, we resort to six different benchmarks such as anagram, test-fmath, test-llong, test-lswlr, test-math,and test-printf. We chose our eight different strategy named taken, not-taken, bimodal, two level adaptive, combination of bimodal and two level adaptive, one bit hash, two bit complementary hash, and our proposed strategy in such as way that they covers both static and dynamic strategy of branch prediction. We measure the accuracy of these branch prediction strategy in terms of branch address prediction rate, branch direction prediction rate and number of misses. Our analysis shows that our proposed strategy shows accuracy more than 80% for both the architecture and all kinds of benchmarks, which proves the efficacy of our proposed strategy with all chosen conventional strategies.**

***Index Terms*—Branch prediction strategy, ISA architecture, static, dynamics, taken, not-taken.**

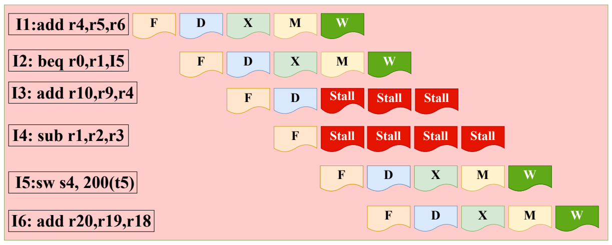
I. Introduction

In recent years, high performance μ-processor are experiencing tremendous insertion of pipeline stage to facilities high clock rates and different level of parallelisms such as *instruction-level parallelism* (ILP), *data-level parallelism* (DLP), *thread-level parallelism* (TLP), *request level parallelism* (RLP) in order to lower the processor execution time and to increase the number of *instructions per cycle* (IPC) [1] [2] [3] [4]. However, pipeline-based μ-processor impedes their performance due to the control dependencies since more than one instruction can be overlapped which obstructs the continuous flow of instruction in the pipeline stages [5]. As pipelines get deeper or the clocking rate gets higher, the penalty imposed by control dependency gets larger [6]. The traditional way to reduce this penalty is predicting the direction of a conditional branch, pre-fetching, decoding, and executing the instruction at the branch target. Hence, the branch prediction method is a critical part of cutting-edge ILP based μ-processors that uses prediction schemes to execute the instructions alongside the anticipated direction [7] [8].

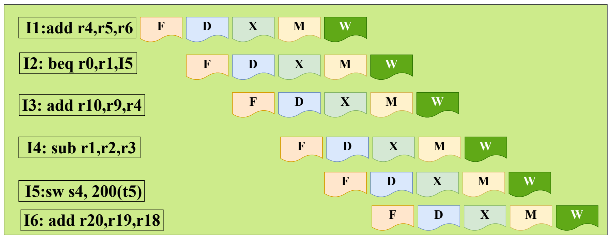
The ultimate goal of the branch predictor is to enhance the flow of instruction in a pipeline stages through the reduction of stall as illustrated in Fig. 1b and 1c, which in turn increases the processor efficiency. However, the branch misprediction also causes stall in the pipeline stages, which includes flushing wrongly fetched instructions, re-fetching the instruction from the other branch as illustrated in Fig. 1d. Hence, a massive amount of substantial work has to be flushed away after a branch miss-prediction [7]. This effect of miss-prediction penalty because more severe for the modern μ-processor as the memory hierarchy is more complex. Therefore, extremely correct branch prediction is needed to address this penalty. Thus, branch prediction has become the focus of computer architecture research and many branch prediction schemes have been proposed to reduce the above mentioned miss-prediction [9].

As mentioned above, in computer architecture, branch prediction plays an important role by achieving high performance, hence, researchers proposed branch prediction schemes to address the time and power wastage due to miss-prediction at different levels in modern microprocessor architectures [10] [11]. Branch prediction schemes can be classified into static schemes and dynamic schemes by the way the prediction is made illustrated in Fig. 2. The most simple prediction scheme is the static prediction [10]. The most straightforward one is predicting branches to be always taken by observing that the majority of branches are taken (also known as taken). Similarly, another straightforward one is predicting branches to be always not-taken by observing that the majority of branches are not taken (also known as not-taken). However, the accuracy of prediction is very low and according to Lee et al [12], these simple strategies' correct prediction rate is about 68%. Hence, to increase the accuracy of the simple taken strategy, another strategy named as back taken was proposed [13], where it predicts that all the branches with lower address were taken and on contrary, all the branches with upper address were not taken. Since loops are terminated with backward branches, back taken strategy was able to predict all loop branches. However, one of the pitfalls of this strategy was if the target address needed to be computed with the program counter before a prediction can be made, which makes this prediction process slower than for other strategies. Furthermore, to enhance the accuracy of back taken prediction, another variant named certain operation taken was proposed [13] where it predicts that all branches with certain operation codes will be taken, while other will be not taken. However, the problem with static branch prediction strategies is that, they take a fixed decision whether a branch will be taken or not taken, and therefore, researchers proposed several dynamic

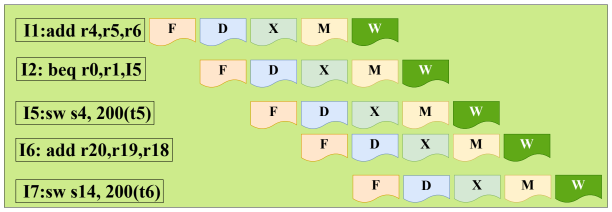
Special thanks to Dr. Ting Zhu, course instructor of CMPE 611, Department of Computer Science & Electrical Engineering, University of Maryland, Baltimore, County, MD-21250, USA.



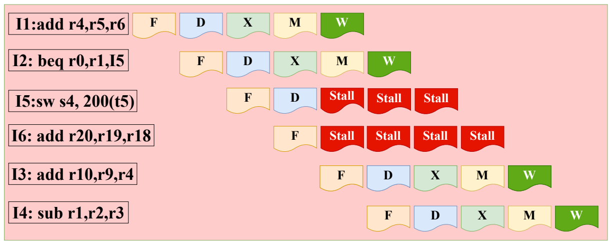
(a) Normal execution without branch prediction.



(b) Correct prediction when branch is not taken.

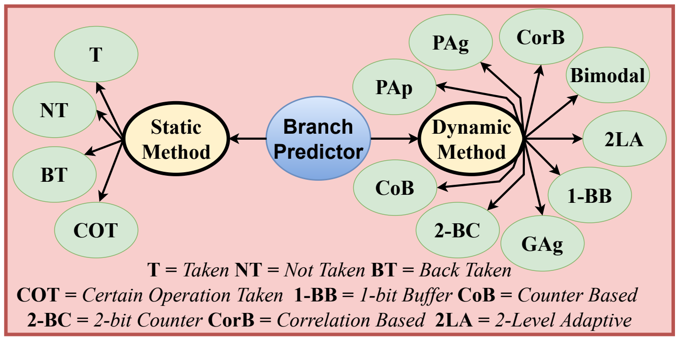


(c) Correct prediction when branch is taken.



(d) Incorrect prediction considering branch is taken

Fig. 1: Illustration of how the stalling affect the performance of the modern μ-processors. Here, in $I1$, the value of $R5$, and $R6$ registers are added and stored in the $R4$ register. There is no stall to perform this operation in five stages of RISC architecture. In $I2$, it manipulates the branch address by comparing $R0$ and $R1$ registers. Since the comparison is calculated by subtracting between $R0$ and $R1$ register, the decision whether the branch will be taken or not would not be ready until it reaches the execution stage and therefore, $I3$ and $I4$ instruction will be fetched since they are presented in the pipeline. However, if the branch is taken, then instructions $I3$ and $I4$ must be flushed and instruction $I5$ should be fetched, which introduces stall as illustrated in Fig. 1a. This stalling phenomenon impedes the performance of the processor. Now, if we use the branch prediction algorithm, which correctly predicts not taken decision as illustrated in Fig. 1b or taken decision as illustrated in Fig. 1c, then stall will reduce will in turn increase the processor efficiency. However, if the algorithm mispredicts then, we have to sacrifice the performance as illustrated in Fig. 1d.



Branch Predictor
Static Method
Dynamic Method
T
NT
BT
COT
PAg
CorB
Bimodal
PAp
2LA
1-BB
CoB
2-BC
GAg
T = Taken NT = Not Taken BT = Back Taken
COT = Certain Operation Taken 1-BB = 1-bit Buffer CoB = Counter Based
2-BC = 2-bit Counter CorB = Correlation Based 2LA = 2-Level Adaptive

Fig. 2: Different branch prediction strategies adopted in different level of modern microprocessor architectures such as dynamic and static control.

branch prediction strategies, where they utilize the history table of previous action to generate a prediction.

Among dynamic branch prediction strategies, one bit branch predictor buffer is the simplest one [10], which comprises of a tiny memory with one bit that indicates whether or not the branch was recently taken. This tiny bit buffer is also served as a suggestion for the next branch prediction. For each of the miss-prediction, the buffer value will be inverted. The major drawback of this strategy is that for an always taken branch, we will miss-predict twice when that branch was not taken. Hence, to address this drawback two bit branch counter was proposed [10], where based on the counter value, it was anticipate whether a branch will be taken or not. When the counter is larger than or equal to one-half of its maximum value 2, the branch is anticipated as taken; otherwise, it is decided as not-taken. On a taken branch, the counter is incremented, whereas on an not-taken branch, it is decremented. This strategy uses only the recent behavior of a branch to predict the future behavior of that branch which may decrease the accuracy of overall prediction. To address this problem, in the GAg strategy [10], the most recent m branches were recorded in an m-bit shift register, named global branch history register, where each bit records whether the branch was taken or not taken. There were another 2m entries global branch history pattern table where each corresponds to the 2-bit counter for a global branch history. The prediction was done based on the most recent m branches. The value coming from the global branch history register was used to query the global branch history pattern table to find the corresponding 2-bit counter. The predict policy of GAg is similar to the two bit counter predictor [10], which predicts a branch as taken if the counter is greater than or equal to one half of its maximum value 2; otherwise, predicts as not-taken. The outcome was shifted left into the global branch history register after the conditional branch was resolved, and the 2-bit counter was incremented on a taken branch and decremented on a not-taken branch, which also similar to the two bit counter. Unlike the global history table, in PAg [10], each individual branch has a branch history register table, where conditional branch's address was used for hashing into this table. Each entry in this table is a branch history register with m bits, recording whether the most recent m branches corresponding to this entry are taken or not. The prediction of a branch is based on the history pattern of the last k outcomes of executing the branch. Each time a conditional branch occurs, it finds the corresponding entry in the branch history register table and gets the branch history register. The branch history register is used to address and make predictions in the global branch history pattern table. After the conditional branch is resolved, the result is shifted to the left in the branch history register at the least significant bit position and is also used to update the 2-bit counter in the global branch history pattern table entry. Instead of having one global branch history pattern table for all the branches like PAg, in PAp, there is a branch history register and a branch history pattern table for each branch [10]. When a conditional branch occurs, it populates the corresponding entry in the branch history register table. It is used to index the branch history pattern table for that branch to find and make predictions for 2-bit counters. After the conditional branch is resolved, the result is shifted to the left in the branch history register at the least significant bit position and is also used to update the 2-bit counter in the branch history pattern table. Another similar strategy named as counter based has been proposed where each dynamic branch with a unique address is given an N-bit counter [14]. The counter value C associated with a branch is utilized for prediction when that branch is about to be executed. The branch is decided to be taken if C is greater than or equal to a predefined threshold value L, else it is predicted not taken. 2^{N-1} is a common value for L. When that branch is resolved, the counter value C is updated. C is incremented by one if the branch is taken; otherwise, it is decremented by one. If C is 2^{N-1}, it will stay with the value as long as the branch is taken. If C is zero, it will remain thus as long as no branch is chosen. Furthermore, to improve the accuracy of prediction algorithm further several advanced strategies like Bimodal, two-level adaptive, correlation, and counter based has been proposed by the researchers [15] [16] [13] [17] [14] [18]. Bimodal branch prediction takes advantage of the bimodal distribution of branch behavior such as taken and not take, and then, attempts to distinguish usually taken from usually not taken branches. This can be done with a table of counters indexed by the low order address bits in the program counter. On the other hand, in two level adaptive predictor, adaptive training method alters the branch prediction algorithm on the basis of information collected at run-time. It is based on the history of branches executed during the current execution of the program. Execution history pattern information is collected on the fly of the program execution by updating the pattern history information in the branch history pattern table of the predictor. Therefore, no pre-runs of the program are necessary.

Most studies of dynamic branch prediction focus on the history of the branch under consideration [14]. With hardware-assisted branch prediction, only the most recent history of a branch is used to predict the outcome of that branch. These branch prediction schemes work well for scientific/engineering workloads where program execution is dominated by inner loops. However, they do not work as well for integer workloads where the outcome of a branch is affected by the outcomes of recently executed branches. When one branch depends on another, in the sense that its outcome depends on the outcome of the other branch, the branches are correlated. Although the presence of branch correlation may cause the behavior of a branch to appear more random, it may shed some light on the condition upon which the branch decision is based. In order to select the proper 2-bit counter assigned to each sub history for prediction, one needs to memorize the branch path leading to the last branch. This can be achieved by using a 2–bit shift register which records the outcomes of the two most recently executed branches. The shift register is then used to select the appropriate counter. The use of a shift register for tracking and selectively relating the correlated information to proper branch sub history is the main idea of the proposed correlation-based branch prediction. Basically, the proposed scheme uses the branch path information to split the history of a branch into several sub-histories and selectively use the proper sub history information for predicting the outcome of the branch.

Hence, in this project, to compare branch address prediction rate, branch direction prediction rate, address prediction hits, direction prediction hits, total number of miss prediction, and number of look-ups, we choose eight different strategies from static and dynamic branch predictor such as taken, not-taken, bimodal, two level adaptive, combination of bimodal and two level adaptive, one bit hash, two bit complementary hash, and finally our proposed strategy [15] [16] [13] [19]. Moreover, we choose two different various of *reduce instruction set computer* (RISC) architecture: $\alpha-$ architecture and *portable instruction set architecture* (PISA) [19]. Furthermore, we adopt six different benchmarks such as Anagram, Testfmath, Test-llong, Test-lswlr, Test-math, Test-printf to evaluate the performance of these strategies [20]. In particular, the key contributions in this work are:

- To implement eight different branch prediction strategies with the help of a computer architecture simulator named as SimpleScalar.
- To compare these strategies between two different architecture such as $\alpha-$ architecture and PISA
- To compare the area, power, and performance between two technology nodes of ASIC implementation.
- To analyze the among of these combination on six different benchmarks such as Anagram, Testfmath, Test-llong, Test-lswlr, Test-math, Test-printf.
- To implement our proposed XOR based hashing strategy with and analyze the outcomes based on hit rate, miss rate, branch address, and direction prediction.

In Section II, background and related work of our project is provided. In Section III, we discuss about our methodology for implementing these eight different strategies with the help of the SimpleScalar simulator [21] [22]. Section IV describes the results obtained in this work, while Section V, concludes this work. Section VI includes supplementary documents of our project.

II. Background & Related Work

Here, in this section, we discuss about eight different strategies that are considered for this project. For fairness in the comparison, we consider both static and dynamic strategies since the former is easier to implement than the later, however, suffers in terms of accuracy. In background & related work, we cover their operation principles, effectiveness as algorithms, and drawbacks.

A. Always Taken Branch Prediction Strategy

This strategy as illustrated in Fig. 3, assumes that the branch which is executing is always going to be taken [13] [12] [23]. In this strategy, branch target address has to be stored in the instructions fetch unit with zero delays. Here, backward branches (i.e. loop branches) are usually taken. It is a very primitive static strategy which has no direction prediction, and therefore, the accuracy for branch prediction is very low, usually 60-70% (approx.). As illustrated in Fig. 3, this strategy has two states naming “Predict Not Taken” and “Predict Taken”. Here, in each state, there is two possible directions. For instance, for “Predict Taken” state:

- Current State: “Predict Taken”; Actual Execution: “Taken”; Next State: “Predict Taken”
- Current State: “Predict Taken”; Actual Execution: “Not Taken”; Next State: “Predict Not Taken”

For “Predict Not Taken” state, opposite type of action occurs.

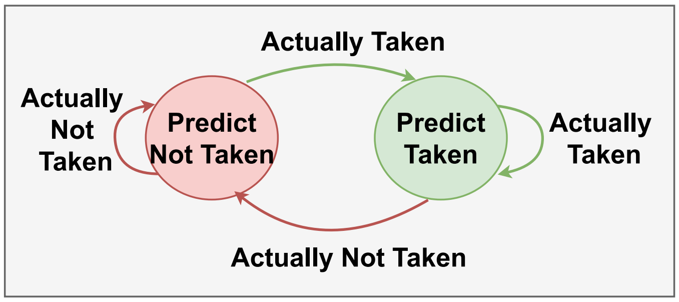


Fig. 3: Taken branch prediction method, where it assumes that the branch which is executing is always going to be taken. As it is a very primitive static strategy with very low accuracy, usually 60-70% (approx.)

B. Always Not-Taken Branch Prediction Strategy

This strategy as illustrated in Fig. 4, assumes that the branch which is executing is always going to be not taken [12] [24] [13]. This strategy executes the instructions after the branch speculatively, and then squashes the instruction execution in the case of miss-prediction. Similar to the taken strategy, it is also a very simple static strategy having no direction prediction, and hence, the accuracy is approximately 30-40%. As illustrated in Fig. 4, this strategy has two states naming ”Predict Not Taken”, which is also similar to the taken strategy. Here, in each state, there is two possible directions. For instance, for “Predict Not Taken” state:

- Current State: “Predict Not Taken”; Actual Execution: “Not Taken”; Next State: “Predict Not Taken”
- Current State: “Predict Not Taken”; Actual Execution: “Taken”; Next State: “Predict Taken”

For “Predict Taken” state, opposite type of action occurs.

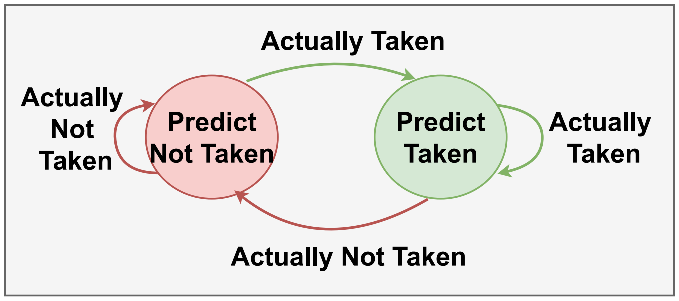


Fig. 4: Not taken branch prediction method, where it assumes that the branch which is executing is always going to be not taken. As it is a very primitive static strategy with very low accuracy, usually 30-40% (approx.)

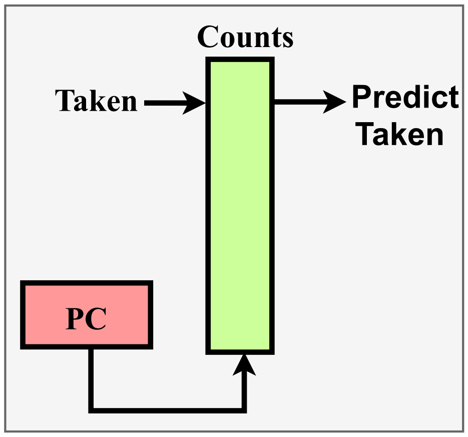


Fig. 5: Bimodal branch prediction method, where it takes advantage of this bimodal distribution of branch behavior and attempts to distinguish always taken branches from always not taken branches.

C. Bimodal Branch Prediction Strategy

Both the above mentioned strategies are very simple static strategies having a very low accuracy, which assume either branch is taken or not taken. Hence, to address this low accuracy of each method, a new strategy was proposed named as bimodal branch prediction as shown in Fig. 5, which takes advantage of this bimodal distribution of branch behavior and attempts to distinguish dynamically taken branches from not taken branches [25] [19]. As shown in Fig. 5, bimodal strategy consists of a stack full of counters, whose index is determined by the LSB bits of the program counter (PC). Each of these counters is two bits long, where the increment or decrement of these counters depend on the state of branch prediction. For each taken branch, the value of these counters increases by one whereas for each not taken branch, the value decreases by one. The most significant bit of these counter determine the direction of prediction, and hence, repeatedly taken branches are likely to be predicted as taken, whereas repeatedly not taken branches are predicted as not taken. Since, the counters stack using in the bimodal method is 2-bit, the predictor can tolerate a branch going an unusual direction one time and keep predicting the usual branch direction, which improve the accuracy of this strategy. For a large counter stack, each branch maps to a unique counter, on the contrary, for small stack, multiple branches share the same counter, resulting in degraded prediction accuracy. One alternate implementation was proposed, where a tag is attached with each counter and use a set-associative lookup to match counters with branches. For a fixed number of counters, a set-associative table has better performance. However, once the size of tags is accounted for, a simple array of counters often has better performance for a given predictor size. This would not be the case if the tags were already needed to support a branch target buffer.

D. Two-Level Adaptive Branch Predictor

As mentioned above, two-level adaptive as illustrated in Fig. 6, is dynamic branch predictor, which alters the branch prediction algorithm on the basis of information collected at run-time [15] [16] [26]. Actually, the run-time decision whether the branch will be taken or not, is done based on the history of branches executed during the current execution of the program. This execution history pattern information is collected on the fly of the program execution by updating the pattern history information in the branch history pattern (BHT) table of the predictor. Therefore, no re-runs of the program are necessary. In Fig. 6, a simple architecture is shown for two-level adaptive branch prediction, which mainly consists of two major data structures such as the branch history register (HR) and the branch pattern history table (PHT). Instead of accumulating statistics by profiling the programs, the execution history information on which branch predictions are based is collected by updating the contents of the history registers and the pattern history bits in the entries of the pattern table depending on the outcomes of the branches. The history register is a shift register which shifts in bits representing the branch results of the most recent history information. All the history registers are contained in a history register table (HRT). The pattern history bits represent the most recent branch results for the particular contents of the history register. Branch predictions are made by checking the pattern history bits in the pattern table entry indexed by the content of the history register for the particular branch that is being predicted. Since the history register table is indexed by branch instruction addresses, the history register table is called a per-address history register table (PHRT). The pattern table is called a global pattern table, because all the history registers access the same pattern table.

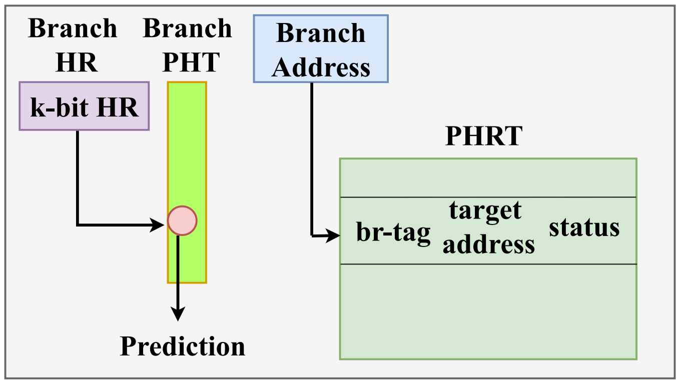


Fig. 6: Two-level adaptive branch prediction method which is based on the history of branches executed during the current execution of the program.

E. Combination of bimodal and two level adaptive method

To improve the accuracy of both bimodal and two level adaptive method, a hybrid prediction has been proposed with the combination of bimodal and two level adaptive branch prediction method. The main difference with the parent strategy is that branch prediction method depends on two branch history tables instead of one. Here, the first table records the history of recent branches through utilizing an array index determined by the low-order bits of the branch address, which solves the issue of degraded performance when multiple branches map to the same smaller stack entry for bimodal prediction. On the counterpart with two level adaptive method, each history table entry records the direction taken by the most recent branches whose addresses map to this entry and the second table is an array of 2-bit counters identical to those used for bimodal branch prediction. However, in hybrid method, the second table is indexed by the branch history stored in the first table. This approach is referred to as local branch prediction since the history used is local to the current branch. On the contrary, through the first table, the hybrid model takes the advantage of other recent branches to make a prediction, which utilizes a single shift register to record the direction taken by the most recent conditional branches. Since the branch history of the first table is global to all branches, this hybrid strategy is called global branch prediction.

As hybrid strategy is a global branch predictor, it takes advantage of two types of patterns. First, the direction taken by the current branch depends strongly on other recent branches. Second, global branch prediction is effective by duplicating the behavior of local branch prediction. This occurs when the global history includes all the local history needed to make an accurate prediction. As above-mentioned for small predictors, the bimodal scheme is relatively good since the branch address bits are used in the bimodal scheme efficiently distinguish different branches. However, as the number of counters doubles, roughly half as many branches share the same counter and the information content of the address bits is high. For large counter tables, as more counters are added, eventually each frequent branch map to a unique counter. Thus, the information content in each additional address bit declines to zero for increasingly large counter tables. The information content of the global history register begins relatively small, but it continues to grow for larger sizes. Since most of the time each branch goes the same direction, the sequence of previous branches and the directions taken by these branches tend to be highly repetitive for any one branch, but perhaps very different for other branches. This behavior allows a global predictor to identify different branches.On the other hand, the global history register can capture more information than just identifying which branch is current, and thus for sufficiently large predictors it does better than bimodal prediction. The global history information is less efficient at identifying the current branch than simply using the branch address. This suggests that a more efficient prediction might be made using both the branch address and the global history. Here the counter table is indexed with a concatenation of global history and branch address bits which is defined as combination of bimodal and two level adaptive as shown in Fig. 7. With the bit selection approach, there is a trade off between using more history bits or more address bits. It performs better than either bimodal or two level adaptive branch prediction since both are essentially degenerate cases. For small sizes, it is best paralleled to the performance of bimodal prediction. However, once there are enough address bits to identify most branches, more global history bits are used, resulting in significantly better prediction results than the bimodal scheme. This method also significantly outperforms simple global prediction for most predictor sizes because the branch address bits more efficiently identify the branch.

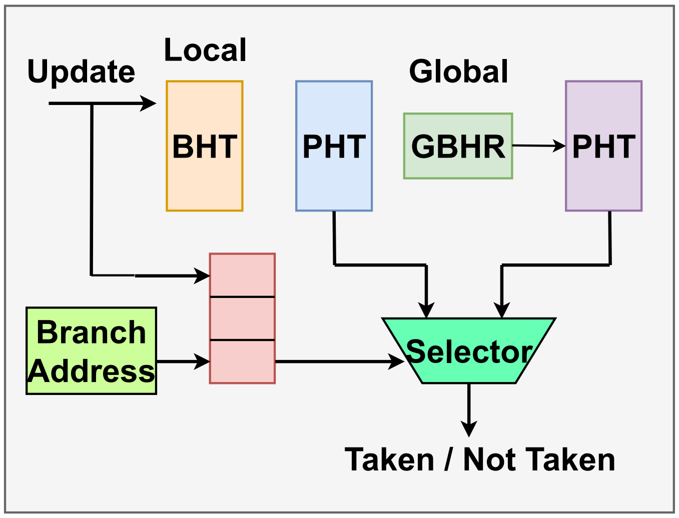


Fig. 7: Combination of bimodal and two level adaptive branch prediction method takes advantages of bimodal and two level adaptive method.

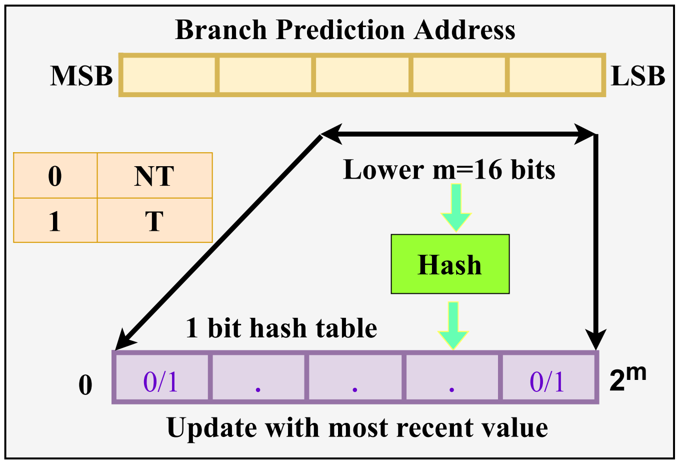


Fig. 8: One bit hash branch prediction method where lower m bits of a branch instruction address are hashed for indexing branch history table (BHT) in the random access memory (RAM)

F. *One bit hash*

Another popular branch prediction strategy is one bit hash which is illustrated in Fig. 8, which is also known as ”Strategy Six” as proposed in [13]. This strategy uses a 1-bit counter to predict the branch direction. The lower m bits of branch instruction address is hashed for indexing branch history table (BHT) in the random access memory (RAM). The indexed memory location contains one-bit history counter, which is the outcome of the most recent execution of that branch. In Fig. 8, it can be seen that the lower (from LSB) m = 16 bits are taken from Branch Prediction Address and hashed for indexing 1 bit hash table (BHT) which is updated with most recent value. This data bit decides the prediction direction for current execution. If the counter value is 1, the branch direction is predicted as taken (T). If the counter value is 0, the branch direction is predicted as not taken (NT). If the prediction is wrong, the history bit is updated accordingly. The default prediction or first time prediction can be initialized as taken or not taken for all the branches. In this project, the default prediction was taken. The size of the 1 bit hash table is determined by the m bits used for hashing. So, the total entries in the 1 bit hash table are 2^{16} for 16 bits. The algorithm 1 for one bit hash is given below.

G. *Two bit complementary hash*

Two bit complementary hash is illustrated in Fig.9, which is designed to deal effectively with the anomalous decisions. The difference between this and the previous method is instead of using a 1-bit counter, it uses N-bit 2’s complement counter for branch prediction direction [13]. Similar to the previous method, the lower m bits of branch instruction address is hashed for indexing branch history table (BHT) in the random access memory (RAM). The N bit 2’s complement history counter is contained in the indexed memory location representing the previous branch execution. In Fig. 9, it has been

Algorithm 1 one bit hash

```
1: Input: branch
2: Output: branch prediction
3:
4: function PREDICT_BRANCH(b)
5:     Direction with branch address $b$
6:     $bits = LOWER\_M\_BITS(b)$
7:     $hash = HASH(bits)$
8:     $val = INDEX(hash)$
9:     if $val == 1$ then
10:        return taken
11:    else if $val == 0$ then
12:        return not_taken
13:
14: function UPDATE_BRANCH(b,actual)
15:    Table with branch address b with real outcome actual
16:    $bits = LOWER\_M\_BITS(b)$
17:    $hash = HASH(bits)$
18:    $val = INDEX(hash)$
19:    $INDEX(hash) = actual$
```

illustrated that the lower (from LSB) m = 16 bits are taken from Branch Prediction Address and hashed for indexing 1 bit hash table (BHT) which is updated with most recent value. If the sign-bit is 0, the branch direction is predicted as taken (T). If the sign-bit is 1, the branch direction is predicted as not taken. If the prediction is wrong, the history bit is updated accordingly. The default prediction or first time prediction can be initialized as taken or not taken for all the branches. In this project, the default prediction was Taken (sign-bit 0). The counter is incremented if the branch is taken and decremented if it is not taken. The size of the 1 bit hash table is determined by the m bits used for hashing. So, the total entries in the 1 bit hash table are 2^{16} for 16 bits. The algorithm 2 for two bit complementary hash is given below.

III. Methodology

In this project, we have implemented eight branch prediction strategies such as taken, not taken, bimodal, two level adaptive, combination of bimodal and two level adaptive, one bit hash, two bit complementary hash and our proposed method. These schemes are executed on six benchmarks such as Anagram, Test-fmath, Test-llong, Test-lswlr, Test-math and Test-printf [20]. The $\alpha-$ architecture and the PISA architecture are utilized as ISA for this project [27] [28]. All these executions are done in SimpleScalar software for evaluating prediction rate, hit rate, number of lookups and miss rate of programs.

A. Benchmark

As aforementioned, we employ six different benchmarks, which contains a set of instructions for evaluation of branch prediction. Benchmarks are written in c language and by running these using a cross gcc compiler to build the executable code. First of all, analysis of these six different benchmarks is presented in terms of the total number of instructions, loads,

Algorithm 2 two bit complementary hash

```
1: Input: branch b
2: Output: branch prediction
3: function PREDICT_BRANCH(b)
4:     Direction with branch address $b$
5:     $bits = LOWER\_M\_BITS(b)$
6:     $hash = HASH(bits)$
7:     $val = INDEX(hash)$
8:     if $val == 0$ then
9:         return taken
10:    else if $val == 1$ then
11:        return not_taken
12:
13: function UPDATE_BRANCH(b,actual)
14:    Table with branch address b with real outcome actual
15:    $bits = LOWER\_M\_BITS(b)$
16:    $hash = HASH(bits)$
17:    $val = INDEX(hash)$
18:    if $val == actual$ then
19:        $INDEX(hash) = INDEX(hash) + 1$
20:    else if $val \neq actual$ then
21:        $INDEX(hash) = INDEX(hash) - 1$
```

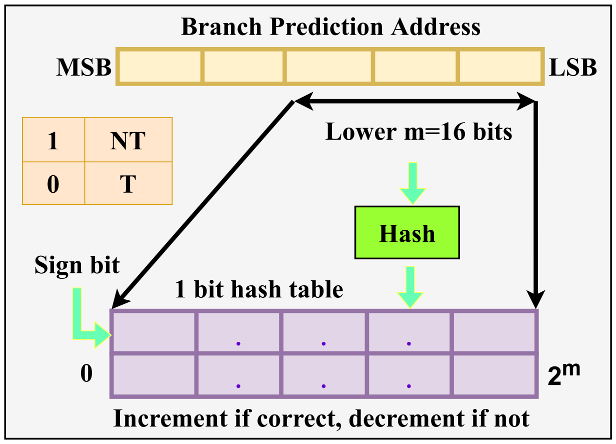

Branch Prediction Address
MSB
LSB
Lower m=16 bits
1 NT
0 T
Hash
Sign bit
1 bit hash table
0
2^m
Increment if correct, decrement if not

Fig. 9: Two bit complementary hash branch prediction method which uses N-bit 2's complement counter for branch prediction.

and stores, and the number of branches, which is illustrated in Fig. 10. For better visibility, we use natural logarithmic value of total number of instructions, loads, and stores, and the number of branches. As illustrated in Fig. 10, the minimum total number of instruction among the benchmarks is 7070, the minimum number of total loads and stores 3906, and the minimum number of branches among the benchmarks are found to be 941. This suggests the ability of these benchmarks to depict the comparison of these strategies in terms of branch address and direction prediction, and total number of misses.

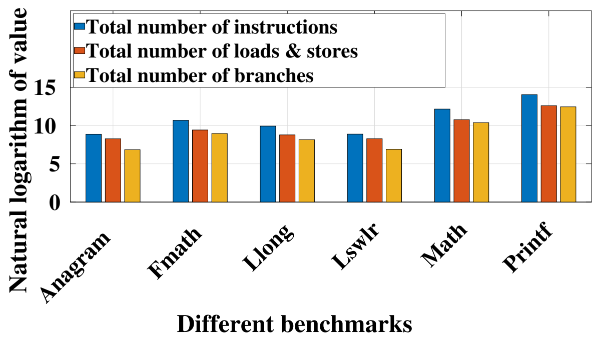
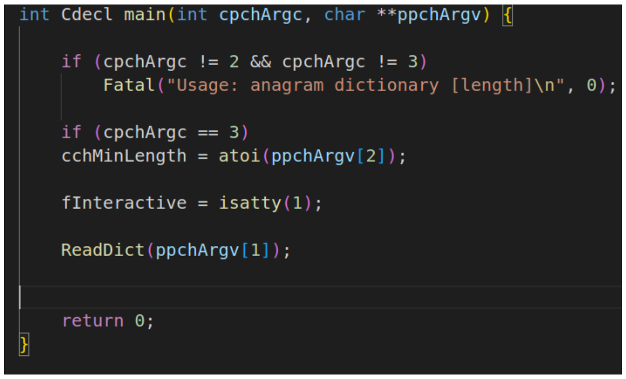


Fig. 10: Different test benchmarks comparison in terms of total number of instructions, the total number of loads stores, and the total number of branches are shown. We take the natural logarithm of the total number of instructions, loads and stores, and branches for visibility.

Now, we describe these benchmarks in brief.

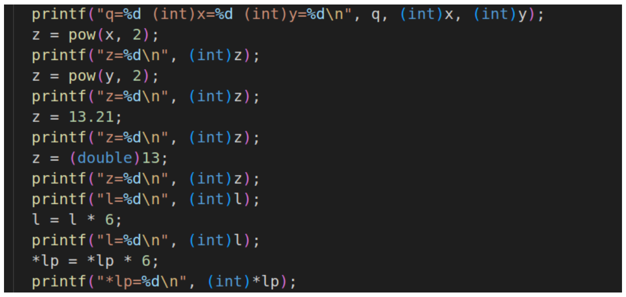
1) Anagram: A subset of c program for finding anagrams for a phrase, based on a dictionary is shown in Fig. 11. There are two tricks. First trick when the user types in a phrase, the phrase is first pre-processed to determine how many of each letter appears. A bit field is then constructed dynamically such that each field is large enough to hold the next power of two larger than the number of times the character appears. The inner loop of an anagram program is doing testing to see if a word fits in the collection of untried letters. Traditional methods keep an array of 26 integers, which are then compared in turn. This means that there are 26 comparisons per word. Second trick before diving into anagram search, how many times each letter appears in the list of candidates is tabulated, and the table is sorted with the rarest letter first. Like most anagram programs, this program does a depth-first search. Although most anagram programs do some sort of heuristics to decide what order to place words in the list of candidates, the search itself proceeds according to a greedy algorithm. When a word that fits is found, it is subtracted and re-cursed.



```
int Cdecl main(int cpchArgc, char **ppchArgv) {

    if (cpchArgc != 2 && cpchArgc != 3)
        Fatal("Usage: anagram dictionary [length]\n", 0);

    if (cpchArgc == 3)
    cchMinLength = atoi(ppchArgv[2]);

    fInteractive = isatty(1);

    ReadDict(ppchArgv[1]);


    return 0;
}
```

Fig. 11: A subset of anagram C implementation, where the main purpose is to find the phrase in a sentence.

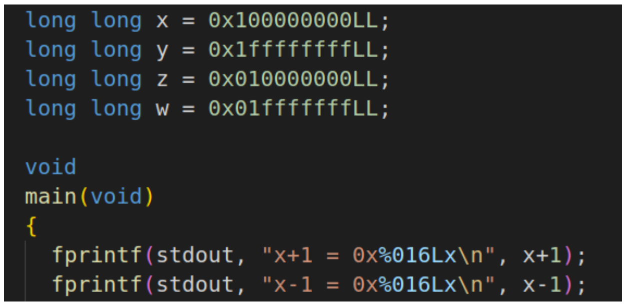
2) Test-fmath: A subset of c program performs various math functions mostly in integers is ahown in Fig. 12. It has done integer-type cast operations which actually converts a floating number into an integer number. When we do type cast, it actually takes a lot of computation at the hardware level. So, if we can avoid it using branch prediction, it may mitigate some computational costs in run-time. Along with that it also executes power function, addition, subtraction, multiplication, etc operation for integer, float, double, and long integer types which are costly. If we do this operation when it is not wanted, computational complexity may be increased which is unwanted.



```
printf("q=%d (int)x=%d (int)y=%d\n", q, (int)x, (int)y);
z = pow(x, 2);
printf("z=%d\n", (int)z);
z = pow(y, 2);
printf("z=%d\n", (int)z);
z = 13.21;
printf("z=%d\n", (int)z);
z = (double)13;
printf("z=%d\n", (int)z);
printf("l=%d\n", (int)l);
l = l * 6;
printf("l=%d\n", (int)l);
*lp = *lp * 6;
printf("*lp=%d\n", (int)*lp);
```

Fig. 12: A subset of fmath C implementation, where the main purpose is to execute mathematical function of integer datatype only.

3) Test-llong: A subset of c program performs computations in a long format is shown in Fig. 13. It executes different types of operations such as addition, subtraction, multiplication, etc among various long hexadecimal. We know that memory is addressed by hexadecimal numbers. So, for doing memory operations, this kind of computation is very helpful. So, branch prediction may speed up the memory access and avoid unwanted memory access.

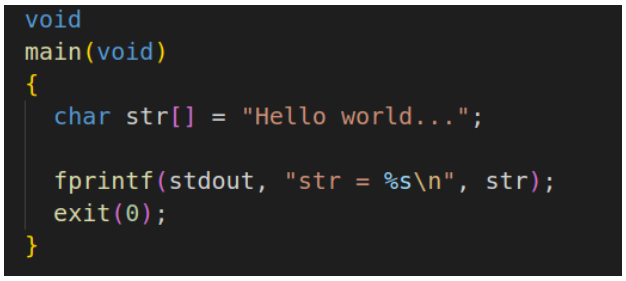


```
long long x = 0x100000000LL;
long long y = 0x1ffffffffLL;
long long z = 0x010000000LL;
long long w = 0x01fffffffLL;

void
main(void)
{
  fprintf(stdout, "x+1 = 0x%016Lx\n", x+1);
  fprintf(stdout, "x-1 = 0x%016Lx\n", x-1);
```

Fig. 13: A subset of llong C implementation, where the main purpose is to execute long integer datatype.

4) Test-lswlr: A subset of c program performs printing Hello World as a string is shown in Fig. 14. At the hardware level, the printf function how deals with a string that is depicted in this code.

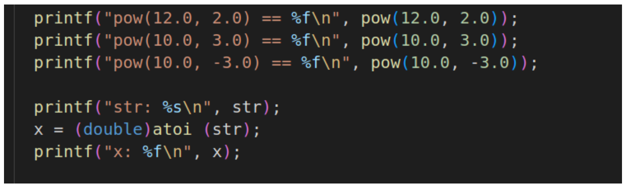
5) Test-math: A subset of c program performs various math computations mostly in integers and displays their result is shown in Fig. 15. This c program executes the power function,



```c
void
main(void)
{
  char str[] = "Hello world...";

  fprintf(stdout, "str = %s\n", str);
  exit(0);
}
```

Fig. 14: A subset of lswlr C implementation, where the main purpose is to execute string datatype especially "Hello world"

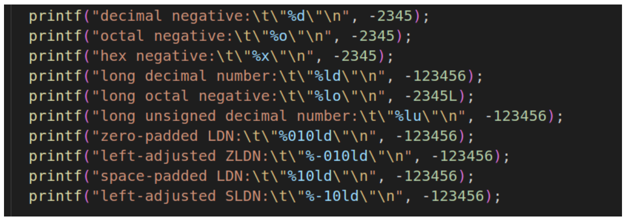
trigonometric function, exponential function, logarithm function, etc. For doing this operation, computational complexity is going to be increased in the hardware level which may hamper the regular execution without proper branch prediction.



```c
printf("pow(12.0, 2.0) == %f\n", pow(12.0, 2.0));
printf("pow(10.0, 3.0) == %f\n", pow(10.0, 3.0));
printf("pow(10.0, -3.0) == %f\n", pow(10.0, -3.0));

printf("str: %s\n", str);
x = (double)atoi (str);
printf("x: %f\n", x);
```

Fig. 15: A subset of math C implementation, where the main purpose is to execute mathematical function for all datatype.

6) Test-printf: A subset of c program displays various print statements is shown in Fig. 16. This program prints different types of data structures, data formats, variables, etc. For creating user-defined data types in C/C++, the structure keyword is used. A structure creates a new user-defined data type that is used to group items of possibly different types into a single type. So, handling this data in hardware level is not so easy. It takes huge computation. Proper branch prediction may save our computation.



```c
printf("decimal negative:\t\"%d\"\n", -2345);
printf("octal negative:\t\"%o\"\n", -2345);
printf("hex negative:\t\"%x\"\n", -2345);
printf("long decimal number:\t\"%ld\"\n", -123456);
printf("long octal negative:\t\"%lo\"\n", -2345L);
printf("long unsigned decimal number:\t\"%lu\"\n", -123456);
printf("zero-padded LDN:\t\"%010ld\"\n", -123456);
printf("left-adjusted ZLDN:\t\"%-010ld\"\n", -123456);
printf("space-padded LDN:\t\"%10ld\"\n", -123456);
printf("left-adjusted SLDN:\t\"%-10ld\"\n", -123456);
```

Fig. 16: A subset of printf C implementation, where the main purpose is to print data structure.

B. Architecture

In this project, we resort to two different architectures and compares the branch prediction strategies among them using above-mentioned benchmarks. These *instruction set architecture* (ISA) are α architecture and PISA architecture. They are described as follows:

1) $\alpha-$ architecture: $\alpha-$ architecture is a 64-bit reduced instruction set computer (RISC) instruction set architecture (ISA) as depicted in Fig. 17. It has 215 instructions with 4 instruction formats. α is implemented in a series of microprocessors that DEC designed and manufactured [27]. These microprocessors are first utilized in a range of DEC workstations and servers, and they eventually become the foundation for practically all of their mid-to-upper-scale offerings. OpenVMS, Tru64 UNIX, Windows NT, Linux, BSD UNIX, and Bell Labs Plan 9 are among the operating systems that supported α. During the development of the α architecture, a port of Ultrix was made, but it was never offered as a product.

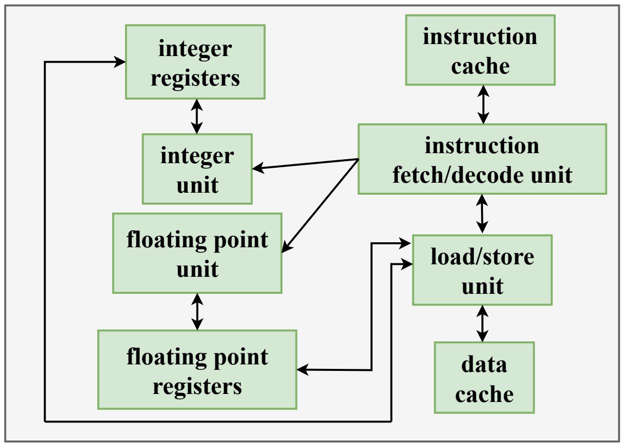


Fig. 17: $\alpha-$ architecture which is a 64-bit reduced instruction set computer (RISC) instruction set architecture (ISA).

2) PISA Architecture: The PISA instruction set architecture (Portable Instruction Set Architecture) is a simple MIPS-like instruction set maintained primarily for instructional use as depicted in Fig. 18. It has 135 instructions with 4 instruction formats. PISA is a free, open-source instruction set architecture (ISA) based on RISC concepts [28]. PISA, unlike most other ISA designs, is available under open source licenses that do not need payment. PISA support is available in a number of open source operating systems, and the instruction set is supported in several prominent software toolchains. A load–store architecture is the PISA architecture. IEEE 754 floating-point instructions are used in its floating-point instructions. PISA is an aesthetically neutral design that places the most-significant parts of immediate values in a fixed spot to speed up sign expansion.

C. Proposed Method

There is a lot of redundancy in the counter index used by the previous methods. If there are enough address bits to identify the branch, the frequent global history combinations must be rather sparse. This effect may be treated as an advantage by hashing the branch address and the global history (1 bit hash table) together. In particular, the exclusive OR between higher m bits (from MSB) and lower m bits (from LSB) of the branch address then hashing with the global history is expected to have more information than either component

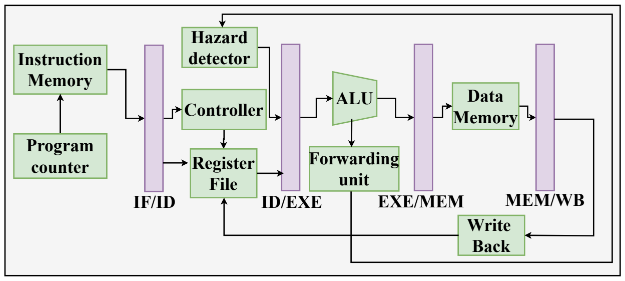


Fig. 18: $PISA-$ architecture which is a simple MIPS-like instruction set maintained primarily for instructional use.

alone. Moreover, since more address bits and global history bits are in use, there is reason to expect better predictions than the previous methods. In Fig. 19, the architecture of the proposed method is depicted. Both one bit hash and two bit complementary hash use the lower m bits of the branch instruction address. In many cases, the lower m-bits may be same for multiple branch addresses. In that case, the previous value will be overwritten or updated by both methods. In our proposed branch prediction method, both the lower and the higher m bits. The XOR function on the two address values has been used to make similar size. By doing this, the problem will not be resolved completely but the probability of the number of times the value will be same is lower than the previous methods. After that the XOR value will be hashed to find the index in the hash table.The value is set to 0 for initialization. If the value of that index is greater than 0, then this method will predict taken, otherwise not taken. For the update operation, the value is incremented if the prediction is correct to give more confidence in it, otherwise the value is decremented to show less confidence. The size of the 1 bit hash table is determined by the m bits used for hashing. So, the total entries in the 1 bit hash table will be 2^{16} for 16 bits. The algorithm 3 for the proposed method is given below.

D. SimpleScalar simulator

The SimpleScalar architectural simulator is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro architectural modeling, and hardware-software co-verification. It is an open source computer architecture simulator, which models a virtual computer system such as cache memory and branch predictor that can be specified by the user [22] [21] [29]. Using the SimpleScalar simulator, users can build modeling applications that simulate real programs running on a range of modern processors and systems. This tool includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. The branch prediction simulator in Simplescalar implements the state of the art algorithms: branch always not taken, branch always taken, bimodal, two-level adaptive and combination of bimodal and two-level adaptive predictors. It also provides software infrastructure for the development of new branch

Algorithm 3 Proposed methodology

```
1:  Input: branch b
2:  Output: branch prediction
3:
4:  function PREDICT_BRANCH(b)
5:      Direction with branch address $b$
6:      $lbits = LOWER\_M\_BITS(b)$
7:      $ubits = HIGHER\_M\_BITS(b)$
8:      $bits = XOR(lbits, ubits)$
9:      $hash = HASH(bits)$
10:     $val = INDEX(hash)$
11:     if $val > 0$ then
12:         return taken
13:     else if $val < 0$ then
14:         return not_taken
15:
16: function UPDATE_BRANCH(b,actual)
17:     Table with branch address b with real outcome $actual$
18:     $lbits = LOWER\_M\_BITS(b)$
19:     $ubits = HIGHER\_M\_BITS(b)$
20:     $bits = XOR(lbits, ubits)$
21:     $hash = HASH(bits)$
22:     $val = INDEX(hash)$
23:     if $val == actual$ then
24:         $INDEX(hash) = INDEX(hash) + 1$
25:     else if $val \neq actual$ then
26:         $INDEX(hash) = INDEX(hash) - 1$
```

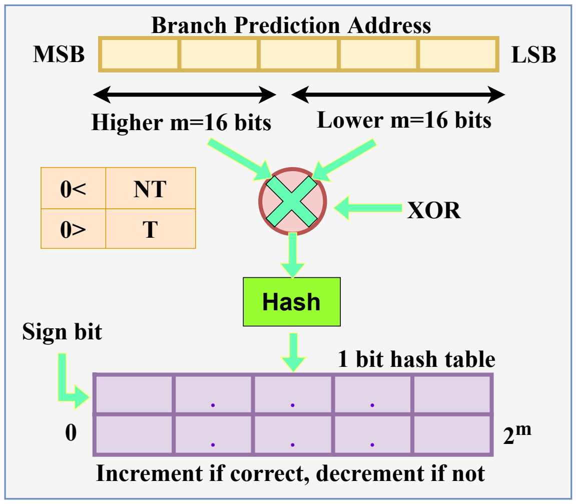
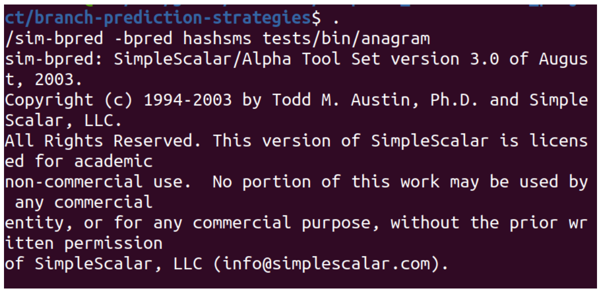


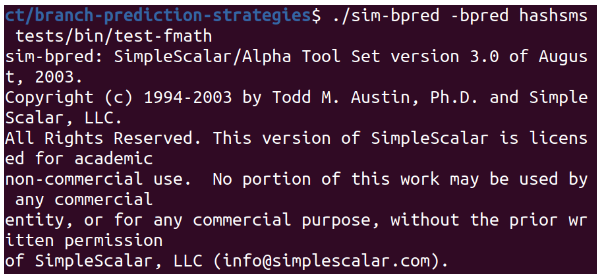
Fig. 19: Proposed branch prediction method which does exclusive or operation between higher m bits and lower m bit for hashing in global history table.

predictors and can be used for evaluating prediction rate, hit rate, number of lookups and miss rate of programs as illustrated in Fig. 20. There are many commands which help simulating different types of work such as sim-fast as a functional simulator, sim-outorder as a micro architecture simulator, sim-eio as check pointing and fast forwarding,



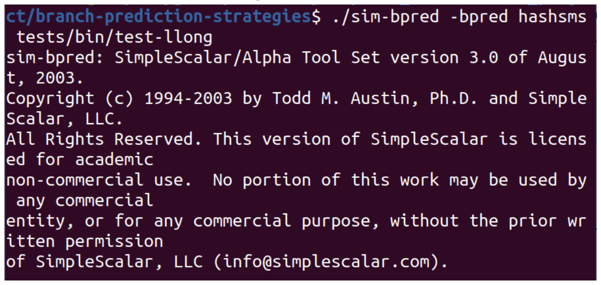
```
ct/branch-prediction-strategies$ .
/sim-bpred -bpred hashsms tests/bin/anagram
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(a) Anagram benchmark.



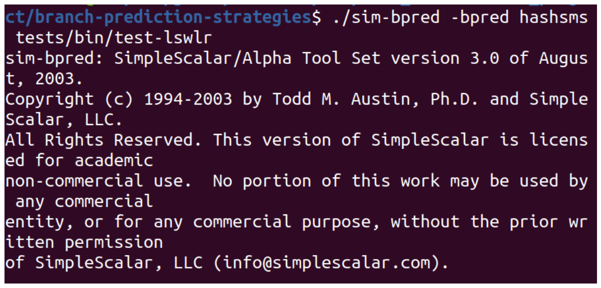
```
ct/branch-prediction-strategies$ ./sim-bpred -bpred hashsms
 tests/bin/test-fmath
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(b) Test-fmath benchmark.



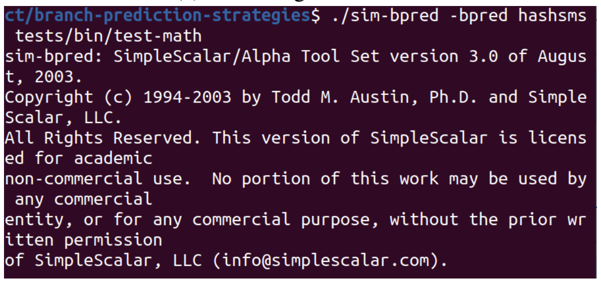
```
ct/branch-prediction-strategies$ ./sim-bpred -bpred hashsms
 tests/bin/test-llong
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(c) Test-llong benchmark.



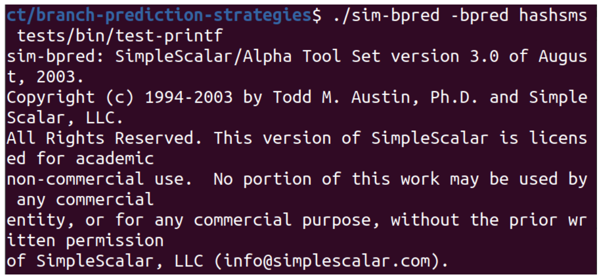
```
ct/branch-prediction-strategies$ ./sim-bpred -bpred hashsms
 tests/bin/test-lswlr
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(d) Test-lswlr benchmark.



```
ct/branch-prediction-strategies$ ./sim-bpred -bpred hashsms
 tests/bin/test-math
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(e) Test-math benchmark



```
ct/branch-prediction-strategies$ ./sim-bpred -bpred hashsms
 tests/bin/test-printf
sim-bpred: SimpleScalar/Alpha Tool Set version 3.0 of Augus
t, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and Simple
Scalar, LLC.
All Rights Reserved. This version of SimpleScalar is licens
ed for academic
non-commercial use.  No portion of this work may be used by
 any commercial
entity, or for any commercial purpose, without the prior wr
itten permission
of SimpleScalar, LLC (info@simplescalar.com).
```

(f) Test-printf benchmark

Fig. 20: An instance of running the proposed method on different benchmarks using SimpleScalar simulator. The proposed method runs in anagram, test-fmath, test-llong, test-lslwr, test-math and test-printf benchmark. A subset of output from anagram, test-fmath, test-llong, test-lslwr, test-math and test-printf benchmark is illustrated in Figures 20a,20b,20c,20d,20e and, 20f

sim-profile as an execution profiler, sim-bpred as a branch prediction simulator, sim-cache as a cache simulator and sim-cheetah as an advanced cache simulator. After installation of SimpleScalar, the choice of ISA should be decided before running the branch prediction in the linux system. If Alpha ISA is chosen, the command “make clean” followed by “make config-alpha” has to be written in the terminal for configuring Alpha ISA. For configuration of PISA ISA, the same rule follows only replacing the name. After making configuration of the ISA, the simulation type, the branch prediction type and the benchmark have to be chosen. For an example if sim-bpred as a branch prediction simulator, bimodal method as a branch predictor and anagram as a benchmark are chosen, the command for executing this will be “./sim-bpred -bpred bimod tests/bin/anagram” where “sim-bpred” represents the simulation type, “bimod” represents the bimodal method and “tests/bin/anagram” represents the directory of the anagram benchmark. In the same way other branch predictors and other benchmarks can be used for branch prediction. By this way, prediction rate, hit rate, number of lookups and miss rate of programs can be observed for different branch predictors and different benchmarks.

IV. Result and Discussion

This section compares the branch address prediction rate, branch direction prediction rate, address prediction hits, direction prediction hits, total number of miss prediction, and number of look-up among different architecture, strategies, and benchmarks. To perform this analysis, we adopt following experimental setup.

A. Experimental setup and data set

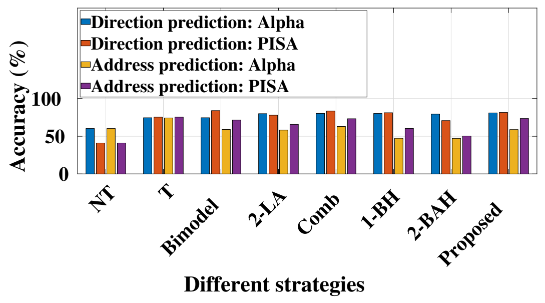
As mentioned in sub-section III-A, we adopt six different benchmarks for our project. For software implementation purpose, we chose SimpleScalar architectural software which run in AMDR Ryzen(R) 7 5700G CPU. For data curation and result analysis, we adopt python framework and Matlab academic edition R2022a.

TABLE I: The performance of eight different strategies on Anagram data-set using α architecture. Here, taken strategy has the highest branch address prediction rate and our proposed strategy has the highest branch direction prediction rate.

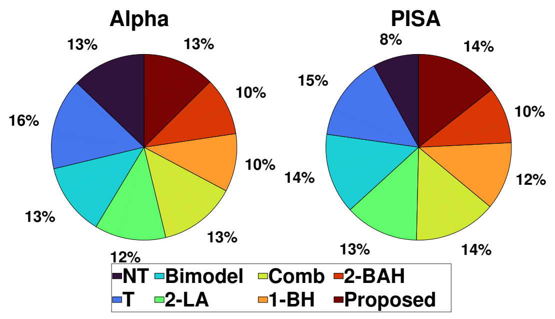
Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	526	526	347	60.25	60.25	873
Taken	646	646	227	**74.24**	74.55	873
Bimod	515	686	187	58.99	74.58	873
2lev	508	698	175	58.19	79.95	873
Comb	550	701	172	63.00	80.30	873
1 bit hash	413	700	173	47.31	80.18	873
2 bit comp hash	412	693	180	47.19	79.38	873
Proposed	426	698	175	58.83	**80.95**	873

TABLE II: The performance of eight different strategies on Anagram data-set using $PISA$ architecture. Here, taken strategy has the highest branch address prediction rate and bimodal strategy has the highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy.

Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	443	443	637	41.02	41.02	1080
Taken	814	814	266	**75.37**	75.37	1080
Bimod	907	772	173	71.48	**83.98**	1080
2lev	842	710	238	65.74	77.96	1080
Comb	791	901	179	73.24	83.43	1080
1 bit hash	652	877	203	60.37	81.20	1080
2 bit comp hash	543	764	316	50.28	70.74	1080
Proposed	556	772	308	73.48	81.48	1080



(a) Accuracy comparison.



(b) Miss rate comparison

Fig. 21: Anagram bench mark for comparing accuracy and miss rate among eight different strategies.

B. *Performance Analysis on Anagram Benchmark*

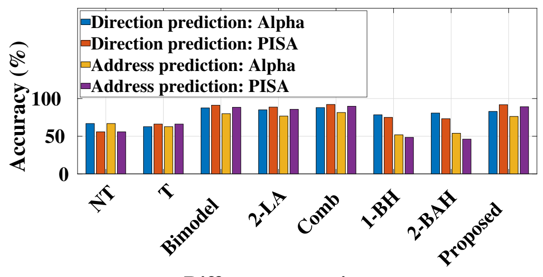
In Fig. 21, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 21a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate. Here, taken strategy has the highest branch address prediction rate and our proposed strategy has the highest branch direction prediction rate for α architecture. For $pisa$ architecture, taken strategy has the highest branch address prediction rate and bimodal strategy has the highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy.In Fig. 21b, combination has the lowest miss-rate for α architecture. Our proposed strategy is close to it. 2 level adaptive has the lowest miss-rate for $PISA$ architecture.Here, our proposed one does well compared to 1 bit hash but not well compared to 2 level adaptive strategy.

TABLE III: The performance of eight different strategies on Test-fmath data-set using α architecture. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy does not perform well compared to the combination strategy.

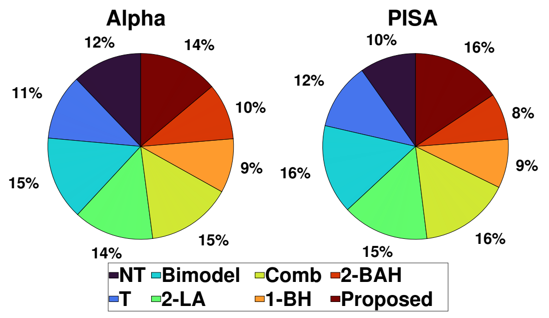
Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	2072	2072	1028	66.84	66.84	3100
Taken	1944	1944	1156	62.71	62.71	3100
Bimod	2478	2714	386	79.94	87.55	3100
2lev	2380	6335	465	76.77	85.00	3100
Comb	2526	2729	371	**81.48**	**88.03**	3100
1 bit hash	1606	2430	670	51.81	78.39	3100
2 bit comp hash	1671	2504	596	53.90	80.77	3100
Proposed	1744	2570	530	76.26	82.90	3100

TABLE IV: The performance of eight different strategies on Test-fmath data-set using $PISA$ architecture. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy is close to the combination strategy.

Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	5768	5768	4572	55.78	55.78	10340
Taken	6839	6839	3501	66.14	66.14	10340
Bimod	9139	9426	914	88.38	91.16	10340
2lev	8860	9171	1169	85.69	88.69	10340
Comb	9284	9528	812	**89.79**	**92.15**	10340
1 bit hash	5008	7752	2588	48.43	74.97	10340
2 bit comp hash	4778	7573	2767	46.11	73.24	10340
Proposed	5083	9445	2395	89.16	91.84	10340



(a) Accuracy comparison.



(b) Miss rate comparison.

Fig. 22: Test fmath bench mark for comparing accuracy and miss rate among eight different strategies.

C. Performance Analysis on Test Fmath Benchmark

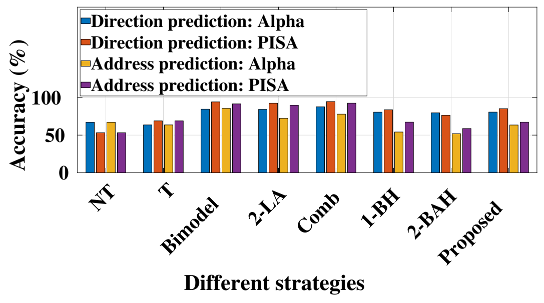
In Fig. 22, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 22a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy does not perform well compared to the combination strategy. In Fig. 22b, combination has the lowest miss-rate for α and $PISA$ architecture.Our proposed strategy performs well than both one bit hash and two bit complementary strategy.

TABLE V: The performance of eight different strategies on Test-llong data-set using α architecture. Here, bimodal strategy has the highest branch address prediction rate and combination strategy has the highest branch direction prediction rate. Our proposed strategy perform better than both one bit hash and two bit complementary.

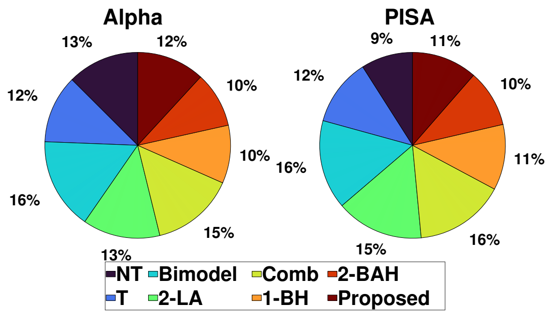
Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	1261	1261	618	67.11	67.11	1879
Taken	1194	1194	685	63.54	63.54	1879
Bimod	1402	1607	272	**85.52**	84.47	1879
2lev	1358	1582	297	72.27	84.19	1879
Comb	1463	1645	234	77.86	**87.55**	1879
1 bit hash	1017	1513	366	54.12	80.52	1879
2 bit comp hash	974	1495	384	51.84	79.56	1879
Proposed	1004	1514	365	53.43	80.57	1879

TABLE VI: The performance of eight different strategies on Test-llong data-set using $PISA$ architecture. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy performs well but not well compared to the combination strategy.

Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	3119	3119	2751	53.13	53.13	5870
Taken	4045	4045	1825	68.91	68.91	5870
Bimod	5369	5525	345	91.47	94.12	5870
2lev	5566	5425	445	89.71	92.42	5870
Comb	5427	5551	316	**92.45**	**94.57**	5870
1 bit hash	3947	4908	962	67.24	83.61	5870
2 bit comp hash	3448	4480	1390	58.74	76.32	5870
Proposed	3944	4995	875	67.19	85.09	5870



(a) Accuracy comparison.



(b) Miss rate comparison.

Fig. 23: Test llong bench mark for comparing accuracy and miss rate among eight different strategies.

D. Performance Analysis on Test Llong Benchmark

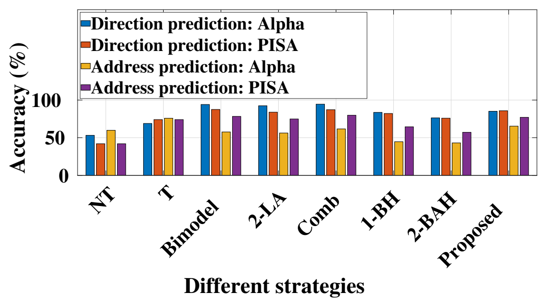
In Fig. 23, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 23a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate.For α architecture, bimodal strategy has the highest branch address prediction rate and combination strategy has the highest branch direction prediction rate. Our proposed strategy perform better than both one bit hash and two bit complementary. For $pisa$ architecture, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy performs well but not well compared to the combination strategy. In Fig. 23b, combination has the lowest miss-rate for α and $PISA$ architecture.Our proposed strategy performs well than other strategies but not well than combination strategy.

TABLE VII: The performance of eight different strategies on Test-lswlr data-set using α architecture. Here, taken strategy has the highest branch address prediction rate and our proposed strategy strategy highest branch direction prediction rate.

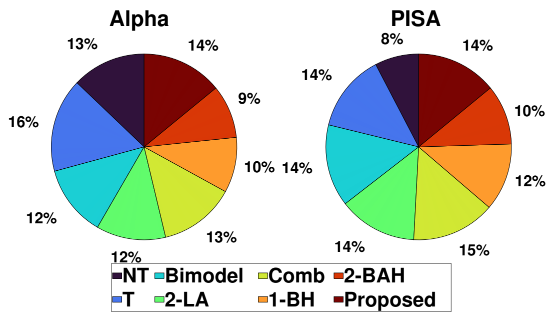
Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	518	518	348	59.82	59.82	866
Taken	657	657	209	**75.87**	75.87	866
Bimod	499	682	184	57.62	78.75	866
2lev	487	687	179	56.24	79.33	866
Comb	534	694	172	61.66	80.14	866
1 bit hash	387	687	179	44.69	79.33	866
2 bit comp hash	373	668	198	43.07	77.14	866
Proposed	393	701	185	65.38	**80.64**	866

TABLE VIII: The performance of eight different strategies on Test-lslwr data-set using $PISA$ architecture. Here, combination strategy has the highest branch address prediction rate and bimodal has the highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy.

Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	588	588	813	41.97	41.97	1401
Taken	1038	1038	363	74.09	74.09	1401
Bimod	1097	1226	175	78.3	**87.51**	1401
2lev	1049	1176	285	74.88	83.94	1401
Comb	1119	1222	179	**79.87**	87.22	1401
1 bit hash	903	1151	250	64.45	82.16	1401
2 bit comp hash	801	1064	337	57.17	75.95	1401
Proposed	799	1192	339	77.03	85.80	1401



(a) Accuracy comparison.



(b) Miss rate comparison.

Fig. 24: Test lswlr bench mark for comparing accuracy and miss rate among eight different strategies.

E. Performance Analysis on Test lswlr Benchmark

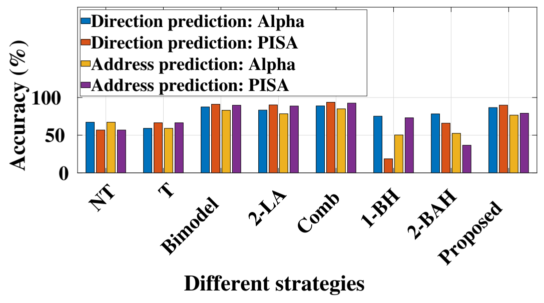
In Fig. 24, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 24a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate.For α architecture, taken strategy has the highest branch address prediction rate and our proposed strategy strategy highest branch direction prediction rate. For $pisa$ architecture, combination strategy has the highest branch address prediction rate and bimodal has the highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy. In Fig. 24b, combination has the lowest miss-rate for α architecture and $PISA$ architecture.Our proposed strategy is not close to it but performs well than other strategies.

TABLE IX: The performance of eight different strategies on Test-math data-set using α architecture. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. If we consider branch direction prediction rate, our proposed strategy is close to the combination strategy.

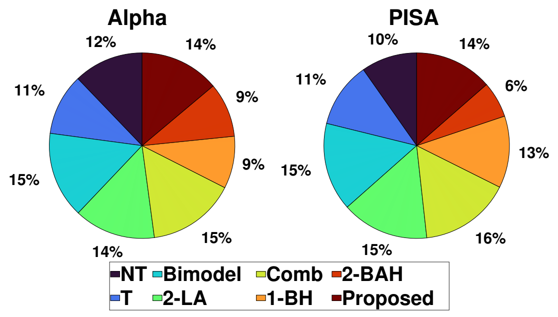
Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	4975	4975	2427	67.21	67.21	7402
Taken	4373	4373	3029	59.08	59.08	7402
Bimod	6148	6476	926	83.06	87.49	7402
2lev	5806	6167	1235	78.44	83.32	7402
Comb	6290	6578	824	**84.98**	**88.87**	7402
1 bit hash	3721	5570	1832	50.27	75.25	7402
2 bit comp hash	3881	5793	1609	52.43	78.26	7402
Proposed	4192	6420	1282	76.63	86.68	7402

TABLE X: The performance of eight different strategies on Test-math data-set using $PISA$ architecture. Here, bimodal strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy if we consider branch direction prediction rate.

Strategy	Address predicted hits	Direction predicted hits	Number of misses	Branch address prediction rate	Branch direction prediction rate	Number of lookups
Not-taken	21938	21938	16653	56.85	56.85	38591
Taken	25661	25661	12930	66.49	66.49	38591
Bimod	34647	35132	3459	**89.78**	**91.04**	38591
2lev	34242	34794	3797	88.73	90.16	38591
Comb	35736	36165	2426	92.60	93.71	38591
1 bit hash	16754	28208	10383	73.09	18.60	38591
2 bit comp hash	14147	25446	13145	36.66	65.94	38591
Proposed	15094	33973	3543	79.11	89.89	38591



(a) Accuracy comparison.



(b) Miss rate comparison.

Fig. 25: Test lswlr bench mark for comparing accuracy and miss rate among eight different strategies.

F. Performance Analysis on Test Math Benchmark

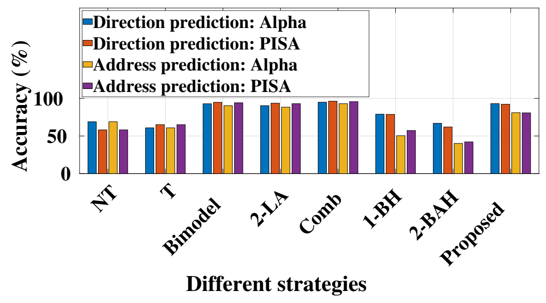
In Fig. 25, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 25a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate. For α architecture, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. If we consider branch direction prediction rate, our proposed strategy is close to the combination strategy. For $pisa$ architecture, bimodal strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy is close to the bimodal strategy if we consider branch direction prediction rate. In Fig. 25b, combination has the lowest miss-rate for α architecture and $PISA$ architecture. Our proposed strategy is close to it and performs well than other strategies.

TABLE XI: The performance of eight different strategies on Test-printf data-set using α architecture. Here, combination strategy has the highest branch address prediction rate and our proposed strategy has the highest branch direction prediction rate.

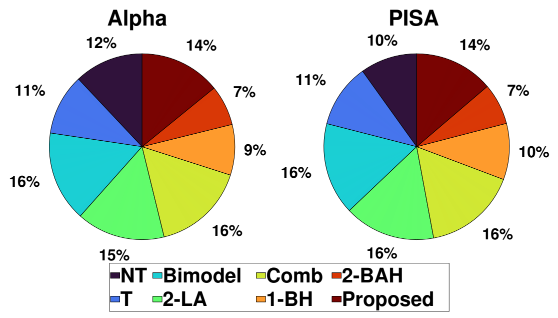
Strategy	**Address predicted hits**	**Direction predicted hits**	**Number of misses**	**Branch address prediction rate**	**Branch direction prediction rate**	**Number of lookups**
Not-taken	109894	109894	49537	68.93	68.93	159431
Taken	96981	96981	62450	60.83	60.83	159431
Bimod	143978	147042	12389	90.31	92.83	159431
2lev	140860	143969	15462	88.35	90.30	159431
Comb	148294	151322	8109	**93.01**	94.91	159431
1 bit hash	80382	125980	33451	50.41	79.02	159431
2 bit comp hash	64019	106548	52883	40.15	66.83	159431
Proposed	65324	147925	51506	80.97	**93.09**	159431

TABLE XII: The performance of eight different strategies on Test-printf data-set using $PISA$ architecture. Here, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy is close to the combination strategy if we consider branch prediction rate.

Strategy	**Address predicted hits**	**Direction predicted hits**	**Number of misses**	**Branch address prediction rate**	**Branch direction prediction rate**	**Number of lookups**
Not-taken	233361	233361	168248	58.11	58.11	401609
Taken	261363	261363	140246	65.08	65.08	401609
Bimod	378117	380405	21204	94.15	94.72	401609
2lev	373619	375930	25679	93.03	93.61	401609
Comb	384406	386630	14979	**95.72**	**96.27**	401609
1 bit hash	230157	316070	85539	57.31	78.70	401609
2 bit comp hash	169260	248874	152735	42.15	61.97	401609
Proposed	163441	349898	151711	80.70	92.22	401609



(a) Accuracy comparison.



(b) Miss rate comparison.

Fig. 26: Test printf bench mark for comparing accuracy and miss rate among eight different strategies.

G. Performance Analysis on Test Printf Benchmark

In Fig. 26, we depict the performance of different branch prediction strategies using anagram benchmark for both α and $PISA$ architecture. In Fig. 26a, we compare the accuracy of branch direction prediction rate and branch direction prediction rate. For α architecture, combination strategy has the highest branch address prediction rate and our proposed strategy has the highest branch direction prediction rate. For $pisa$ architecture, combination strategy has the highest branch address prediction rate and highest branch direction prediction rate. Our proposed strategy is close to the combination strategy if we consider branch prediction rate. In Fig. 26b, combination has the lowest miss-rate for α architecture and $PISA$ architecture.Our proposed strategy is not close to it but performs well than few other strategies.

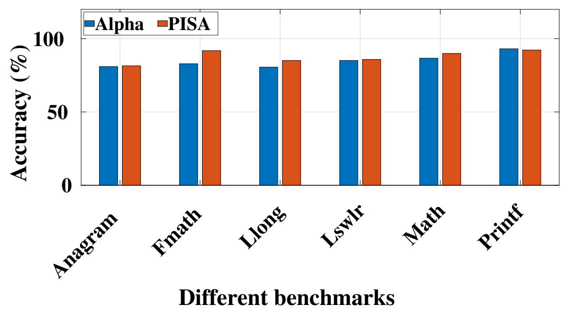


Fig. 27: Our proposed strategy performance on six different benchmarks. Our proposed strategy has accuracy more than 80% for both the architecture.

H. Performance Analysis of our Proposed Strategy

From the Fig. 27, we can see that our proposed strategy works well for all benchmarks of both α and $PISA$ architecture. It is 80% or upper than that. We have analyzed branch address prediction rate, branch direction prediction rate and number of misses for evaluating our performance. Sometimes, our proposed strategy shows the best performance. Sometimes, it shows close performance to the best one.

V. Conclusion

In this project, we have analyzed seven different branch prediction strategies and developed a novel branch prediction strategy. Then, we have tested their performance on two different architecture named α and $PISA$ architecture along with six different benchmarks. We have measured the performance in terms of branch address prediction rate, branch direction prediction rate and number of misses. Our proposed strategy shows better performance than similar one bit hash and two bit complementary hash presented in [13]. Our strategy achieves more than 80% accuracy in both the architecture and all six benchmarks. In future, we will try to implement more accurate strategy such as tournament predictor as it is a multi-level branch predictor that chooses between global and local predictor.

References

[1] Y. Mao, H. Zhou, X. Gui, and J. Shen, “Exploring convolution neural network for branch prediction,” *IEEE Access*, vol. 8, pp. 152 008–152 016, 2020.

[2] D. A. Jiménez and C. Lin, “Dynamic branch prediction with perceptrons,” in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] Z. Su and M. Zhou, “A comparative analysis of branch prediction schemes,” *University of California at Berkeley, Computer Architecture Project*, 1995.

[5] C.-K. Lin and S. J. Tarsa, “Branch prediction is not a solved problem: Measurements, opportunities, and future directions,” *arXiv preprint arXiv:1906.08170*, 2019.

[6] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, “Branchnet: A convolutional neural network to predict hard-to-predict branches,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 118–130.

[7] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, “Power issues related to branch prediction,” in *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE, 2002, pp. 233–244.

[8] P. Chaudhary *et al.*, “Implemented static branch prediction schemes for the parallelism processors,” in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. IEEE, 2019, pp. 79–83.

[9] T. Ball and J. R. Larus, “Branch prediction for free,” *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 300–313, 1993.

[10] G. Luo and H. Guo, “Software-based and hardware-based branch prediction strategies and performance evaluation.”

[11] T. Mitra and A. Roychoudhury, “A framework to model branch prediction for wcet analysis,” in *2nd Workshop on Worst Case Execution Time Analysis (WCET)*, 2002.

[12] J. K. Lee and A. J. Smith, “Branch prediction strategies and branch target buffer design,” *Computer*, vol. 17, no. 01, pp. 6–22, 1984.

[13] J. E. Smith, “A study of branch prediction strategies,” in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215.

[14] S.-T. Pan, K. So, and J. T. Rahmeh, “Improving the accuracy of dynamic branch prediction using branch correlation,” in *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 1992, pp. 76–84.

[15] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *Proceedings of the 24th annual international symposium on Microarchitecture*, 1991, pp. 51–61.

[16] ——, “Alternative implementations of two-level adaptive branch prediction,” *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 124–134, 1992.

[17] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, “Analysis of branch prediction via data compression,” *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 128–137, 1996.

[18] N. Gloy, M. D. Smith, and C. Young, “Performance issues in correlated branch prediction schemes,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE, 1995, pp. 3–14.

[19] S. McFarling, “Combining branch predictors,” Citeseer, Tech. Rep., 1993.

[20] R. Kumar, A. K. Saha, and J. T. Yen, “Towards a more efficient trace cache,” *TC*, vol. 64, p. 1.

[21] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[22] T. Austin, D. Ernst, E. Larson, C. Weaver, R. Desikan, R. Nagarajan, J. Huh, B. Yoder, D. Burger, and S. Keckler, “Simplescalar tutorial (for release 4.0),” in *International Symposium on Microarchitecture (MICRO-34)*, 2001.

[23] G. S. Tyson, “The effects of predicated execution on branch prediction,” in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 196–206.

[24] S. Nain and P. Chaudhary, “Implementation and comparison of bi-modal dynamic branch prediction with static branch prediction schemes,” *International Journal of Information Technology*, vol. 13, no. 3, pp. 1145–1153, 2021.

[25] I. Bate and R. Reutemann, “Efficient integration of bimodal branch prediction and pipeline analysis,” in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’05)*. IEEE, 2005, pp. 39–44.

[26] C. Egan, G. Steven, P. Quick, R. Anguera, F. Steven, and L. Vintan, “Two-level branch prediction using neural networks,” *Journal of Systems Architecture*, vol. 49, no. 12-15, pp. 557–570, 2003.

[27] G. Darcy, R. Brender, S. Morris, and M. Iles, “Using stimulation to develop and port software,” *Digital Technical Journal*, vol. 4, pp. 181–181, 1993.

[28] V. Herdt, D. Große, and R. Drechsler, “Towards specification and testing of risc-v isa compliance¡sup¿¡/sup¿,” in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 995–998.

[29] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” *ACM SIGARCH computer architecture news*, vol. 25, no. 3, pp. 13–25, 1997.

VI. Supplementary Document

A. Installation of SimpleScalar

For installing SimpleScalar on ubuntu, please follow this tutorial step by step video

B. Running of SimpleScalar

After installation, please copy all our codes and paste it in the directory where all the installation files exist. For running the alpha ISA, please run the following commands.

- $ make clean
- $ make config-alpha
- $ make

For running the PISA ISA, please run the following commands.

- $ make clean
- $ make config-pisa
- $ make

For running the branch prediction, please run the following command,

- $ ./sim-bpred -bpred bimod tests/bin/anagram

Here “sim-bpred” represents the simulation type, “bimod” represents the bimodal method and “tests/bin/anagram” represents the directory of the anagram benchmark. In the same way other branch predictors and other benchmarks can be used for branch prediction.

Besides of ”bimod”, there are 5 default options

- nottaken
- taken
- bimod
- 2lev
- comb

For the other 3 strategies we implemented,

- hash01 - One bit hash
- hashsign - Two bit complementary hash
- hashsms - Our proposed method

Besides of ”anagram”, there are 6 benchmarks

- anagram
- test-fmath
- test-llong
- test-lslwr
- test-math
- test-printf

C. Our Video DemostrationS

Here, we attach video demonstration on Alpha and PISA architecture for the reference of the reader.