Report for CSC3050 project 3

1. Introduction

This report is for CSC3050 project 3 Cache Simulator. In the project, First, we implement a single-layer cache and analyze the performance. Second, we create a three-level inclusive/exclusive cache (with an optional victim cache). Finally, we embed the tree-level cache into a C++ RV-32I pipeline simulator we implemented in Project 2. This project references He Hao's project [1]

2. How to Run

To run the project, users have 2 options: use the Vscode-based CMAKE tool or use the Bash-based tool. This report will focus on the former option.

2.1. Environment

The original project is built under 64bit Ubuntu 22.04.4 LTS. As CMakeLists.txt was provided, it is able to build for different devices.

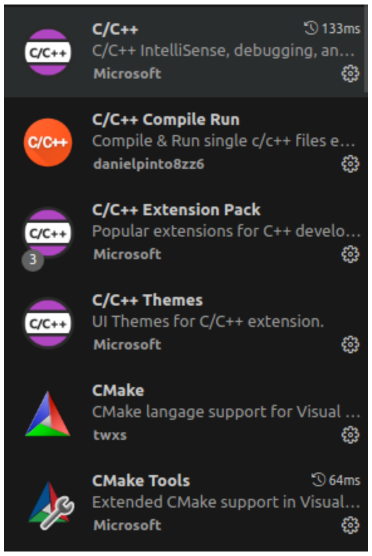
2.2. Build with vs code on Ubuntu

You can run `build.sh` to build the whole project.

2.2.5. More options:

The project is built using vscode. Please follow the steps below to make and compile the project.

- i. Download the `C/C++` extensions, `Cmake` extension, and `C/C++ Complie Run` extension (shown below).



C/C++ 133ms
C/C++ IntelliSense, debugging, an…
Microsoft

C/C++ Compile Run
Compile & Run single c/c++ files e…
danielpinto8zz6

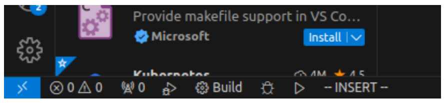
C/C++ Extension Pack
Popular extensions for C++ develo…
Microsoft

C/C++ Themes
UI Themes for C/C++ extension.
Microsoft

CMake
CMake langage support for Visual …
twxs

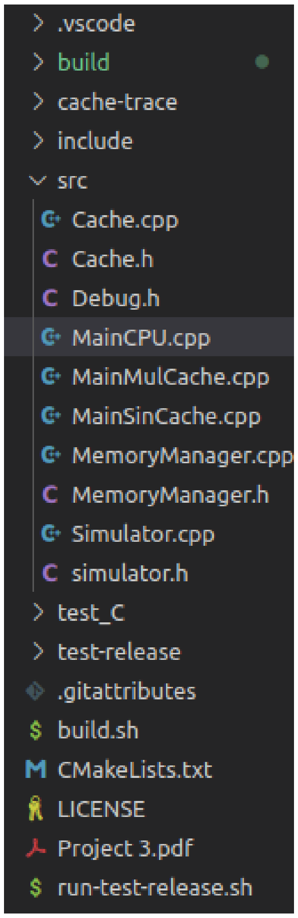
CMake Tools 64ms
Extended CMake support in Visual…
Microsoft

- ii. After downloading, it will show "build", "debug(a bug in)" and "run (a triangle img)" on the left side of the bottom.



Provide makefile support in VS Co…
Microsoft Install

- iii. Open the project file `CSC3050-project-3` in vscode (**please open the whole project file, do not just open MainCPU.cpp!**). You will find the project structure below:



```
> .vscode
> build
> cache-trace
> include
v src
    Cache.cpp
    Cache.h
    Debug.h
    MainCPU.cpp
    MainMulCache.cpp
    MainSinCache.cpp
    MemoryManager.cpp
    MemoryManager.h
    Simulator.cpp
    simulator.h
> test_C
> test-release
  .gitattributes
  build.sh
  CMakeLists.txt
  LICENSE
  Project 3.pdf
  run-test-release.sh
```

- iv. Now, users can try to build or run the program. **caution: this may not work, if don't work, try the 5th step**

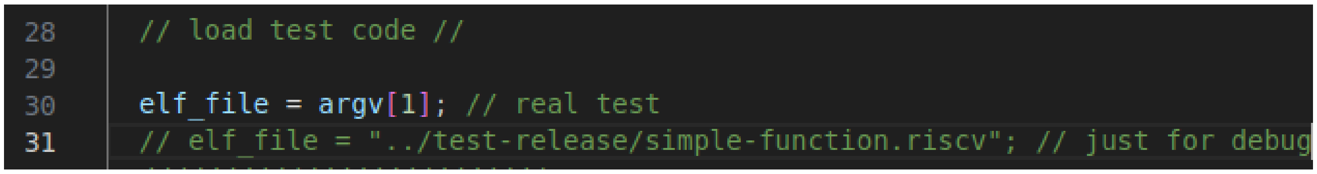
- v. If users cannot build the program, please remove the `build` file in the project, and create a new `build` file. click `build` bottom shown in stage 2.
- vi. Then users may be able to run the project.

2.3. ELF source option (For `MainCPU.cpp`)

To load the ELF file, users have to move the ELF file into `test-release` and have two options.

- i. Use bash to run `run-test-release.sh`
- ii. Load single ELF.

Users can change the option in `./src/MainCPU.cpp` by choosing to use `argv[1]` as the path or manually enter the path.



```
// load test code //

elf_file = argv[1]; // real test
// elf_file = "../test-release/simple-function.riscv"; // just for debug
```

2.4. Trance file read option

Users can find the place to change the trace path in `MainSinCache.cpp` , at line **79** or in `MainMulCache.cpp` at line **21**

Please make sure your path is valid. As the executed file is in `./build` directory, so we should use `../cache-trace/tarce1.trace` for most of the cases. However, if you prefer to use `./build/xxx.run` to run the executed file, you have to use `./cache-trace/trace1.trace` instead. If you are really confused by these path, you can just directly use the absolute path.

2.5. CMakelist.txt setings

The content of CMakelist.txt is below:

```cmake
cmake_minimum_required(VERSION 3.0)

project(RISCV-Simulator)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_FLAGS "-O0 -Wall")

include_directories(${CMAKE_SOURCE_DIR}/include)

# 1. single value excutable

#add_executable(
#    CacheSin
#    src/MainSinCache.cpp
#    src/MemoryManager.cpp
#    src/Cache.cpp
#)

# 2. multi value excutable

#add_executable(
#    CacheMul
#    src/MainMulCache.cpp
#    src/MemoryManager.cpp
#    src/Cache.cpp
#)

# 3. CPU embedding

add_executable(
    Simulator
    src/MainCPU.cpp
    src/Simulator.cpp
    src/MemoryManager.cpp
    src/Cache.cpp
)
```

You can see three `add_executable`, each one is one part. If you want to execute a different part, **please clear up the `build` file and change the executable**.

2.5. `build.sh`

Users can directly use the `build.sh` to make the project.

3. Implementation

3.1. Read trace file

The trace file is constructed by instruction: `r/w <adress>` like:

```
r 0x122e80
r 0x134608
r 0xd1e98
w 0xd3678
```

So we only need to decode `w/r` and address. and call `setByte(<address>, 0)` or `getByte(<address>)` (`0` in `setByte` is the value written to cache or memory. In the trace file, write value doesn't matter)

3.2. Cache class interface: `Cache.h`

To provide users with the basic interface of the Cache class, which is essential for the rest of the discussion.

```
class Cache {
public:
  struct Policy { // policy: all configration
    // In bytes, must be power of 2
    bool hasVictom = false; // do we have victom cache?
    uint32_t cacheSize;
    uint32_t blockSize;
    uint32_t blockNum;
    uint32_t associativity;
    uint32_t hitLatency;  // in cycles
    uint32_t missLatency; // in cycles
  };

  struct Block { // each cache block
    bool valid; // valid bit: 1 for valid (have data), 0 for clear
    bool modified; // 1 for modified
    uint32_t tag; // tag of the cache line
    uint32_t id; // id of the cache line
    uint32_t size;
    uint32_t lastReference;
    std::vector<uint8_t> data; // use a uint8_t vector as memory
    Block() {}
    Block(const Block &b) : valid(b.valid), modified(b.modified), tag(b.tag), id(b.id), siz
  };

  struct Statistics {...}; // record data

  Cache( // initializer
        MemoryManager *manager, // pass in MemoryManager, to make sure cache can write/loac
        Policy policy, // pass in policy as a structure
        Cache *lowerCache = nullptr, // nullptr means it is the lowest cache, connect to Me
        bool writeBack = true, // writeback / write through policy
        bool writeAllocate = true, // write allocate or not
        bool isExclusive = false, // set if it is exclusive cache?
        bool isVictom = false // if there is it a victom cache?
    );

  bool inCache(uint32_t addr); // check if a address is in the cache
  uint32_t getBlockId(uint32_t addr); // id: the id of the block in a set
  uint8_t getByte(uint32_t addr, uint32_t *cycles = nullptr); // main read interface
  void setByte(uint32_t addr, uint8_t val, uint32_t *cycles = nullptr); // main write inter

  void printInfo(bool verbose);
  void printStatistics();
```

```
    void writeBlockToVictom(Block&);

    Statistics statistics;

  private:
    uint32_t referenceCounter;
    bool writeBack;     // default true
    bool writeAllocate; // default true
    bool isExclusive = false; // is it exclusive? default false
    bool isVictom = false; // is this cache a victom cache?
    MemoryManager *memory;
    Cache *lowerCache;
    Cache *victomCache = nullptr;
    Policy policy;
    std::vector<Block> blocks; // how many blocks?

    void initCache(); // initialize the cache (mainly initialize the block vector)
    void loadBlockFromLowerLevel(uint32_t addr, uint32_t *cycles = nullptr); // get a block f
    void loadBlockFromMemory(uint32_t addr, uint32_t *cycles = nullptr); // get a block from
    uint32_t getReplacementBlockId(uint32_t begin, uint32_t end); // find out which Block id
    void writeBlockToLowerLevel(Block &b); // write to lower level
    bool isInLowerCache(uint32_t addr); // is it in lower cache?

    // Utility Functions
    bool isPolicyValid();
    bool isPowerOfTwo(uint32_t n); // check if it is power of two
    uint32_t log2i(uint32_t val); // return log2
    uint32_t getTag(uint32_t addr); // get tag for each possible record
    uint32_t getId(uint32_t addr); // which set it is in?
    uint32_t getOffset(uint32_t addr); // get offset of each block
    uint32_t getAddr(Block &b); // get the address
  };
```

3.3. Cycle count method

We count cycles in `getByte()` and `setByte()` methods. We count Numhit and add hit latency whenever the data we want is in the cache. If they are not in the cache, we will count Nummiss and add miss latency (miss latency is the hit latency of the next layer). So we will count latency cycles both at the miss cache and the hit cache. But when we embed the cache into the CPU, we only count hit latency.

4. Single-level cache

4.1. Implementation

Initialize the cache object with `lowerCache = nullptr` which can create a Single-level cache as the `lowerCache` pointer is `nullptr` , the cache will know that they are the lowest cache that directly reads/writes data from memory.

4.2. Experiment

The experiment of single-level caches is aimed to show the difference among different cache `policy` settings. We choose `trace 1` as our test trace. Our policy search range is shown below:

Settings	Range
Cache Size	4KB to 1MB, incremented by 4X.
Block Size	32Bytes to 256Bytes incremented by 2X.
Associativity	2 to 32 incremented by 2X.
Write Back	True or False.
Write Allocate	True of False.

Table 1: setting search range

We will traverse all possible combinations. For each setting combination, we will go through the whole trace and record the outcome in the format below:

acheSize	blockSize	associativity	writeBack	writeAllocate	missRate	totalCycles
...	...	...	...	...	...	...

Table 2: records sample

The outcome is stored in `cache-trace/trace1.trace.csv` . Based on this csv file, we can compare different settings.

4.3. Analysis

4.3.1. Analysis 1: variation of average miss rate of different block size and cache size

We count the average miss rate of different block sizes and cache sizes, plotting it in a graph.

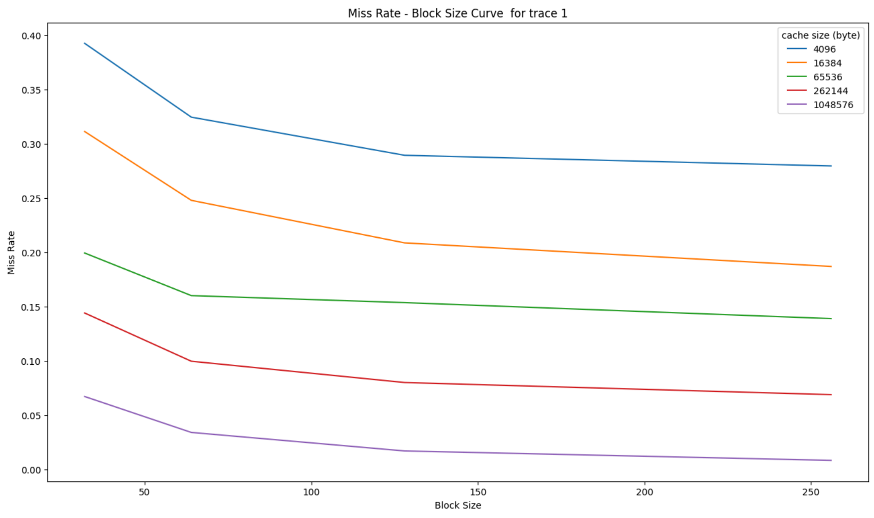


Figure 1: Average miss rate versus Cache size & Block size figure

In **Figure 1**, we can see that generally speaking, when the cache size is fixed, if we increase the block size, we can find out that the miss rate will decrease. Moreover, the decrease rate is slowing down. This is because when we increase the block size, we can handle more **spatial locality**. However, the reason why the decrease rate slows down is that if the cache size is fixed, when we increase the block size, the number of blocks will decrease. That is, we may handle less **temporal locality**. This will make the miss rate decline slower and slower as the block size increases.

Comparing different cache sizes, we can observe that when the cache size increases, the average miss rate will decline for every block size. This is because when the cache size increases, we can have more and more space to store data. So for fixed block size, we can save more blocks to better handle **temporal locality**. Meanwhile, as the block size is fixed, the capability to dealing **spatial locality** is same. So the performance will be improved.

4.3.2. Analysis 2: variation of average miss rate of different associativity and cache size

We count the average miss rate of different associativity and cache sizes, plotting it in a graph.

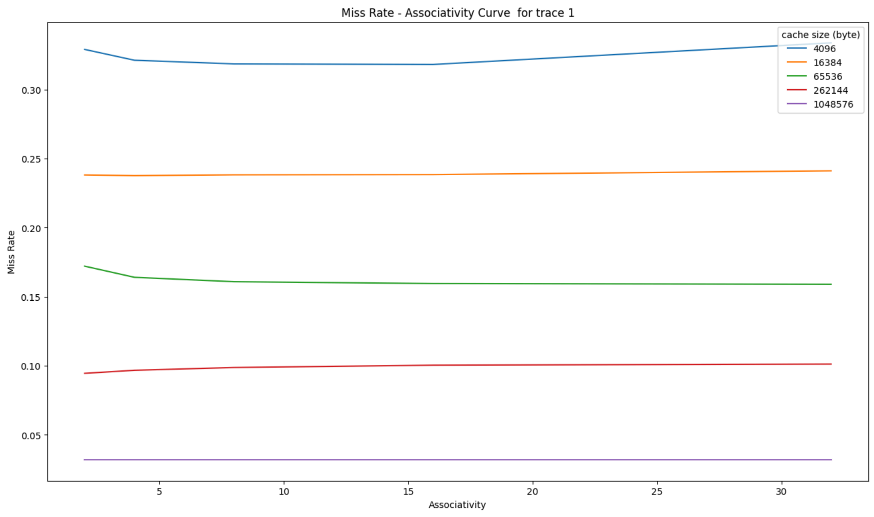


Figure 2: Average miss rate versus Cache size & Associativity figure

In **Figure 2** we can see the variation of the average miss rate of different associativity and cache sizes. if we fix associativity, the situation is similar to **Figure 1**. This means that for every fixed associativity, when the cache size increases, the miss rate will decline. The reason is similar too, the increased cache size provides more space to handle **temporal and spatial locality**. However, things become more tricky for fixed cache size. For fixed cache size, when we increase the associativity, the miss rate may remain or even increase. For larger cache sizes, this situation will become more and more obvious. Theoretically, if the replace algorithm is proper, the increases in associativity may help deal with the miss rate as it can provide a more flexible mapping strategy. But the experiment shows it is not the point.

We do not have a simple and general explanation of this phenomenon. But we can provide a possible answer. The **`LRU` (Least Recently Used) replace algorithm** which we used to find replace block (when the block set is full) may be not efficient for some particular access order. eg, for a 3-line associative set, if we access block tag `1, 2, 3, 4, 1, 2, 3, 4` , the miss rate will be 100% as the cache will be full when we access 4, and we have to discord 1 (and we have to load it back one step later). To access 1, we have to discard 2 which will be accessed on step later too! This issue will happen more and more frequently as the associative size increases, more and more blocks will map to the same set. This may cause an increase in the miss rate.

5. Multi-level cache

5.1. Implementation

The core method of implementing multilevel cache is `lowerCache` pointer. This pointer constructs a `linked list` structure that links every higher-level cache to the lower cache. This pointer allows the higher-level cache load data blocks from lower-level caches (or memory) or write data blocks to lower-level cache (or memory).

Inclusive cache can be easily implemented in this structure. Whenever we encounter cache misses, just recursively load it from the lower cache using the lower-cache pointer. We only need to write the data back if we have to change the data (`dirty bit == 1`). This writ back operation will execute recursively.

However Exclusive cache is much more different. The exclusive cache needs to remove the data block from the lower cache when we move it up. And we can directly read data from memory (via MemoryManager pointer). We only move discord data to the lower-level cache. So we have to recursively check whether a data block exists in the lower cache. Otherwise, we have to directly load it from memory.

Victim cache is implemented specially. We initialize the victim cache as a member object of the common cache. The victim cache's `lowerCache pointer` is pointed to the lower-level cache of the common cache it initialized in. It is just a backup for discarded block lines from the common cache. The only things that need to be recognized is that when the common cache discards the block, it will write to the victim cache ignoring the dirty bit. However, the block may still write to the lower cache if the dirty bit is 1.

5.2. Experiment

The experiment of multi-level caches is aimed to show the difference among different multi-cache policies (such as inclusive or not, has victim or not). However, the basic structure will be fixed. We choose `trace 1` as our test trace. Our policy search range is shown below:

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

Table 3: fixed structure for Multi-level structure

Besides, we provide a similar single-level cache for comparison.

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

Table 4: fixed structure for Single-level structure

We manually set the I/O latency of memory as 100 cycles, this setting will remain in the later discussions.

5.3. Outcome

We compare single-level and multiple-level cache in trace 1. We record Num Read, Num Write, Num Hit, Num Miss and Total Cycle to compare different implementations for different multi-level cache settings.

5.3.1. Comparing Single-level cache and Multi-level cache

We set the Multi-level cache as an inclusive, non-victim cache.

Level	Num Read	Num Write	Num Hit	Num Miss	Total Cycle	Miss Rate
Single-level (non-victim)	232611	50903	228814	54700	6973914	0.309

Table 4: Single-level Cache Performance

Level	Num Read	Num Write	Num Hit	Num Miss	Total Cycle	Miss Rate
L1	232611	50903	228814	54700	768422	0.309
L2	3500800	816064	4287772	29092	34984296	0.006
L3	1861888	320896	2175254	7530	44258080	0.003

Table 5: Inclusive Multi-level Cache without Victim Cache Performance

Compared with the single-level cache, and the multi-level cache, we can find out that the L1 cache and the single-level cache have the same statistics except for the total cycle. It is because almost every setting are same for two cache objects except hit latency cycles. Comparing L1, L2 and L3, the miss rate declines as the cache level decreases. This is because of the increase in cache size.

We can find out that the multi-level structure does not change the miss rate of L1. However, it can decline the Cycle it needs to get from lower caches. Compared with the single-level cache, the miss rate of L2 and L3 is much smaller.

5.3.2. Comparing Inclusive and Exclusive Cache

The statistic of the Inclusive cache is provided in **Table 5** so we only provide the Performance of the exclusive cache (without the victim cache) here.

Level	Num Read	Num Write	Num Hit	Num Miss	Total Cycle	Miss Rate
L1	232611	50903	228814	54700	3490002	0.309
L2	1674368	1801856	3448070	28154	30406900	0.008
L3	18880	22208	40741	347	849520	0.008

Table 6: Exclusive Multi-level Cache without Victim Cache Performance

Comparing exclusive cache and inclusive cache, we can find out that exclusive cache has smaller cycle times for L2 and L3 but has more cycles for L1. The miss rate of exclusive cache slightly increases for L2 and L3. L1 remains the same statistics.

First, the increase in L1's total time cycle is that the exclusive cache can load data lines from memory directly if they are not in the cache. So the miss latency may be high especially when the cache does not warm up. On the other hand, the L1 cache in inclusive cache will only load data from L2, so L1's latency will be smaller.

Second, the exclusive cache has a better performance of L2's and L3's cycle time. This is because when the data line does not exist in the cache, the inclusive cache must first load it to L3, then load it to L2, and finally load it to L1. So the latency of L2 and L3 will both increase. However, the exclusive cache directly loads data from the cache. So it will not increase the total cycle of L2 and L3. In trace 1, this provides exclusive cache with a better overall performance compared with inclusive cache.

Thirdly, the cycle time of L3 increases significantly in the exclusive cache, this may be because L3 may have less frequency of communicating with memory as most of the memory load operation is directly handled by L1 cache.

Finally, the Miss Rate slightly increased maybe because the L2 and L3 do not warm up as the trace size is too small. We can observe that the number of read and write in L2 and L3 is relatively small.

5.3.3. Performance of Victim cache

Our victim cache will only apply to L1, the settings of our victim cache are below:

Level	Capacity	Associativity	Block Size	Write Policy	Hit Latency
L1 victim	512 Bytes (8 blocks)	8 way	64 Bytes	Write Back	1 Cycle

Table 7: Settings for victim cache

We compare the performance of the victim cache under the Inclusive cache structure. The performance of inclusive cache without victim cache is provided in **Table 5**. Here we will provide the performance with cache in L1 cache:

Level	Num Read	Num Write	Num Hit	Num Miss	Total Cycle	Miss Rate
L1	232611	50903	228814	54700	763375	0.309
L2	3454656	816064	4241631	29089	34615168	0.006
L3	1861696	321088	2175254	7530	44258080	0.003

Table 8: Inclusive Multi-level Cache with Victim Cache Performance

We can observe that the introduction of victim cache reduces the total cycle of most caches. Moreover, it reduces the I/O number of the lower cache. This is mainly because the victim cache can act as a buffer for L1, so we can get data from the victim cache instead of lower caches. This will decline the I/O number of the lower cache. As the hit latency of the victim cache is relatively small, it can reduce the total cycle of L1. However the size of the victim cache is small, so it may not make a large difference.

6. Embedding to CPU

6.1. Implementation

We embedded the change into the CPU we defined in Project 2 by embedding it into MemoryManager. The MemoryManager of Project 2 naturally supports cache. As the main I/O interface of the Cache class is `Cache::setByte()` and `Cache::getByte()` , MemoryManager also implements `MemoryManager::setByte()` and `MemoryManager::getByte()` to use Cache's interface. And any other I/O function such as `MemoryManager::setInt()` will only call `MemoryManager::setByte()` . So we only need to embed the cache interface into `MemoryManager::setByte()` and `MemoryManager::getByte()` then the cache is successfully embedded into the RV32I CPU.

6.2. Experiment

Our experiment compares the CPI of the ideal CPU with **zero memory access latency** and our inclusive three-level Cache without victim cache defined in section 5 (**see Table 3**). The executable file is compiled in RV-32I ISA, in the `test_C` file.

6.3. Outcome

The outcome is shown below:

executable	CPI for ideal CPU	CPI of Cache	instruction number
ackermann.riscv	1.71	1.72	335363
helloworld.riscv	1.61	1.71	160
matrixmulti.riscv	1.66	1.66	189112
quicksort.riscv	1.94	1.95	84963

Table 9: CPI comparison

Generally, the cache CPU performance is slightly worse than the ideal CPU. This is trivial because the cache CPU has to consider latency between different layers of cache and a high I/O latency from memory while the ideal CPU has zero latency. However, it seems that cache can make the performance of the CPU close to the ideal CPU. This may partly be due to high temporal and spatial locality in instruction fetching.

Comparing the difference of CPI between the ideal CPU and cache CPU, we can find out that the more complex the executable file is, the less difference between two CPUs. That may partly be because if the executable file is too simple (such as print hello word), the cache may still be cold.

The spatial and temporal locality will affect the performance too. For higher spatial and temporal locality tasks such as matrix multiplication, the CPI of the cache CPU is much closer to the ideal CPU.

To sum up, the embedding of cache makes the CPU's CPI performance close to ideal situations. Considering the memory I/O latency is high (100 cycles), this is a huge improvement as we have to access memory almost every cycle to fetch instructions. If we do not use cache, we have to wait at least 100 cycles for each instruction.

7. Reference

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