RISC-V Simulator: Dynamic Branch Prediction and Data Cache

Md Rayhanul Islam

December 30, 2024

Abstract

This project implements three dynamic branch predictors (static, 1-level, and 2-level) and three data caching approaches (Direct Mapped cache, 2-way set associative, and 4-way set associative) on the RISC-V simulator. It also explores the performance of each of the branch predictors considering branch misprediction rate and evaluates the efficacy of different caching approaches using cache hit rate.

1 Introduction

The performance of computer systems largely depends on several factors related to processors, including cycles per instruction (CPI), clock rate, bandwidth, and memory. However, even processors with high clock rates and CPI can perform poorly if they remain idle during certain cycles. If processors become idle, then having many powerful processors with multiple cores is ineffective. To maximize efficiency, a steady stream of instructions must be delivered to the processor to maintain uninterrupted computation. Unfortunately, incorrect branch predictions can lead to the fetching of wrong instructions. In contrast, a lack of readily available instructions or data (often waiting to be loaded from memory to registers) can further degrade performance.

To address these challenges, various techniques have been developed to ensure processors fetch correct instructions and have immediate access to the required data. Branch target address prediction and preloading mechanisms are employed to enhance performance by reducing delays. Existing literature highlights dynamic branch prediction methods that enable processors to accurately anticipate subsequent instructions, which reduces pipeline stalls caused by conditional branches. Similarly, caching strategies have been introduced to store data close to the processor so that the processor has faster access and reduces the latency by loading data directly from memory.

Dynamic branch prediction introduces enhanced control flow handling by incorporating data structures such as the Branch Target Buffer (BTB), Branch History Table (BHT), and Branch History Shift Register (BHSR). These mechanisms aim to reduce penalties from branch mispredictions by predicting the outcome of branch instructions during the fetch stage. Among many ways to predict branches, this project concentrates on three approaches, which are static approach, 1-level approach, and 2-level approach. A brief description of these approaches is provided in section 3.1.

Caching approaches vary in how they map memory blocks to cache lines. There exist many ways to cache data. in this process, we considered Direct-mapped (DM) cache, 2-Way Set Associative, and 4-Way Set Associative cache. Each approach has some trade-off between simplicity, performance, and hardware complexity. A detailed description of these caching approaches along index and offset bits is provided in section 4.1, and the result of different caching on different test cases is provided in section 3.2.

2 Implementation

The whole project is implemented on top of the RISC-V simulator mainly used for programming assignments. In order to implement the functionality of the dynamic branch predictor, the following functions are implemented, provided in the following Listing. The full implementation of these functions is available in *sim_stages.c* file.

```
unsigned int BTB_lookup(unsigned int inst_addr)

unsigned int BTB_target(unsigned int inst_addr)

void BTB_update(unsigned int inst_addr, unsigned int branch_target)

unsigned int predict_direction(unsigned int inst_addr)

void direction_update(unsigned int direction, unsigned int inst_addr)
```

The functionalities of three different caching approaches are provided using the following functions shown in the below Listing.

```
unsigned int dcache_lookup_4_way(unsigned int addr_mem)

void dcache_update_4_way(unsigned int addr_mem, int line)

unsigned int dcache_lookup_2_way(unsigned int addr_mem)

void dcache_update_2_way(unsigned int addr_mem, int line)

unsigned int dcache_lookup_DM(unsigned int addr_mem)

void dcache_update_DM(unsigned int addr_mem)
```

3 Branch Predictors

The performance of branch prediction strategies, such as static, 1-level dynamic, and 2-level, depends on the program's characteristics. In the following, each of the branch predictors is explained briefly in section 3.1, along with the scenario where they excel, and the experimental result of branch predictors are analyzed in section 3.2.

3.1 Description of Branch Predictors

This section briefly explains each of the branch predictors with the scenario where they perform well.

Static branch predictor. The static branch predictor assumes that every conditional branch will not be taken, and it further considers that the program will continue executing the next sequential instruction. This is the simplest form of branch prediction, which does not track past behavior or use complex prediction algorithms. Overall, the *always not taken* branch predictor is an effective strategy for programs with mostly *sequential code or rarely taken branches*.

1-level dynamic predictor. A 1-level branch predictor is a simple dynamic branch prediction mechanism that uses a BHT to predict the direction of branches. The BHT is indexed using the lower bits of the program counter (PC), and each entry in the table typically contains a 2-bit. These counters keep track of whether a branch is likely to be *taken or not taken* based on its recent behavior. A 1-level branch predictor works well when branches have consistent behavior that does not depend on global history or complex patterns.

Consider the following loop, where the branch at the end of the loop ($i < 100$) will be taken 99 times and not taken once. The BHT entry for this branch will quickly learn that it is usually "taken" and predicts "taken" for most iterations. The predictor will mispredict only on the final iteration when the branch is "not taken."

```
for (int i = 0; i < 100; i++) {
```

```
    // Loop body
}
```

Overall, a 1-level branch predictor excels in scenarios where:

- *Loops with Fixed Iterations*. Loops like for or while with predictable exit conditions (iterating a fixed number of times) are ideal.
- *Static Branches*. Branches that are almost always taken or not taken, such as error-checking conditions or initialization checks.

2-level branch predictor. A 2-level branch predictor is an advanced dynamic branch prediction mechanism that uses both local and global branch history to make more accurate predictions using BTB, BHT, and BHSR. 2-level branch predictors excel in situations where branch outcomes are correlated with the outcomes of other recent branches. They are particularly effective for:

- *Nested loops*. Where the behavior of an inner loop affects the outer loop's branching pattern.
- *Correlated branches*. When the outcome of one branch influences another.
- *Complex control flow*. In programs with intricate decision-making structures.

Consider the following example. Here, the outer loop runs 100 times. The inner loop's iteration count depends on the outer loop's counter. and the if statement's behavior might depend on both loop counters. In such a scenario, the 2-level predictor outperforms others (static and 1-level predictor).

```
    for (int i = 0; i < 100; i++) {
    for (int j = 0; j < i; j++) {
        if (data[j] > threshold) {
            process(data[j]);
        }
    }
}
```

3.2 Result Analysis of Branch Prediction

The experimental results of the three branch predictors implemented in this project are summarized in Table 1. To obtain these results, we conducted experiments with forwarding and out-of-order execution enabled for all test cases. The branch prediction type or data caching type was modified as required for each specific scenario. In the following, we will provide an analysis of why certain branch predictors performed better in specific test cases.

Test case	# branches	Static	1-level	2-level
Binary_search.out	5	40	40	40
Binary_search_itr.out	60	48.33	38.33	26.67
Colliding_cache.out	0	0	0	0
Colliding_cache_itr.out	10	90	30	80
Correlation.out	44	54.55	54.55	20.45
Correlation_itr.out	450	55.33	51.33	6.44
Derivative.out	31	96.77	9.68	25.81
Derivative_itr.out	320	96.56	4.69	9.44
Derivative_unrolled.out	4	75	75	75
Endian_reverse.out	4	75	75	75
Endian_reverse_itr.out	50	78	30	26
Fibonacci.out	10	90	30	80
Fibonacci_itr.out	330	90.61	10.61	14.24
Matrix_mul.out	22	31.72	31.72	31.72
Matrix_mul_itr.out	310	51.29	49.35	16.77
Nested_loops.out	32	71.88	43.75	34.38
Nested_loops_itr.out	330	72.42	29.70	8.18
Twos_complement.out	1	0	0	0
Twos_complement_itr.out	11	45.00	15.00	25.00
Wordsearch.out	12	8.33	8.33	8.33
Wordsearch_itr.out	130	14.62	10.00	10.00

Table 1: Branch misprediction rates of all branch predictors across various test cases. The first column lists the test case names, while the second column indicates the number of branches in each program. The third column, labeled "Static," shows the misprediction rate (%) when the static branch predictor is used. The fourth column, "1-level," represents the misprediction rate (%) with 1-level branch prediction enabled. The final column, "2-level," displays the misprediction rate (%) with 2-level branch prediction enabled.

In test cases related to binary search, all prediction types work equally because the branch not taken scenario dominates in the binary search's assembly code. The assembly code shows that it randomly changes its branch target based on whether mid is equal to the target or it is less than the target to select the left or right part of the search array. That is why 1-level and 2-level predictors could not capture the branch behavior. However, the 2-level branch predictor works best in binary_search_itr because it has more complex branch types, and BHSR can accurately represent the behavior of the branch.

According to the assembly code, Colliding_cache does not have any branch instructions, and Colliding_cache_itr does have branch instructions with a simple loop, and all local branch target addresses are distinct, which is why the 1-level branch type works well here. The code analysis shows that the correlation code structure is complex, with calling a function multiple times using $jalr$, and 1-level branch predictors could not capture the branch behaviors. That is why the 2-level branch predictor works best in correlation test cases, as it can capture the global history through BSHR.

In the test case (Derivative.out), 1-level prediction works well because the assembly program contains a simple loop, which iterates over 0 to 31, and the branch history table can store proper branch behavior. Similarly, 1-level also works best in Derivative_itr.out because of the same reason as Derivative.out. 2-level branch prediction also works well here compared to static branch predictor because the assembly code has loops, and BHSR can store branch behavior. The static branch predictor performs worst as it always predicts a branch not taken even though the assembly code has a loop. On the other hand, all three branch prediction approaches work poorly in Derivative_unrolled.out because the loop only repeats 4 times, which is not sufficient to learn the pattern for 1-level and 2-level branch predictors.

The assembly code of endian_reverse contains procedure calls, and the loop is repeated only 4 times, which is the reason why the 1-level and 2-level branch predictor performs poorly. As the branch is taken for 4 times, the static predictor predicts that the branch is always not taken. That is why it also performs poorly. In the endian reverse iteration, the 2-level branch predictor performs well compared

to the 1-level branch predictor and static branch predictor because it has procedure calls like *jal* and *jalr*, and a branch history shift register can able to capture the branch behavior.

In the two test cases of Fibonacci, the 1-level branch predictor works best because the assembly code contains a simple loop with simple branch conditions; therefore, BHT can capture the behavior perfectly. The test case, *Matrix_mul*, contains an inner, outer, and dot_product loop, which prevents static and 1-level branch predictors from correctly capturing branch behavior. In *Matrix_mul_itr*, it contains inner, outer, and dot_product loop, which helps BHSR to capture the proper branch behavior. That is the reason the 2-level branch predictor works better in *Matrix_mul_itr* and archives the lowest branch misprediction rate.

The 2-level branch predictor has the lowest branch misprediction rate in two test cases of nested loops because BHSR can capture the branch behavior of nested loops, and the 1-level and static predictor could not. The test case titled 'twos_complement' has achieved the best performance from all three branch predictions with a branch misprediction rate of 0. In the *two_complement_itr*, 1-level and 2-level branch predictor outperforms static predictors because the assembly code contains a loop.

In the test case *wordsearch*, the branch not taken prediction works most of the time, that is the reason the static predictor's misprediction rate is equal to the misprediction rate of 1-level and 2-level predictors. In the *wordsearch_itr*, the misprediction rate among all three branch predictors are very close; to be more specific, 1-level and 2-level have a misprediction rate of 10, and static has around 14, which implies most of the branch instructions are not taken, and 1-level and 2-level predictor can accurately predict the branch behaviors.

4 Data Caching

The caching in this project is implemented in three ways: Direct mapped cache, 2-way set associative cache, and 4-way set associative cache.

4.1 Description of Caching

A short description of these caching methods is provided below.

Direct-Mapped (DM) cache. In a direct-mapped cache, each memory block maps to exactly one specific cache line based on its address. This approach is simple and fast, as it uses a straightforward indexing mechanism to locate data. However, it is prone to conflict misses, which occur when multiple memory blocks map to the same cache line, causing frequent evictions and reloads even if there is available space in the cache. In this project, the first 4 bits are used for offset, then the next 4 is used for the index, and the remaining bit is used for the tag.

2-Way Set Associative Cache. A 2-way set associative cache divides the cache into sets, with each set containing two cache lines. Each memory block can be stored in either of the two lines within a specific set, which is determined by the block's address. This increased flexibility reduces conflict misses compared to direct-mapped caches, as there are more options for storing blocks in a set. In this project, first 4 bits are used for offset, then next 3 bits are used for index, and the remaining bits are used for tag.

4-Way Set Associative Cache. In a 4-way set associative cache, each set contains four cache lines, providing even greater flexibility for storing memory blocks within a set. This further minimizes conflict misses, making it well-suited for workloads with high contention or irregular memory access patterns. In this project, first 4 bits are used for offset, then next 2 bits are used for index, and the remaining bits are used for tag.

4.2 Result Analysis of Caching

Table 2 presents the cache hit rate of different caching mechanism on different test cases.

Test case	# memory access	DM cache	2-way cache	4-way cache
Binary_search.out	4	75.00	75.00	75.00
Binary_search_itr.out	40	97.50	97.50	97.50
Colliding_cache.out	28	78.57	78.57	78.57
Colliding_cache_itr.out	280	97.86	97.86	97.86
Correlation.out	0	0	0	0
Correlation_itr.out	0	0	0	0
Derivative.out	63	92.06	92.06	92.06
Derivative_itr.out	630	99.21	99.21	99.21
Derivative_unrolled.out	66	92.42	92.42	92.42
Endian_reverse.out	2	50.00	50.00	50.00
Endian_reverse_itr.out	20	95.00	95.00	95.00
Fibonacci.out	0	0	0	0
Fibonacci_itr.out	0	0	0	0
Matrix_mul.out	20	95.00	95.00	95.00
Matrix_mul_itr.out	200	99.50	99.50	99.50
Nested_loops.out	24	91.67	91.67	91.67
Nested_loops_itr.out	240	99.17	99.17	99.17
Twos_complement.out	0	0	0	0
Twos_complement_itr.out	0	0	0	0
Wordsearch.out	6	83.33	83.33	83.33
Wordsearch_itr.out	60	98.33	98.33	98.33

Table 2: The memory hit access of three different caching approaches. The first column, 'Test case', represents the test case name, and The second column represents the number of memory accesses. The third column, 'DM cache,' represents the hit ratio of the Direct caching approach. The fourth column, 2-way cache, means the caching approach of the 2-way caching approach. The fifth column represents the hit ratio of the 4-way caching approach.

The experimental result shows the number of memory accesses for some test cases, such as *twos_complement.out*, *twos_complement_itr.out*, *correlation.out*, *correlation_itr.out*, *Fibonacci.out*, and *Fibonacci_itr.out*, is 0. A close inspection of the assembly code of these test cases shows that these test cases' assembly codes do not have any load instructions. Therefore, the program does not access any memory access when running on these test cases, which results cache hit rate of 0.

In the remaining test cases, such as *Binary_search.out*, *Binary_search_itr.out*, *Derivative_unrolled.out*, *Endian_reverse.out* , *Endian_reverse_itr.out*, *Matrix_mul.out*, *Matrix_mul_itr.out*, *nested_loops.out*, *nested_loops_itr.out*, *wordsearch.out*, *wordsearch_itr.out*, *Colliding_cache.out*, *Colliding_cache_itr.out*, *Derivative_itr.out*, and *Colliding_cache_itr.out*, although the cache hit rate varies from test case to test case, but for an individual test case, the cache hit rate is the same among all caching approaches.

In the test cases *Binary_search.out*, and *Binary_search_itr.out*, the memory access pattern inherently avoids conflicts because data are stored in a contiguous array, which helps to achieve the same level of cache hit rate among all caching approaches.

Consider the test cases *Matrix_mul* and *Matrix_mul_itr*, which involve inner, outer, and dot product loops in their assembly code. In theory, a 4-way set associative cache should outperform the other caching strategies because both DM cache and 2-way set associative cache are more prone to conflict misses. However, in this example, all caching methods achieve the same level of cache hit rate—99.5% for *Matrix_mul_itr* and 95% for *Matrix_mul*. This is due to the memory access patterns in these test cases, which naturally help to minimize cache conflicts.

5 Conclusion

In summary, 1-level and 2-level branch predictors generally outperform static branch predictors in most cases. Regarding cache performance, all caching approaches demonstrate equivalent cache hit rates in this experiment.