CSC3050 Project3

1. Introduction:

Cache is an important component of a CPU system that has a significant impact on computer performance by reducing memory access times. The focus of this project is to simulate the cache in the RISC-V architecture, including:

- Single-level Cache Simulation: Implement a cache simulatot that enables the single-level cache simulation. Simulate different cache conditions for single-level cache and compare how different cache conditions affects the cache performance.
- Multi-level Cache Simulation: Implement the multi-level cache simulation and further exam how multi-level cache can improve the performance.
- Integration with CPU Simulation: Integrate the cache simulator with CPU simulator that simulates a pipeline CPU with multi-level cache.

2. Environment, Compilation, and Execution

2.1. Environment

The testing environment for the simulator is Linux, programming language is C++, build system is CMake.
The project should be compile and run in Linux. CMake is needed.

2.2. Compilation

Single-level cache simulation: change directory to /single

Multi-level (inclusive) cache simulation: change directory to /mul_inclusive

Multi-level (exclusive) cache simulation: change directory to /mul_exclusive

Multi-level (inclusive with victim) cache simulation: change directory to /mul_victim

Integration with CPU simulation: change directory to /CPU

Then use these lines in the command line to compile:
mkdir build

```
cd build
cmake ..
make
```

or simply run : bash build.sh

2.3. Execution

Single-level cache: change directory to /single/build
run in the command line:
./Single ../trace1.trace

Multi-level (inclusive) cache: change directory to /mul_inclusive/build
run in the command line:
./Inclusive ../trace1.trace

Multi-level (exclusive) cache: change directory to /mul_exclusive/build
run in the command line:
./Exclusive ../trace1.trace

Multi-level (inclusive with victim) cache: change directory to /mul_victim/build
run in the command line:
./Victim../trace1.trace

Integration with CPU simulation: change directory to /CPU/build, make and then run in the command line. -c: with cash; default: without cache
./Simulator ../riscv-elf/ackermann.riscv -c

3. Single-Level Cache Simulation

3.1 Implementation details

The code implements a single-level cache simulator designed to manage data between the processor and memory.
The core components are built around the Cache class, which handles the logic for caching data and is integrated with a MemoryManager for simulating lower-level memory access.
a. Cache Configuration: The Cache::Policy struct defines the cache's configuration, including the size of the cache (cacheSize), the size of each block within the cache (blockSize), the total number of blocks (blockNum), the degree of associativity (associativity), and latencies associated with cache hits (hitLatency) and misses

(missLatency).

b. Initialization: Upon instantiation, the Cache constructor initializes the cache based on the specified policy. It also determines if the cache operates with write-back or write-allocate policies.

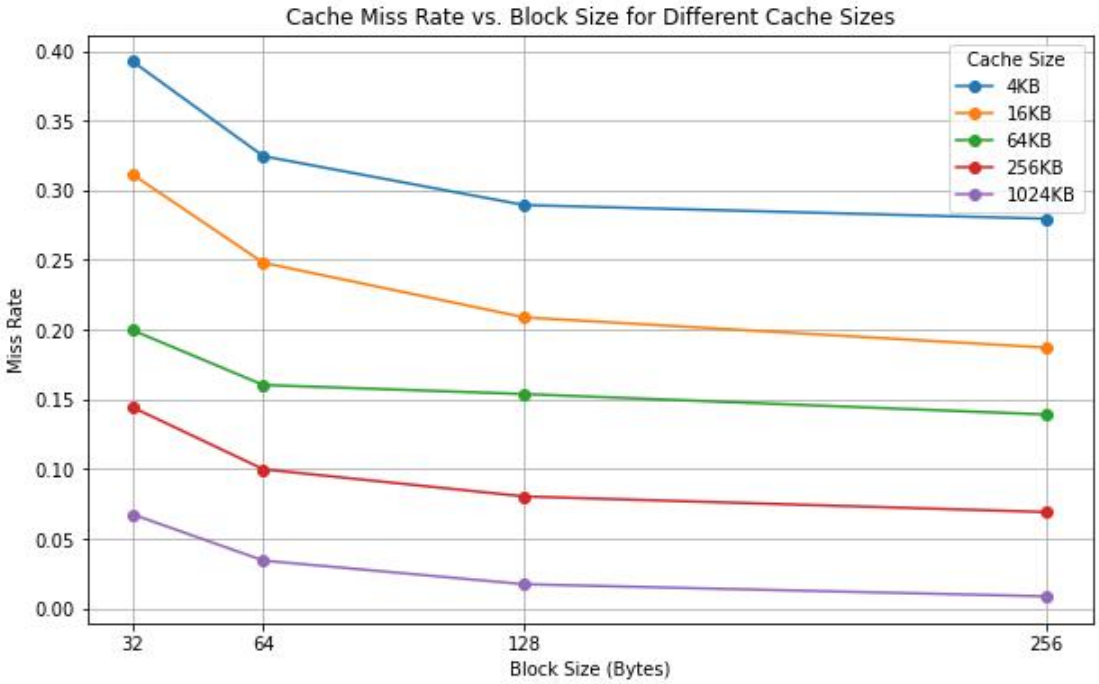
c. Read and Write Operations:
Cache hit: If a requested address is present and valid (getBlockId identifies the block's ID from the address), the operation is quickly completed using the data within the cache (getByte returns the byte directly). Then updates the hit count, and resets the block's last access time.
Cache miss: If a requested address isn't present in the cache, it results in a cache miss, and getByte loads the required data block from main memory. The function loadBlockFromLowerLevel loads the data into a new cache block. It also evaluates whether there is a need to replace an existing block. In cases where replacement is necessary, the least recently used (LRU) block is chosen for eviction. For write-back configurations, any blocks marked as modified are written back to the main memory before they are replaced.
Data transferring: The simulator checks each block's validity and tag to ensure proper alignment with memory addresses within cache blocks. Data is transferred in predetermined block sizes, allowing the simulator to effectively handle both read and write operations from memory or lower cache levels.
Output: Results are output to a CSV file with columns for each cache configuration and its corresponding miss rate and total cycles.

3.2 Performance Evaluation

3.2.1 Miss Rate with Block Size under different cache sizes
Impact of **Cache Size**:
As cache size increases, miss rate significantly decrease. For example, for a block size of 32 bytes, the miss rate drops from 0.3924 at 4KB to 0.0674 at 1MB. This indicates that larger caches can store more data blocks, reducing the need to access lower levels of storage due to cache misses, thus enhancing performance.

Impact of **Block Size**:
Within each cache size configuration, miss rates generally decrease as block size increases. This trend is especially caused by smaller cache sizes, where the reduction in miss rate is more significant. Increasing the block size reduces the miss rate because larger blocks mean more data is loaded into the cache at once, decreasing the likelihood of cache misses in the near future.

Combined Effect of Block Size and Cache Size:
Combinations of larger caches and larger block sizes seem to offer the best miss rate performance.



3.2.2 Miss Rate with Associativity under different cache sizes

Impact of **Cache Size**:

Miss rate generally decrease as cache size increases. Larger caches can hold more data, reducing the frequency of cache misses. This is especially clear as we move from 4 KB to 1024 KB cache sizes across different associativity levels. The larger cache size allows more data to be stored, thus reducing the miss rate.

Impact of **Associativity**:

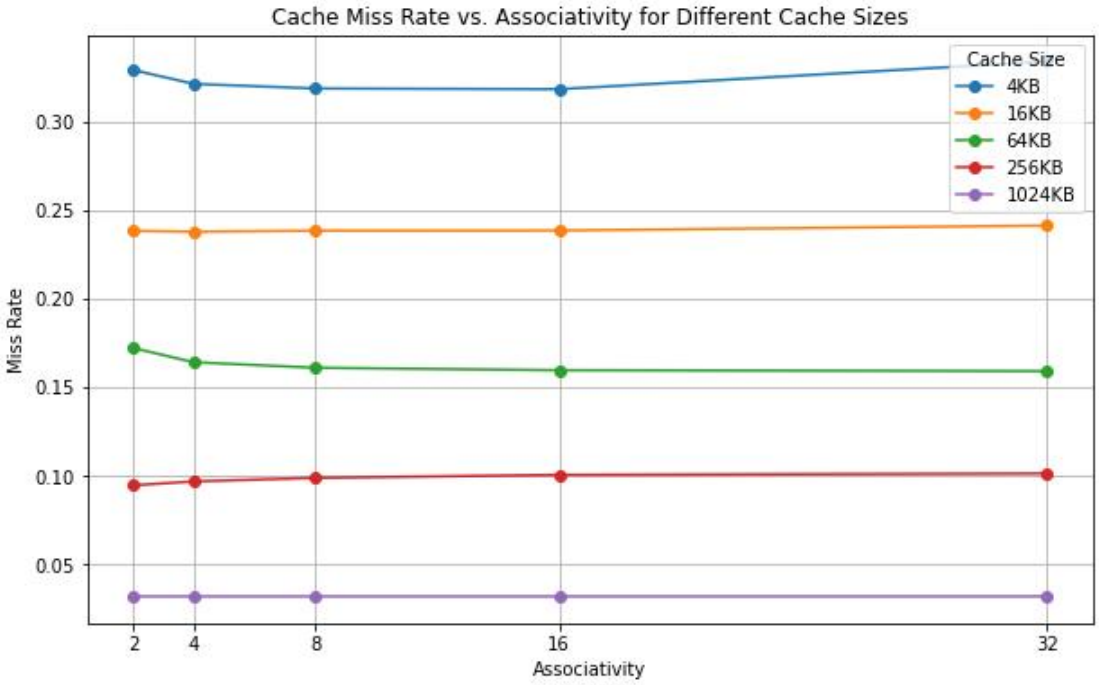
The influence of associativity on miss rates varies based on the cache size:

In smaller caches (4KB), increasing associativity initially lowers the miss rate slightly until it reaches a certain point (associativity=16), beyond which the miss rate increases. This could be due to the fact that higher associativity caches have more blocks that need to be managed, leading to more conflict misses when data from different addresses maps to the same set in the cache.

For medium cache sizes (16KB and 64KB), as associativity increases, miss rates decrease up to a certain point before plateauing or slightly increasing.

In larger caches (256KB), increasing associativity leads to an increase in miss rates. This unusual behavior might suggest inefficiencies or specific workload characteristics that do not benefit from higher associativity in larger caches.

For the largest cache size tested (1MB), the miss rate remains constant regardless of associativity, indicating that for very large caches, the impact of associativity on miss rates becomes negligible for the given workload.



4. Multi-Level Cache Simulation

4.1 Implementation details

Inclusive Cache:
All data present in a lower-level cache is also contained within all higher-level caches. This approach simplifies the management of data consistency. However, this can result in inefficient use of cache space, as the same data is duplicated across multiple cache levels.

Exclusive Cache:
A block stored in a higher-level cache (e.g., L1) is not duplicated in any lower-level caches (e.g., L2, L3).
If a data request hits in the higher-level cache, the data is directly read from it and returned to the CPU. If there is a miss in the higher-level cache but a hit in the lower-level cache, the data from the lower-level cache is read and returned to the CPU.
`loadBlockFromLowerLevel()`: Loads a block from the lower cache level or memory into the current cache, and invalidate the corresponding block in the lower cache if it exists (due to the exclusivity policy).
If a cache line eviction happen, this evicted cache line is written to the lower level cache or memory.

Victim Cache: A small cache used to store cache lines that are evicted from the L1 cache. This cache acts as a temporary buffer to retain data that might otherwise be discarded, potentially avoiding costly data retrievals from lower-level caches or main memory. When a cache line is evicted from the L1 cache, it is transferred to the victim cache instead of being directly moved to the L2 cache. If there's a subsequent miss in the L1 cache but a hit in the victim cache, the data can be quickly restored to the L1 cache. This process helps reduce the time taken to retrieve data from slower memory components.

4.2 Performance Evaluation

4.2.1. Multi level V.S. Single level

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	717519
L2	3500800	816064	4287772	29092	0.006739	34984296
L3	186188	320896	2175254	7530	0.003450	44250880

Table1. Multi Cache (inclusive), Write Back, Write Allocate

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	3271427

Table2. Single Cache, Write Back, Write Allocate

The total cycles for L1 in the single-level cache are 3,271,427, which is drastically higher than 717,519 in a multi-level cache.
This suggests that handling all operations within a single cache layer, without the support of additional cache levels, requires significantly more time, likely due to more frequent accesses to slower main memory. In contrast, in multi-level cache, frequent access data is efficiently managed at higher levels (L1) and less accessed data filters down to lower levels (L2 and L3).
Overall, the multi-cache configuration offers a more balanced approach to handling various types of memory accesses, optimizing both speed and capacity across different cache levels. Conversely, a single-cache system, while simpler, may suffer from increased latency and reduced overall system performance due to higher dependency on slower memory components.

4.2.2. Inclusive V.S. Exclusive

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	717519
L2	3500800	816064	4287772	29092	0.006739	34984296

L3	186188	320896	2175254	7530	0.003450	44250880

Table3. Inclusive, Write Back, Write Allocate

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	1051063
L2	3500800	3484416	6902447	82769	0.011849	57395376
L3	5297216	1665344	6874565	87995	0.012638	146390800

Table4. Exclusive, Write Back, Write Allocate

The data presented in Tables 3 and 4 reflect the differences in cache performance between inclusive and exclusive caching strategies, both operating under a write-back, write-allocate policy.

The total cycles and miss rate in the exclusive setup are generally higher across L2 and L3 caches.
The comparison between exclusive and inclusive cache strategies reveals significant differences due to their data management approaches.

Miss Rates:
The exclusive strategy shows a higher miss rate in L2 and L3.
This difference may due to: in exclusive cache, the lower level does not have the same date as in the higher level, so when write, it is more likely that a miss occurs.
The inclusive strategy tends to have lower miss rates, reflecting its ability to quickly access duplicate data stored across multiple levels.

Total Cycles:
The exclusive strategy shows a higher number of total cycles, indicating increased operational costs due to frequent data movement.
When write-back, in inclusive cache, only write to lower cache when the evicted block has been modified.

```
if (this->writeBack && replaceBlock.valid && replaceBlock.modified) {
  this->writeBlockToLowerLevel(replaceBlock);
  this->statistics.totalCycles += this->policy.missLatency;
}
```

But in exclusive cache, as long as a block is evicted, regardless of whether it has been modified, it should be write to the lower level. Because the lower level does not contain the original data.

```
// Unlike inclusive, the evicted block should be sent to lower level,
// regardless of whether this block has been modified.
if (this->writeBack && replaceBlock.valid) {
  this->writeBlockToLowerLevel(replaceBlock);
  this->statistics.totalCycles += this->policy.missLatency;
}
```

This is also the reason why in L2 and L3, the num write increase significantly.

The num read in L3 also increases, partly due to the higher miss rate in L2.

These findings suggest that while exclusive cache can minimizes redundancy and improve storage efficiency, it does so at the cost of more complex data transfers and retrievals, increased processing overhead and complexity in cache management. Conversely, the inclusive strategy results in fewer total cycles but retains data redundancy across cache levels, with the cost of efficiency in storage.
The choice between inclusive and exclusive caching strategies depends on the specific performance goals and constraints of a system, balancing between quick access times and efficient use of cache space.

4.2.3 With Victim Cache V.S. Without

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	717519
L2	3500800	816064	4287772	29092	0.006739	34984296
L3	186188	320896	2175254	7530	0.003450	44250880

Table 5: without Victim Cache, Write Back, Write Allocate

	Num Read	Num Write	Num Hit	Num Miss	Miss Rate	Total Cycle
L1	181708	50903	177911	54700	0.235157	245362
L2	4289408	1632000	5892320	29088	0.004912	47820540
L3	1861632	320704	2174806	7530	0.003450	44249120
Victim	3500800	816064	4249842	67022	0.015526	4888010

Table 6: with Victim Cache, Write Back, Write Allocate

The victim cache acts as an intermediary buffer that quickly provides recently evicted data, minimizing the need to retrieve it from slower, lower-level storage.
When miss in L1 but hit in Victim cache, it still counted as a miss in L1, so the miss rate of L1 does not change. But the total cycle of L1 decreases, as it does not need to frequently visit lower caches
The miss rate of L2 decreases, as Victim cache provides a temporary storage for evicted data from L1, which can be quickly accessed if needed again.
However, the total cycle of L2 increase, may due to the increased write-back operations. The presence of a Victim Cache might increase the number of blocks that need to be written back from the L1 Cache to the L2 Cache. Especially under write-back policies, when modified blocks are moved from the L1 Cache to the Victim Cache and eventually need to be written back to the L2 Cache, this increases write operations.
The victim cache exhibits more reads than writes, partly because the L1 reads more than writes. When L1 miss a read or write, it will go to the Victim.

Overall, the victim cache in an inclusive cache hierarchy enhances system performance by serving as a quick-access buffer for evicted data, thereby reducing access times and improving efficiency across the caching system.

5 Integration with CPU Simulator

3.1 Implementation

Because in project 2, the main.cpp and simulator.cpp had already been implemented, only need to add cache based on project 2's simulator.
a. Initialize cache in the mainCPU.cpp.
b. Modify functions including getByte and setByte in memory_manager and added functions getByteNoCache and setByteNoCache.
The simulator is in 64 bit. When encounter a branch, always taken.
The access time to memory is set to 100 cycles.

3.2 Performance Evaluation

File name	CPI without Cache	CPI with Cache
ackermann	39.3694	1.7482
helloworld	22.4605	1.8947
matrixmulti	13.4775	1.5599
quicksort	49.2559	1.9030
test_arithmetic	19.5434	1.7071
test_branch	28.6011	1.6782
test_syscall	24.3407	1.6953

Table 7: CPI Comparison

After introducing a cache, the CPI (Cycles Per Instruction) significantly decreases.
The CPI with a cache is fairly consistent across different programs, hovering around 1.5 to 1.9.
Programs with better locality have a bigger decrease rate in CPI.
In contrast, programs like quicksort and ackermann involves many recursive calls, and spend most of their CPU cycles using a small part of the code to manipulate a small amount of data. Their locality results in maximum performance gain due to the introduction of the cache.
The test_arithmetic, test_branch, and test_syscall involve arithmetic operations, branching, and system calls respectively. The improvement in CPI across these tests illustrates the general advantage of caching in diverse operations, reducing cycle counts by improving access times to frequently used instructions and data.

This data underscores the critical role of caches in modern computing architectures,

particularly for performance-intensive applications. By minimizing the need to fetch data from slower main memory, caches enable faster execution of programs, effectively lowering the CPI and enhancing overall system performance.

References

[1] Hao He, “RISCV-Simulator,” https://github.com/hehao98/RISCV-Simulator, 2019.