# REPORT OF CS3230 PJ1 TASKC

## *Block Matrix Multiplication Considering Cache*

I. algorithm description

The basic idea is to decide the partition size by the input cache size. Then multiply by block matrices.

A. partition size

The sub matrices tend to be square and the length should satisfy the following inequality.

$$pttSize + pttSize^2 \le S$$

For simplification, I calculate as $pttSize = \left\lfloor \sqrt{S} - 1 \right\rfloor$.

B. partition method of A ("5 for loop" or "6 for loop")

Shall we partition A vertically? It seems to be more elegant and symmetrical. But actually it's not as efficient as not doing so. (I test both methods with the 10 sets of data, and the output is shown as a line graph in last page.)

Because when we multiply two sub matrices, Sub A is scanned row by row in sequence but sub B is scanned many times as a whole one. So sub B is almost always in cache. If we continue scanning A downwards and multiply it with sub B, we'll use sub B almost without cost.

Otherwise, when we scan the second block line of A, we access sub B again. But this time, sub B is not in cache so we must pay for it.

Actually it's some kind of *thrashing*. I avoid it by scanning the blocks of B in single direction.

C. uniformity

The partition are usually imperfect and there may be some rectangular sub matrices in B. To make the code more uniform and simple, I decide the bound condition of for loop by the position of the sub matrix.

II. efficiency comparison

A. In Task 2, the usages of the same entry are dispersed. The cache is much smaller than the matrices, so when an entry is accessed again, usually it's kicked out of cache before, which leads to a cache miss.

B. In Task 3, the usages of the same entry are clustered. The sub matrices are smaller than the cache, so when an entry is accessed again, usually it's still in cache, which leads to a cache hit.

C. Let's consider the total accesses too.

Task 2 has $(2x+1)\,p \cdot q$.

Task 3 has about $\left(2x + \dfrac{x}{pttSize}\right) p \cdot q$.

The small difference of access numbers can easily be made up of the improvement of cache hit rate.

III. anomaly

LIFO has anomaly. (Analysis is in IV.)

After testing and thinking, I haven’t found any other anomaly.

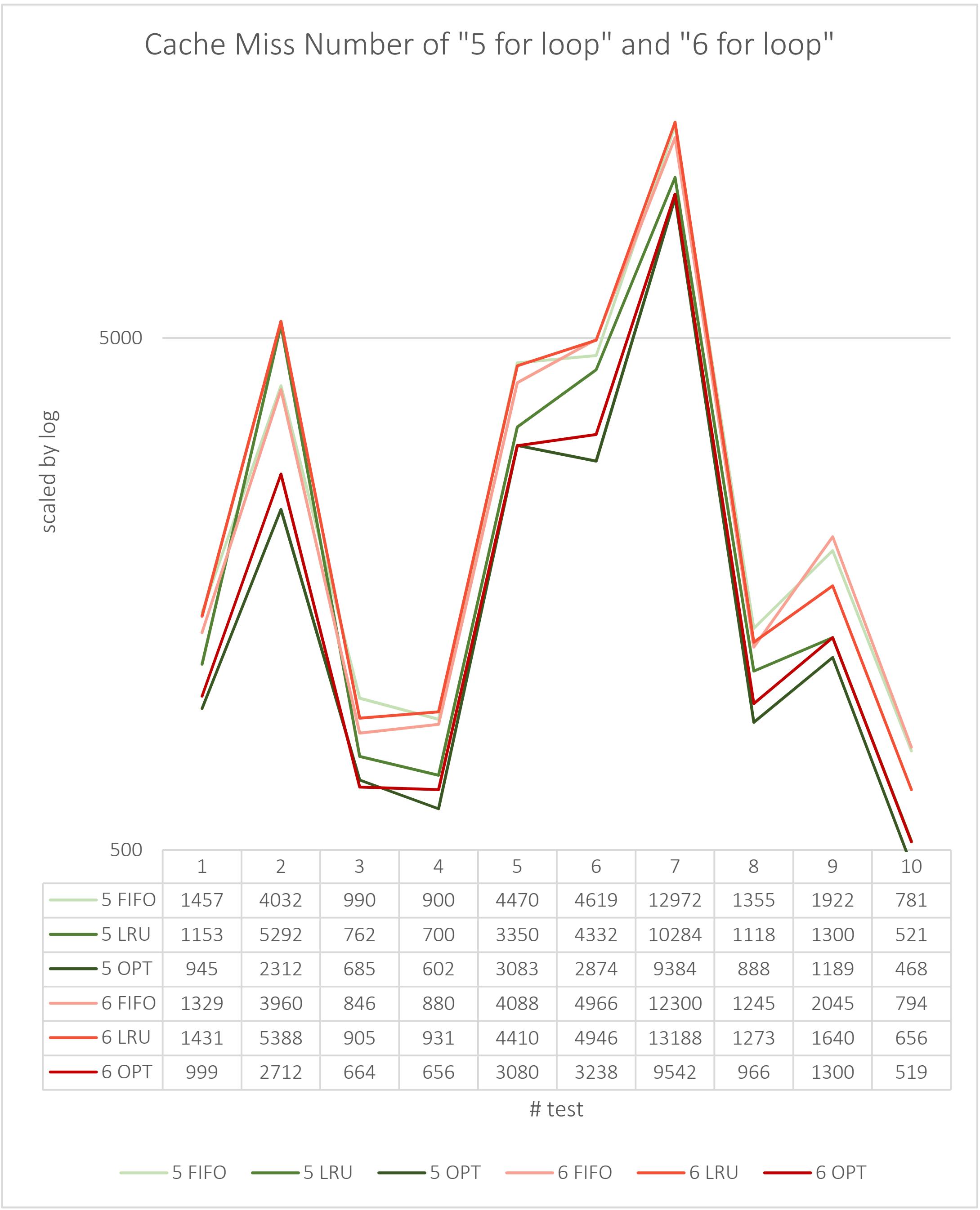
IV. LIFO

LIFO violates our original intension of turning to block matrix multiplication.

A. By multiplying small matrices, we cluster the usages of entries so as to cut down cache miss. So once we access an entry, we expect to access it again soon. In other word, we should keep the recently accessed entry in cache for revisiting.

B. But as for LIFO, we choose the newest one as victim. The case we use LIFO is that once we access an entry, we expect not to access it again soon.

In conclusion, LIFO can only lead to worse performance, which is true in testing. That’s why we don’t consider it for this task.



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 FIFO | 1457 | 4032 | 990 | 900 | 4470 | 4619 | 12972 | 1355 | 1922 | 781 |
| 5 LRU | 1153 | 5292 | 762 | 700 | 3350 | 4332 | 10284 | 1118 | 1300 | 521 |
| 5 OPT | 945 | 2312 | 685 | 602 | 3083 | 2874 | 9384 | 888 | 1189 | 468 |
| 6 FIFO | 1329 | 3960 | 846 | 880 | 4088 | 4966 | 12300 | 1245 | 2045 | 794 |
| 6 LRU | 1431 | 5388 | 905 | 931 | 4410 | 4946 | 13188 | 1273 | 1640 | 656 |
| 6 OPT | 999 | 2712 | 664 | 656 | 3080 | 3238 | 9542 | 966 | 1300 | 519 |