

Lab 6: Cache

Learning Outcomes:

After completing this lab, you will be able to:

- Clearly define what caching is and its role in computer systems, emphasising its significance in improving data access performance.
 - Demonstrate the ability to analyze cache performance metrics, including hit rates, miss rates, Average memory access time and their impact on overall system efficiency.
 - Explain the need for and the methods of caching in RISC-V & recognize the patterns in machine code that benefit from caching
 - Identify and describe the different mechanisms of cache associativity
-

Introduction

Caching

Caching in CPUs is a crucial aspect of computer architecture that significantly enhances the speed and efficiency of data access and program execution. It involves the use of small, high-speed memory units called caches, which are situated closer to the CPU core than the main memory (RAM). The primary purpose of CPU caching is to reduce the latency of accessing frequently used data and instructions, thereby improving overall system performance.

Example

- This lab will use the following simple C code to analyze the use of the cache in the Ripes tool. To reduce the number of assembly instructions used in our programs, we will avoid using the standard C library and use a set of assembly instructions instead to load, run and exit the 'main' function.
- First, disable importing the standard library automatically by going to Edit->Settings->Compiler and adding the '-nostdlib' flag in linker arguments.
- Then use the following code to create a new program

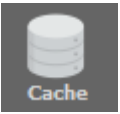

```
asm("li sp, 0x100000"); // SP set to 1 MB
asm("jal main"); // call main
asm("mv a1, a0"); // save return value in a1
asm("li a7, 10"); // prepare ecall exit
asm("ecall"); // now your simulator should stop
```

```
int main() {
    int n = 0;
    int k = 4;
```

```

    for(int i=0; i<10; i++){
        n += k;
    }
    return 0;
}

```

- As mentioned earlier, the assembly instructions at the top will handle the overall execution of the program.
- Analyze the C code. You can see that it is a simple code where a constant k is added to the variable n, 10 times.
- Before compiling the program, go to the Cache tab of the tool by clicking the  icon. In the **L1 Data cache** tab, set the Preset as direct-mapped and Replacement policy as **LRU(Least Recently Used)**
- Then, go to the editor and compile the code. Analyze the assembly code generated carefully and identify the objective of each instruction.
- Then, using the  button, execute instruction by instruction. Switch between the Editor and Cache tabs and see how and when data is loaded to the Cache. Refer to the ‘Observe Caching in RISC-V’ section below to further understand the UI of the Cache tab
- Notice how the variables n, k and i are loaded and stored in the cache. Whenever these variables are called to the registers, they are loaded from the Cache and not the memory, saving a considerable amount of time (Cache hit). Observe how a Cache hit is indicated in Green in the diagram of the Cache tab.
- In the Cache tab, under statistics, you can see the relevant information regarding Cache accesses. Observe how these change as the code progresses.
- Since this program is small and all the required data can be held in the Cache, cache misses do not exceed 2.

Observe Caching in RISC-V

In this section, we will observe caching in the context of the RISC-V.

In RISC-V, we have a helpful dashboard which visualize and helps you to understand how caching works inside the CPU. Here are some important things to keep an eye on that will help you get a better understanding:

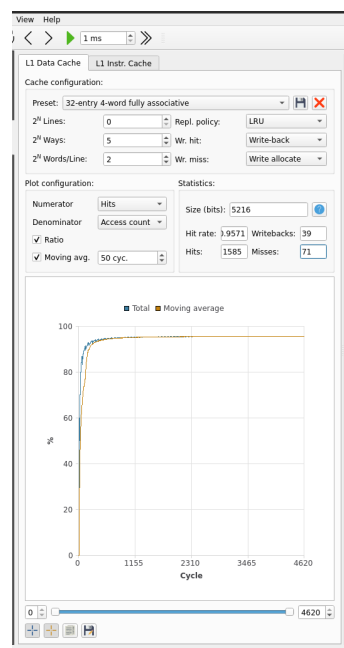
- You can visualise the cache using the Cache tab as mentioned above.

- On the right side of the screen, it shows you the current cache details.

Access address: 0000000000011111111111111000

Index	V	D	LRU	Tag	Word 0	Word 1	Word 2	Word 3
1118	1	1	8	0x0000fffa	0x000097e	0x0000988	0x0000992	0x000099c
1110	1	1	0	0x0000fffe	0x0000a1e	0x000008c	0x000000f	0x0000104
1117	1	1	7	0x0000fffb	0x00009a6	0x00009b0	0x00009ba	0x00009c4
1116	1	1	6	0x0000fffc	0x00009ce	0x00009d8	0x00009e2	0x00009ec
1115	1	1	5	0x0000fffd	0x00009f6	0x0000a00	0x0000a0a	0x0000000
1104	1	1	4	0x0000ffbd	0x0000000	0x0000000	0x000000a	0x0000014
1103	1	1	3	0x0000ffbe	0x000001e	0x0000028	0x0000032	0x000003c
1102	1	1	2	0x0000ffbf	0x0000046	0x0000050	0x000005a	0x0000064
1101	1	1	1	0x0000ffc0	0x000006e	0x0000078	0x0000082	0x000008c
1131	1	1	31	0x0000ffe3	0x00005e6	0x00005f0	0x00005fa	0x0000604
1130	1	1	30	0x0000ffe4	0x000060e	0x0000618	0x0000622	0x0000000
1129	1	1	29	0x0000ffe5	0x0000636	0x0000640	0x000064a	0x0000000
1128	1	1	28	0x0000ffe6	0x000065e	0x0000668	0x0000672	0x0000000
1127	1	1	27	0x0000ffe7	0x0000686	0x0000690	0x000069a	0x0000000
1126	1	1	26	0x0000ffe8	0x00006ae	0x00006b8	0x00006c2	0x0000000
1125	1	1	25	0x0000ffe9	0x00006d6	0x00006e0	0x00006ea	0x0000000
1124	1	1	24	0x0000ffea	0x00006fe	0x0000708	0x0000712	0x0000000
1123	1	1	23	0x0000ffeb	0x0000726	0x0000730	0x000073a	0x0000000
1122	1	1	22	0x0000ffec	0x000074e	0x0000758	0x0000762	0x0000000
1121	1	1	21	0x0000ffed	0x0000776	0x0000780	0x000078a	0x0000000
1120	1	1	20	0x0000ffee	0x000079e	0x00007a8	0x00007b2	0x0000000
1119	1	1	19	0x0000ffef	0x00007c6	0x00007d0	0x00007da	0x0000000
1118	1	1	18	0x0000fff0	0x00007ee	0x00007f8	0x0000802	0x000080c
1117	1	1	17	0x0000fff1	0x0000816	0x0000820	0x000082a	0x0000834
1116	1	1	16	0x0000fff2	0x0000836	0x0000848	0x0000852	0x0000000
1115	1	1	15	0x0000fff3	0x0000866	0x0000870	0x000087a	0x0000884
1114	1	1	14	0x0000fff4	0x0000886	0x0000898	0x00008a2	0x0000000
1113	1	1	13	0x0000fff5	0x00008b6	0x00008c0	0x00008ca	0x00008d4
1112	1	1	12	0x0000fff6	0x00008de	0x00008e8	0x00008f2	0x00008fc
1111	1	1	11	0x0000fff7	0x0000906	0x0000910	0x000091a	0x0000924
1110	1	1	10	0x0000fff8	0x000092e	0x0000938	0x0000942	0x000094c
1109	1	1	9	0x0000fff9	0x0000956	0x0000960	0x000096a	0x0000000

- On the left side you can see numeric values and a graph related to caching. You can change various parameters to get various observations.
- Observe them one by one for better understanding.



- You can watch different colours on the caching graph, and each colour means something specific. Here are a few colour codes to understand what's happening:

Cache Hit

A cache hit is indicated by a green-coloured cell.

Exercise 1 - Matrix

Copy the below program into the Ripes code editor, having the language set as C.

```
asm("li sp, 0x100000"); // SP set to 1 MB
asm("jal main"); // call main
asm("mv a1, a0"); // save return value in a1
asm("li a7, 10"); // prepare ecalls exit
asm("ecall"); // now your simulator should stop

#define W 2 // Matrix order

void mmul(const int a[W][W], const int b[W][W], int c[W][W]) {
    for (int row = 0; row < W; row++) {
        for (int col = 0; col < W; col++) {
            for (int k = 0; k < W; k++) {
                c[row][col] += a[row][k] * b[k][col];
            }
        }
    }
}

int main() {
    int A[W][W];
    A[0][0] = 3;
    A[0][1] = 4;
    A[1][0] = 5;
    A[1][1] = 6;

    int B[W][W];
    B[0][0] = 9;
    B[0][1] = 10;
    B[1][0] = 11;
    B[1][1] = 12;

    int C[W][W];


    mmul(A, B, C);


    return 0;
}
```

Compile and execute the program.

This code is a simple code which multiplies two 2x2 matrices.

Observe that the cache is getting filled gradually, as values get assigned to the 2D arrays.

Next pause the program at the loop execution by clicking on the  button and execute the

program step-by-step using  button. Observe how the cache hits and misses change. At the first execution of the loop notice the cache miss count is around 7. When the loop

progresses you can observe how matrix values are accessed from the cache for the multiplication operation.

Can you explain the variation of cache miss count as the loop progresses?

Next, expand the matrices to size 5x5 and observe the cache. Note the row length of the matrices is larger than the block size of the cache, which causes the last value of the previous row to be replaced upon matrix initialization. Therefore, a cache miss occurs when the last value is called.

Exercise 2 - 1D array

In some computer programs, loading data from cache to registers does not always work. There will be cache hits as well as cache misses. When there is a cache miss, data needs to be brought from the main memory into the cache memory first, and then to the registers. Such a cache miss scenario can be observed in this exercise.

Copy the below program into the Ripes code editor, having the language set as C.

```
asm("li sp, 0x100000"); // SP set to 1 MB
asm("jal main");        // call main
asm("mv a1, a0");        // save return value in a1
asm("li a7, 10");        // prepare ecall exit
asm("ecall");            // now your simulator should stop

int main() {
    int array[260];

    // Fill the array with values
    for (int i = 0; i < 260; i++) {
        array[i] = i * 10;
    }


    int num_read;


    // Access the array in a contiguous manner to induce cache misses
    for (int i = 0; i < 100; i += 1) {
        num_read = array[i];
    }

    return 0;
}
```

Compile and execute the program.

Observe that the cache is getting filled gradually, as the for loop keeps adding elements into the array. You may notice that the existing values in the cache are getting overwritten when the cache size is exceeded.

Pause the execution at the beginning of the second for loop by clicking on  button at the toolbar, and observe that the cache misses are around 81, for 260 write operations of elements into the memory.

Now execute the program step-by-step using the  buttons at the toolbar, and observe how the cache hits and misses change. Try to relate the values with the RISC-V instruction that is being executed and observe that the cache misses are now increasing for most iterations in the loop. Try to justify reasons for why such cache misses take place.

Change the **cache configuration preset** to different options such as **direct-mapping, fully associative, and set-associative** options and observe the order of getting the cache memory filled. Also, change the **replacement policy** to **random** and observe the behaviour of replacing the values in the cache when the cache is completely filled.

For each configuration, observe that a 4-word block is brought into the cache memory for each cache miss. This **prefetching** technique helps to avoid future cache misses. Explain further the advantages of prefetching using the spatial and temporal locality properties of the cache memory.