An Overview of Cache replacement policy

Name: Qi Zhao

University: George Washington University

Department: Computer Science

Author Note

This paper is prepared for CSCI 6461, Computer Architecture, taught by Morris Lancaster

An Overview of Cache replacement policy

**Abstract**

This article first introduces the features and characteristics of seven conventional cache replacement policies including Belady’s Min, FIFO, LIFO, LRU, MRU, LFU, RR. Then three promising replacement policies are detailed and studied. Finally, a comparative analysis of the efficiencies and performances of these replacement policies are conducted.

**Keywords**

Cache, Replacement policies, LRU, Hit, Miss

**Introduction**

As the key components of the memory hierarchy of modern computer system, Cache is the primary technique that is used to bridge the gap between the computer processor and main memory. Each level in the memory hierarchy serves as a cache that stores data or instructions that are fetched from the lower level and may be required by the higher level. Figure1 visualize the relations among memory hierarchy levels.

Cache memory design is based on two locality principles. One is called temporal locality, which means the data or instructions that are referenced will be referenced again in the future. The other is called spatial locality, which means the data or instructions stored in memory, near to the data or instructions which have been referenced, will be referenced in the future.

There are three major cache design schemes:

1. Direct mapping, in which any data is mapped to a unique place and don’t need replacement.
2. Fully associative, in which data could be mapped to any empty cache block. If there is no empty cache, a data stored in the cache will be replaced by the new data.
3. Set associative, in which a cache is divided into many sets and allow data to be mapped into any empty block within a cache set.

A cache hit occurs when the requested data can be found in the cache, while a cache miss occurs when it cannot. Cache hit means successfully reading data from the cache, which is faster than reading the data from the lower storage level. But Cache miss will cause extra penalty because of the time cost to access the lower storage level.

When the cache is full and a cache miss occurs, there will be a replacement policy to select the victim block and replace the victim block with the missed block. This article will focus on the replacement policies. The features of conventional replacement policies will be introduced in the section “Conventional Replacement policies”. And three detailed promising replacement policies will be introduced in the section “Three promising replacement policies.”

**Conventional Replacement Policies**

***Belady’s Min***

This is an optimal algorithm, in which the block that will be referenced in the furthest future should be replaced by the missed block. The cache that implements this policy will achieve the best performance. However, because the future access pattern generally cannot be forecasted, this policy is impossible to apply in practical scenarios. Belady’s Min is always used to make the comparison with other policies.

***FIFO***

This is a first in first out algorithm which behaves like a queue data structure. Regardless of how often or how many times the stored blocks are referenced, this policy simply chooses the first accessed block to be evicted when the missed block need an empty place to be stored.

***LIFO***

This is a last in first out algorithm which behaves like a stack data structure. This policy simply chooses the last accessed block to be the victim replaced by the missed block, with no regard to how often or how many times the stored blocks are referenced.

***Least recently used (LRU)***

In this policy, the least recently used block will be replaced by the missed block. Each block will hold a few bits serving as the counter which will increment every time when the block is referenced. When the cache is full and a cache miss occurs, the block with the least counter will be replaced. Figure2 illustrates this policy’s working process:

In the pattern illustrated by figure 2, the access sequence is A, B, C, D, E, D, F. The counter records the recency of the reference of the block. When the cache needs to access E, because the cache is full and A has the smallest counter, so the LRU policy replaces the A with E.

### *Most recently used (MRU)*

The MRU policy, in which the most recently used block (or the block that hold the biggest counter) will be replaced by the missed block, is a contrast to LRU. Figure 3 illustrates this policy:

In the pattern illustrated by figure 3, the access sequence is A, B, C, D, E, C, D, B. When the cache needs to access E, because the cache is full and D has the highest counter, block D is replaced by block E.

### *Least-frequently used (LFU)*

### In this policy, the least frequently used block will be replaced by the missed block when the cache is full. Each block will hold a counter that counts the frequency of the reference of each block. Figure 4 illustrates the working process of this policy:

In the pattern illustrated by figure4, when D is referenced, the counter of D will increment 1. When F is referenced, because E hold the least counter, E will be replaced by F. And the counter of F will be initialized to 1.

***Random replacement (RR)***

This policy randomly chooses a block as the victim when the missed block needs an empty space. Figure 5 illustrates the working process of this policy:

In the pattern illustrated by figure 5, the reference sequence is A,B,C,D,E,B,D,F. when the E is referenced, because the cache is full, the RR randomly choose block B as the victim and replace B with E.

**Three Promising Replacement Policies:**

***Re-reference Interval Prediction (RRIP)***

This policy is introduced in 2010 by a few experts. The authors mainly introduced two RRIP policies: Static RRIP (SRRIP), Dynamic RRIP (DRRIP) (Aamer, J., Kevin, B. T., Simon, C. S. J., Joel, E., 2010, p.60-p.71).

Figure 6(a) presents a typical stack access pattern that repeats N times. As recency-friendly access pattern, every block in this pattern has a near-immediate re-reference interval. That means LRU, not other replacement policies, will favor this pattern.

Figure 6(b) presents an access pattern in which k working sets repeat for N times. LRU will be suitable if the cache is capable of holding all the working sets, that is, cache size is bigger than k. However, if the working sets k is bigger than cache size, LRU will give 0 cache hit because of the cache thrashing. For this patter, the optimal replacement policy should preserve part of the working sets, but LRU cannot achieve this.

Figure 6(c) presents an infinite streaming access pattern. For this access pattern, the block that appeared before will not appear in the later sequence, so any replacement policy doesn’t work for this pattern.

Figure 6(d) presents a mixed access pattern, which combines the recency-friendly access pattern, scan access pattern and thrashing access pattern. For this access pattern, if k+m is less than the size of the cache, then LRU works well. But if k+m is bigger than the size of the cache, the LRU will discard the working sets that will be referenced after the scan pattern. This will degrade the performance of the cache.

*SRRIP:*

In order to improve the LRU so that it can work well in the thrashing pattern and mixed pattern, SRRIP was developed. In this policy, each block holds an M-bits counter that store its reference prediction value (RRPV). Every time when a cache miss occurs, this policy will choose the block whose RRPV is (2^M -1) to be replaced by the missed block whose RRPV is set as

(2^M – 2). If there is no such block exists in the cache, then the RRPV of each block will be increment by 1 until one victim is found. Every time when a cache hit occurs, this policy will minus the RRPV of the block being referenced by 1, unless its RRPV is 0 (Aamer, J., Kevin, B. T., Simon, C. S. J., Joel, E., 2010, p.61).

Figure 7 illustrates how the LRU, NRU, and SRRIP work in the mixed access pattern, and why the SPPIP outperform LRU and NRU when working in mixed access pattern. We can see that SRRIP will learn and update the RRPV of the block during the access pattern. This strategy not only prevents the blocks with distance re-reference interval polluting the cache but also allows RRIP more time to learn and improve the re-reference prediction.

According to Aamer et al. (2010, p.65), for active working set size w (w<A), associativity A and scan length L, M-bit SRRIP is scan-resistance when:

L <= (2^M – 1) \* (A – w)

*DRRIP:*

SRRIP is scan-resistant, but when the re-reference interval of all blocks are bigger than the size of the cache, the SRRIP will cause cache thrashing and give 0 cache hit. In order to solve this problem, Aamer et al. first introduced Bimodal RRIP (2010, p.65), in which a large part of the cache blocks will be assigned distant re-reference interval (i.e., 2^M – 1) prediction, while a small part of the cache blocks will be assigned long re-reference interval (i.e., 2^M – 2). The BRRIP is thrashing-resistant.

But for non-thrashing access pattern, the BRRIP will largely degrade the performance of the cache. So, the Aamer et al. (2010) introduce dynamic RRIP policy (p.65), in which the cache will dynamically choose the appropriate policy between the SRRIP and BRRIP. A part of the cache blocks serves as the monitor of these two replacement policies. The DRRIP will apply the policy whose performance in the monitor blocks is better to the remained cache blocks.

The experts design some experiment to demonstrate the efficiency of the SRRIP and DRRIP. According to Aamer et al. (2010*), SRRIP and DRRIP outperform LRU by an average of 4% and 10% on a single-core processor with a 16-way 2MB LLC. While on a 4-core CMP with a 16-way 8MB shared LLC, SRRIP and DRRIP outperform LRU by an average of 7% and 9%. RRIP also outperforms LFU, the state-of-the-art scan-resistant replacement algorithm by 2.5%. For the cache configurations under study, RRIP requires 2X hardware than LRU and 2.5X less hardware than LFU (p.70, italics originated).*

***Evict Write (EW LRU)***

This policy considers the different influence on cache performance between write and read operations. Write is considered less important than read to the overall throughput. Because the write-block has been buffered and the execution can continue even before the write-block arrive at the main memory. According to this principle, Marius et al. (2016) developed a replacement policy which evicts the victim blocks in the following order:

1. Write-only blocks
2. Read-write blocks
3. Read-only blocks

Each block needs two bits to store the write or non-write, read or non-read information. The principle that EW LRU follows is that a block that was read at least one time will be read again in the future. So, it is better for them to stay in the cache to receive future hits. The strategy is to reduce the execution time by reducing the read misses.

Algorithm 1 (showed in appendixes section) is portable to other LRU-like policies. For example, if this algorithm is extended to SRRIP policy, the EW SRRIP will compare the predicted re-reference interval within the block sets of write-only, read-write, or read-only, and then evict the biggest re-reference interval block in the write-only block set.

Figure 8 illustrates the difference between the LRU and EW LRU. The left side of the picture shows the contents of the cache, the right side of this picture shows the sequence of access pattern. In step 5, the cache is full, so the LRU or EW LRU need to choose a victim block to be replaced. LRU chooses the least recently used block A to be replaced by E. While EW LRU chooses the write-only block C to be replaced by E. The victim decisions of following steps are almost same like the former one. For the last two steps, cache miss occurs in LRU cache, while cache hit occurs in EW LRU cache.

Marius et al. (2016) used some benchmarks to test the EW LRU policy. And the experimental results (Marius Geanta, Lavinia Ghica, Nicolae Tapus, 2016) show that *the EW LRU obtain a maximum 26% better execution time than LRU in a quad core simulation and 2.9% average speedup improvement for the parallel benchmarks tested. In addition, it is demonstrated that the proposed enhancements are portable and can be extended from basic LRU*

*to more advanced policies such as RRIP. Finally, scalability was tested by running a simulation of an eight-core processor, which got 30% speedups with EW LRU.*

***Recency Frequency Replacement (RFR)***

According to Akash et al. (2014), this policy combines LRU and LFU. The main steps applying this policy are:

1. Weighing LRU and LFU
2. Fusion of LRU and LFU
3. Predict the line to be replaced

Algorithm 2 (showed in appendixes section) demonstrates how to weighing LRU and LFU (K.M. A., Akash, S., Divyalakshmi, G., Monica, S. C., 2014).

Stated by Akash et al. (2014), the RFR policy makes the victim decision based on the weight RF values, which is determined by the following equation:

Weight(RF) = Weight\_LRU \* C(LRU) + Weight\_LFU \* C(LFU).

Where the C(LRU) and C(LFU) are priority constants for recency and frequency respectively.

Akash et al. (2014) used CUBEMACH simulator to conduct a comparative analysis of various replacement policies.And the experimental results show that by testing different SPECCPU2000 benchmarks, RFR policy gives around 9% better performance in terms of miss ratio when compared with LRU, LFU, and FIFO (K.M. A., Akash, S., Divyalakshmi, G., Monica, S. C., 2014).

**Conclusion**

Cache replacement policy plays an important role in improving the cache performance. The conventional replacement policies such as FIFO, LIFO, RR and etc. are easy to implement but not optimal solutions for block replacement. Until now, LRU is the most commonly used policy because it achieves a balance between the implementation complexity and cache performance. In order to improve the cache performance further, especially for the last level cache, the experts developed some very promising replacement policies basing on LRU. RRIP outperform LRU by an average of at least 5%, and it performs better in the multicore environment. EW LRU outperforms LRU up to 26%, which prove that the EW policy could be extended to various replacement policies such as LFU and RRIP. RFR achieves 9% less missing ratio than LRU, but RFR requires more hardware overhead which needs to be further optimized.

**References**

Aamer, J., Kevin, B. T., Simon, C. S. J., Joel, E. (2010, June 19-23). High performance cache replacement using re-reference interval prediction (RRIP). *Association for Computing Machinery,* 38(3), 60-71. Doi: [10.1145/1815961.1815971](https://doi.org/10.1145/1815961.1815971)

Marius Geanta, Lavinia Ghica, Nicolae Tapus. (2016, September 8-10). Leverage cache replacement policy in multicore processors, 2016 IEEE 12th International Conference on ICCP, Cluj-Napoca, Romania, 2016.Cluj-Napoca, Romania: IEEE. Doi: [10.1109/ICCP.2016.7737182](https://doi-org.proxygw.wrlc.org/10.1109/ICCP.2016.7737182)

K.M, A., Akash, S., Divyalakshmi, G., Monica, S.C. (2014, September 24-27). A hybrid cache replacement policy for heterogeneous multi-cores, 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), New Delhi, India, 2014. New Delhi, India: IEEE. Doi: [10.1109/ICACCI.2014.6968209](https://doi-org.proxygw.wrlc.org/10.1109/ICACCI.2014.6968209)

**Appendixes**

*Algorithm 1:*

This is the Eviction Decision Algorithm for EW Policy (Marius Geanta, Lavinia Ghica, Nicolae Tapus, 2016)

Procedure\_find\_eviction\_block( cahce\_set )

Lru\_blcok\_read = lru\_blcok\_write\_only = lru\_block\_read\_write = None

for block in cache\_set do

if was\_read( block ) then

if was\_written( block ) then

if last\_reuse( block ) < last\_reuse( lru\_block\_read\_write ) then

lru\_block\_read\_write = block

else if last\_reuse ( block ) < last\_reuse ( lru\_block\_read ) then

lru\_block\_read = block

end if

else

if last\_reuse( block ) < last\_eruse( lru\_blcok\_write\_only ) then

lru\_block\_write\_only = block

end if

end if

end for

if lru\_block\_write is None then

if lru\_block\_read\_write is None then

return lru\_block\_read

else

return lru\_block\_read\_write

end if

end if

return lru\_block\_write

end procedure

*Algorithm 2:*

Weight\_Recency

{ for each hit in cache

If weight\_lru\_block > Weight\_LRU[current\_block]

weight\_lru\_block = weight\_lru\_block -1

Weight\_LRU[current\_block] = associativity -1

}

Weight\_Frequency

{

val\_LFU[n] ----counter value of all lines, n is the number of lines in a cache set

Weight\_LFU[n] ---weight of all lines

Weight\_LFU[n] = {0}

for( i=0;i<n-1;i++)

if (val\_LFU[i] > cal\_LFU[i+1])

Weight\_LFU[i] ++

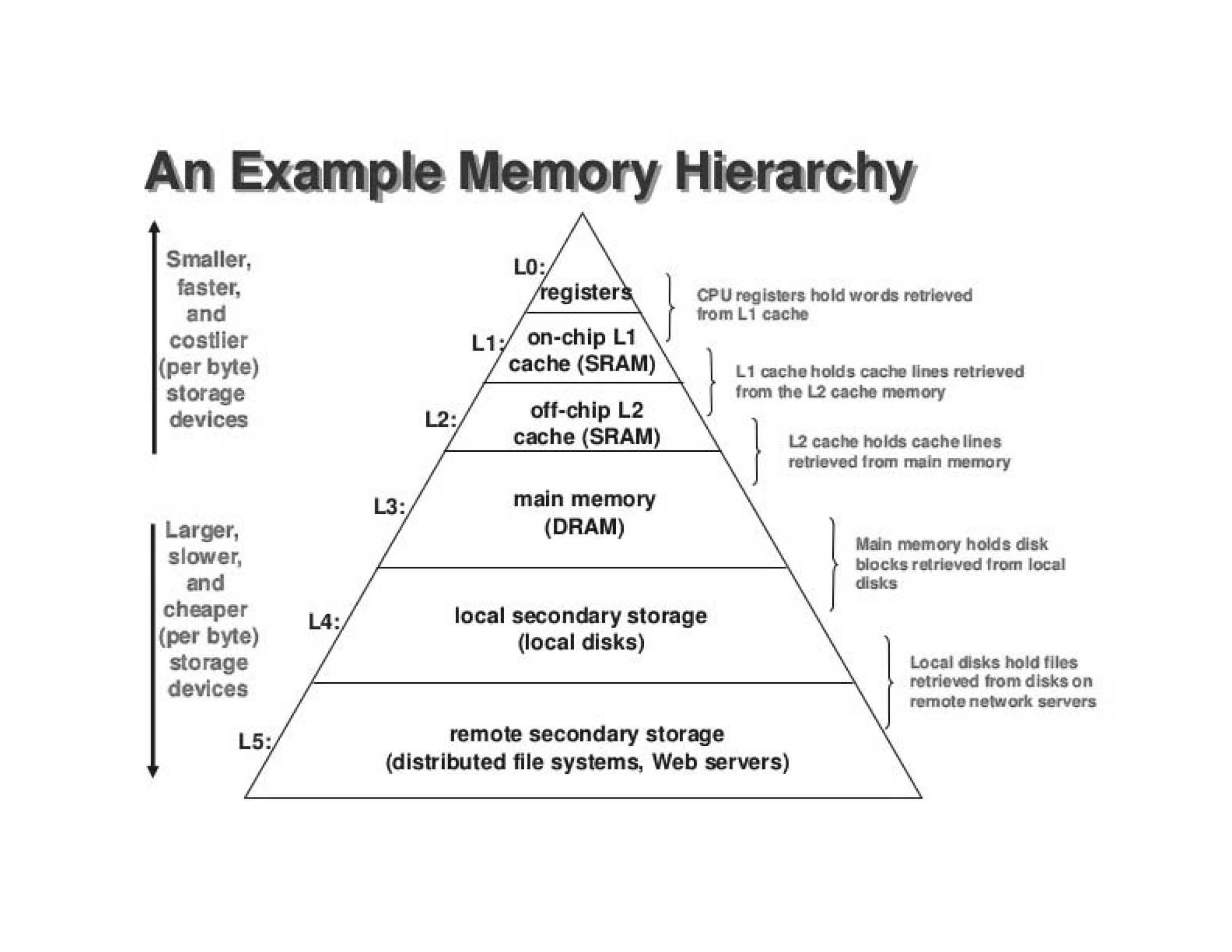
Else

Weight\_LFU[i+1] ++

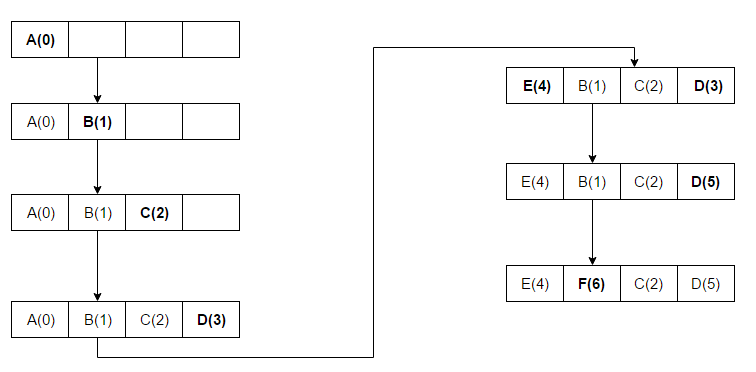
}

**Figure Captions and Figures**

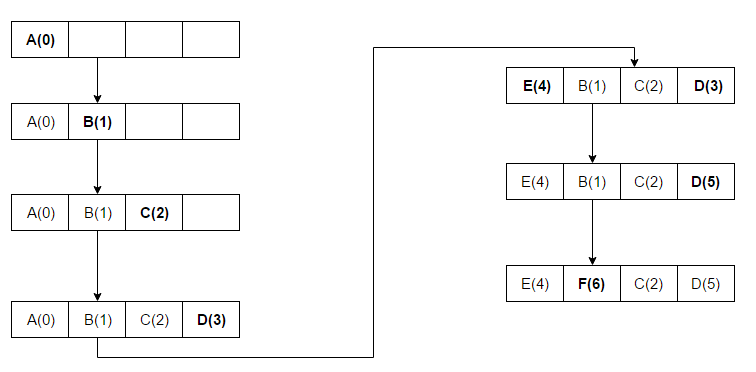
*Figure 1:* Memory Hierarchy. Adapted from "The memory hierarchy," by Abhishek Garg, 2015, SlideShare. Retrieved April 6, 2018, from <https://www.slideshare.net/abhishekgarg73/the-memory-hierarchy>.



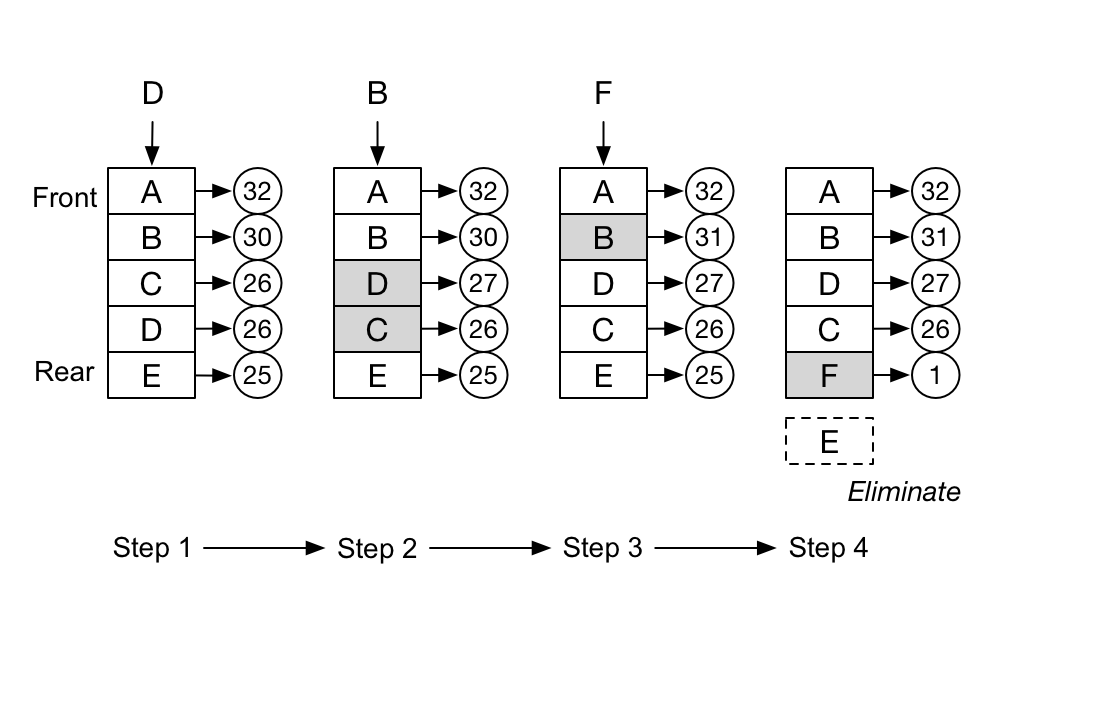
*Figure 2:* Lru example. Adapted from “Cache replacement policies," *Wikipedia*. Retrieved April 6, 2018, from <https://en.wikipedia.org/wiki/Cache_replacement_policies>.



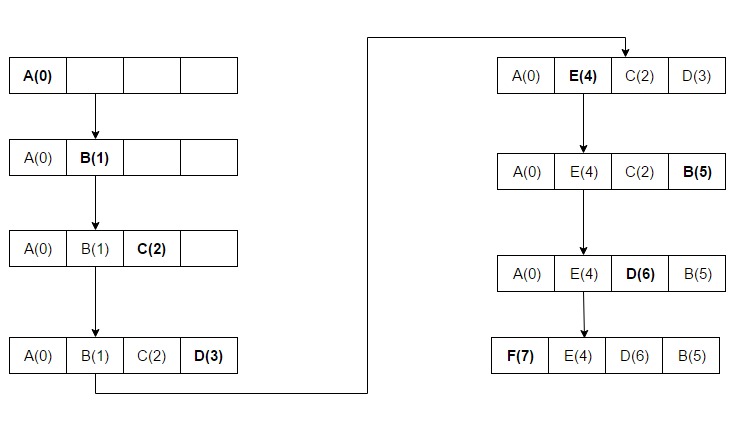
*Figure 3*: Mru example. Adapted from “Cache replacement policies," *Wikipedia*. Retrieved April 6, 2018, from <https://en.wikipedia.org/wiki/Cache_replacement_policies>.



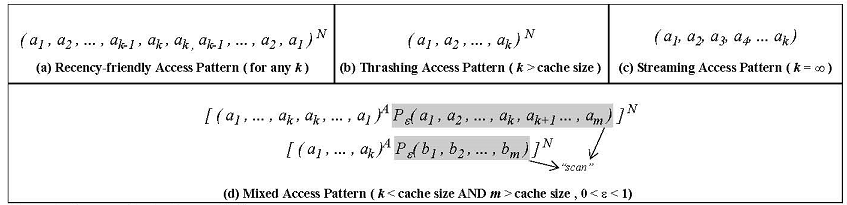
*Figure 4*: LFU Cache Elimination Process. Adapted from “LRU and LFU Cache Algorithms", *RiXu Online*. Retrieved April 6, 2018, from <https://xuri.me/2016/08/13/lru-and-lfu-cache-algorithms.html>.



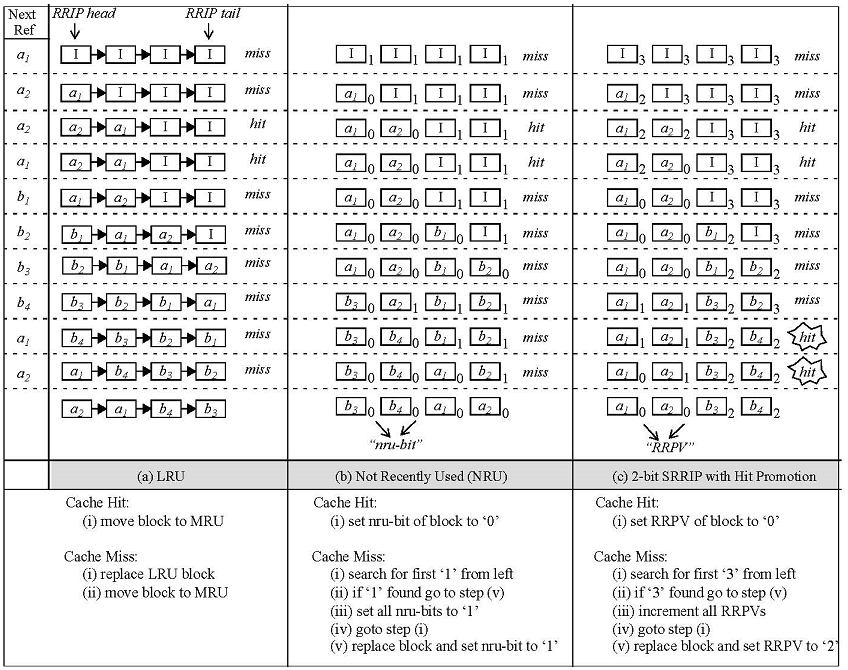
*Figure 5*: RR example. Adapted from “Cache replacement policies," *Wikipedia*. Retrieved April 6, 2018, from <https://en.wikipedia.org/wiki/Cache_replacement_policies>.



*Figure 6*: Common Cache Access Patterns. Adapted from “ High performance cache replacement using re-reference interval prediction (RRIP)”, by Aamer, J., Kevin, B. T., Simon, C. S. J., Joel, E., 2010, June 19, *ACM*, 38(3), p.61.



*Figure 7*: Behavior of LRU, NRU, and SRRIP for a Mixed Access Pattern. Adapted from “High performance cache replacement using re-reference interval prediction (RRIP)”, by Aamer, J., Kevin, B. T., Simon, C. S. J., Joel, E., 2010, June 19, *ACM*, 38(3), p.64.



*Figure 8*: LRU vs EW LRU Scenario in a 4-way Associative Cache. Adapted from “Leverage cache replacement policy in multicore processors”, by Marius Geanta, Lavinia Ghica, Nicolae Tapus, 2016, September 8, *2016 IEEE 12th International Conference on ICCP*.

