Cache-Friendly Programming

Ruiqi Feng

2023.1

Abstract

In this report three designs of the cache set will be introduced and their performance will be analyzed. A cache simulator is implemented to investigate how replacement policies affect the hit rates of the cache, whose result matches theoretical analysis and experiments. Some advice on cache-friendly programming is given based on these results. In the end, several cache writing policies are discussed.

1 Introduction

As we know, the cache is a component in the CPU that improves its performance. To fully realize its potential, we will discuss the design and strategies used in the cache and give some practical programming advice.

2 Design of the Cache Set

It is known to all that the cache in the computer consists of cache sets, each of which is made of cache lines. Because of the storage hierarchy, the memory space is much larger than the cache space. The map from the memory address to the cache address is therefore similar to a hash map, where several bits in the middle of the memory address string are extracted to serve as the index of sets in the cache. The number of lines in a set, in turn, causes differences in the influenced cache lines in the replacement policy. The symbols below are consistent with those in Ref.[2]

2.1 Different Set Design

2.1.1 Direct Mapping

If $E = 1$, all memory addresses with the same middle bits will be loaded to the same location in the cache. In case of a cache miss, this line will be replaced. Two problems then arise: the usage of the cache will remain low if the cache is not warmed-up, and the hit rate will be small if different pages of memory are frequently loaded[2].

2.1.2 Set-associative Mapping

In order to increase the hit rate, one cache set can have multiple lines, which enables various replacement policies to be implemented. The search in the same cache set can be parallelized using hardware acceleration[10].

2.1.3 Fully-associative Mapping

If each cache set has $E = \frac{C}{B}$ cache lines, or there are $S = 1$ cache sets, the procedure of every replacement has to traverse all cache lines. Parallelization is possible, and thus fully-associative mapping would outperform set-associative mapping if extra time consumption in a larger set is neglected.

2.2 Conclusions

Generally speaking, the direct mapping is minor to set-associative mapping in terms of time consumption. This is because the hit rate of the former scheme is no higher than that of the latter, among which the most extreme condition is the conflict miss. If details of implementations are taken into consideration, direct mapping can

be better than set-associative mapping under certain circumstances. In general, however, the set-associative mapping is better than the direct mapping.

Meanwhile, the number of lines in each set under the set-associative scheme has an effect on the performance as well. In principle, replacement policies can realize better hit rates if the number of lines is increased, but extra time consumption and the price will be a problem. Therefore, modern CPUs usually have a 2-way to 16-way set-associative cache. The fully associative cache is usually used on smaller caches like TLB for the same reasons.

3 Replacement Policies

In case of a cache miss and there is no empty block (line) in this set, one of the lines is required to be evicted so that the new contents can be loaded. To attain the highest possible hit rate, the line to be evicted is expected not to be accessed in the future.

3.1 Different Strategies

3.1.1 Belady Strategy

There exists a theoretically optimal algorithm[1], called the Belady replacement policy, in which the cache line which will be accessed in the most distant future is evicted[6][1]. Unfortunately, the Belady strategy is impractical for it requires our exact prediction of the future. We will be talking about other replacement policies and the Belady strategy can serve as a benchmark.

3.1.2 Least Recently Used

A feasible cache replacement policy can be designed to exploit the locality of the memory. One strategy is to replace the line with the earliest last access time. It is an attempt to predict the future by assuming the contents that have been long ignored are more unlikely to be accessed shortly, and thus is an approximation of the Belady strategy. The LRU strategy is close to the theoretical optimal according to experimental results and is widely adopted in real applications [8][7].

3.1.3 Least Frequently Used

An alternative way to take advantage of the locality is to evict the least frequently accessed contents in a given period T in the past. Though its spirits are similar to LRU, it can be seen that the parameter T affects the performance of LFU. While if T is too large, the response of the cache to the change in the memory space will experience higher latency, if T is too small, the calculated frequencies are more likely dominated by noise, and thus would not be able to reflect the actual frequency of recent accesses well. There is therefore a balance to search for.

3.1.4 Not Frequently Used (Aging)

Yet another method to make use of temporal locality is to evict the line with the accumulated minimum access count. The problem is obvious: accesses a long time ago should be neglected. Aging is proposed to solve this problem, where the accumulated access counts exponentially decay, erasing the effect of ancient accesses[8][9]. The high efficiency and simplicity of bit-wise shift make this strategy more practical.

This method is substantially equivalent to the LFU algorithm. The time average over a sliding window (LFU) can be considered as a convolution with a rect function

$$\int \mathrm{rect}(-\frac{t}{T}+\frac{1}{2})a(t)dt \tag{1}$$

where $a(t)$ is the count of new accesses to a certain cache line at time t, while NFU corresponds to the convolution with an exponential function (modulated by a step function, of course):

$$\int H(-t)(\frac{1}{2})^{-t}a(t)dt \tag{2}$$

[1]The proof is provided in Supplementary Materials Sec 1.

The trade-off between the high-frequency components and the ability to filter high-frequency noises remains. [2]

3.1.5 First In First Out

This scheme resembles a queue. FIFO exploits the locality that an earlier accessed element is less likely to be accessed in the future. Its performance is generally worse than NFU or LRU because of the case where a variable is loaded a long time ago but has been repetitively accessed[9]. Though FIFO itself is seldom used, there are some variants of FIFO[8][9].[3]

3.1.6 Random

Randomly select a cache line to evict. It has the advantage of simple implementation.

3.2 Simulation of cache performance

To further investigate the efficiency of different replacement strategies, I implemented a cache enumerator where the workflow of the cache, including the selection of the cache line to evict and the replacement with new contents, is modeled. Then a simulation of a matrix multiplication algorithm on this simulator is executed. Hits and misses are counted to evaluate the hit rate and thus the performance of the cache.

The multiplication of two matrices can be realized with three nested loops. However, different sequences of the loops will significantly influence the performance of the program, as discussed in Ref.[2].

In our simulation, the cache consists of 2 sets, 16 lines each, and each block stores 8 doubles.

3.3 Results

3.3.1 NFU/LFU

First, we need to decide the parameter of NFU and LFU strategies. Unfortunately, to get the average over a sliding window, a queue is maintained for every cache line, which drastically increased the time complexity of our simulation to an unfeasible degree. We will therefore stick to the NFU algorithm.

In our implementation, a 1-bit right shift is used to implement the exponential decay. Whenever a line is accessed, its frequency value f gets $f \leftarrow f + (1 << l - 1)$ where l is the maximum length of the bit string of f. This *aging* scheme, coincidentally, makes the NFU very similar to the LRU strategy.

Any line is guaranteed to have an f larger than that of any other lines with a last access time earlier than itself, as long as the deciding bit in f is recent enough so that has not been shifted out of the right end. This is to say, if the aging period is infinitely long, NFU is equivalent to LRU. In reality, however, the frequency bit can not take up infinite space. Nevertheless. we would notice similarities between NFU and LRU in our results.

To determine the optimal length of f in our problem, we simulate and evaluate the hit rate of different f and on different sizes of matrices. The results are shown in Fig.3.3.1. Though the permutation of i, j, k is 6, 3 pairs are equivalent because of symmetry. In our problem, the optimal length of f is estimated as 30.

3.3.2 NFU/LRU/FIFO

We now compare the three different strategies, at different sizes of matrices and different multiplication algorithms. The results are shown in Fig.3.3.2. In all cases, NFU and LRU are close to each other and are generally better than FIFO. In the ijk configuration, the hit rate of the FIFO method experiences a sharp drop near $\frac{\text{cache size}}{\text{matrix size}} = 1$. Before this point, the cache is not filled and no replacement is required. Thus, this demonstrates the relatively weak ability of FIFO to detect the pattern of recurring accesses. A similar situation occurs at ikj loops as well.

Next, we give an asymptotic analysis of the hit rate at a large matrix size. If the size of the matrix is sufficiently large, neighboring attempts to access elements in different rows will result in a cache miss. Meanwhile, sequential accesses to elements in the same row at step 1 are not able to use the temporal locality but only the spatial locality, which is the following elements being loaded to the cache. Therefore this performance depends

[2] I even suspect NFU is just an alias of LFU.

[3] Due to limited space (and time), I would not discuss them.

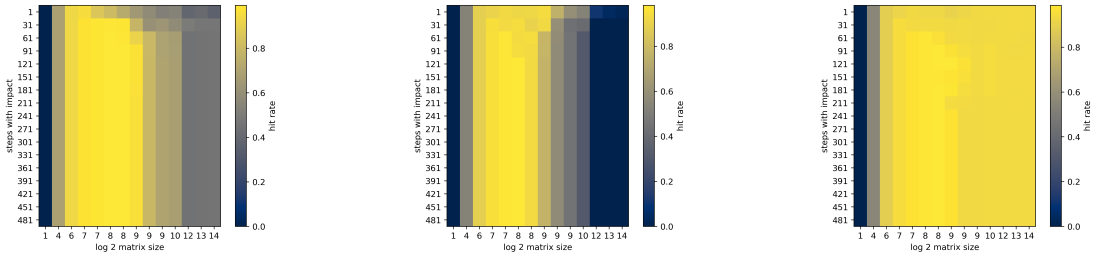

Figure 1: Left: loop order ijk; Center: loop order jki; Right: loop order ikj. The y-axis corresponds to f, and the x-axis is the size of the square matrix. The color represents the hit rate. When f is large enough, the hit rate is independent of f, where NFU approaches LRU. It can already be noticed that ikj is better than ijk and jki is the least cache-friendly.

only on the size of cache blocks. Our cache has 8 words in each block, so the hit rates at large matrix size are [4]:

config	ijk	jki	ikj
hit rate	0.625	0.912	0.0

which corresponds with the results in Fig.3.3.2.

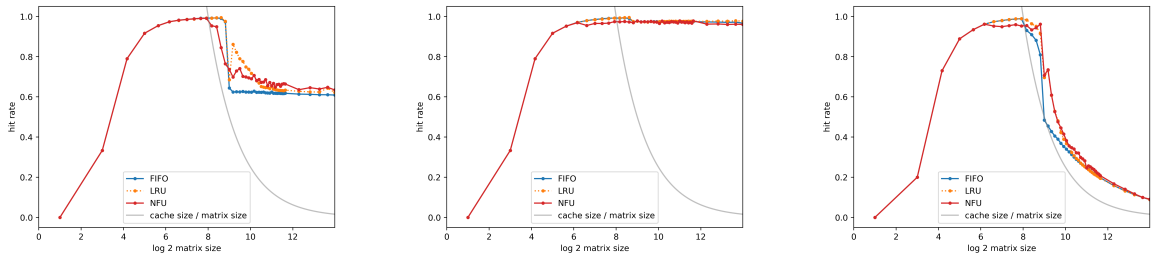

Figure 2: Left: loop order ijk; Center: loop order jki; Right: loop order ikj. Three different replacement policies are plotted. The gray line shows the ratio between cache size and matrix size. When the matrix size is small, the hit rates are relatively low because the program terminates before the cache gets warmed up.

3.4 Conclusion

As discussed above, an important factor to consider is the computational expenses to implement a certain replacement strategy. Thus, the LRU method or equivalent NFU method is often implemented. Good practice in terms of cache-friendly programming, then, is to improve the locality.

3.4.1 Small Data Size

Data frequently accessed should be small enough to fit into the cache. To realize so, large loops could be broken up into small portions. An example would be to convert the matrix multiplication into the multiplication of block-wise matrices. As is proved in Supplemental Materials Sec 2, multiplication by block consumes m^3 times the original time, where m^2 is the number of block matrices. Besides, as can be seen in Fig.3.3.2, the size of the matrix should not be too small to achieve higher hit rates.

3.4.2 Order of Access

Sometimes simply swapping the order of certain loops drastically increases the cache performance. This is by accessing data with good spatial locality. An example would be to traverse the matrix in the same row

[4] assuming in ikj and ikj, the values that are unchanged in the inner loop are stored in registers

as possible. As shown in Fig.3.4.2., the hit rates in our simulation show a similar trend as the experimental results from Ref.[2]. This further confirms that by switching certain loops, the spatial locality of programs can be improved.

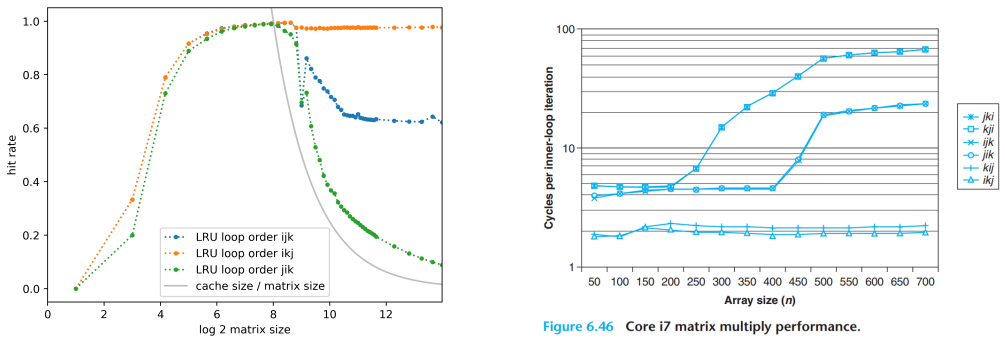

Figure 6.46 Core i7 matrix multiply performance.

Figure 3: Left: hit rates in 3 different matrix multiplication algorithms, using LRU replacement policy; Right: time in CPU clock of 3 different matrix multiplication algorithms

3.4.3 Recurring Access

The hierarchy of memory is intrinsically in favor of temporal locality. A rule of thumb would be to use higher level storage as possible, for example, store frequently accessed variables in registers. The same thing applies to the cache, so it would be generally good to *keep* frequently accessed data in the cache. To do so, the program should repetitively access the same contents if possible. This is a promising approach insensitive to the specific replacement policy used, and as shown in Fig. 3.3.2, in the center subfigure 3 policies all perform well.

4 Writing Policies

4.1 Different Write Strategies

4.1.1 Write Back

The cache is also used in writing data back to the memory. At a write hit, the *write-back* strategy only updates the contents in the cache. Because all reading first searches in the cache for the data, updating the cache alone is adequate for the system to function, regardless of the mismatch between the cache and the memory. A dirty bit is set so that when this block gets evicted, the contents in it are updated to the memory.

At a write miss, the *write back* policy is usually paired with the *write allocate* strategy[3], where the block written is also loaded to the cache.

4.1.2 Write Through

In the *write through* policy, a write hit is followed by updating both the cache and the memory. This ensures the cache always matches the memory. At a write miss, the *write through* policy is usually accompanied by the *no-write allocate* strategy, where the cache would not be updated at a write miss[4][5].

4.1.3 Cache Aside

In my understanding, the *cache aside* policy is similar to the *write though* policy. The difference lies in that the *cache aside* policy requires the client to implement cache writing policies, but in the *write through/read through* scheme the access procedure is encapsulated in the cache itself. The *cache aside* policy is often mentioned in the context of database applications, and it is more robust when concurrency is used[4].

4.2 Conclusion

We can select the appropriate writing policy according to the task.

If frequent writing is required, *writing back* would be a good idea because its writing latency is much shorter. Besides, when accompanied by a *write allocate* policy, the cache hit rate can be improved. In this case, a write miss is similar to a read miss[3].

When the task is read-heavy, on the contrary, *write through* is better. Its disadvantage in time efficiency is not significant in this case, but it can avoid the data loss caused by the mismatch between the cache and the memory, and is easier and cheaper to implement.

When the workload is static, a warm-up can be done to improve the performance of the cache. It is basically loading the data to be used into the cache before the execution of the program.

5 Summary

In fact, the cache designs are applicable to many areas, from databases to virtual memory. This demonstrates again the power of the hierarchy of computer storage. To best realize the potential of the cache, we need to understand the basic structure of the cache as discussed in Sec.2. Though we are usually unable to adjust the reading/writing policy of our CPU, it is beneficial to understand how they work. Some specific cache-friendly programming suggestions are listed in Sec.3.4. It is sometimes difficult to know the exact strategies the machine is using[2], so it is generally good to increase the locality of the program. This higher level of perspective is also beneficial to the portability of high-performance programs.

References

[1] Hyokyung Bahn and Sam Noh. "Characterization of Web Reference Behavior Revisited: Evidence for Dichotomized Cache Management". In: vol. 2662. Feb. 2003, pp. 1018–1027. ISBN: 978-3-540-40827-7. DOI: 10.1007/978-3-540-45235-5_100.

[2] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136108040.

[3] *Cache (computing)*. https://en.wikipedia.org/wiki/Cache_(computing).

[4] *Cache-Aside pattern*. https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside.

[5] *Differences between disk Cache Write-through and Write-back*. https://forum.huawei.com/enterprise/en/differences-between-disk-cache-write-through-and-write-back/thread/203781-891.

[6] Akanksha Jain and Calvin Lin. "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 78–89. DOI: 10.1109/ISCA.2016.17.

[7] Douglas W. Jones. *22C:116, Lecture Notes*. University Lecture. 1995.

[8] *Page replacement algorithm*. https://en.wikipedia.org/wiki/Page_replacement_algorithm#cite_note-3.

[9] Herber Tanenbaum Andrew S.; Bos. *Modern Operating Systems*. 4th. USA: Pearson, 2015. ISBN: 9780133591620.

[10] Bei YU. *Computer Organization Design*. University Lecture. 2022.