

# CSC3050 Assignment3 Report

Songlin Zhao, 120090346

May 10th, 2024

## 1 Usage

### 1.1 Run Single-Level Cache Simulation

```
cd /the_working_space
chmod +x build1.sh
./build1.sh
# Now the single-level cache simulator will be runned,
# and the .csv file ./src/analysis_p1.csv will be generated
```

### 1.2 Run Multi-Level Cache Simulation

```
chmod +x build2.sh
./build2.sh
# Now the multi-level cache simulator will be runned,
# and the .csv file ./src/analysis_p2.csv will be generated
```

### 1.3 Run Integration with CPU Simulator

```
chmod +x build3.sh
./build3.sh
./build/Simulator riscv-elf/quicksort.riscv # run without cache
./build/Simulator -c riscv-elf/quicksort.riscv # run with cache
# the statistics will be printed in the terminal
```

## 2 Single-Level Cache Simulation

In the first part of this project, I implemented a single-level cache. The cache takes parameters including underlying memory, hit latency, cache size, block size, associativity, write back, write allocate, .etc.

## 2.1 Implementation Details

A cache consists of *cacheSize/blockSize* blocks, and each block contains the following parameters. The valid bit, dirty bit, tag, set number are defined by the cache. The lastAccess parameter records the last access cycle of the current block, and data is an array of uint\_8, which is essentially an array of bytes.

---

```
struct Block {
    bool valid;    // valid bit
    bool dirty;    // dirty bit
    uint32_t tag;  // tag
    uint32_t setNum;
    uint32_t lastAccess;
    std::vector<uint8_t> data; // data in each block, an array of uint_8
};
```

---

Each Cache contains *cacheSize/blockSize* blocks, which is represented by an array of blocks (`std::vector<Block> blocks`). Two essential functions of the Cache class is `get_byte()` and `set_byte()`.

### 2.1.1 get\_byte

The `uint8_t get_byte(uint32_t addr)` function takes the address of the memory as the parameter, and return a 8-bit unsigned integer. The overall logic of this function is as follows:

1. Given address, get the tag number, set number, and the offset.
2. Check whether the block of this address is already in cache, i.e., the valid bit is 1, and the set number matches. If so, just return the byte in cache.
3. If not in cache, this function tries to find the addr in memory, and bring the whole block from memory to cache. If there is no enough space for the new block, the cache will evict a block from the cache using LRU algorithm. Finally, return the byte which is newly put in cache.

### 2.1.2 set\_byte

The `void set_byte(uint32_t addr)` function takes the address of the memory as the parameter. The overall logic of this function is as follows (assuming write back and write allocate policy is used):

1. Given address, get the tag number, set number, and the offset.
2. Check whether the block of this address is already in cache, i.e., the valid bit is 1, and the set number matches. If so, write the corresponding byte in cache, and set dirty bit to 1.

3. If not in cache, this function tries to find the addr in memory, and bring the whole block from memory to cache. If there is no enough space for the new block, the cache will evict a block from the cache using LRU algorithm. Finally, write the corresponding byte in cache, and set dirty bit to 1.

### 2.1.3 Write Back and Write Allocate

There are four cases for write back and write allocate policies:

1. Write-Back = True, Write-Allocate = True: When a write miss occurs, the cache block containing the target address is loaded into the cache, and then the write operation is performed on the cached block. The modified block is marked as dirty but is not immediately written back to main memory. The dirty data is only written back to the main memory when the cache block is evicted.
2. Write-Back = True, Write-Allocate = False: When a write operation occurs, the data is written directly to main memory. However, if the data is already in the cache, it is updated in the cache and marked as dirty. It will only be written back to the main memory upon eviction.
3. Write-Back = False, Write-Allocate = True: When a write miss occurs, the cache block is loaded into the cache, and the write operation is executed on the cache. However, all writes to the cache will also be written to main memory.
4. Write-Back = False, Write-Allocate = False: When a write operation occurs, the data is written directly to main memory. No write back is used.

## 2.2 Performance analysis

The following image shows how different block size affects cache performance. We can observe that when holding all other conditions, the larger the block size, the better the cache performance. This could be explained by the fact that larger block size can bring a larger block of data during a cache miss, which increases the possibility of the next hit.

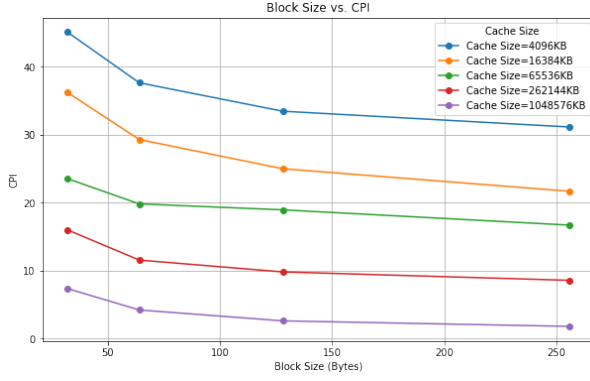


Figure 1: Block Size vs. CPI

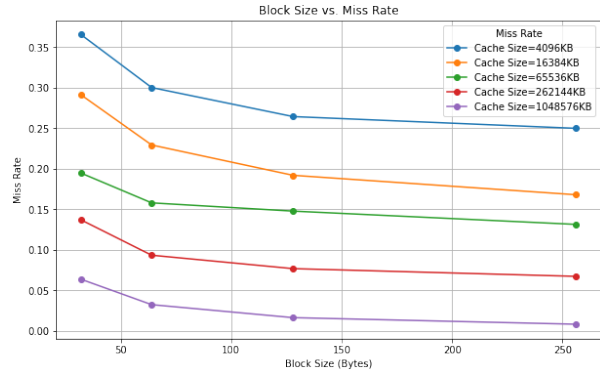


Figure 2: Block Size vs. Miss Rate

Figure 3: Effects of block size on performance

The following image shows how different cache size affects cache performance. We can observe that when holding all other conditions, when we increase the cache size, the performance improves significantly. However, under the same cache size, the performance of different block sizes does vary that much. The larger cache size means more faster memories, which will definitely increase the performance.

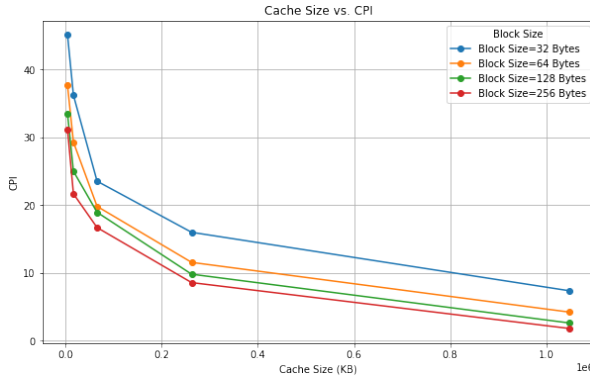


Figure 4: Cache Size vs. CPI

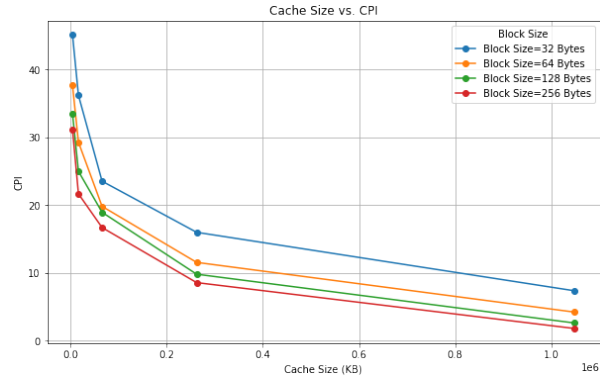


Figure 5: Cache Size vs. Miss Rate

Figure 6: Effects of cache size on performance

The following image demonstrates the influence of associativity on cache performance. When the cache size is 4KB, increase the associativity can cause better performance at first, but worse performance latter. However, when the cache size is 16KB, the relationship is positive, while when the cache size is 64KB, the relationship is negative. This shows that increasing associativity does not necessarily mean increasing or decreasing the performance.

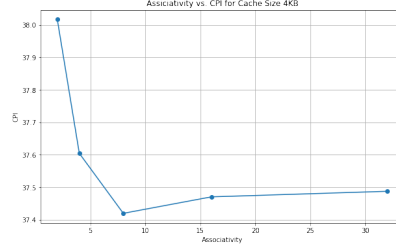


Figure 7: Cache Size 4KB

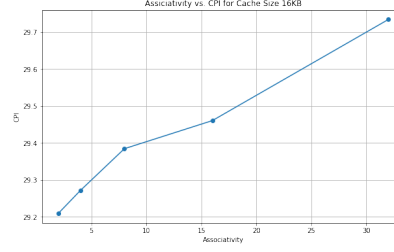


Figure 8: Cache Size 16KB

Figure 9: Effects of cache size on performance

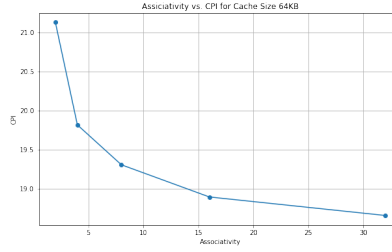


Figure 10: Cache Size 64KB

Figure 11: Effects of cache size on performance

## 3 Multi-Level Cache Simulation

### 3.1 Implementation Details

#### 3.1.1 Inclusive Cache

The `get_byte()` and `set_byte()` function is modified to enable multi-level inclusive cache. In inclusive cache, if a block of data is present at a certain level, all lower levels will contains that block of data.

1. When there is a cache miss, this function recursively get the byte from lower caches and place them in the current level until the data is present, which maintains inclusivity.
2. When there is a eviction occurs, we need to write the data to lower level. We only write to the next lower level, not all lower levels.
3. I implemented the back invalidation when the data is evicted from a lower level, all data from higher levels will also be evicted. However, by using LRU and recording the lastAccess variable of all blocks, there will be no case when the data is evicted from a lower level where higher level caches contains that block of data.
4. The dirty bit in multi-level cache means the block in that level is different from lower level cache, so during writing to an address, the dirty bit is only set to the level 1 cache. All lower caches is not dirty.

### 3.1.2 exclusive Cache

For exclusive cache, a block of data only exists in a certain level of cache. To pertain this property, the following eviction and replacement policy is used.

1. When there is a cache miss, the function finds the exact level of cache that contains this block of data, and the data is directly returned to the L1 cache.
2. When there is a eviction occurs, we need to write the data to lower level. We only write to the next lower level, not all lower levels.
3. When a block is brought up from a lower level cache or evicted to a lower level cache, the dirty bit is preserved.

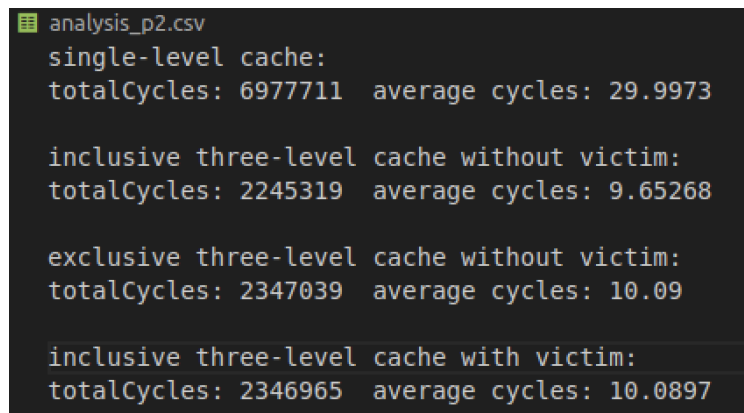
### 3.1.3 Victim Cache

The victim cache is only connected to the L1 cache, and stores the recently evicted cache from the L1 cache. The victim cache is only 8 blocks and it is fully associative.

1. When a data access results in a miss in the L1 cache, the program checks the victim cache to see if the requested data is available there. If the data is in victim cache, just return it. If the data is not found in the victim cache either, the data is then fetched from the L2 cache.
2. The victim cache uses LRU policy to handle which block to evict.

## 3.2 Performance analysis

The following figure shows the performance of single-level cache, inclusive three-level cache without victim cache, exclusive three-level cache without victim cache, and inclusive three-level cache with victim cache.



```
analysis_p2.csv
single-level cache:
totalCycles: 6977711  average cycles: 29.9973

inclusive three-level cache without victim:
totalCycles: 2245319  average cycles: 9.65268

exclusive three-level cache without victim:
totalCycles: 2347039  average cycles: 10.09

inclusive three-level cache with victim:
totalCycles: 2346965  average cycles: 10.0897
```

Figure 12: The Cache performance of different settings

Level	Capacity	Associativity	Block Size	Write Policy	Hit Latency
L1	16 KB	1 way	64 Bytes	Write Back	1 CPU Cycle

Table 3: Default cache setting used in single-level cache simulation.

Level	Capacity	Associativity	Block Size	Write Policy	Hit Latency
L1	16 KB	1 way	64 Bytes	Write Back	1 CPU Cycle
L2	128 KB	8 ways	64 Bytes	Write Back	8 CPU Cycle
L3	2 MB	16 ways	64 Bytes	Write Back	20 CPU Cycle

Table 4: Default cache setting used in multi-level cache simulation.

Figure 13: Default cache setting used in multi-level cache simulation

### 3.2.1 Single and Multi

The performance of multi-level cache is 3 times faster than single-level cache. This is because the addition of L2 and L3 cache shortens the distance between cpu and memory.

### 3.2.2 Inclusive and Exclusive

The performance of inclusive and exclusive cache is similar, while exclusive cache might be a little bit slower. Although exclusive cache saves more space for lower-level data, it can be a little bit slower. This can be explained by the fact that the L1 cache is small. **For inclusive cache, the block is only evicted from L1 cache when the L1 cache is full, and the dirty bit is 1. However, for exclusive cache, the block is evicted only if the L1 cache is full, no matter whether it is dirty.** As a result, it causes more cycles during testing because exclusive cache takes more time to write to L2 caches. The other latencies for inclusive and exclusive caches are the same. They both checks all absent caches during cache miss. **During cycles calculation, if a cache miss occurs, inclusive cache might both look up and write to the cache, exclusive cache might only look up the cache without writing to it, however, I only calculated once for the latency to mimic the real-world behaviour.**

### 3.2.3 With and Without victim

The addition of victim cache increases the total cycles a little bit, because the size of the victim cache is only 8 blocks, and accessing it causes extra time. **The access latency of the victim cache is set to 2.** However, during each cache miss, 2 extra cycles will be added, but the chances to find a block in victim cache is really low, so the addition of victim cache would increase the total cycles. **If I set the hit latency of victim cache to be 0, the caches with victim indeed works a little bit faster, which conforms my expectation.**

## 4 Integration with CPU Simulator

### 4.1 Implementation Details

The time to access memory is set to be 100. The settings of cache is set to be in Table 4. In this part, I used the MemoryManager.cpp, BranchPredictor.cpp, Simulator.cpp, and MainCPU.cpp from the reference. I used my own Cache.cpp, and a little modification to the MainCPU.cpp and Simulator.cpp.

### 4.2 Performance analysis

```
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59
58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14
13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
-----
STATISTICS
Number of Instructions: 183682
---Run without Cache---
Number of Cycles: 5182013
Avg Cycles per Instruction: 49.2083
Branch Prediction Accuracy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7316
Number of Data Hazards: 8648
Number of Memory Hazards: 23398
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3
```

Figure 14: Without Cache

```
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3 $ ./build/simulator -c riscv-elf/quickSort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59
58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14
13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
-----
STATISTICS
Number of Instructions: 183682
---Run with Cache---
Number of Cycles: 214068
Avg Cycles per Instruction: 2.6724
Branch Prediction Accuracy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7316
Number of Data Hazards: 8648
Number of Memory Hazards: 23398
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3
```

Figure 15: With Cache

Figure 16: quicksort.riscv

```
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3 $ ./build/Simulator riscv-elf/ackermann.riscv
Ackermann(0,0) = 1
Ackermann(0,1) = 2
Ackermann(0,2) = 3
Ackermann(0,3) = 4
Ackermann(0,4) = 5
Ackermann(1,0) = 2
Ackermann(1,1) = 3
Ackermann(1,2) = 4
Ackermann(1,3) = 5
Ackermann(1,4) = 6
Ackermann(2,0) = 3
Ackermann(2,1) = 5
Ackermann(2,2) = 7
Ackermann(2,3) = 9
Ackermann(2,4) = 11
Ackermann(3,0) = 5
Ackermann(3,1) = 13
Ackermann(3,2) = 29
Ackermann(3,3) = 61
Ackermann(3,4) = 125
Program exit from an exit() system call
-----
STATISTICS
Number of Instructions: 438765
---Run without Cache---
Number of Cycles: 16944864
Avg Cycles per Instruction: 39.3367
Branch Prediction Accuracy: 0.5045 (Strategy: Always Not Taken)
Number of Control Hazards: 48012
Number of Data Hazards: 279916
Number of Memory Hazards: 47774
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3
```

Figure 17: Without Cache

```
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3 $ ./build/Simulator -c riscv-elf/ackermann.riscv
Ackermann(0,0) = 1
Ackermann(0,1) = 2
Ackermann(0,2) = 3
Ackermann(0,3) = 4
Ackermann(0,4) = 5
Ackermann(1,0) = 2
Ackermann(1,1) = 3
Ackermann(1,2) = 4
Ackermann(1,3) = 5
Ackermann(1,4) = 6
Ackermann(2,0) = 3
Ackermann(2,1) = 5
Ackermann(2,2) = 7
Ackermann(2,3) = 9
Ackermann(2,4) = 11
Ackermann(3,0) = 5
Ackermann(3,1) = 13
Ackermann(3,2) = 29
Ackermann(3,3) = 61
Ackermann(3,4) = 125
Program exit from an exit() system call
-----
STATISTICS
Number of Instructions: 438765
---Run with Cache---
Number of Cycles: 764123
Avg Cycles per Instruction: 1.7739
Branch Prediction Accuracy: 0.5045 (Strategy: Always Not Taken)
Number of Control Hazards: 48012
Number of Data Hazards: 279916
Number of Memory Hazards: 47774
songlin@songlin-virtual-machine: ~/Desktop/CSC3050/Project3
```

Figure 18: With Cache

Figure 19: ackermann.riscv

For quicksort.riscv, the CPI is 49.2 without cache, but it is 2.07 with cache. For ackermann.riscv, the CPI is 39.3 without cache, but it is 1.77 with cache. The Cache significantly improves the performance of programs.