Lab 2

Letong Han

2021533062

0. Designs and Implementations

- src/Cache.h

- src/Cache.cpp

- src/MainCache.h

- src/MainCacheOptimized.cpp

+ src/CacheLayer.hpp

|- class CacheLayer

+ src/CacheLayer.cpp,

+ src/L1L2Cache.hpp

|- class L1L2Cache

|- class CacheWarpper

+ src/L1L2Cache.cpp

We also modified MainCPU.cpp and MemoryManager.cpp to replace original cache with ours.

The stack size is reduce to 64 kilo bytes.

0.1 CacheLayer

As its name suggests, **CacheLayer** provides only one layer of Cache. It is not responsible for interacting with other layers of Cache.

Provided replacement policies: **LRU**, **RRIP** and **Optimal**, details in Section 2.

0.2 L1L2Cache

We adapted from the original Cache.hpp / Cache.cpp and provided a new class **L1L2Cache** to allow different replacement policies and inclusion policies for L1 and L2 while maintaining the same interface as the original **Cache**.

For simplicity, cache lines in different layers are required to be the same size and **L1L2Cache** is a **write-back** and **write-allocate** cache.

Provided inclusions policies: **Inclusive**, **Exclusive** and **Non-Inclusive**, details in Section 1.

Also, it by default dumps the trace of memory access to **dump.trace**, which is useful for section 2 and 3.

0.3 CacheWrapper

We also implemented class **CacheWrapper** to provide a uniform and easy-to-use interface for generating caches with different configurations.

Tunable parameters including:

(1) The size of each cache block

(2) Number of cache blocks in L1

(3) Associativity of L1

(4) Replacement policy of L1: **LRU/RRIP/Optimal**

(5) Number of cache blocks in L2

(6) Associativity of L2

(7) Replacement policy of L2: **LRU/RRIP/Optimal**

(8) Is there a victim cache and the victim cache size

(9) Inclusion policy for L1 and L2: **Inclusive/Exclusive/Non-Inclusive**

(10) When L1/L2's replacement policy is **Optimal**, it is required to provide trace of memory access.

0.4 Experiment

Throughout the lab, we use **AMAT** (Average Memory Access Time) as an index to compare different caches. However, some workload is not counted into the "cycles" variable in get/setByte, including installing cache line after a miss, backward invalidation to maintain inclusive hierachy or writing a cache line down to lower-level memory hierachy. How this kind of **occupancy** in L1/L2 caches converts to **latencies** of get/setByte are not specified, and is not taken into consideration in this lab.

1. Mutli-level Caches

The original inclusion policy is Non-Inclusive, since

(1) A cacheline can exist in both L1 and L2, so it is not Exclusive.

(2) It has no backward invalidation when a cacheline in L2 is evicted, so it is not Inclusive.

In conclusion, it is Non-Inclusive (NINE).

1.1 Exclusive

demonstration code in **(src/Section1.cpp) exclusive()**

Implementation:

(1) Hit L1: update lastReference/RRPV only

(2) Hit L2: move the cache line from L2 to L1

(3) L1 eviction: install the cache line to L2

(4) L2 eviction: write back to memory if dirty

(5) L2 Miss: install the cache line to L1 only

We use a 2 byte fully-associative L1, 4 byte fully-associative L2 cache for demonstration. Replacement policy for both layer is LRU.

```
Initial State
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 2: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 3: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  0 (L1, L2 miss)
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 3) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 2: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 3: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  1 (L1, L2 miss)
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 3) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 2: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 3: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  2 (L1, L2 miss)
--------------------------L1--------------------------
Block 0: addr 2 tag 0x2 id 0 valid   unmodified (last ref 9) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 4) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 2: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 3: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  3 (L1, L2 miss)
--------------------------L1--------------------------
Block 0: addr 2 tag 0x2 id 0 valid   unmodified (last ref 9) 0
Block 1: addr 3 tag 0x3 id 0 valid   unmodified (last ref 12) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 4) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
Block 2: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 3: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

As we can see, this configuration is exclusive since cache line cannot exist in both L1 and L2, it only install cache line to L1 on a L2 miss and cache line evicted from L1 are placed in L2.

1.2 Inclusive

demonstration code in **(src/Section1.cpp) inclusive()**

Implementation:

(1) Hit L1: update lastReference/RRPV only

(2) Hit L2: install the cache line to L1

(3) L1 eviction: write to L2 if dirty

(4) L2 eviction: write to memory if dirty, notify L1 for backward invalidation

(5) L2 Miss: install the cache line to both L1 and L2

We use a 2 byte fully-associative L1, 2 byte fully-associative L2 cache for demonstration. Replacement policy for both layer is LRU.

```
Initial State
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  0 (L1,L2 miss)
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 2) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 3) 0
Block 1: addr 0 tag 0x0 id 0 invalid unmodified (last ref 0) 0
------------------------------------------------------
```

```
Read  1 (L1,L2 miss)
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 2) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 4) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 3) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
------------------------------------------------------
```

```
Read  0 (L1 hit)
--------------------------L1--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 5) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 4) 0
--------------------------L2--------------------------
Block 0: addr 0 tag 0x0 id 0 valid   unmodified (last ref 3) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
------------------------------------------------------
```

```
Read  2 (L1,L2 miss)
--------------------------L1--------------------------
Block 0: addr 2 tag 0x2 id 0 valid   unmodified (last ref 7) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 4) 0
--------------------------L2--------------------------
Block 0: addr 2 tag 0x2 id 0 valid   unmodified (last ref 9) 0
Block 1: addr 1 tag 0x1 id 0 valid   unmodified (last ref 6) 0
------------------------------------------------------
```

We show this cache is inclusive by demonstrating backward invalidation caused by a L2 eviction.

The backward invalidation happen in this step. Note that Block 1 is the LRU block in L1, however since Inclusive hierachy requires cache line to be installed to both L1 and L2 and Block 0 is the LRU block in L2, the Block 0 in L1 was set to invalid by backward invalidation. So instead of Block 1, Block 0 was evicted even though it is not the LRU block.

2. Cache Replacement

2.1 RRIP

We implemented SRRIP (Static Re-Reference Interval Prediction). Let M be the number of bits of RRPV.

The newly arrived cacheline has RRPV of $2^M - 2$; when a block is referenced, its RRPV is set to 0.

During eviction, we first look for blocks with RRPV of $2^M - 1$ in a set, if not found, increment all blocks' RRPV by 1. Repeat the above procudure until a block with RRPV of $2^M - 1$ is found.

2.2 Optimal

The algorithm to calculate an cache line's next reference is given below.

```
// calculate the begin address of a cache line
inline uint32_t get_cacheline(uint32_t addr, uint32_t cacheline_bits) {
  return addr & (~((1 << cacheline_bits) - 1));
}

// next_ref_map: the next reference time of each cache line, the cache line is determined
by its begin address
// next_ref[i]:  the next reference time of the cache line that trace[i].addr is on
for (int i = trace.size() - 1; i >= 0; --i) {
  auto cacheline_addr = get_cacheline(trace[i].addr, cacheline_bits);
  if (!next_ref_map.count(cacheline_addr)) {
    next_ref[i] = trace.size();
  } else {
    next_ref[i] = next_ref_map[cacheline_addr];
  }
  next_ref_map[cacheline_addr] = i;
}
```

We give the cache "hints" through

```
std::map<uint32_t,uint32_t> next_ref_map
```

which give the cacheline's next reference time.

On a replacement, the block with farthest reference in the future is evicted

After each r/w operation, this map will be updated to reflect the up-to-date status.

```
next_ref_map[get_cacheline(addr, cacheline_bits)] = next_ref[trace_index];
trace_index++;
```

2.3 Result

code in **src/Section2.cpp**

Configuration:

Cache Line: 16 bytes

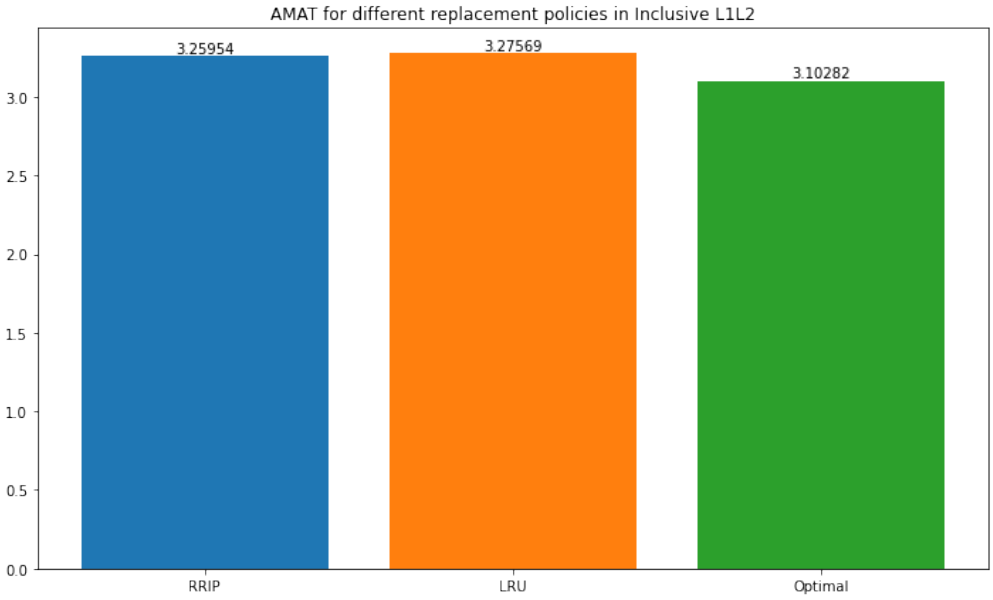
L1: 512 bytes, 4-way-set-associative

L2: 2048 bytes, 8-way-set-associative

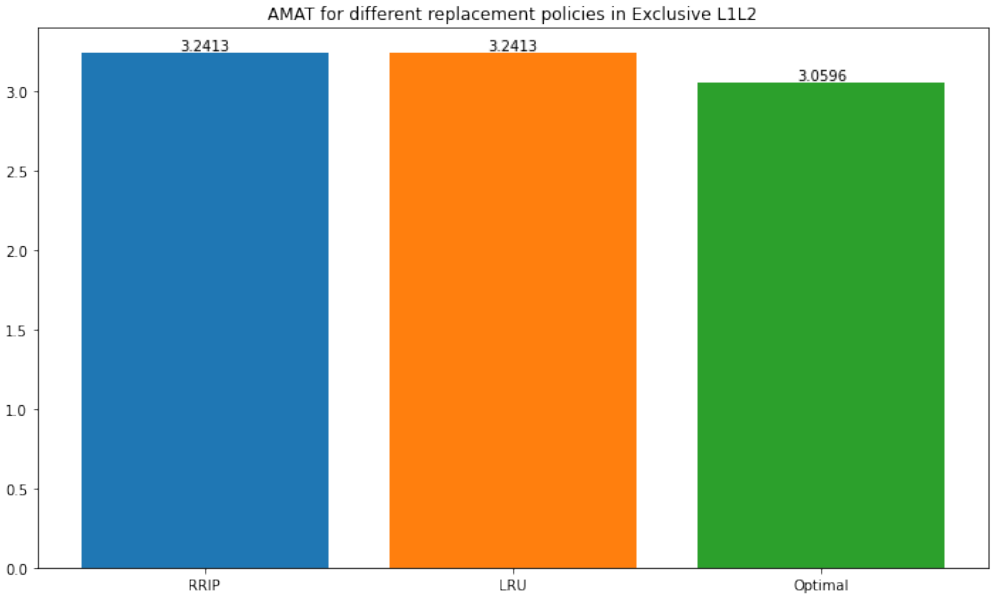
Trace: lab2/transpose.trace, which was generated by lab2/tranpose.c, which performs a matrix transpose.

In experiments, L1 L2 adopt the same replacement policies.

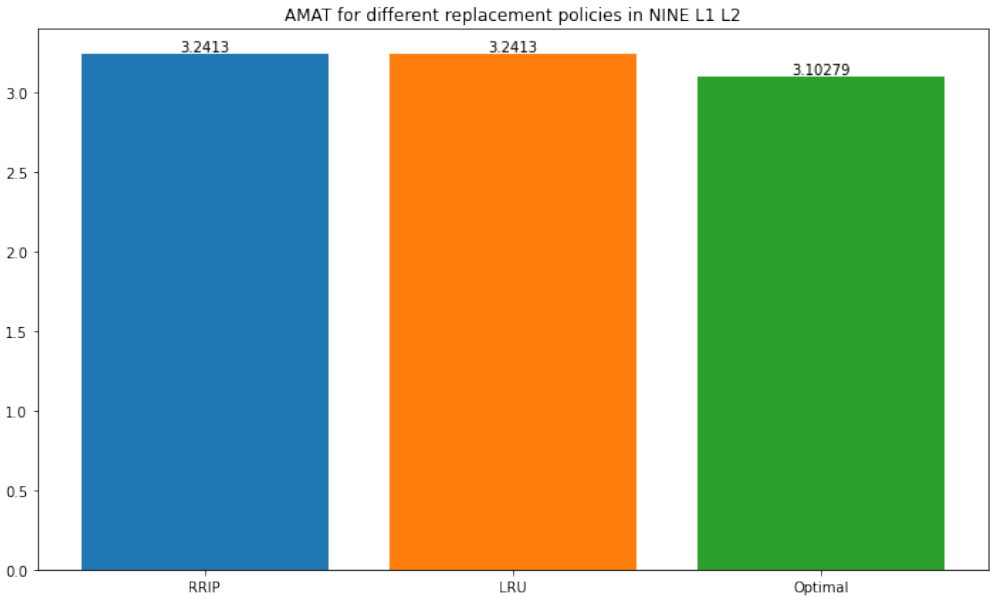
2.3.1 Inclusive



2.3.2 Exclusive



2.3.3 Non-Inclusive



Thus, we have shown the performance differance of **LRU**, **RRIP** and **Optimal** in Inclusive, Exclusive and Non-Inclusive caches, which is **Optimal**>=**{LRU**, **RRIP}**.

Another thing to note is that even with only two bits per cache line, RRIP's peformace is nearly equal and sometimes better than true-LRU, which demonstrates its effectiveness.

3.Victim Cache

3.1 Fit Victim Cache Between L1 and L2

According to the slides, when a dirty cacheline is evicted from L1, it will be sent to both victim cache and lower-level memory hierachy.

(1) Hit victim cache: Move the cache line from victim cache to L1. If there is any eviction, follow (2). If L1L2 is inclusive and that cache line is not present in L2, must sent this cache line to L2.

(2) Evict from L1: Send cache line to both victim cache and L2. This is the only way for victim cache to get populated.

(3) Evict from Victim Cache: Simply discard the cache line since it is content has already been sent to lower-level cache/memory.

Also note that in our implementation, we assume L1 look up and victim cache lookup are performed in parallel, so latencies are:

(1) L1 hit: 2 cycles

(2) Victim cache hit: 1 cycles:

(3) L2 hit: 8 cycles:

(4) L2 miss: 100 cycles

3.2 Result

code in **src/Section3.cpp**

trace is **lab2/victim.trace**, which is generated by **lab2/gen_victim_trace.cpp**

Configuration:

Cache Line: 16 bytes

L1: 1024 bytes, direct-mapped

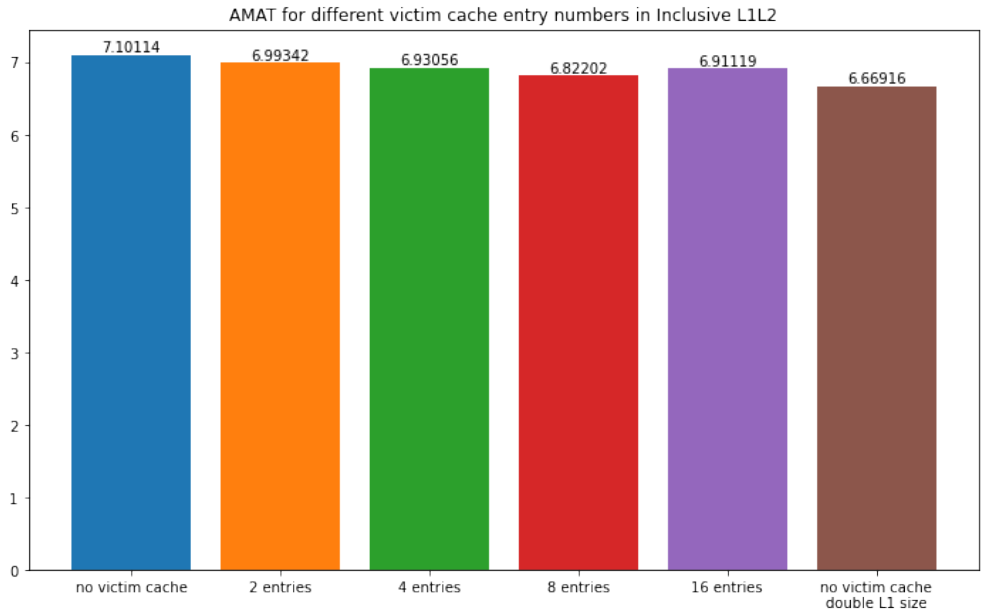
L2: 4096 bytes, 4-way-set-associative

Victim cache: fully-associative

In experiments, L1 and L2 both adopt the LRU for replacement policy.

3.2.1 Inclusive

Note that upon a victim cache hit, we must check the if cache line exists in L2 and if not, we have to sent it to L2. Or the inclusive hierachy of L1 and L2 would be violated.



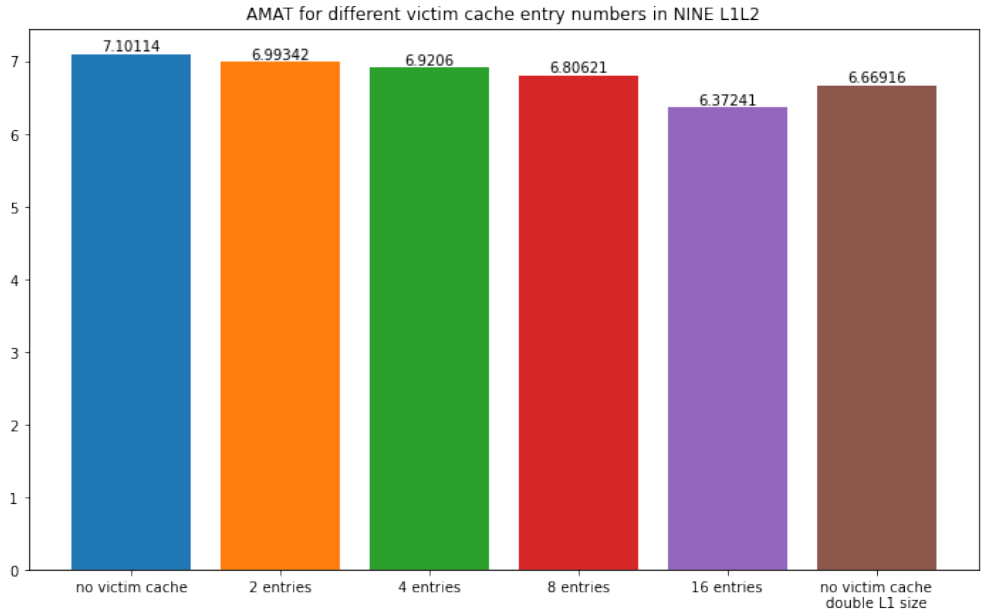
The performace drop from 8 entries to 16 entries may due to the following reasons:

(1) L2 only sees L1 misses, as a result, the most recently used cache line in L1 may be the LRU cache line in L2 and is more likely to be evicted. When the eviction happens, the cache line in L1 is invalidated, causing more misses.

(2) Whenever a cache line is brought back to L1 from victim cache, we have to ensure it is in L2. Sending a cache line to L2 may lead to backward invalidations, which worsens the (1) problem.

The combination of (1) (2) leads to poor performance.

3.2.2 Non-Inclusive



For Non-Inclusive L1L2 cache, the problem mentioned in 3.2.1 does not exist.

On this trace, adding a fully-associative victim cache of 1/4 L1 cache size achieves better performance than simply doubling L1 cache size.

This shows our victim cache is capable of "adding" associativity to the direct-mapped L1and reduces conflict misses.

3.2.3 Exclusive

However, when it comes to Exclusive cache, our implementation of victim cache differs from the one in the slides, and it is more similiar to the one mentioned in the mid-term.

be sent to both victim cache and lower-level memory hierarchy. This classmate thinks it might be more efficient if the victim cache is used as a *filter*, which means that any cache line evicted from upper-level cache is only sent to victim cache. When the victim cache is full, we apply cache replacement to make space and the victim cache line is sent to lower-level memory hierarchy.

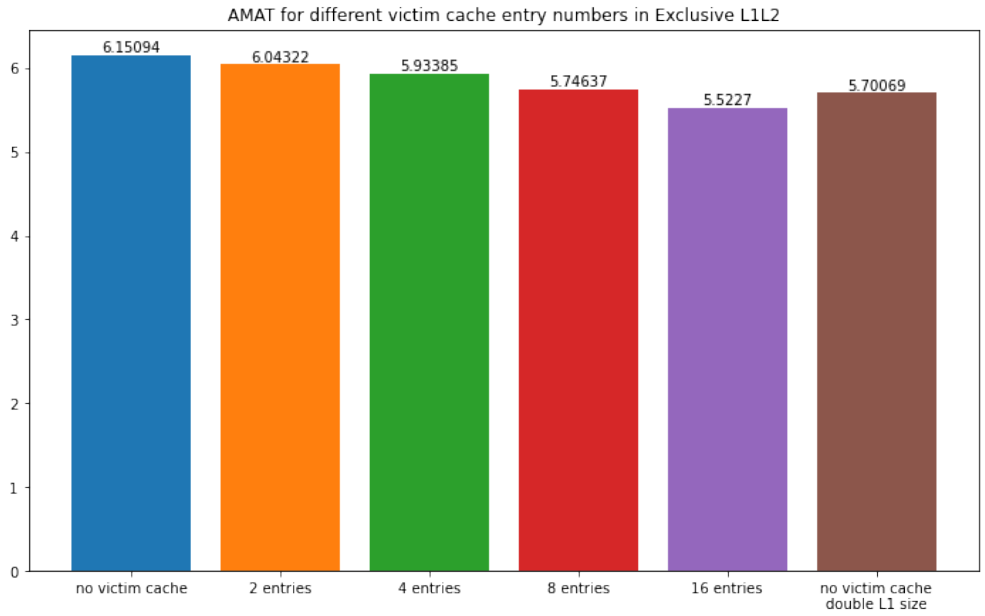
(1) Hit victim cache: Move the cache line from victim cache to L1. If there is any eviction, follow (2).

(2) Evict from L1: Send cache line to victim cache **only**. This is the only way for victim cache to get populated.

(3) Evict from Victim Cache: Send the cache line to lower-level cache/memory if it is dirty.

We made this modification since although we only need to consider L1/L2 for exclusive hierachy and it is legal to have a cache line in both victim cache and L2, a hit in of that cache line can bring it back to L1, which violates the exclusive hierachy. To fix this, we can either (1) check and invalidate a L2 cache line when a victim cache hit or (2) when evicted from L1, bypass L2 and write directly to memory. Option (1) requires a L2 look up on every victim cache hit, while (2) issue a write to memory on every L1 eviction if the evicted cache line is dirty.

However, after the modification was made, there are no such problem, as L1, victim cache and L2 would be exclusive to each other.



On this trace, adding a fully-associative victim cache of 1/8 L1 cache size achieves nearly equal performance compared to simply doubling L1 cache size.

4.Evaluation with Applications

Configuration:

Cache Line: 32 bytes

L1: 1024 bytes, 4-way-set-associative

L2: 4096 bytes, 4-way-set-associative

Victim cache: 8 entries, fully-associative

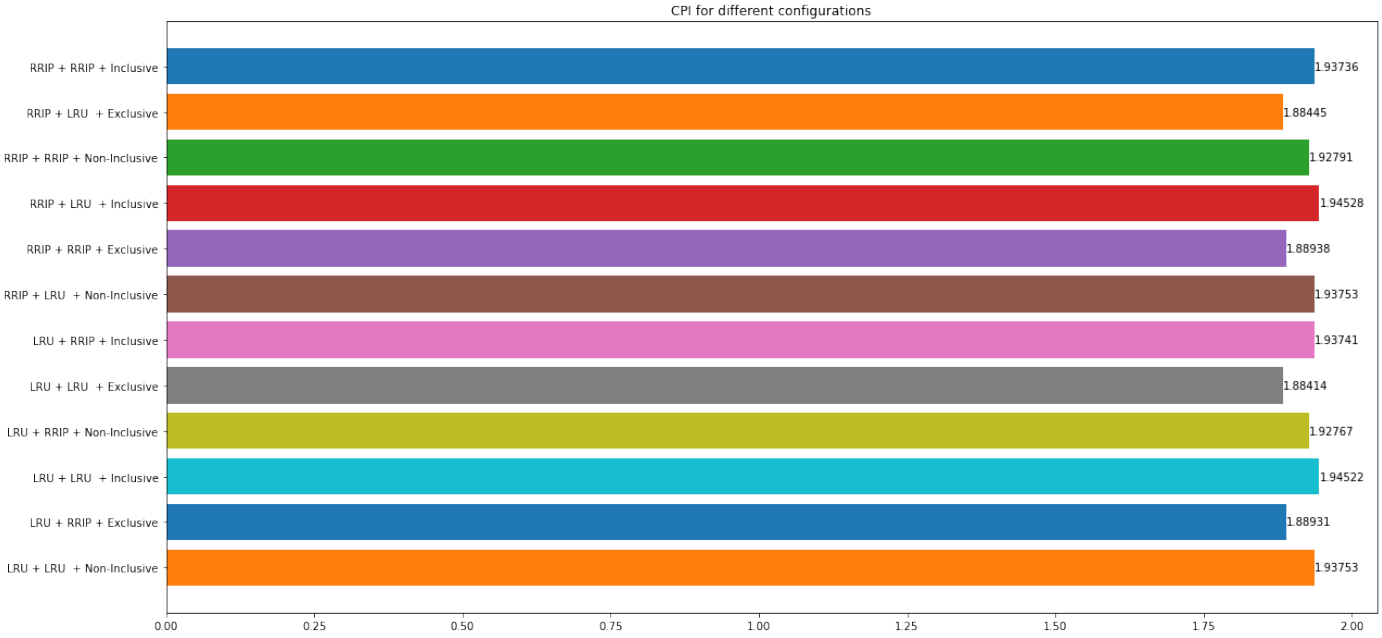
Since all variants excutes same amount of instructions, we use CPI (Cycles Per Instruction) for performance comparison.

All programs' optimization level were set to O3.

4.1 quicksort.c

code: **lab2/quicksort.c**, which implements quick sort.

Experiment Variantion	L1 Replacement Policy	L2 Replacement Policy	Inclusion Policy	CPI	AMAT
1	RRIP	RRIP	Inclusive	1.93736	2.06624
2	RRIP	LRU	Exclusive	1.88445	2.05798
3	RRIP	RRIP	Non-Inclusive	1.92791	2.06397
4	RRIP	LRU	Inclusive	1.94528	2.06692
5	RRIP	RRIP	Exclusive	1.88938	2.05858
6	RRIP	LRU	Non-Inclusive	1.93753	2.0649
7	LRU	RRIP	Inclusive	1.93741	2.0663
8	LRU	LRU	Exclusive	1.88414	2.05793
9	LRU	RRIP	Non-Inclusive	1.92767	2.06394
10	LRU	LRU	Inclusive	1.94522	2.06691
11	LRU	RRIP	Exclusive	1.88931	2.05856
12	LRU	LRU	Non-Inclusive	1.93753	2.0649



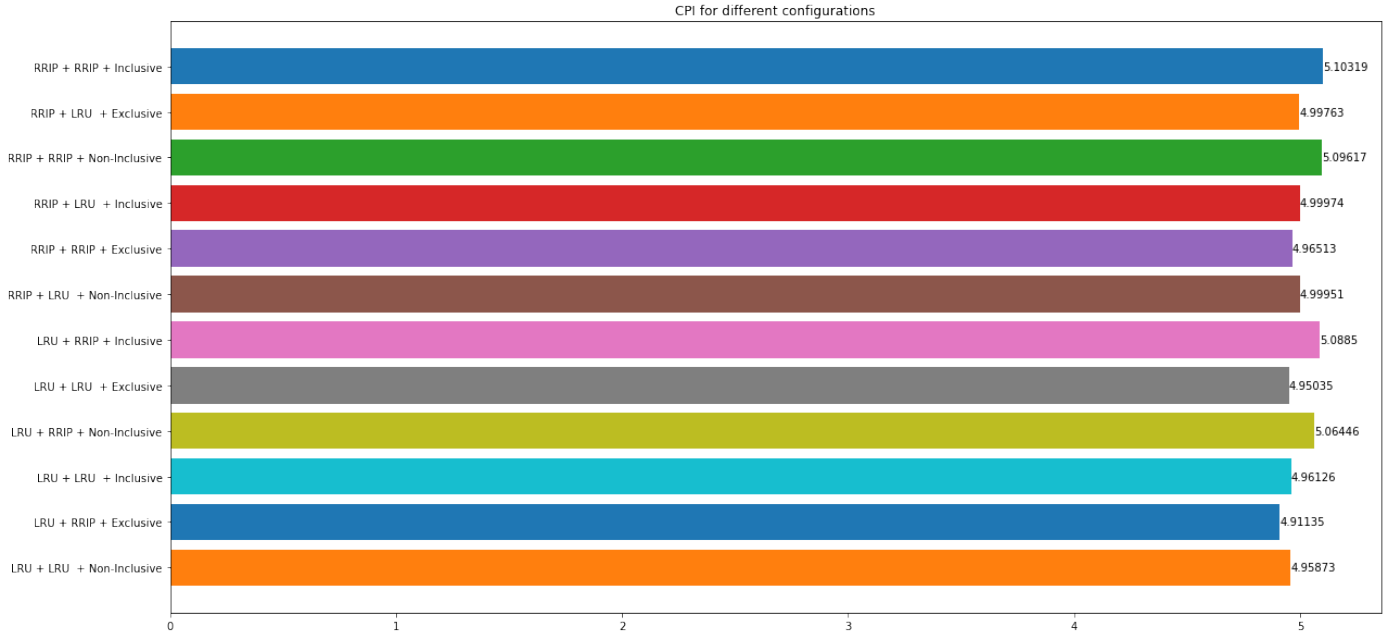
For this program, the replacement policy does not have much influence on performance.

Variations with exclusive cache have the best performance.

4.2 matrixmulti.c

code: **lab2/matrixmulti.c**, which implements matrix multiplication.

Experiment Variantion	L1 Replacement Policy	L2 Replacement Policy	Inclusion Policy	CPI	AMAT
1	RRIP	RRIP	Inclusive	5.10319	2.15962
2	RRIP	LRU	Exclusive	4.99763	2.15347
3	RRIP	RRIP	Non-Inclusive	5.09617	2.15559
4	RRIP	LRU	Inclusive	4.99974	2.15549
5	RRIP	RRIP	Exclusive	4.96513	2.15276
6	RRIP	LRU	Non-Inclusive	4.99951	2.15351
7	LRU	RRIP	Inclusive	5.0885	2.15904
8	LRU	LRU	Exclusive	4.95035	2.152
9	LRU	RRIP	Non-Inclusive	5.06446	2.15449
10	LRU	LRU	Inclusive	4.9612	2.15424
11	LRU	RRIP	Exclusive	4.91135	2.15112
12	LRU	LRU	Non-Inclusive	4.95873	2.15219



Variation of LRU (L1) + RRIP (L2) + Exclusive has the best performance.