Advanced Computer Architecture: Lab 1 Report

Due on Octobor 18th at 11:59am

Name: **Zhenghong Yu**
Mailbox: **yuzhh1@shanghaitech.edu.cn**
Student ID: 2020533156

Contents

0.1 Preknowledge

The task1 is implemented in the branch **Task1**, the task2 is implemented in the branch **Task2**.

1 Task1

1.1 Requirement of Exclusive Cache

Data only has one copy in the cache, different level of cache do not have intersection data.

1.2 Analysis of Inclusive Cache Implementation

The simulatior has already provided a inclusive cache implementation. The author use nested recursion to realize it. **Cache::getByte()** is used to load the data of the current level cache, if success, read and return the value directly, if not, enter **Cache::loadBlockFromLowerLevel()** to get the target block from a lower cache, find a victim block and replaced it. **Cache::setByte()** first confirm whether the writing data is in the current level cache, if yes, modified it and return. If not, first use **Cache::loadBlockFromLowerLevel()** to load the data to the L1 cache, then modified it.
It is a complex implementation since the author tried to realize many functions in a same function interface, which means a smaller change might cause the whole program crash, and actually it is a very dirty work to follow his structure to implement a exclusive cache. So I'll state some new functions while reuse as much his codes as possible.

1.3 Implementation of Exclusive Cache

The most important question of a exclusive cache is how to keep the data only one copy in all level of cache. And in my opinion,
First, each time we read a data from memory, we only read it to the L1 cache.
Second, each time we read a data from L2/L3 cache to L1 cache, we will invalid the old data of the L2/L3 cache.
Since each time we will replaced the block when we want to evited a data, the exclusive property is still kept, we don not need to worry about that. Here is a basic thought:

- Read a byte.
  - If L1 hit, just read
  - If L1 miss, read a byte from the L2 cache.
    - If L2 hit, return it to L1 cache and invalid the cache line
    - If L2 miss, read a byte from the L3 cache and directly return
      - If L3 hit, return it to L1 cache and invalid the cache line
      - If L3 miss, read a byte from memory and return directly
- L1 cache evict a block
  find a replacement block and write it to L2 cache
  - If L2 cache evict a block
  - find a replacement block and write it to L3 cache
    - If L3 cache evict a block
    - Write it to memory if it is modified
- Write a byte

  - If L1 hit
    * If write through, write it to block and memory
    * If write back, write it to block and mark it as modified
  - If L1 miss, load the block from lower cache
    * If write through, write it to block and memory
    * If write back, write it to block and mark it as modified

Following is my code implementation, to give it a brief look, I'll write a small mind map before listing all the code.

- **Cache::getByte** in **Cache.cpp**
  - If hit, read the byte and return
  - If miss, call **loadBlockFromLowerLevelExclusive** to get the data to the L1 cache
  - Find a block to be replaced
  - Evict it and call **writeBlockToLowerLevelExclusive** if it is valid
  - Read the byte and return

```
uint8_t Cache::getByte(uint32_t addr, uint32_t *cycles)
{
  this->referenceCounter++;      /* add reference */
  this->statistics.numRead++;    /* add read num */

  /* If in cache, read 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].lastReference = this->referenceCounter;
    if (cycles)
      *cycles = this->policy.hitLatency;
    return this->blocks[blockId].data[offset];
  }

  /* If miss, read it from lower cache */
  this->statistics.numMiss++;    /* add miss number */
  /* add miss lantency */
  this->statistics.totalCycles += this->policy.missLatency;

  /* Set the vector to store the data read from lower cache */
  std::vector<uint8_t> tmp(this->policy.blockSize);
  this->loadBlockFromLowerLevelExclusive(addr, tmp, cycles);
  /* After this step, the data should be in the L1 level */

  /* Get block to be replaced */
  Block b;
```

```
  b.valid = true;
  b.modified = false;
  b.tag = this->getTag(addr);
  b.id = this->getId(addr);
  b.size = this->policy.blockSize;
  b.data = std::vector<uint8_t>(b.size);
  b.data = tmp;

  /* Find a replaced block */
  uint32_t id = this->getId(addr);
  uint32_t blockIdBegin = id * this->policy.associativity;
  uint32_t blockIdEnd = (id + 1) * this->policy.associativity;
  uint32_t replaceId = this->getReplacementBlockId(blockIdBegin, blockIdEnd);
  Block replaceBlock = this->blocks[replaceId];
  if (this->writeBack && replaceBlock.valid && replaceBlock.modified)
  { /* Write it back to memory */
    this->writeBlockToLowerLevelExclusive(addr, replaceBlock);
    this->statistics.totalCycles += this->policy.missLatency;
  }
  this->blocks[replaceId] = b;

  /* The block is in top level cache now, return directly */
  if ((blockId = this->getBlockId(addr)) != -1)
  {
    uint32_t offset = this->getOffset(addr);
    this->blocks[blockId].lastReference = this->referenceCounter;
    return this->blocks[blockId].data[offset];
  }
  else
  {
    fprintf(stderr, "Error: data not in top level cache!\n");
    exit(-1);
  }
}
```

- **Cache::setByte** in **Cache.cpp**
  - If hit, write the data
    if write through, write to memory
  - If miss, call **loadBlockFromLowerLevelExclusive** to get the data to the L1 cache
  - Find a block to be replaced
  - Evict it and call **writeBlockToLowerLevelExclusive** if it is valid
  - Write the byte, write to memory if it is writethrough

```
void Cache::setByte(uint32_t addr, uint8_t val, uint32_t *cycles)
{
  this->referenceCounter++;      /* add reference */
  this->statistics.numWrite++;   /* add write num */
```

```

/* 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;
  if (!this->writeBack)
  {   /* If write through, write to memory */
    this->memory->setByteNoCache(addr, val);
    this->statistics.totalCycles += this->policy.missLatency;
  }
  if (cycles)
    *cycles = this->policy.hitLatency;
  return;
}

/* Else, load the data from cache */
this->statistics.numMiss++;
this->statistics.totalCycles += this->policy.missLatency;

if (this->writeAllocate)
{
  /* If miss, call Exclusive function to get the data */
  std::vector<uint8_t> tmp(this->policy.blockSize);
  this->loadBlockFromLowerLevelExclusive(addr, tmp, cycles);
  /* Now we have the data at L1 */
  /* Get block to be replaced */
  Block b;
  b.valid = true;
  b.modified = false;
  b.tag = this->getTag(addr);
  b.id = this->getId(addr);
  b.size = this->policy.blockSize;
  b.data = std::vector<uint8_t>(b.size);
  b.data = tmp;

  /* Find replace block */
  uint32_t id = this->getId(addr);
  uint32_t blockIdBegin = id * this->policy.associativity;
  uint32_t blockIdEnd = (id + 1) * this->policy.associativity;
  uint32_t replaceId = this->getReplacementBlockId(blockIdBegin, blockIdEnd);
  Block replaceBlock = this->blocks[replaceId];
  if (this->writeBack && replaceBlock.valid && replaceBlock.modified)
  { /* write back to memory */
```

```
      this->writeBlockToLowerLevelExclusive(addr, replaceBlock);
      this->statistics.totalCycles += this->policy.missLatency;
    }

    this->blocks[replaceId] = b;

    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;
      if (!this->writeBack)
      {
        this->memory->setByteNoCache(addr, val);
        this->statistics.totalCycles += this->policy.missLatency;
      }
      return;
    }
    else
    {
      fprintf(stderr, "Error: data not in top level cache!\n");
      exit(-1);
    }
  }
  else
  {
    if (this->lowerCache == nullptr)
    {
      this->memory->setByteNoCache(addr, val);
    }
    else
    {
      this->lowerCache->setByte(addr, val);
    }
  }
}
```

- **Cache::loadBlockFromLowerLevelExclusive** in **Cache.cpp**
  - If the cache hit, return the data, mark it invalid
  - If miss, call **loadBlockFromLowerLevelExclusive** to get the data from a lower cache and directly return it

```
void Cache::loadBlockFromLowerLevelExclusive(uint32_t addr,
std::vector<uint8_t> &a, uint32_t *cycles = nullptr)
{
  if (this->lowerCache == nullptr)
  { /* Read directly from memory */
```

```
    uint32_t bits = this->log2i(this->policy.blockSize);
    uint32_t mask = ~((1 << bits) - 1);
    uint32_t blockAddrBegin = addr & mask;
    for (uint32_t i = blockAddrBegin; i < blockAddrBegin
    + this->policy.blockSize; ++i)
    {
      a[i - blockAddrBegin] = this->memory->getByteNoCache(i);
    }
    if (cycles)
      *cycles = 100;
  }
  else
  { /* Read from low level cache */
    this->lowerCache->statistics.numRead++; /* add read num */
    int blockId;
    if ((blockId = this->lowerCache->getBlockId(addr)) != -1)
    {   /* If hit, return data directly */
      uint32_t offset = this->lowerCache->getOffset(addr);
      this->lowerCache->statistics.numHit++;
      this->lowerCache->statistics.totalCycles
      += this->lowerCache->policy.hitLatency;
      this->lowerCache->blocks[blockId].lastReference =
      this->lowerCache->referenceCounter;
      if (cycles)
        *cycles = this->lowerCache->policy.hitLatency;
      a = this->lowerCache->blocks[blockId].data;
      this->lowerCache->blocks[blockId].valid = false;
    }
    else
    {   /* If miss, find in lowercahce */
      this->lowerCache->statistics.numMiss++;
      this->lowerCache->statistics.totalCycles
      += this->lowerCache->policy.missLatency;
      this->lowerCache->loadBlockFromLowerLevelExclusive(addr, a, cycles);
    }
  }
}
```

- **Cache::writeBlockToLowerLevelExclusive** in **Cache.cpp**
  - Find a victim cache block
  - If not full, directly write
  - Else, call **writeBlockToLowerLevelExclusive**

```
void Cache::writeBlockToLowerLevelExclusive(uint32_t addr, Block &b)
{
  if (this->lowerCache == nullptr)
  { /* If it is in memory */
    uint32_t addrBegin = this->getAddr(b);
```

```
    for (uint32_t i = 0; i < b.size; ++i)
    {
      this->memory->setByteNoCache(addrBegin + i, b.data[i]);
    }
  }
  else
  {
    int blockId;
    if ((blockId = this->lowerCache->getBlockId(addr)) != -1)
    {
      fprintf(stderr, "================error================");
    }
    Block tmp;
    tmp.valid = b.valid;
    tmp.modified = b.modified;
    tmp.tag = this->lowerCache->getTag(addr);
    tmp.id = this->lowerCache->getId(addr);
    tmp.size = this->lowerCache->policy.blockSize;
    tmp.data = std::vector<uint8_t>(tmp.size);
    tmp.data = b.data;

    uint32_t id = this->lowerCache->getId(addr);
    uint32_t blockIdBegin = id * this->lowerCache->policy.associativity;
    uint32_t blockIdEnd = (id + 1) * this->lowerCache->policy.associativity;
    uint32_t replaceId =
    this->lowerCache->getReplacementBlockId(blockIdBegin, blockIdEnd);
    Block replaceBlock = this->lowerCache->blocks[replaceId];
    if (this->writeBack && replaceBlock.valid &&
        replaceBlock.modified)
    { /* write back to memory */
      this->lowerCache->writeBlockToLowerLevelExclusive(addr, replaceBlock);
      this->lowerCache->statistics.totalCycles +=
      this->lowerCache->policy.missLatency;
    }

    this->lowerCache->blocks[replaceId] = tmp;
  }
}
```

1.4 Different Write Policy

Since the author of the simulatior has already realize two write policy, write back and write through, which is decided at the initialization of **Cache::Cache**. If it is write back, give the parameter of the write back to be **true**, if it is write through, give the parameter of the write through to be **false**.
Here is the declaretion in **MainCPU.cpp** which initialize the cache.

```
l3Cache = new Cache(&memory, l3Policy, nullptr, 0);
l2Cache = new Cache(&memory, l2Policy, l3Cache, 0);
l1Cache = new Cache(&memory, l1Policy, l2Cache, 0);
```

The implementation of write back/through is already showed before, here is a brief introduction, each time when the cpu want to write a data, it will try to find it in the cache, if hit, the write back will modified it, set it as modified and write back it to memory at other time, the write through will both modified it and it copy in the memory. If miss, cpu will get the data from memory to the L1 and do the same thing.

1.5 Correctness

1.5.1 Program Result Correctness

This section I prove that this implementation of exclusive cache will not cause program mistake, take **quicksort.c** in test file as the example

Write back:

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 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: 109340
Number of Cycles: 162364
Avg Cycles per Instrcution: 1.4849
Branch Perdiction Accuacy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7315
Number of Data Hazards: 97777
Number of Memory Hazards: 29293
----------------------------------
```

Write through:

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 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: 109340
Number of Cycles: 162364
Avg Cycles per Instrcution: 1.4849
Branch Perdiction Accuacy: 0.4925 (Strategy: Always Not Taken)
Number of Control Hazards: 7315
Number of Data Hazards: 97777
Number of Memory Hazards: 29293
----------------------------------
```

As showed before, the implementation is correct.

1.5.2 Cache Correctness

To check whether the exclusive cache is implemented correctly, I add some code when there is a evit and replace happens.

```
Cache *tmpcache = this->lowerCache;
while (tmpcache != nullptr)
{
    if ((blockId = tmpcache->getBlockId(addr)) != -1)
    {
        fprintf(stderr, "================error================");
    }
    tmpcache = tmpcache->lowerCache;
}
```

As follows, I will check whether all other level of cache has the same copy of this block, if yes it will output error. To get a large test, I use **/cache-trace/1.trace** which has almost 10000 write and read opertion to test it. And here is the terminal output.

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./CacheSim ../cache-trace/1.trace
-------- STATISTICS ----------
Num Read: 2743
Num Write: 2720
Num Hit: 0
Num Miss: 5463
Total Cycles: 64496
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 5463
Num Write: 0
Num Hit: 2
Num Miss: 5461
Total Cycles: 127096
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 5461
Num Write: 0
Num Hit: 0
Num Miss: 5461
Total Cycles: 546100
-------- STATISTICS ----------
Num Read: 5463
Num Write: 0
Num Hit: 2
Num Miss: 5461
Total Cycles: 127096
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 5461
Num Write: 0
Num Hit: 0
Num Miss: 5461
Total Cycles: 546100
-------- STATISTICS ----------
Num Read: 5461
Num Write: 0
Num Hit: 0
Num Miss: 5461
Total Cycles: 546100
Result has been written to ../cache-trace/1.trace.csv
```

We can see there is no error ouput, which proves that my cache implementation is correct.

1.6 Efficacy

In this section, I propose a question of the original implementation of inclusive cache, the author read and write byte each time when visit cache, which will cause a situation, that if I want to visit an int data, the first byte of data will miss and the cache will get the block from the memory, then the remaining three bytes will get cache hit. It is odd because we should treat a cache line as a whole instead of as many bytes. The directly result is as follows,

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./CacheSim ../cache-trace/1.trace
-------- STATISTICS ----------
Num Read: 6
Num Write: 4
Num Hit: 0
Num Miss: 10
Total Cycles: 80
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 640
Num Write: 0
Num Hit: 630
Num Miss: 10
Total Cycles: 5240
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 640
Num Write: 0
Num Hit: 630
Num Miss: 10
Total Cycles: 13600
-------- STATISTICS ----------
Num Read: 640
Num Write: 0
Num Hit: 630
Num Miss: 10
Total Cycles: 5240
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 640
Num Write: 0
Num Hit: 630
Num Miss: 10
Total Cycles: 13600
-------- STATISTICS ----------
Num Read: 640
Num Write: 0
Num Hit: 630
Num Miss: 10
Total Cycles: 13600
Result has been written to ../cache-trace/1.trace.csv
```

It is a kind of ridiculousness. As I only put 10 read/write in **1.trace**, there is nearly 630 hit in lower cache! This is my implementation as I treat a whole cache line as a whole, the result is,

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./CacheSim ../cache-trace/1.trace
-------- STATISTICS ----------
Num Read: 6
Num Write: 4
Num Hit: 0
Num Miss: 10
Total Cycles: 80
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 10
Num Write: 0
Num Hit: 0
Num Miss: 10
Total Cycles: 200
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 10
Num Write: 0
Num Hit: 0
Num Miss: 10
Total Cycles: 1000
-------- STATISTICS ----------
Num Read: 10
Num Write: 0
Num Hit: 0
Num Miss: 10
Total Cycles: 200
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 10
Num Write: 0
Num Hit: 0
Num Miss: 10
Total Cycles: 1000
-------- STATISTICS ----------
Num Read: 10
Num Write: 0
Num Hit: 0
Num Miss: 10
Total Cycles: 1000
Result has been written to ../cache-trace/1.trace.csv
```

which the all cache hit in author's implementation is a cache miss at my implementation. In a word, due to the wrong implementation of the author's statistics of cache access, compare efficacy with his result is a meaningless thing.

2 Task2

2.1 SDBP

We get a **sampler tag array**, **predictor table** to help realize **SBDP**.
The sampler tag array is interested in, for example, 32 cache sets in LLC, has 12 entries. Each entry has 15-bit partial tags, 15-bit partial PCs, one prediction bit, one valid bit, and four bits to maintain LRU position information.
The predictoe table has three arrays to trace confidence, each is indexed by a hash value of blocks' signature.
Each time when we have a access to the LLC(a L2 cache miss)

- If the block is not sampled, just do what it should do.
- If the block is sampled, take it's tag and the pc to the sampler
  - If has already stored, add the pc to the signature, get a truncated sum
  - If not, use LRU to replace a block, set the signature and the tag

At the same time,

- Take the tag and the PC to the predictor tables and get a confidence, compare with the threshold.
  - If larger, the block is dead and will firstly replaced at next replacement.
  - If not, nothing happens.

Each time when we evit a block from L3 cache, which means this block is dead,

- If the block is not sampled, just do what it should do.
- If the block is sampled, take it's tag and the pc to the sampler
  - If has already stored, hash it's signature to the predictor tables, add the counter by one
  - If not, just do what it should do

2.2 Implementation

The following is new declared variables or functions to help realize **SDBP**.

```
struct SamplerEntry
{
  uint32_t tag;      /* Partial tag */
  uint64_t trace;    /* Partial Signature */
  bool valid;        /* Valid bit */
  uint32_t used;     /* Refence to LRU*/
} samplerEntry;

std::vector<SamplerEntry> sampler(12);  /* Entries */
uint32_t lastRefence = 0;               /* Global refence */

/* This function is used to use LRU in entries array */
uint32_t getReplacementEntry(void)
{
  for (int i = 0; i < 12; i++)
  {
    if (sampler[i].valid = false)
    {
      return i;
    }
  }

  uint32_t min = sampler[0].used;
  uint8_t num = 0;
  for (int i = 0; i < 12; i++)
  {
    if (sampler[i].used < min)
    {
      min = sampler[i].used;
      num = i;
    }
  }
  return num;
}

/* This function is used to check whether a block
    has already stored in entries table */
int32_t getEntryId(uint32_t tag)
{
  for (int i = 0; i < 12; i++)
  {
    if (sampler[i].tag == tag)
    {
      return i;
    }
  }
```

```
  return -1;
}

/* Entries to get confidence by hash from three table */
int32_t getConfidence(uint64_t pc, uint32_t tag)
{
  uint8_t conf1 = sample_table1[pc + tag];
  uint8_t conf2 = sample_table2[pc & tag];
  uint8_t conf3 = sample_table3[pc ^ tag];
  return (conf1 + conf2 + conf3);
}
```

The following are functions which changed.
Cache::Cache add new initialization.

```
Cache::Cache(MemoryManager *manager, Policy policy, Cache *lowerCache,
    bool writeBack, bool writeAllocate)
{
  ...
  if (this->lowerCache == nullptr)
  {
    for (int i = 0; i < 12; i++)
    {
      sampler[i].valid = false;
      sampler[i].trace = 0;
      sampler[i].tag = 0;
    }
  }
  ...
}
```

Cache::getByte

```
uint8_t Cache::getByte(uint32_t addr, uint32_t *cycles, uint64_t pc)
{
    ...
    if ((blockId = this->getBlockId(addr)) != -1)
    {
        ...
        if (this->lowerCache == nullptr)
        {   /* If it is LLC */
          int32_t tag = getTag(addr);
          uint32_t setindex = this->getId(addr);
          if ((setindex % 128) == 0)
          {  /* If it is sampled */
            int32_t EntryId;
            if ((EntryId = getEntryId(tag)) != -1)
            {   /* If in the sampler array */
              lastRefence++;
              sampler[EntryId].trace += pc;
              sampler[EntryId].used = lastRefence;
```

```
            }
            else
            {   /* If not in the sampler array */
              lastRefence++;
              EntryId = getReplacementEntry();
              sampler[EntryId].trace = pc;
              sampler[EntryId].valid = true;
              sampler[EntryId].tag = tag;
              sampler[EntryId].used = lastRefence;
            }
          }

          uint8_t confidence = getConfidence(pc, getTag(addr));
          if (confidence >= 8)
          { /* If larger than the threshold, dead */
            this->blocks[blockId].dead = true;
            // printf("A block is dead.\n");
          }
        }
        ...
    }
}
```

Cache::setByte

```
void Cache::setByte(uint32_t addr, uint8_t val, uint32_t *cycles, uint64_t pc)
{
    ...
    if ((blockId = this->getBlockId(addr)) != -1)
    {
        ...
        if (this->lowerCache == nullptr)
        {   /* If it is LLC */
          int32_t tag = getTag(addr);
          uint32_t setindex = this->getId(addr);
          if ((setindex % 128) == 0)
          {  /* If it is sampled */
            int32_t EntryId;
            if ((EntryId = getEntryId(tag)) != -1)
            {   /* If in the sampler array */
              lastRefence++;
              sampler[EntryId].trace += pc;
              sampler[EntryId].used = lastRefence;
            }
            else
            {   /* If not in the sampler array */
              lastRefence++;
              EntryId = getReplacementEntry();
              sampler[EntryId].trace = pc;
              sampler[EntryId].valid = true;
```

```
                sampler[EntryId].tag = tag;
                sampler[EntryId].used = lastRefence;
              }
            }

            uint8_t confidence = getConfidence(pc, getTag(addr));
            if (confidence >= 8)
            { /* If larger than the threshold, dead */
              this->blocks[blockId].dead = true;
              // printf("A block is dead.\n");
            }
          }
          ...
      }
}
```

We update the replacement policy, since only the last level cache has dead blocks, there is no meaning to specified LLC. If there is dead block, use dead block, otherwise, still LRU policy.

Cache::getReplaceBlockId

```
uint32_t Cache::getReplacementBlockId(uint32_t begin,
    uint32_t end, uint64_t pc)
{
    /* First check invalid and dead */
    for (uint32_t i = begin; i < end; ++i)
    {
        if (!this->blocks[i].valid)
            return i;
        if (this->blocks[i].dead)
        {
            return i;
        }
    }

    /* Otherwise use LRU */
    uint32_t resultId = begin;
    uint32_t min = this->blocks[begin].lastReference;
    for (uint32_t i = begin; i < end; ++i)
    {
        if (this->blocks[i].lastReference < min)
        {
          resultId = i;
          min = this->blocks[i].lastReference;
        }
    }
    return resultId;
}
```

2.3 Correctness

Since the **SDBP** only change the replacement policy, I just use **quicksort.c** to test it's correctness.

```
ubuntu@VM-4-7-ubuntu:~/RISCV-Simulator/build$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 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
```

2.4 Efficacy and Comparation

2.4.1 quicksort.c

SDBP

```
Program exit from an exit() system call
---------- CACHE STATISTICS ----------
-------- STATISTICS ----------
Num Read: 854384
Num Write: 4231660
Num Hit: 4951959
Num Miss: 134085
Total Cycles: 2125400
Hit rate: 36.931492
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4290720
Num Write: 4210880
Num Hit: 8370216
Num Miss: 131384
Total Cycles: 72213128
Hit rate: 63.708031
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4204288
Num Write: 4197952
Num Hit: 8270931
Num Miss: 131309
Total Cycles: 184921920
Hit rate: 62.988304
------------ STATISTICS ------------
Number of Instructions: 103670
Number of Cycles: 151185
Avg Cycles per Instrcution: 1.4583
Branch Perdiction Accuacy: 0.4926 (Strategy: Always Not Taken)
Number of Control Hazards: 7314
Number of Data Hazards: 86448
Number of Memory Hazards: 23398
-----------------------------------
```

LRU

```
Program exit from an exit() system call
---------- CACHE STATISTICS ----------
-------- STATISTICS ----------
Num Read: 854384
Num Write: 4231660
Num Hit: 4954727
Num Miss: 131317
Total Cycles: 2099008
Hit rate: 37.731041
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4202144
Num Write: 4193888
Num Hit: 8264715
Num Miss: 131317
Total Cycles: 71329440
Hit rate: 62.937130
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4202144
Num Write: 4136608
Num Hit: 8207435
Num Miss: 131317
Total Cycles: 183858500
Hit rate: 62.500935
------------ STATISTICS ------------
Number of Instructions: 103670
Number of Cycles: 143089
Avg Cycles per Instrcution: 1.3802
Branch Perdiction Accuacy: 0.4926 (Strategy: Always Not Taken)
Number of Control Hazards: 7314
Number of Data Hazards: 86448
Number of Memory Hazards: 23398
-----------------------------------
```

2.4.2 ackermann.c

SDBP

```
---------- CACHE STATISTICS ----------
-------- STATISTICS ----------
Num Read: 3034928
Num Write: 4631364
Num Hit: 7516144
Num Miss: 150148
Total Cycles: 2395976
Hit rate: 50.058235
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4804736
Num Write: 4779168
Num Hit: 9451194
Num Miss: 132710
Total Cycles: 80913252
Hit rate: 71.216896
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4246720
Num Write: 4239200
Num Hit: 8354611
Num Miss: 131309
Total Cycles: 186595420
Hit rate: 63.625576
------------ STATISTICS -----------
Number of Instructions: 430753
Number of Cycles: 720948
Avg Cycles per Instrcution: 1.6737
Branch Perdiction Accuacy: 0.5045 (Strategy: Always Not Taken)
Number of Control Hazards: 48010
Number of Data Hazards: 279916
Number of Memory Hazards: 47774
-----------------------------------
```

LRU

```
---------- CACHE STATISTICS ----------
-------- STATISTICS ----------
Num Read: 3034928
Num Write: 4631364
Num Hit: 7534975
Num Miss: 131317
Total Cycles: 2099008
Hit rate: 57.380043
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4202144
Num Write: 4193888
Num Hit: 8264715
Num Miss: 131317
Total Cycles: 71329440
Hit rate: 62.937130
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4202144
Num Write: 4136608
Num Hit: 8207435
Num Miss: 131317
Total Cycles: 183858500
Hit rate: 62.500935
------------ STATISTICS -----------
Number of Instructions: 430753
Number of Cycles: 576124
Avg Cycles per Instrcution: 1.3375
Branch Perdiction Accuacy: 0.5045 (Strategy: Always Not Taken)
Number of Control Hazards: 48010
Number of Data Hazards: 279916
Number of Memory Hazards: 47774
-----------------------------------
```

2.4.3 matrixmulti.c

SDBP

```
---------- CACHE STATISTICS ----------
-------- STATISTICS ----------
Num Read: 1449885
Num Write: 4216340
Num Hit: 5524798
Num Miss: 141427
Total Cycles: 2188040
Hit rate: 39.064663
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4525664
Num Write: 4226496
Num Hit: 8620910
Num Miss: 131250
Total Cycles: 74213860
Hit rate: 65.683121
---------- LOWER CACHE ----------
-------- STATISTICS ----------
Num Read: 4200000
Num Write: 4194528
Num Hit: 8263284
Num Miss: 131244
Total Cycles: 184755980
Hit rate: 62.961231
------------ STATISTICS -----------
Number of Instructions: 226082
Number of Cycles: 350237
Avg Cycles per Instrcution: 1.5492
Branch Perdiction Accuacy: 0.3764 (Strategy: Always Not Taken)
Number of Control Hazards: 40667
Number of Data Hazards: 112620
Number of Memory Hazards: 13164
-----------------------------------
```

LRU

```
---------- CACHE STATISTICS ----------
-------- STATISTICS ---------
Num Read: 1434093
Num Write: 4215948
Num Hit: 5518817
Num Miss: 131224
Total Cycles: 2097528
Hit rate: 42.056461
---------- LOWER CACHE ----------
-------- STATISTICS ---------
Num Read: 4199168
Num Write: 4190944
Num Hit: 8258888
Num Miss: 131224
Total Cycles: 71279104
Hit rate: 62.937328
---------- LOWER CACHE ----------
-------- STATISTICS ---------
Num Read: 4199168
Num Write: 4133632
Num Hit: 8201576
Num Miss: 131224
Total Cycles: 183722720
Hit rate: 62.500580
------------ STATISTICS -----------
Number of Instructions: 225440
Number of Cycles: 318948
Avg Cycles per Instrcution: 1.4148
Branch Perdiction Accuacy: 0.3765 (Strategy: Always Not Taken)
Number of Control Hazards: 40678
Number of Data Hazards: 110957
Number of Memory Hazards: 11735
-----------------------------------
```

Since this we can conclude that **SDBP** can minorly improve cache efficacy.

2.5 Some analyze and observations

Since I'v already said the disadvantage of the miss rate calculation method before, the comparesion is a little bit meaningless, but I still observate some thing interesting.
First, SDBP will betterly performed compare with the LRU policy when the L1 and L2 cache is small, this is because there will more accesses to the LLC(since it is a three-level cache).
Also, since the three test grogramms is quite small, the differece between SDBP and LRU is reasonable not significant.
I also tried some other test, and I noticed that SDBP will perform worsely if the whole accesses are random and widely. This is because the sampler tag array has only 12 entries, if I randomly and widely access data, the former entries will be replaced quickly by the later entries. The SDBP will degenerate to LRU policy, but it is not a big problem at normal operating system, since many access are nearly both at space and time. But it will become a problem when there is a situation the page manager has a lot of fragmentations.