CS359: Computer System Architecture, Fall 2016

Report: Cache Lab

Name：兰鸥羽
Student ID：5140309001

---

1 Purpose

The lab has two parts. I need to understand the organization and the behavior of a cache memory and its impact on C programs and data structures.

- Part A: Writing a Cache Simulator

  In this part, it is required to write a cache simulator in C which simulates the hit/miss/evict behaviour of a cache memory on *valgrind* memory traces. The structure of the cache, including the set size, the associativity and the block size, is arbitrary and designated by the command-line arguments. And it should use LRU replacement policy. It should output the exact total number of hits, misses, and evictions.

- Part B: Optimizing Matrix Transpose

  In this part, it is required to write a matrix transpose function with no more than 12 local variables in C as few cache misses as possible in directed mapped cache memory with 1KB cache size and 32-byte block size. The function should be correct and effective for three different-sized matrices: 32*32, 64*64 and 61*67.

2 Implement

2.1 Part A: Writing a Cache Simulator

2.1.1 Physical Organization

To simulate the cache memory, we need to first simulate its physical organization. We know that a cache is composed of one or more sets, a set is composed of one or more lines, a line is composed of the tag, set index, valid bit and block offset. Therefore, I define the *struct* for the Line, Set and Cache. Different from the realistic cache memory, I do not store the block offset, set index and the data in the address in the Line simulator because of useless in this task.

Additionally, I add a *int* cycle in it for the simulation of RLU replacement policy. And I write the *cache_init*, *cache_destroy* function to initial and destroy the cache, especially allocate and free the space, in the beginning and at the end respectively.

2.1.2 Cache Operation

For each instruction, we get the operation (load, store, modify or else), the address and the size information. Because in this task, all we need to do is count the total miss/hit/eviction number, so I recognize the *load* and *store* operation as the same, using *cache_op* function; and call this function twice for the *modify* operation, while ignoring other operations and the size information. All we need to do is to find whether the provided address has been stored in the cache simulator, if not, choose the least recent used one and replace it with the new address.

For each provided address, we know that the first t-bit is for tag, following s-bit for set index and last b-bit for block offset. Firstly, I extract the set index bits from the address to determine which set to find or put it. Then search the particular set with *tag* extracted from the address. It is worth to mention that we abandon the set index and block offset information here because useless afterwards. Secondly, I search every line in the set to check the *valid* information and compare the extracted tag with the *tag* in each line. If it finds a valid line with the same tag, then HIT! If not, MISS! Call the *cache_miss* function which searches the whole set, chooses the first invalid line to put the new tag in. If not, choose the line with largest *cycle* and replace it with the new tag, that is EVICTION! Whichever situation, it is necessary to call *set_cycle* function which set the new put line's cycle 0, then increase every line's cycle in this set by 1 for later LRU replacements.

2.2 Part B: Optimizing Matrix Transpose

2.2.1 Simple Blocking

With the hint of blocking, I decide to divide the input matrix into multiple bunches (alias to distinguish with the item of block in cache memory), and deal with each bunch separately in order. The problem is how to choose the bunch size. Since the block size is 32 bytes and the *int* size is 4 bytes, each byte contains 8 *int*s. First observing the 32*32 matrix, I divide it in blocks and tag each block with the set number mapped to. I discover that the blocks are mapped into the same set every 8 rows. So I think the 8*8 bunch may be the best. However, blocks mapped into the same set only every 4 rows in 64*64 matrix, there are maybe some better choice. Then considering the diagonal confliction, the better solution is to assign the value in the diagonal at the last of each bunch row. The *miss* produced by the diagonal is also unavoidable for the next row assignment, so it is worthwhile. And we should consider the matrix whose row number and

column number are not same when programming. Then I choose different-size bunches for each different-sized matrix and observe the *miss* number (Table 1.).

Matrix\Bunch	8*8	4*4	16*8	16*4	8*4	Standard
32*32	287	439	287	415	319	300
64*64	4643	1795	4643	1747	1747	1300
61*67	2118	2423	1953	1902	2059	2000

Table 1. The *miss* number in different matrix and bunch size

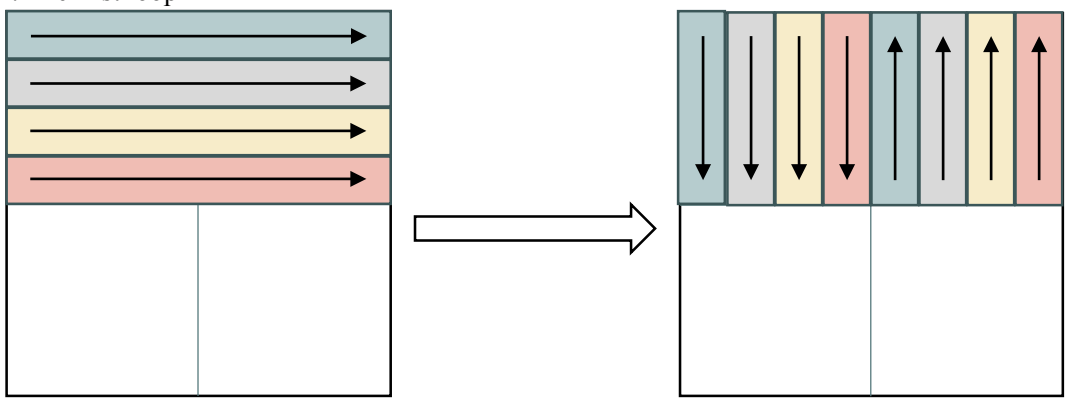
We can see that the 32*32, 64*64 and 61*67 matrix have least *miss* number with the bunch size of 8*8/16*8, 16*6/8*4 and 16*4 respectively. However, whatever the bunch size is, the miss number of 64*64 matrix has never been below the standard. So there must be some better algorithms.

2.2.2 Blocking with Local Variables

Notice that the lab description expresses that we can use 12 local variables at most in the transpose function, but I only use 4 local variables in *2.2.1* for loops. It is time to utilize the local variables to reduce the conflict miss. In my opinion, the reason why 64*64 matrix transpose produce so many *miss*es is that the blocks are mapped into the same set every 4 lines while the block contains 8 *int*s at one time, which results that we have not use enough data before a *miss* happens. After deep consideration, I think we can still divide the matrix into bunches of 8*8 size. However, we need to divide each bunch into sub-bunched of 4*4 and use the local variables and unused space in B matrix for temporal locality.

For every bunch, I write 2 loops. In every step of the first loop, I read one block of the bunch in A in the first four lines and store the whole block (8 *int*s) into 8 local variables (x0~x7), which happens only 1 *miss*. Then assign x0~x7 to B, which causes 4 *miss*es. What I want to do is to transpose the upper left sub-bunch and use the upper right sub-bunch of B as a temporal space for upper right of A which should be transposed into bottom left of B. And the order in the temporal space should be reversed to avoid the conflict miss in latter moving.

Fig 1. The first loop



In every step of the second loop, I read two columns in the bottom sub-bunches of A to variables, which causes 4 misses. Then move one line of the upper right sub-bunch to the bottom left in B, assign two line of right with variables, which causes two *miss*es.

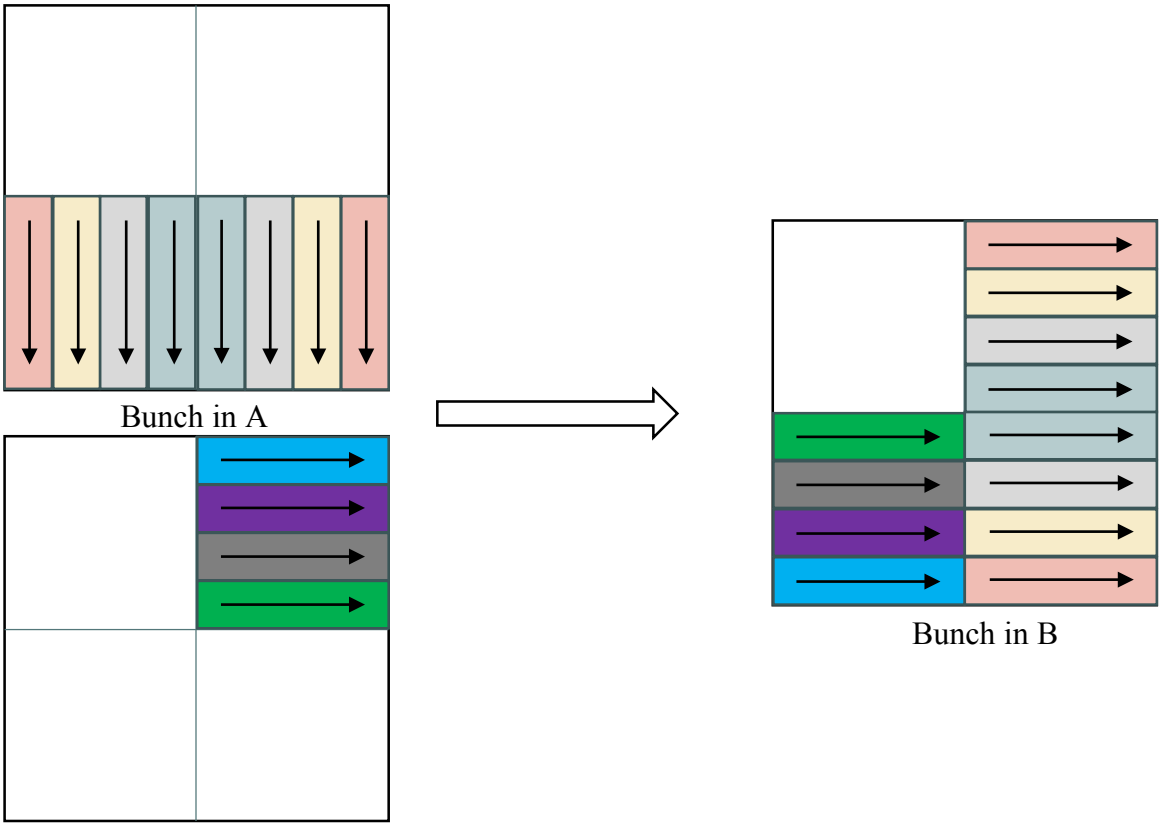


Fig 2. The second loop

The above two figures(Fig 1. Fig 2.) show the process of the loop respectively, where one color represents one step in the particular loop while the arrow represents the order of data placement. Using this algorithm, both 32*32 and 64*64 matrices have the least miss, 283 and 1243 respectively. However, it is not suitable for matrices whose row number and column number are different.

2.2.3 Blocking Combination

According to the performance, I choose the *2.2.1* algorithm for 61*67 matrix and *2.2.2* algorithm for 32*32 and 64*64 matrices. And combine them in the *transpose_submit* function. I have not test it with other size of input matrix. Maybe it is not well-designed enough.

3 Conclusion

This lab helps me understand the cache memory much better, and makes me recognize the importance of considering cache during coding. During the lab, I confront with many problems, which have been solved after deep consideration. The detail has been included in the *Impletion*

section. This lab teaches me that software and hardware is complementary. Even if the hardware is unchanged, we can still improve the performance a lot.