Report-Lab2

Yunxin Yang

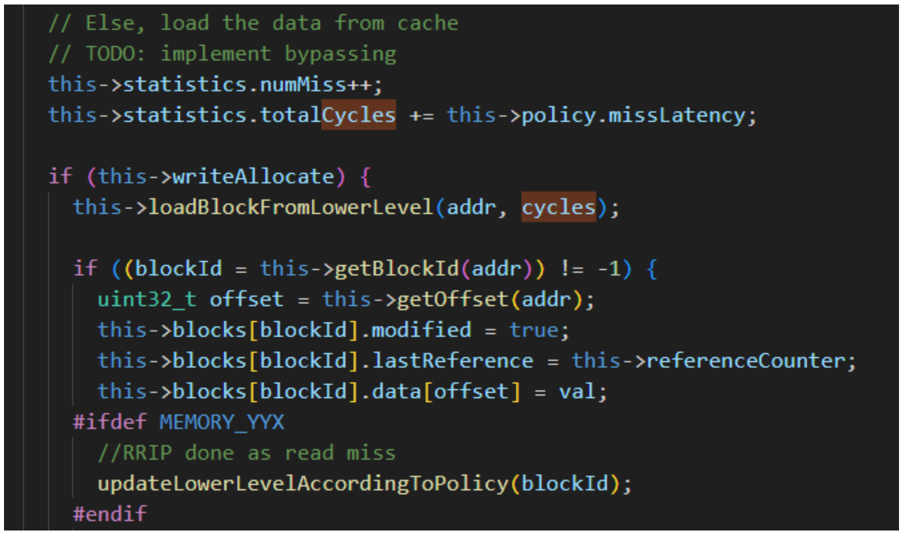
2023233204

1. Answers to the question in section 1

Inclusion policy is at this point between the multi-level caches is **NINE(not-inclusive-not-exclusive).**

2. Description of your designs and implementations;

ps: I modify a little bit to original NINE cache that when write miss on write allocate and after load cache block, the write originally will just write current level but not lower level. I add the section that after it read the cache block, it should **continue write just like cache hit,** so I let it roll according to the write hit policy after load cache on write allocate. This modification will facililate the inclusive and exclusive policy's implementation.



```
// Else, load the data from cache
// TODO: implement bypassing
this->statistics.numMiss++;
this->statistics.totalCycles += this->policy.missLatency;

if (this->writeAllocate) {
  this->loadBlockFromLowerLevel(addr, cycles);

  if ((blockId = this->getBlockId(addr)) != -1) {
    uint32_t offset = this->getOffset(addr);
    this->blocks[blockId].modified = true;
    this->blocks[blockId].lastReference = this->referenceCounter;
    this->blocks[blockId].data[offset] = val;
  #ifdef MEMORY_YYX
    //RRIP done as read miss
    updateLowerLevelAccordingToPolicy(blockId);
  #endif
```

what is added is in #ifdef MEMORY_YYX section.

`updateLowerLevelAccordingToPolicy` : used to keep write to lower level according to the write policy and inclusion policy

```
void Cache::updateLowerLevelAccordingToPolicy(int blockId) {
    if (!this->writeBack) {
      if (EXCLUSIVE == this->inclusionType){
        // just write memory
        this->writeBlockToMemory(this->blocks[blockId]);
      }else{
        this->writeBlockToLowerLevel(this->blocks[blockId]);
      }
      this->statistics.totalCycles += this->policy.missLatency;
    }else if(INCLUSIVE == this->inclusionType){
    // writeback but inclusive, need to write down until last level of cache
    // in this case, high level must have this cache line since inclusive; so will always go to this code in every level
      this->writeBlockToLowerLevelWithoutMemory(this->blocks[blockId]);
      this->statistics.totalCycles += this->policy.missLatency;
    }
}
```

1. Multi-level Caches

Inclusive

1. Read Cache hit : the same as origin
2. Read Cache miss: add `backInvalidation`
   a. when it need to evict block, I add `backInvalidation` in `loadBlockFromLowerLevel(addr, cycles);` to guarantee the inclusive policy that it need to invalidate the cache line in upper cache.

   ```
   // whenever valid be replaced, need to check
     if (replaceBlock.valid && INCLUSIVE == this->inclusionType)
       backInvalidation(this->upperCache, addr);
   ```

   b. so I firstly add upperCache in the constructor to define the upperCache
   c. details in `backInvalidation`

   just to traverse every upperCache to invalid the addr's cache block

   ```
   void Cache::backInvalidation(Cache *cache, uint32_t addr){
     if (nullptr == cache) return;
     // If in cache, return directly
     int blockId = this->getBlockId(addr);
     if (-1 == blockId)
       return;
     this->blocks[blockId].valid = false;
     // recurse
     backInvalidation(this->upperCache, addr);
   }
   ```

3. Write Cache hit

   when the policy is writeback, the cache should not write to memory but write to all level cache, so I add the logic in `updateLowerLevelAccordingToPolicy(blockId);` , which will be added at the section when write cache hit

   ```
   // If in cache, write to it directly
     int blockId;
     if ((blockId = this->getBlockId(addr)) != -1) {
   ...
       #ifdef MEMORY_YYX
       //RRIP
       this->blocks[blockId].RRIPid = 0;
       this->updateLowerLevelAccordingToPolicy(blockId);
   ...
       #endif
   ...
     }
   ```

   in `updateLowerLevelAccordingToPolicy(blockId);`

   ```
   void Cache::updateLowerLevelAccordingToPolicy(int blockId) {
       if (!this->writeBack) {
         if (EXCLUSIVE == this->inclusionType){
           // just write memory
           this->writeBlockToMemory(this->blocks[blockId]);
         }else{
           this->writeBlockToLowerLevel(this->blocks[blockId]);
         }
         this->statistics.totalCycles += this->policy.missLatency;
       }else if(INCLUSIVE == this->inclusionType){
       // writeback but inclusive, need to write down until last level of cache
       // in this case, high level must have this cache line since inclusive; so will always go to this code in every level
         this->writeBlockToLowerLevelWithoutMemory(this->blocks[blockId]);
         this->statistics.totalCycles += this->policy.missLatency;
       }
   }
   ```

when the policy is writeback and inclusive, `writeBlockToLowerLevelWithoutMemory` will be called , which only write to lower cache without memory

```
void Cache::writeBlockToLowerLevelWithoutMemory(Cache::Block &b) {
  uint32_t addrBegin = this->getAddr(b);
  if (this->lowerCache != nullptr) {
    for (uint32_t i = 0; i < b.size; ++i) {
      this->lowerCache->setByte(addrBegin + i, b.data[i]);
    }
  }
}
```

4. write cache miss

   since I add `updateLowerLevelAccordingToPolicy` in writeallocate logic, the logic will be the same as talked in above.

   other part remain unchanged.

Exclusive

1. Read Cache hit : the same as origin
2. Read Cache miss: since will load cache from lower level, I need to invalid the cache line in all the lower level. So I added below code in `Cache::loadBlockFromLowerLevel`

```
#ifdef MEMORY_YYX
  if (EXCLUSIVE == this->inclusionType && this->lowerCache){
    // invalid next level cache
    this->lowerCache->exclusiveInvalidation(addr);
  }
#endif
```

`exclusiveInvalidation`:

```
void Cache::exclusiveInvalidation(uint32_t addr){
  int blockId = this->getBlockId(addr);
  if (-1 == blockId)
    return;
  this->blocks[blockId].valid = false;
}
```

3. Write cache hit

   when it comes to write through, the cache should just keep the block in L1 and memory without any higher level cache,

```
void Cache::setByte(uint32_t addr, uint8_t val, uint32_t *cycles) {
...
// If in cache, write to it directly
  int blockId;
  if ((blockId = this->getBlockId(addr)) != -1) {
    uint32_t offset = this->getOffset(addr);
    this->statistics.numHit++;
    this->statistics.totalCycles += this->policy.hitLatency;
    this->blocks[blockId].modified = true;
    this->blocks[blockId].lastReference = this->referenceCounter;
    this->blocks[blockId].data[offset] = val;
    #ifdef MEMORY_YYX
    //RRIP
    this->blocks[blockId].RRIPid = 0;
    this->updateLowerLevelAccordingToPolicy(blockId);
....
}
```

   in `updateLowerLevelAccordingToPolicy`

```
void Cache::updateLowerLevelAccordingToPolicy(int blockId) {
    if (!this->writeBack) {
      if (EXCLUSIVE == this->inclusionType){
        // just write memory
        this->writeBlockToMemory(this->blocks[blockId]);
      }else{
        this->writeBlockToLowerLevel(this->blocks[blockId]);
      }
      this->statistics.totalCycles += this->policy.missLatency;
    }...
}
```

when it is EXCLUSIVE and write through, the write for the lower level will just for memory. `writeBlockToMemory`

```
void Cache::writeBlockToMemory(Cache::Block &b) {
  uint32_t addrBegin = this->getAddr(b);
  for (uint32_t i = 0; i < b.size; ++i) {
    this->memory->setByteNoCache(addrBegin + i, b.data[i]);
  }
}
```

2. Cache Replacement

RRIP

the policy is suit for scan operation

- structure: the Block of Cache should add `int RRIPid;` for record

```
struct Block {
...
  #ifdef MEMORY_YYX
    int RRIPid;
  #endif
...
  };
```

- when initialize , all the block's RRIPid is `(1<<policy.associativity)-1;`

```
void Cache::initCache() {
...
    b.RRIPid = (1<<policy.associativity)-1;
...
  }
}
```

- when cache hit (read and write)the block's RRIPid is to 0
  `this->blocks[blockId].RRIPid = 0;`
- when cache miss need to `loadBlockFromLowerLevel` , the insert block's RRIPid is to `(1<<(this->policy.associativity))-2`

```
void Cache::loadBlockFromLowerLevel(uint32_t addr, uint32_t *cycles) {
...
  // Initialize new block from memory
  Block b;
...
  b.RRIPid = (1<<(this->policy.associativity))-2;
```

- when pick the one to evict in `Cache::getReplacementBlockId`

1. traverse to see if there is a block's RRIPID = `(1<<(this->policy.associativity))-1`, if yes then retrun this blockid for eviction

2. otherwise, all RRIPID in the block will add 1 until condition 1 meets.

```
case RRIP:{
    // if RRIPid is (1<<associaty)-1 ,then is invalid or need to be evicted
    uint32_t max_RRIPid = this->blocks[begin].RRIPid;
    uint32_t max_index = begin;
    const uint32_t FULLRRIPid = (1<<(this->policy.associativity))-1;
    for (uint32_t i = begin; i < end; ++i) {
      if (this->blocks[i].RRIPid < 0){
        // robust
        printf("this->blocks[i].RRIPid not initialized\n");
        exit(-1);
      }
      if (FULLRRIPid == this->blocks[i].RRIPid) {
        resultId = i;
        return resultId;
      }else if (this->blocks[i].RRIPid > max_RRIPid){
        max_RRIPid = this->blocks[i].RRIPid;
        max_index = i;
      }else if (FULLRRIPid < this->blocks[i].RRIPid){
        // Robust test
        printf("FULLRRIPid < this->blocks[i].RRIPid\n");
        exit(-1);
      }
    }
    // no FULLRRIPid == this->blocks[i].RRIPid, add everyone's RRID with (FULLRRIPid - max_RRIPid)
    uint32_t add_RRIP = FULLRRIPid - max_RRIPid;
    resultId = max_index;
    for (uint32_t i = begin; i < end; ++i) {
      this->blocks[i].RRIPid += add_RRIP;
    }
    break;
  }
```

BELADY

add `getGlobalUniqueBlockIDFromAddr` to get GlobalUniqueBlockID to identify a block

```
uint32_t Cache::getGlobalUniqueBlockIDFromAddr(uint32_t addr) {
  uint32_t offsetBits = log2i(policy.blockSize);
  uint32_t idBits = log2i(policy.blockNum / policy.associativity);
  return  (this->getTag(addr) << idBits) | this->getId(addr);
}
```

思路：

1. 当输入数据时，调用 `preInputAddrGlobalBlockIDForBelady(traceFilePath);`

   用于初始化每一个addr 对应的 `GlobalUniqueBlockID` 出现的时间 `counter`

   `this->globalBlockID2Times` 为一个 `std::unordered_map<uint32_t, std::vector<int>>` 用于记录，每个globalBlockID出现的次数

   ```
   void Cache::preInputAddrGlobalBlockIDForBelady(const char * traceFilePath){
     // Read and execute trace in cache-trace/ folder
     std::ifstream trace(traceFilePath);
     if (!trace.is_open()) {
       printf("Unable to open file %s\n", traceFilePath);
       exit(-1);
     }

     char type; //'r' for read, 'w' for write
     uint32_t addr;
     uint32_t counter = 0; // time after may start at 1 the same as referenceCounter
     while (trace >> type >> std::hex >> addr) {
       counter++;
       uint32_t globalBlockID = this->getGlobalUniqueBlockIDFromAddr(addr);
       this->globalBlockID2Times[globalBlockID].push_back(counter);
     }
   }
   ```

2. when actually do the mem call `updateGlobalBlockIDCurrentTime`

   to update the addr's `GlobalUniqueBlockID` appear times, being record by `std::unordered_map<uint32_t, int> globalBlockIDCurrentOrder;`

```
 int main(int argc, char **argv) {
  ...
  char type; //'r' for read, 'w' for write
  uint32_t addr;
  while (trace >> type >> std::hex >> addr) {
#ifdef MEMORY_YYX
    // Belady
    if (BELADY == replacementType){
      l1cache->updateGlobalBlockIDCurrentTime(addr);
      l2cache->updateGlobalBlockIDCurrentTime(addr);
    }
#endif
```

```
void Cache::updateGlobalBlockIDCurrentTime(uint32_t addr){
  uint32_t globalBlockID = this->getGlobalUniqueBlockIDFromAddr(addr);
  int currentOrder = 0;
  if (!globalBlockIDCurrentOrder.count(globalBlockID)){
    // then default globalBlockIDCurrentOrder[addr] as 0
    currentOrder = 0;
  }else{
    currentOrder = globalBlockIDCurrentOrder[globalBlockID]+1;
  }
  globalBlockIDCurrentOrder[globalBlockID] = currentOrder;
  // printf("updateGlobalBlockIDCurrentTime globalBlockIDCurrentOrder[%u] = %d of addr:%0x\n", globalBlockID, currentOrder, addr);
}
```

3. when pick the one to evict in `Cache::getReplacementBlockId`

   compare which block have the longest next appear time add choose it as the victim

```
case BELADY:{
    // Find invalid block first
    for (uint32_t i = begin; i < end; ++i) {
      if (!this->blocks[i].valid){
        return i;
      }
    }
    // Otherwise use BELADY
    uint32_t nextAppearTime_t = getNextSameGlobalBlockIDTimeFromGlobalBlockID(getGlobalUniqueBlockIDFromBlock(this->blocks[begin]));
    uint32_t max_t = nextAppearTime_t;
    for (uint32_t i = begin; i < end; ++i) {
      nextAppearTime_t = getNextSameGlobalBlockIDTimeFromGlobalBlockID(getGlobalUniqueBlockIDFromBlock(this->blocks[i]));
      if (nextAppearTime_t > max_t) {
        resultId = i;
        max_t = nextAppearTime_t;
      }
    }
    break;
  }
```

```
uint32_t Cache::getNextSameGlobalBlockIDTimeFromGlobalBlockID(uint32_t globalBlockID){
  int currentOrder = 0;
  if (!globalBlockIDCurrentOrder.count(globalBlockID)){
    printf("ERROR: no possible that !globalBlockIDCurrentOrder.count(globalBlockID:%d) of addr: %0x, \
    please check if you have updateGlobalBlockIDCurrentTime() before doing mem operation\n", globalBlockID);
    exit(-1);
  }else{
    currentOrder = globalBlockIDCurrentOrder[globalBlockID];
  }
  int nextOrder = currentOrder+1;
  if (nextOrder < globalBlockID2Times[globalBlockID].size()){
    return globalBlockID2Times[globalBlockID][nextOrder];
  }else if (globalBlockID2Times[globalBlockID].size() == nextOrder){
    return MAXNEXTAPPEARTIME;
  }else{
    printf("unable to appear that globalBlockID2Times[globalBlockID].size() < nextOrder");
    exit(-1);
  }
}
```

3. Victim Cache

ps: only can use victim cache when L1 cache is direct-mapped

use `std::queue<Block> victimCache;` as the victim cache's structure

1. when `getBlockID` (function to judge if cache hit): if cache miss , add victim cache logic below
   a. traverse the victim cache, if hit then swap the block with the cache miss block in L1 cache and write the cache miss block to lower level
   b. if no hit, then continue

```
uint32_t Cache::getBlockId(uint32_t addr) {
  uint32_t tag = this->getTag(addr);
  uint32_t id = this->getId(addr);
  // printf("0x%x 0x%x 0x%x\n", addr, tag, id);
  // iterate over the given set
...
  // Victim Cache below
  // fail first
  if (this->ifUsingVictimCache){
    bool ifVictimCacheHit = 0;
    if (policy.associativity != 1){
      fprintf(stderr, "Using Victim cache but policy.associativity != 1\n");
      exit(-1);
    }
    // Full associatity
    const int victimCacheSize = victimCache.size();
    for (size_t i = 0; i < victimCacheSize; i++)
    {
      // 访问队列的前端元素
      Cache::Block frontBlock = victimCache.front();
      victimCache.pop();
      if (getGlobalUniqueBlockIDFromAddr(addr) == getGlobalUniqueBlockIDFromBlock(frontBlock)){
        // if hit then add latency, if miss, then L1 miss, use L1 miss latency is ok because it assumes L2 and victim cache concurrentl
        this->statistics.totalCycles += this->victimCacheLatency;
        // Full associatity, if hit, swap
        if (ifVictimCacheHit){
          fprintf(stderr, "ERROR: hit 2 times in VictimCache：%d\n", i);
          exit(-1);
        }
        victimCache.push(this->blocks[id]);
        writeBlockToLowerLevel(this->blocks[id]);
        this->blocks[id] = frontBlock;
        ifVictimCacheHit = 1;
        // printf("victimCache hit\n");
      }else{
        // other remain the same
        victimCache.push(frontBlock);
      }
    }
    if (victimCache.size() != victimCacheSize){
      fprintf(stderr, "ERROR: after swap, victimCache.size() != victimCacheSize\n");
      exit(-1);
    }
    if (ifVictimCacheHit){
      return id;
    }
  }

  return -1;
}
```

2. if L1 cache miss and victim miss and need to evict , add eviction to victim cache too in `Cache::loadBlockFromLowerLevel`

```
void Cache::loadBlockFromLowerLevel(uint32_t addr, uint32_t *cycles) {
...
  // victimCache
  if (replaceBlock.valid && this->ifUsingVictimCache){
    // FIFO
    if (victimCache.size() == victimCacheCapacity){
      victimCache.pop();
    }
    victimCache.push(replaceBlock);
  }
...
  this->blocks[replaceId] = b;
}
```

4. Evaluation with Application(s)

in `MainCPU.cpp` , I change the code below to suit for the variations for the policy

1. use myself made constructor for cache to designate the policy
2. use only 2 level cache

```
int main(int argc, char **argv) {
  if (!parseParameters(argc, argv)) {
    printUsage();
    exit(-1);
  }

  // Init cache
  Cache::Policy l1Policy, l2Policy, l3Policy;
#ifdef MEMORY_YYX
  l1Policy.cacheSize = 32 * 1024;
  l1Policy.blockSize = 64;
  l1Policy.blockNum = l1Policy.cacheSize / l1Policy.blockSize;
  if (ifUseVictimCache){
    l1Policy.hitLatency = 0;
    l1Policy.associativity = 1;
  }else{
    l1Policy.hitLatency = 2;
    l1Policy.associativity = 8;
  }
  l1Policy.missLatency = 8;

  l2Policy.cacheSize = 256 * 1024;
  l2Policy.blockSize = 64;
  l2Policy.blockNum = l2Policy.cacheSize / l2Policy.blockSize;
  l2Policy.associativity = 8;
  l2Policy.hitLatency = 8;
  l2Policy.missLatency = 20;

  int victimCacheCapacity = 4;
  int victimCacheLatency = 2;
  // use default write policy : both true
  l2Cache = new Cache(&memory, l2Policy, nullptr, true, true, inclusionType, nullptr, L2ReplacementType);
  l1Cache = new Cache(&memory, l1Policy, l2Cache, true, true, inclusionType, nullptr, L1ReplacementType, ifUseVictimCache, victimCacheCapac
  l2Cache->setUpperCache(l1Cache);
#else
//  origin code here
#endif
```

3. Evaluation and analysis on your results, test case(s), etc.

1. Multi-level Caches

I use the `01-mcf-gem5-xcg.trace` and `CacheOptimized (modify a little bit to suit for different policy)` to test different cache inclusion policy under LRU replacement policy under `writeback and writeallocate` policy

NINE

```
inclusion type: NINE
replacement type: LRU
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 185983
Num Miss: 46628
Total Cycles: 821310
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2984192
Num Write: 610560
Num Hit: 3572505
Num Miss: 22247
Total Cycles: 31058240
```

INCLUSIVE

```
---------------------
inclusionPolicy:1
inclusion type: INCLUSIVE
replacement type: LRU
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 185983
Num Miss: 46628
Total Cycles: 1228534
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2984192
Num Write: 3868352
Num Hit: 6830297
Num Miss: 22247
Total Cycles: 443955776
```

EXCLUSIVE

```
---------------------
inclusionPolicy:2
inclusion type: EXCLUSIVE
replacement type: LRU
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 185983
Num Miss: 46628
Total Cycles: 821310
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2984192
Num Write: 610560
Num Hit: 3547400
Num Miss: 47352
Total Cycles: 33114400
```

We can see that the NINE policy performs the best since it dont need more restraint for cache coherence which will entail more write.

EXCLUSIVE outperforms INCLUSIVE as INCLUSIVE will write to all level each time when writeback while EXCLUSIVE may not.

NINE out performs EXCLUSIVE since EXCLUSIVE need to invalidate block in lower level when read from lower level

2. Cache Replacement

I use the `01-mcf-gem5-xcg.trace` and `CacheOptimized (modify a little bit to suit for different policy)` to test different cache inclusion policy under NINE inclusion policy under `writeback and writeallocate` policy

LRU

```
---------------------
inclusionPolicy:0
inclusion type: NINE
replacement type: LRU
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 185983
Num Miss: 46628
Total Cycles: 821310
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2984192
Num Write: 610560
Num Hit: 3572505
Num Miss: 22247
Total Cycles: 31058240
```

RRIP

```
---------------------
inclusionPolicy:0
inclusion type: NINE
replacement type: RRIP
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 190058
Num Miss: 42553
Total Cycles: 779060
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2723392
Num Write: 468160
Num Hit: 3169286
Num Miss: 22266
Total Cycles: 27856588
```

BELADY

```
---------------------
inclusionPolicy:0
inclusion type: NINE
replacement type: BELADY
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 198288
Num Miss: 34323
Total Cycles: 715920
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 2196672
Num Write: 358080
Num Hit: 2540634
Num Miss: 14118
Total Cycles: 21914672
```

We can see that BELADY outperforms most since it is the optimized policy.

RRIP outperforms LRU since it is more useful for scan.

3. Victim Cache

I use the `01-mcf-gem5-xcg.trace` and `CacheOptimized (modify a little bit to suit for different policy)` to test different cache inclusion policy under NINE inclusion policy under LRU under `writeback and writeallocate` policy.

in this case , the victim cache latency set as the same as L1hitlatency

with victim cache

```
---------------------
inclusionPolicy:0
argc:5
You designate a fourth parameter, meaning you are deciding if you are using victim cache
you are using the version with victim cache
inclusion type: NINE
replacement type: LRU
Using victim cache
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 184760
Num Miss: 47851
Total Cycles: 837414
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 3062464
Num Write: 691776
Num Hit: 3732022
Num Miss: 22218
Total Cycles: 32333776
```

without victim cache but with L1 still direct-mapped

```
---------------------
inclusionPolicy:0
argc:5
You designate a fourth parameter, meaning you are deciding if you are using victim cache
you are using the version without victim cache for comparison
inclusion type: NINE
replacement type: LRU
L1 Cache:
-------- STATISTICS ----------
Num Read: 181708
Num Write: 50903
Num Hit: 184529
Num Miss: 48082
Total Cycles: 838474
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 3077248
Num Write: 678080
Num Hit: 3733114
Num Miss: 22214
Total Cycles: 32336212
```

we can see that victim cache do accelerate the performance but just a little `Total Cycles: 837414` < `Total Cycles: 838474` (L1), `Total Cycles: 32333776` < `Total Cycles: 32336212` (L2)

4. Evaluation with Application(s)

the parameters for cache is as below

```
l1Policy.cacheSize = 32 * 1024;
  l1Policy.blockSize = 64;
  l1Policy.blockNum = l1Policy.cacheSize / l1Policy.blockSize;
  if (ifUseVictimCache){
    l1Policy.hitLatency = 0;
    l1Policy.associativity = 1;
  }else{
    l1Policy.hitLatency = 2;
    l1Policy.associativity = 8;
  }
  l1Policy.missLatency = 8;

  l2Policy.cacheSize = 256 * 1024;
  l2Policy.blockSize = 64;
  l2Policy.blockNum = l2Policy.cacheSize / l2Policy.blockSize;
  l2Policy.associativity = 8;
  l2Policy.hitLatency = 8;
  l2Policy.missLatency = 20;

  int victimCacheCapacity = 4;
  int victimCacheLatency = 2;
```

I use 24 variations to test

the basic 12 variations is below

ExperimentVariation	L1 ReplacementPolicy	L2 ReplacementPolicy	InclusionPolicy
1	RRIP	RRIP	Inclusive
2	RRIP	LRU	Inclusive
3	RRIP	RRIP	Exclusive
4	RRIP	LRU	Exclusive
5	RRIP	RRIP	Non-Inclusive
6	RRIP	LRU	Non-Inclusive
7	LRU	RRIP	Inclusive
8	LRU	LRU	Inclusive
9	LRU	RRIP	Exclusive
10	LRU	LRU	Exclusive
11	LRU	RRIP	Non-Inclusive
12	LRU	LRU	Non-Inclusive

with victim cache and without make the 12 variations double to 24.

When with victim cache, the L1 is set to direct-mapped.

I run the test `matrixmulti.riscv` and `quicksort.riscv` given in the original folder

in `matrixmulti.riscv` :

all the 12 variations with victim cache:

```
------------ STATISTICS -----------
Number of Instructions: 225452
Number of Cycles: 322241
Avg Cycles per Instrcution: 1.4293
Branch Perdiction Accuacy: 0.3765 (Strategy: Always Not Taken)
Number of Control Hazards: 40680
Number of Data Hazards: 116740
Number of Memory Hazards: 11735
-----------------------------------
```

all the 12 variations without victim cache:

```
------------ STATISTICS -----------
Number of Instructions: 225452
Number of Cycles: 344963
```

```
Avg Cycles per Instrcution: 1.5301
Branch Perdiction Accuacy: 0.3765 (Strategy: Always Not Taken)
Number of Control Hazards: 40680
Number of Data Hazards: 113697
Number of Memory Hazards: 11735
-----------------------------------
```

We can see that the victim cache did make the difference. The reason why the choice like replacement type and inclusion policy don’t make the difference is perhaps that the memory operation in this application is not enough for cache.

in `matrixmulti.riscv` :

all the 12 variations with victim cache:

```
------------ STATISTICS -----------
Number of Instructions: 103682
Number of Cycles: 144555
Avg Cycles per Instrcution: 1.3942
Branch Perdiction Accuacy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7316
Number of Data Hazards: 93856
Number of Memory Hazards: 23398
-----------------------------------
```

all the 12 variations without victim cache:

```
------------ STATISTICS -----------
Number of Instructions: 103682
Number of Cycles: 186164
Avg Cycles per Instrcution: 1.7955
Branch Perdiction Accuacy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7316
Number of Data Hazards: 81947
Number of Memory Hazards: 23398
-----------------------------------
```

We can see that the victim cache did make the difference. The reason why the choice like replacement type and inclusion policy don’t make the difference is perhaps that the memory operation in this application is not enough for cache.