

Summary of Lab 1

Name: hu weiming Major: Master in CS

Student No: 2020233177 E-mail: huwm1@shanghaitech.edu.cn

Optimal Replacement Policy and Inclusive to Non-inclusive

The work of Cache in Sniper simulator

We all know that the Optimal Replacement Policy is based on the future use of the cache block. But actually it is impossible to obtain cache blocks for future use in practice. Maybe we could predict according to some regular sequence which appeared in past executions or use a lookahead-stack like the paper teacher Wang gave. The method I use is to adjust according to the existing cache block access sequence, which is the traditional OPT policy.

Computer security technology, ASLR will randomly arrange the address space of the program to prevent attackers from jumping to the address they need. In the Linux system, ASLR is divided into three levels: 0, 1, and 2. 0 means no randomization, that is, turn off ASLR. 1 and 2 represent partial randomization and complete randomization, respectively. The ASLR level is set to 0 to ensure that the program has the same address twice when the program is executed.

Focus on AccessSingleLine

Now talking about the understanding of the Sniper simulator. Every cache accessing would call **accessSingleLine** of **cache.cc**, and **splitAddress** would handle the addr, then I got the key parameter **set_index**, **tag** and **block_offset**. Then the CPU chooses to read or write according to **access_type**. Logically, the function **read_line** and **write_line** both call the function **updateReplacementIndex**. In the Sniper the replacement policy is LRU. Correspondingly, when a cache miss happen, the function **insertSingleLine** would be called and finish the replacement and insert. The function insert of **cache_set.cc** get the index of cache block to be replaced by **getReplacementIndex**.

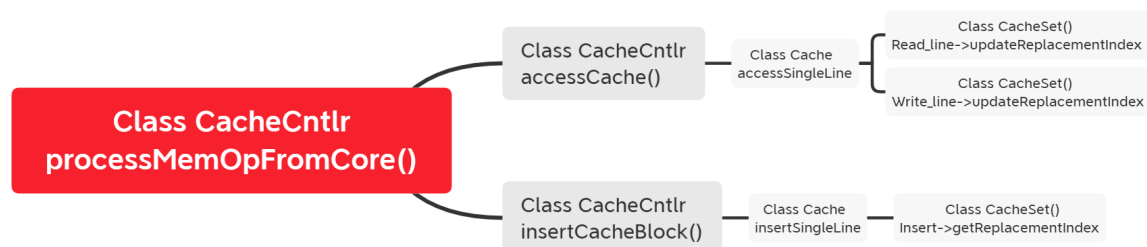


Figure 3 access and insert

File IO operation

As I said before, after running a process I will get the cache accessing sequence, and the memory address wouldn't change without ASLR. So I use a file IO operation in the function **accessSingleLine** to write the **set_index** and **tag** into a text file. When running the program for the second time, I read the data for the text file and set up a two-dimensional array to store all cache accessing sequence. It is noted that when **accessSingleLine** is called for the first time, I store the all data into the array. It means that only reading the file once to reduce the latency. Both of them modified in function **accessSingleLine** of file **cache.cc**.

```
// *****read file operation
if(read_condition==0){
    ifstream infile;
    infile.open("future_list.txt",ios::in);
    while (!infile.eof())
    {
        infile>>set_index>>tag;
        future_list[read_condition][0]=set_index;
        future_list[read_condition][1]=tag;
        read_condition++;
    }
    infile.close();
}
//*****read file operation
```

Figure 2 read file

```
// *****write file operation
ofstream outfile;
outfile.open("future_list.txt",ios::app);
outfile <<set_index<<" "<< tag << endl;
outfile.close();
//*****write file operation
```

Figure 3 write file

But I am sorry to say that this place may need to be implemented manually. Running the program for the first time to execute the **write file** area and comment out the **read file** area. In the second time, executing the **read file** area and comment out the **write file** area.

My OPT Policy ideas

After sorting out the process of CPU cache access, I will describe my idea of implementing OPT below. I noticed that the replacement policy selected by Sniper is LRU. In order to avoid a lot of modification of interfaces, classes and passing parameters, I directly replaced the algorithm with OPT in LRU. I store the cache accessing sequence in an array named **future_list**, the **set_index** and **tag** of all cache accessing sequences are stored in the array. I use a global variable **counter_access** to mark the position of the currently accessed instruction in the **future_list**. Through **future_list** and **counter_access**, after entering the replacement function, I can also know the **set_index** and **tag** currently accessed.

The OPT idea of mine is that each cache block in the set has its own value of **next**, which represents the distance between the next access of the this cache block and current accessing. It is stored in the array **m_opt_bits**. Every time there is a block to be evicted, finding the cache block with the max **m_opt_bits[i]** in the set, which represents the cache block that is the farthest to be accessed next time. Returning this index to implement **getReplacementIndex**. Every time the

cache accessing is executed, as with LRU, the next value of the cache block must be updated. I let all `m_opt_bits[i]` of the set must be subtracted 1, which means that the cache accessing sequence has went ahead one bit, and then the value of "next" of the current access need to be updated. According to the result of the comparison, the miss rate is relatively low when the length of searching times is set to 8000, so each time the value of next is updated, it only looks forward at most 8000 bits. Equivalent to the length of the sliding window is 8000.

Main Code of OPT

```
UInt32
CacheSetLRU::getReplacementIndex(CacheCntlr *cntlr)
{
    for (UInt32 i = 0; i < m_associativity; i++) //same as LRU find the invalid
first
    {
        if (!m_cache_block_info_array[i]->isValid())
        {
            m_opt_bits[i]=FindtheNextAccess(counter_access); //update the value of
next
            all_set_opt_bits[set_index][i]=m_opt_bits[i]; //synchronize it to
all_set_opt_bits
            return i;
        }
    }
    for(UInt8 attempt = 0; attempt < m_num_attempts; ++attempt) //same as LRU
    {
        UInt32 index = 0;
        UInt8 max_bits = 0;
        for (UInt32 i = 0; i < m_associativity; i++) //find the MAX of next
        {
            if (m_opt_bits[i] > max_bits && isValidReplacement(i))
            {
                index = i;
                max_bits = m_opt_bits[i];
            }
        }
        LOG_ASSERT_ERROR(index < m_associativity, "Error Finding OPT bits");

        bool qbs_reject = false;
        if (attempt < m_num_attempts - 1)
        {
            LOG_ASSERT_ERROR(cntlr != NULL, "CacheCntlr == NULL, QBS can only be
used when cntlr is passed in");
            qbs_reject = cntlr-
>isInLowerLevelCache(m_cache_block_info_array[index]);
        }
        qbs_reject=false;
        if (qbs_reject) //this modification is about inclusive policy, could
ignore it
        {
            cntlr->incrementQBSLookupCost();
            continue;
        }
        else
        {
            m_opt_bits[index]=FindtheNextAccess(counter_access);
        }
    }
}
```

```

        m_set_info->incrementAttempt(attempt);
        return index;
    }
}
LOG_PRINT_ERROR("Should not reach here");
}

```

This is the code of **getReplacementIndex**. I use the class of LRU and modified it on this basis, so I did not modify the attempt and qbs_reject part. I added a **FindtheNextAccess** function as follows.

```

UInt32
CacheSetLRU::FindtheNextAccess(UInt32 counter_access){ //find the next of
current accessing
    UInt32 set_index=Cache::future_list[counter_access][0]; //set_index of
current accessing
    IntPtr tag=Cache::future_list[counter_access][1]; //tag of current accessing
    UInt32 i;
    for(i=counter_access+1;i<counter_access+num_lookahead+1;i++) //look forward
    {
        if(set_index==Cache::future_list[i][0])
        {
            if(tag==Cache::future_list[i][1])
            {
                return i-counter_access; //return the value of next(distance)
            }
        }
    }
    return i-counter_access;
}

```

This is the code of **FindtheNextAccess**. Actually I think this is not the best implementation. With the lookahead, the method in the paper shared by Teacher Wang can be tried. However, due to the limitations of personal programming ability and knowledge, I chose this method of directly matching "next".

```

void
CacheSetLRU::updateReplacementIndex(UInt32 accessed_index)
{
    for(UInt32 j=0;j<512;j++){ //total 512 set
        for(UInt32 i=0;i<m_associativity;i++){
            if(all_set_opt_bits[j][i]>0) //avoid out of bounds
                all_set_opt_bits[j][i]--; //every access, the value of next will
-1
        }
    }
    m_opt_bits[accessed_index]=FindtheNextAccess(counter_access);
    //synchronize it to all_set_opt_bits
    all_set_opt_bits[set_index][accessed_index]=m_opt_bits[accessed_index];
    counter_access++;
}

```

This is the code of **updateReplacementIndex**. Every access we need to update the **all_set_opt_bits** array that stores the "next" information of all blocks. The conditional judgment is added to avoid data out of bounds when the initialization logic is flawed, to avoid the value of "next" is reduced to below 0. Then update the "next" of the current access and synchronize it to **all_set_opt_bits**. Every access will call **updateReplacementIndex**, so **counter_access** is incremented here.

Testing of Result

test case: lab0.exe

First of all, I have been using the executable program lab0.exe in Lab 0 when I performed the pre-code testing. So I will use this program as the basis of miss rate test first, and then optimize the algorithm. So I first posted the results of the lab0.exe test and compared the OPT algorithm with Sniper's own LRU algorithm. The results are as follows.

1 num_lookahead=8000

`int num_lookahead=8000;` The look forward access sequence is 8000.

L2 TLB		L2 TLB	
num accesses	151	num accesses	193
num misses	143	num misses	142
miss rate	94.70%	miss rate	73.58%
mpki	0.84	mpki	0.83
Cache Summary		Cache Summary	
Cache L1-I		Cache L1-I	
num cache accesses	20437	num cache accesses	20439
num cache misses	1039	num cache misses	1681
miss rate	5.08%	miss rate	8.22%
mpki	6.11	mpki	9.88
Cache L1-D		Cache L1-D	
num cache accesses	55078	num cache accesses	55079
num cache misses	3029	num cache misses	3352
miss rate	5.50%	miss rate	6.09%
mpki	17.81	mpki	19.70
Cache L2		Cache L2	
num cache accesses	4105	num cache accesses	5083
num cache misses	3508	num cache misses	3542
miss rate	85.46%	miss rate	69.68%
mpki	20.63	mpki	20.82

miss rate of LRU(left)

miss rate of OPT(right)

In fact, for Cache L1-I, not only did not optimize the hit rate, but got a relatively poor result, 600 more cache misses. Other caches are not performing well. It actually reduces the **miss rate** of L2 TLB and Cache L2, which may be a valuable place. But in fact, if you observe carefully, you will find that **num cache misses** has not decreased, but **num cache accesses** has increased. Therefore, either my implementation ideas are flawed, or there are still many things I don't understand in Sniper. Unfortunately, there is no more time to perfect it.

2 num_lookahead=2000

`int num_lookahead=2000;` The look forward access sequence is 2000.

L2 TLB			L2 TLB		
num accesses		154	num accesses		221
num misses		142	num misses		142
miss rate		92.21%	miss rate		64.25%
mpki		0.83	mpki		0.83
Cache Summary			Cache Summary		
Cache L1-I			Cache L1-I		
num cache accesses		20437	num cache accesses		20437
num cache misses		1033	num cache misses		1698
miss rate		5.05%	miss rate		8.31%
mpki		6.07	mpki		9.98
Cache L1-D			Cache L1-D		
num cache accesses		55069	num cache accesses		55069
num cache misses		3038	num cache misses		3335
miss rate		5.52%	miss rate		6.06%
mpki		17.86	mpki		19.61
Cache L2			Cache L2		
num cache accesses		4111	num cache accesses		5088
num cache misses		3501	num cache misses		3535
miss rate		85.16%	miss rate		69.48%
mpki		20.59	mpki		20.79

miss rate of LRU(left)

miss rate of OPT(right)

There is indeed a certain improvement for L2 Cache. However, according to the feature of the cache, ensuring the hit rate of L1 Cache is definitely the most critical. The hit rate alone may not be able to describe the impact on performance, but considering the hit rate, there is no doubt that this OPT is not a success. Teacher Wang or TA are welcome to criticize and correct any ill-consideration.

test case: sort.exe num_lookahead=2000

For this quick sort of 100 numbers, the number of instructions processed is larger than that of lab0.exe, which is about 1.5-2 times that of lab0.exe. In this test, OPT performed pretty well, and the optimization of L2 Cache miss was very large. The scale of this quick sort algorithm is relatively small, the size of the array is 100.

L2 TLB			L2 TLB		
num accesses		152	num accesses		1430
num misses		137	num misses		138
miss rate		90.13%	miss rate		9.65%
mpki		0.54	mpki		0.54
Cache Summary			Cache Summary		
Cache L1-I			Cache L1-I		
num cache accesses		31742	num cache accesses		31742
num cache misses		1972	num cache misses		1903
miss rate		6.21%	miss rate		6.00%
mpki		3.79	mpki		15.43
Cache L1-D			Cache L1-D		
num cache accesses		83631	num cache accesses		83638
num cache misses		3527	num cache misses		3325
miss rate		4.22%	miss rate		3.98%
mpki		11.77	mpki		15.10
Cache L2			Cache L2		
num cache accesses		3989	num cache accesses		7800
num cache misses		3438	num cache misses		3438
miss rate		86.19%	miss rate		44.08%
mpki		13.53	mpki		13.53

miss rate of LRU-sort(left)

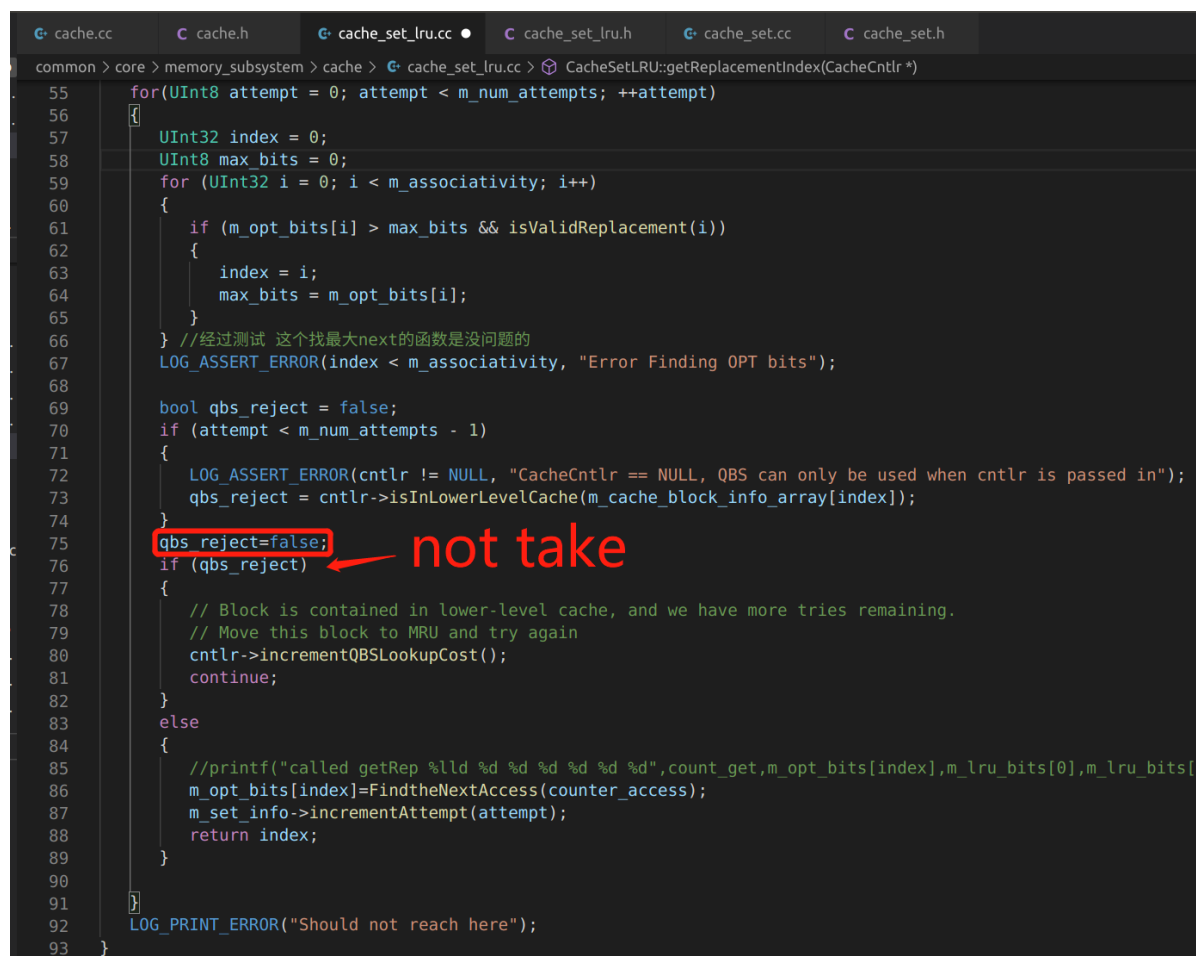
miss rate of OPT-sort(right)

test case: to be updated...

I haven't thought of a more interesting example for the time being. If I understand deeply in running a segment of code in the future, I may be able to add this part. Of course, the premise is that the OPT algorithm is correct. Because there may be many loopholes in the OPT in this lab, it may not be a real "OPT".

Inclusive to Non-inclusive

After discussing with classmates in the office, I think there must be an interface related to inclusive policy in the replacement algorithm. So focus on the parameters **qbs_reject** and **attempt**. The processing of inclusive policy may be included in **isInLowerLevelCache**. Therefore, the solution I chose is to set **qbs_reject** to false, that is, not to follow the conditional branch of if execution. After comparing the cache miss data before and after performing this operation, I found that this operation hardly affects the miss rate of L1 and L2 Cache. Therefore, the judgment about the interface of the inclusive policy must be in the replacement policy is not necessarily correct.



```
55 for(UINT8 attempt = 0; attempt < m_num_attempts; ++attempt)
56 {
57     UInt32 index = 0;
58     UInt8 max_bits = 0;
59     for (UInt32 i = 0; i < m_associativity; i++)
60     {
61         if (m_opt_bits[i] > max_bits && isValidReplacement(i))
62         {
63             index = i;
64             max_bits = m_opt_bits[i];
65         }
66     } //经过测试 这个找最大next的函数是没问题的
67     LOG_ASSERT_ERROR(index < m_associativity, "Error Finding OPT bits");
68
69     bool qbs_reject = false;
70     if (attempt < m_num_attempts - 1)
71     {
72         LOG_ASSERT_ERROR(cntlr != NULL, "CacheCntlr == NULL, QBS can only be used when cntlr is passed in");
73         qbs_reject = cntlr->isInLowerLevelCache(m_cache_block_info_array[index]);
74     }
75     qbs_reject=false;
76     if (qbs_reject)
77     {
78         // Block is contained in lower-level cache, and we have more tries remaining.
79         // Move this block to MRU and try again
80         cntlr->incrementQBSLookupCost();
81         continue;
82     }
83     else
84     {
85         //printf("called getRep %lld %d %d %d %d %d %d",count_get,m_opt_bits[index],m_lru_bits[0],m_lru_bits[
86         m_opt_bits[index]=FindtheNextAccess(counter_access);
87         m_set_info->incrementAttempt(attempt);
88         return index;
89     }
90 }
91 LOG_PRINT_ERROR("Should not reach here");
92 }
93 }
```

Besides, in addition to comparing their miss rate, I have not found another way to verify inclusive to non-inclusive. I think there is a detailed processing method for L1 L2 in cache_cntlr.cc. This file describes a complete set of CPU control flow.

Challenge

This part content is the thinking process of the lab 1. Except for the deeper understanding of CPU cache, instruction stream and memory, I also got a lot in C++ programming and Object-Oriented Programming. This part can be regarded as a chat or note. I have some understanding before that the replacement algorithm is a strategy to implement the block replacement in the set. To find the operation of the cache, you still need to pay attention to the cache_set. The write, read, and insert inside are the operations that really change the cache. The replacement policy is just a decision who to modify.