

# Cache replacement policies

In computing, **cache algorithms** (also frequently called **cache replacement algorithms** or **cache replacement policies**) are optimizing instructions—or **algorithms**—that a **computer program** or a hardware-maintained structure can follow in order to manage a cache of information stored on the computer. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

## Contents

**Overview**

**Policies**

- Bélády's Algorithm
- First In First Out (FIFO)
- Last In First Out (LIFO)
- Least Recently Used (LRU)
- Time aware Least Recently Used (TLRU)<sup>[5]</sup>
- Most Recently Used (MRU)
- Pseudo-LRU (PLRU)
- Random Replacement (RR)
- Segmented LRU (SLRU)
- Least-Frequently Used (LFU)
- Least Frequent Recently Used (LFRU) <sup>[11]</sup>
- LFU with Dynamic Aging (LFUDA)
- Low Inter-reference Recency Set (LIRS)
- Adaptive Replacement Cache (ARC)
- Clock with Adaptive Replacement (CAR)
- Multi Queue (MQ) caching algorithm|Multi Queue (MQ)
- Pannier: Container-based caching algorithm for compound objects

**See also**

**References**

**External links**

## Overview

The average memory reference time is<sup>[1]</sup>

$$T = m \times T_m + T_h + E$$

where

- T** = average memory reference time
- m** = miss ratio = 1 - (hit ratio)
- T<sub>m</sub>** = time to make a main memory access when there is a miss (or, with multi-level cache, average memory reference time for the next-lower cache)
- T<sub>h</sub>**= the latency: the time to reference the cache when there is a hit
- E** = various secondary effects, such as queuing effects in multiprocessor systems

There are two primary figures of merit of a cache: The latency, and the hit rate. There are also a number of secondary factors affecting cache performance.<sup>[1]</sup>

The "hit ratio" of a cache describes how often a searched-for item is actually found in the cache. More efficient replacement policies keep track of more usage information in order to improve the hit rate (for a given cache size).

The "latency" of a cache describes how long after requesting a desired item the cache can return that item (when there is a hit). Faster replacement strategies typically keep track of less usage information—or, in the case of direct-mapped cache, no information—to reduce the amount of time required to update that information.

Each replacement strategy is a compromise between hit rate and latency.

Hit rate measurements are typically performed on **benchmark** applications. The actual hit ratio varies widely from one application to another. In particular, video and audio streaming applications often have a hit ratio close to zero, because each bit of data in the stream is read once for the first time (a compulsory miss), used, and then never read or written again. Even worse, many cache algorithms (in particular, LRU) allow this streaming data to fill the cache, pushing out of the cache information that will be used again soon (cache pollution).<sup>[2]</sup>

Other things to consider:

- Items with different cost: keep items that are expensive to obtain, e.g. those that take a long time to get.
- Items taking up more cache: If items have different sizes, the cache may want to discard a large item to store several smaller ones.
- Items that expire with time: Some caches keep information that expires (e.g. a news cache, a DNS cache, or a web browser cache). The computer may discard items because they are expired. Depending on the size of the cache no further caching algorithm to discard items may be necessary.

Various algorithms also exist to maintain cache coherency. This applies only to situation where *multiple* independent caches are used for the *same* data (for example multiple database servers updating the single shared data file).

# Policies

## Bélády's Algorithm

The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm. Since it is generally impossible to predict how far in the future information will be needed, this is generally not implementable in practice. The practical minimum can be calculated only after experimentation, and one can compare the effectiveness of the actually chosen cache algorithm.

|                 |   |   |   |   |   |   |   |   |   |   |
|-----------------|---|---|---|---|---|---|---|---|---|---|
| Access Sequence | 5 | 0 | 1 | 2 | 0 | 3 | 1 | 2 | 5 | 2 |
| Frame1          | 5 | 5 | 5 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Frame 2         |   | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 |
| Frame 3         |   |   | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 |
|                 | f | f | f | f |   | f |   |   | f |   |

At the moment when a page fault occurs, some set of pages is in memory. In the example, the sequence of '5', '0', '1' is accessed by Frame 1, Frame 2, Frame 3 respectively. Then when '2' is accessed, it replaces value '5', which is in frame 1 since it predicts that value '5' is not going to be accessed in the near future. Because a real-life general purpose operating system cannot actually predict when '5' will be accessed, Bélády's Algorithm cannot be implemented on such a system.

## First In First Out (FIFO)

Using this algorithm the cache behaves in the same way as a FIFO queue. The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.

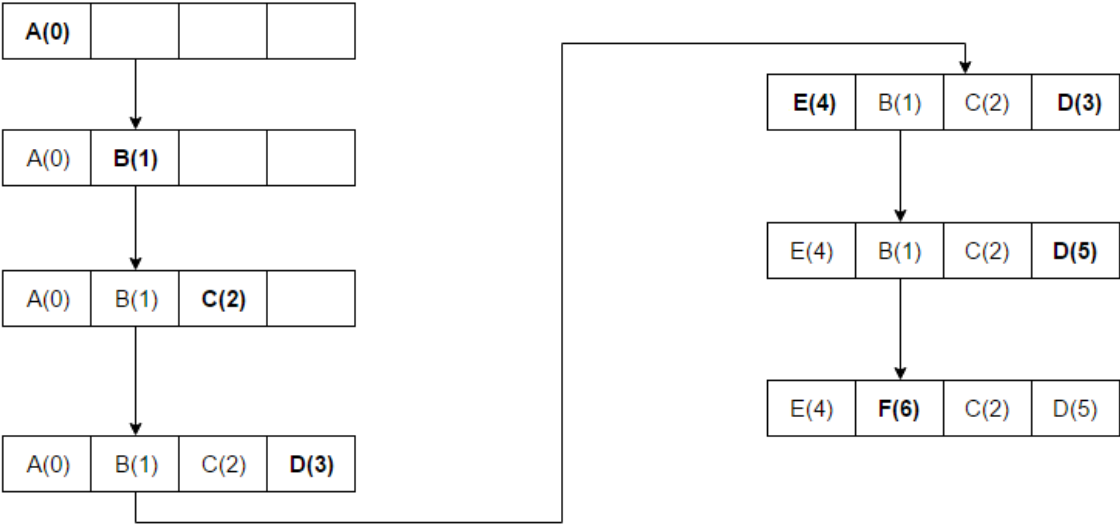
## Last In First Out (LIFO)

Using this algorithm the cache behaves in the exact opposite way as a FIFO queue. The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.

## Least Recently Used (LRU)

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes. LRU is actually a family of caching algorithms with members including 2Q by Theodore Johnson and Dennis Shasha,<sup>[3]</sup> and LRU/K by Pat O'Neil, Betty O'Neil and Gerhard Weikum.<sup>[4]</sup>

The access sequence for the below example is A B C D E D F.



In the above example once A B C D gets installed in the blocks with sequence numbers (Increment 1 for each new Access) and when E is accessed, it is a miss and it needs to be installed in one of the blocks. According to the LRU Algorithm, since A has the lowest Rank(A(0)), E will replace A.

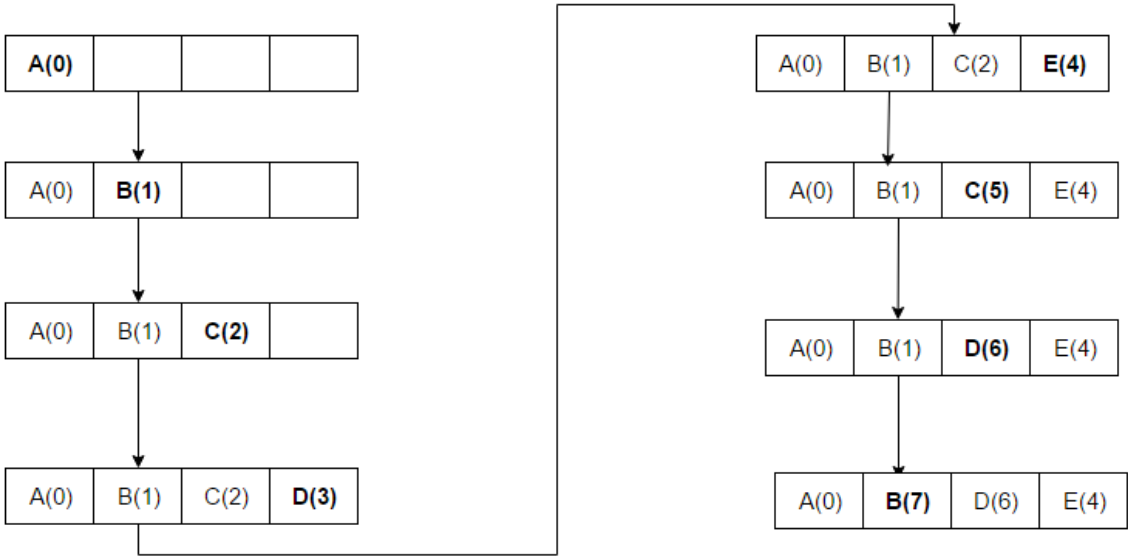
Time aware Least Recently Used (TLRU)<sup>[5]</sup>

The Time aware Least Recently Used (TLRU) is a variant of LRU designed for the situation where the stored contents in cache have a valid life time. The algorithm is suitable in network cache applications, such as Information-centric networking (ICN), Content Delivery Networks (CDNs) and distributed networks in general. TLRU introduces a new term: TTU (Time to Use). TTU is a time stamp of a content/page which stipulates the usability time for the content based on the locality of the content and the content publisher announcement. Owing to this locality based time stamp, TTU provides more control to the local administrator to regulate in network storage. In the TLRU algorithm, when a piece of content arrives, a cache node calculates the local TTU value based on the TTU value assigned by the content publisher. The local TTU value is calculated by using a locally defined function. Once the local TTU value is calculated the replacement of content is performed on a subset of the total content stored in cache node. The TLRU ensures that less popular and small life content should be replaced with the incoming content.

Most Recently Used (MRU)

Discards, in contrast to LRU, the most recently used items first. In findings presented at the 11th VLDB conference, Chou and DeWitt noted that "When a file is being repeatedly scanned in a [Looping Sequential] reference pattern, MRU is the best replacement algorithm."<sup>[6]</sup> Subsequently, other researchers presenting at the 22nd VLDB conference noted that for random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns) MRU cache algorithms have more hits than LRU due to their tendency to retain older data.<sup>[7]</sup> MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

The access sequence for the below example is A B C D E C D B.



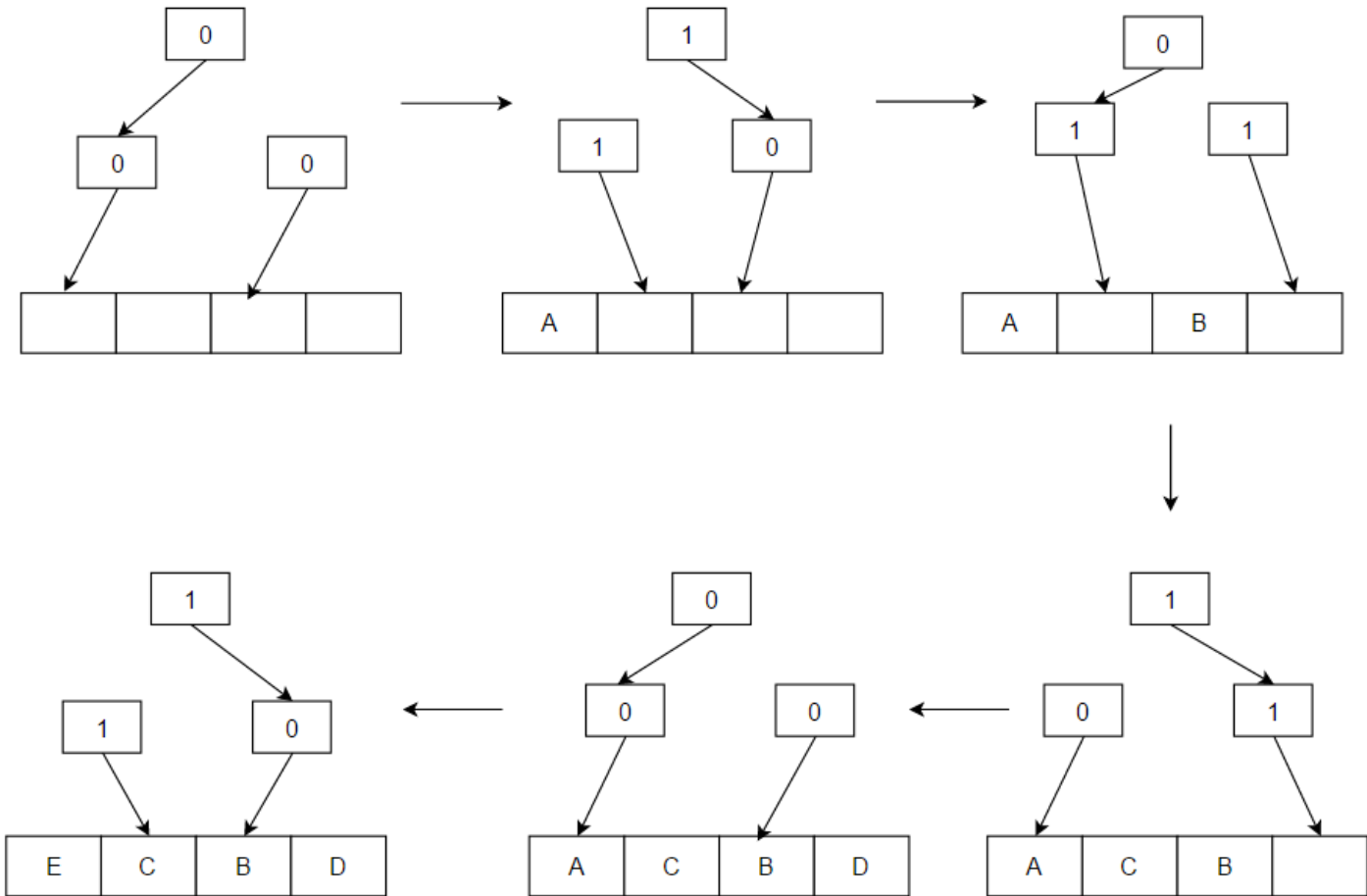
Here, A B C D are placed in the cache as there is still space available. At the 5th access E, we see that the block which held D is now replaced with E as this block was used most recently. Another access to C and at the next access to D, C is replaced as it was the block accessed just before D and so on.

Pseudo-LRU (PLRU)

For CPU caches with large associativity (generally >4 ways), the implementation cost of LRU becomes prohibitive. In many CPU caches, a scheme that almost always discards one of the least recently used items is sufficient. So many CPU designers choose a PLRU algorithm which only needs one bit per cache item to work. PLRU typically has a slightly worse miss ratio, has a slightly better latency, uses slightly less power than LRU and lower overheads compared to LRU.

In the following example, it is clearly shown how Bits work as a binary tree of 1-bit pointers that point to the less recently used subtree. Following the pointer chain to the leaf node identifies the replacement candidate. Upon an access all pointers in the chain from the accessed way's leaf node to the root node are set to point to subtree that does not contain the accessed way.

The access sequence is A B C D E.

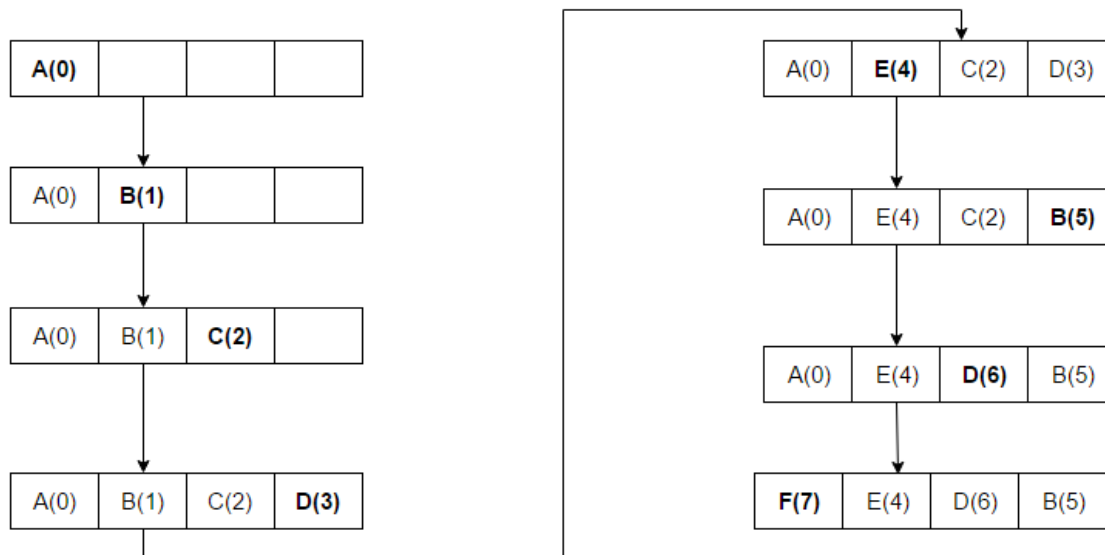


The principle here is simple to understand if we only look at the arrow pointers. When there is an access to a value say 'A' and the we cannot find it in the cache then load it from memory and **place it at the block where the arrows are pointing** go from top to bottom and when you place that block **make the arrows point away** from that block go from bottom to top. In the above example we see how 'A' was placed followed by 'B', 'C' and 'D'. Then as the cache became full 'E' replaced 'A' as that was where the arrows were pointing at that time. On the next access, the block where 'B' is being held will be replaced.

Random Replacement (RR)

Randomly selects a candidate item and discards it to make space when necessary. This algorithm does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors.<sup>[8]</sup> It admits efficient stochastic simulation.<sup>[9]</sup>

The access sequence for the below example is A B C D E B D F



### Segmented LRU (SLRU)

SLRU cache is divided into two segments, a probationary segment and a protected segment. Lines in each segment are ordered from the most to the least recently accessed. Data from misses is added to the cache at the most recently accessed end of the probationary segment. Hits are removed from wherever they currently reside and added to the most recently accessed end of the protected segment. Lines in the protected segment have thus been accessed at least twice. The protected segment is finite, so migration of a line from the probationary segment to the protected segment may force the migration of the LRU line in the protected segment to the most recently used (MRU) end of the probationary segment, giving this line another chance to be accessed before being replaced. The size limit on the protected segment is an SLRU parameter that varies according to the I/O workload patterns. Whenever data must be discarded from the cache, lines are obtained from the LRU end of the probationary segment.<sup>[10]</sup>

### Least-Frequently Used (LFU)

Counts how often an item is needed. Those that are used least often are discarded first. This works very similar to LRU except that instead of storing the value of how recently a block was accessed, we store the value of how many times it was accessed. So of course while running an access sequence we will replace a block which was used least number of times from our cache. E.g., if A was used (accessed) 5 times and B was used 3 times and others C and D were used 10 times each, we will replace B.

### Least Frequent Recently Used (LFRU) <sup>[11]</sup>

The Least Frequent Recently Used (LFRU) cache replacement scheme combines the benefits of LFU and LRU schemes. LFRU is suitable for 'in network' cache applications, such as Information-centric networking (ICN), Content Delivery Networks (CDNs) and distributed networks in general. In LFRU, the cache is divided into two partitions called privileged and unprivileged partitions. The privileged partition can be defined as a protected partition. If content is highly popular it is pushed into privileged partition. If it is require replacing content from privileged partition, the replacement is done as follows: LFRU evicts content from unprivileged partition, push content from privileged partition to unprivileged partition, and finally insert new content in privileged partition. In the above procedure the LRU is used for the privileged partitions and approximated LFU (ALFU) scheme is used for the unprivileged partition; hence together is called LFRU. The basic idea is to filter out the locally popular contents with ALFU scheme and push the popular contents to one of the privileged partition.

### LFU with Dynamic Aging (LFUDA)

A variant called LFU with Dynamic Aging (LFUDA) that uses dynamic aging to accommodate shifts in the set of popular objects. It adds a cache age factor to the reference count when a new object is added to the cache or when an existing object is re-referenced. LFUDA increments the cache ages when evicting blocks by setting it to the evicted object's key value. Thus, the cache age is always less than or equal to the minimum key value in the cache.<sup>[12]</sup> Suppose when an object was frequently accessed in the past and now it becomes unpopular, it will remain in the cache for a long time thereby preventing the newly or less popular objects from replacing it. So this Dynamic aging is introduced to bring down the count of such objects thereby making them eligible for replacement. The advantage of LFUDA is it reduces the cache pollution caused by LFU when cache sizes are very small. When Cache sizes are large few replacement decisions are sufficient and cache pollution will not be a problem.

### Low Inter-reference Recency Set (LIRS)

A page replacement algorithm with an improved performance over LRU and many other newer replacement algorithms. This is achieved by using reuse distance as a metric for dynamically ranking accessed pages to make a replacement decision. LIRS effectively address the limits of LRU by using recency to evaluate Inter-Reference Recency (IRR) for making a replacement decision. The algorithm was developed by Song Jiang and Xiaodong Zhang.

|      |   |   |   |   |   |   |   |   |   |    | Recency | IRR |
|------|---|---|---|---|---|---|---|---|---|----|---------|-----|
| A5   |   |   |   |   |   |   |   | X |   |    | 0       | INF |
| A4   |   | X |   |   |   |   | X |   |   |    | 2       | 3   |
| A3   |   |   |   | X |   |   |   |   |   |    | 4       | INF |
| A2   |   |   | X |   | X |   |   |   |   |    | 3       | 1   |
| A1   | X |   |   |   |   | X |   | X |   |    | 1       | 1   |
| TIME | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |         |     |

In the above figure, "x" represents that a block is accessed at time t. Suppose if block A1 is accessed at time 1 then Recency will become 0 since this is the first accessed block and IRR will be 1 since it predicts that A1 will be accessed again in time 3. In the time 2 since A4 is accessed, the recency will become 0 for A4 and 1 for A1 because A4 is the most recently accessed Object and IRR will become 4 and it will go on. At time 10, the LIRS algorithm will have two sets LIR set = {A1, A2} and HIR set = {A3, A4, A5}. Now at time 10 if there is access to A4, miss occurs. LIRS algorithm will now evict A5 instead of A2 because of its largest recency.

### Adaptive Replacement Cache (ARC)

Constantly balances between LRU and LFU, to improve the combined result.<sup>[13]</sup> ARC improves on SLRU by using information about recently evicted cache items to dynamically adjust the size of the protected segment and the probationary segment to make the best use of the available cache space. Adaptive replacement algorithm is explained with the example.<sup>[14]</sup>

### Clock with Adaptive Replacement (CAR)

Combines the advantages of Adaptive Replacement Cache (ARC) and CLOCK. CAR has performance comparable to ARC, and substantially outperforms both LRU and CLOCK. Like ARC, CAR is self-tuning and requires no user-specified magic parameters. It uses 4 doubly linked lists: two clocks T1 and T2 and two simple LRU lists B1 and B2. T1 clock stores pages based on "recency" or "short term utility" whereas T2 stores pages with "frequency" or "long term utility". T1 and T2 contain those pages that are in the cache, while B1 and B2 contain pages that have recently been evicted from T1 and T2 respectively. The algorithm tries to maintain the size of these lists  $B1 \approx T2$  and  $B2 \approx T1$ . New pages are inserted in T1 or T2. If there is a hit in B1 size of T1 is increased and similarly if there is a hit in B2 size of T1 is decreased. The adaptation rule used has the same principle as that in ARC, invest more in lists that will give more hits when more pages are added to it.

### Multi Queue (MQ) caching algorithm|Multi Queue (MQ)

The Multi Queue Algorithm or MQ was developed to improve the performance of second level buffer cache for e.g. a server buffer cache. It is introduced in a paper by Zhou, Philbin, and Li.<sup>[15]</sup> The MQ cache contains an  $m$  number of LRU queues:  $Q_0, Q_1, \dots, Q_{m-1}$ . Here, the value of  $m$  represents a hierarchy based on the lifetime of all blocks in that particular queue. For example, if  $j > i$ , blocks in  $Q_j$  will have a longer lifetime than those in  $Q_i$ . In addition to these there is another history buffer  $Q_{out}$ , a queue which maintains a list of all the Block Identifiers along with their access frequencies. When  $Q_{out}$  is full the oldest identifier is evicted. Blocks stay in the LRU queues for a given lifetime, which is defined dynamically by the MQ algorithm to be the maximum temporal distance between two accesses to the same file or the number of cache blocks, whichever is larger. If a block has not been referenced within its lifetime, it is demoted from  $Q_i$  to  $Q_{i-1}$  or evicted from the cache if it is in  $Q_0$ . Each queue also has a maximum access count; if a block in queue  $Q_i$  is accessed more than  $2^i$  times, this block is promoted to  $Q_{i+1}$  until it is accessed more than  $2^{i+1}$  times or its lifetime expires. Within a given queue, blocks are ranked by the recency of access, according to LRU.<sup>[16]</sup>

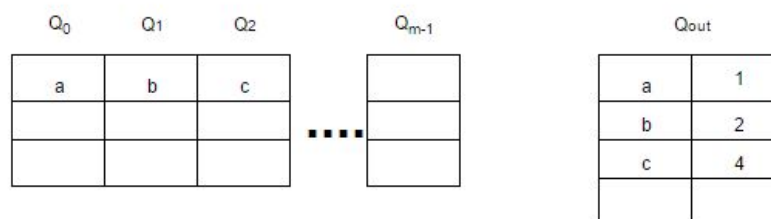


Fig.

We can see from Fig. how the  $m$  LRU queues are placed in the cache. Also see from Fig. how the  $Q_{out}$  stores the block identifiers and their corresponding access frequencies.  $a$  was placed in  $Q_0$  as it was accessed only once recently and we can check in  $Q_{out}$  how  $b$  and  $c$  were placed in  $Q_1$  and  $Q_2$  respectively as their access frequencies are 2 and 4. The queue in which a block is placed is dependent on access frequency( $f$ ) as  $\log_2(f)$ . When the cache is full, the first block to be evicted will be the head of  $Q_0$  in this case  $a$ . If  $a$  is accessed one more time it will move to  $Q_1$  below  $b$ .

### Pannier: Container-based caching algorithm for compound objects

Pannier<sup>[17]</sup> is a container-based flash caching mechanism that identifies divergent (heterogeneous) containers where blocks held therein have highly varying access patterns. Pannier uses a priority-queue based survival queue structure to rank the containers based on their survival time, which is proportional to the live data in the container. Pannier is built based on Segmented LRU (S2LRU), which segregates hot and cold data. Pannier also uses a multi-step feedback controller to throttle flash writes to ensure flash lifespan.

## See also

- Cache-oblivious algorithm
- Locality of reference
- Distributed cache

## References

- Alan Jay Smith. "Design of CPU Cache Memories". Proc. IEEE TENCON, 1987. [1] (<http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-357.pdf>)
- Paul V. Bolotoff. "Functional Principles of Cache Memory" ([http://alasir.com/articles/cache\\_principles/](http://alasir.com/articles/cache_principles/)). 2007.
- <http://www.vldb.org/conf/1994/P439.PDF>
- O'Neil, Elizabeth J.; O'Neil, Patrick E.; Weikum, Gerhard (1993). "The LRU-K Page Replacement Algorithm for Database Disk Buffering" (<http://doi.acm.org/10.1145/170035.170081>). *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. New York, NY, USA: ACM: 297–306. doi:10.1145/170035.170081 (<https://doi.org/10.1145%2F170035.170081>). ISBN 0-89791-592-5.
- Bilal, Muhammad; et al. "Time Aware Least Recent Used (TLRU) Cache Management Policy in ICN" (<http://ieeexplore.ieee.org/document/6779016/>). *IEEE 16th International Conference on Advanced Communication Technology (ICACT)*.
- Hong-Tai Chou and David J. DeWitt. *An Evaluation of Buffer Management Strategies for Relational Database Systems*. (<http://www.vldb.org/conf/1985/P127.PDF>) VLDB, 1985.
- Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. *Semantic Data Caching and Replacement*. (<http://www.vldb.org/conf/1996/P330.PDF>) VLDB, 1996.
- ARM Cortex-R series processors manual (<http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexr/index.html>)
- An Efficient Simulation Algorithm for Cache of Random Replacement Policy [2] (<http://www.springerlink.com/index/L324G2U075540681.pdf>)
- Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. In *Computer*, 1994.
- Bilal, Muhammad; et al. "A Cache Management Scheme for Efficient Content Eviction and Replication in Cache Networks" (<https://arxiv.org/abs/1702.04078>). *IEEE Access*.
- <https://arxiv.org/pdf/1001.4135.pdf>
- Nimrod Megiddo and Dharmendra S. Modha. *ARC: A Self-Tuning, Low Overhead Replacement Cache*. ([http://www.usenix.org/events/fast03/tech/full\\_papers/megiddo/megiddo.pdf](http://www.usenix.org/events/fast03/tech/full_papers/megiddo/megiddo.pdf)) FAST, 2003.
- <http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html>
- Yuanyuan Zhou, James Philbin, and Kai Li. *The Multi-Queue Replacement Algorithm for Second Level Buffer Caches*. (<http://static.usenix.org/event/usenix01/zhou.html>) USENIX, 2002.
- Eduardo Pinheiro , Ricardo Bianchini, Energy conservation techniques for disk array-based servers, Proceedings of the 18th annual international conference on Supercomputing, June 26-July 01, 2004, Malo, France
- Cheng Li, Philip Shilane, Fred Douglass and Grant Wallace. *Pannier: A Container-based Flash Cache for Compound Objects*. (<http://dl.acm.org/citation.cfm?id=2814734>) ACM/IFIP/USENIX Middleware, 2015.

## External links

- Definitions of various cache algorithms ([http://www.usenix.org/events/usenix01/full\\_papers/zhou/zhou\\_html/node3.html](http://www.usenix.org/events/usenix01/full_papers/zhou/zhou_html/node3.html))
- Fully associative cache (<http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Memory/fully.html>)
- Set associative cache (<http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Memory/set.html>)
- Direct mapped cache (<http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Memory/direct.html>)
- Slides on various page replacement schemes including LRU ([http://www.powershow.com/view/95163-NzkyO/4\\_4\\_Page\\_replacement\\_algorithms\\_powerpoint\\_ppt\\_presentation](http://www.powershow.com/view/95163-NzkyO/4_4_Page_replacement_algorithms_powerpoint_ppt_presentation))
- Caching algorithm for flash/SSDs (<http://dl.acm.org/citation.cfm?id=2814734>)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Cache\_replacement\_policies&oldid=820802062"

**This page was last edited on 16 January 2018, at 18:08.**

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.