An experimental comparison of cache algorithms

Trausti Saemundsson
Research Methodology, Reykjavik University

November 21, 2012

Abstract

Computers store data in a hierarchy of memories ranging from expensive fast memories to cheap and slow memories. It is common to store data in fast memories to try to prevent requests to the slower ones and this is referred to as caching. But when the fastest memory becomes full and a new element must be inserted, some other element has to be replaced. There are various approaches to decide which element to remove and these approaches are often called "cache algorithms" or "page replacement algorithms". Various algorithms have been studied and their performance often depends on the workload. This paper provides an overview of some much studied cache algorithms as well as a performance comparison of those algorithms by using real life request logs.

1 Introduction

To cache is to store data from a slow memory in a faster memory. This is done to minimize the requests to the slow memory and thus reduce memory access latency. Cache is used in various applications, in hard disks, web servers, databases and CPUs to name a few.

Computers contain many different memories and they form a hierarchy with respect to speed, cost and capacity. A typical CPU today contains at least 3 cache blocks, which are called L_1, L_2 and L3. These are the fastest memories. The Random Access Memory is slower than the CPU cache but faster than any Hard Disk Drive. An overview of access times in computer hardware is shown in Table 1. As can be seen from this overview, it is more than a million times faster to retrieve data from the L_1 cache than from a hard disk. It is therefore useful to cache frequently used data.

Table 1: Read access latency of computer hardware [1, 2]

L_1 cache reference	0.5 ns
L_2 cache reference	7 ns
Main memory reference	100 ns
Read 1 MB from memory	250,000 ns
Read 1 MB from a Solid State Drive	1,000,000 ns
Hard Disk Drive seek	10,000,000 ns
Read 1 MB from a Hard Disk Drive	30,000,000 ns

We use replacement policies (cache algorithms) to choose which element to remove from the cache when we need space for a new element. The element that the replacement policy chooses is then removed from the cache and this is called to *evict* the element from the cache. When we get a request to an element we first check whether the element is stored in the cache. If the element is in the cache we get a *cache hit*; otherwise a *cache miss* occurs and the element must be fetched from a slow memory. Cache algorithms that do not depend on knowing the future are called *online* algorithms and those that do called *offline* algorithms.

In this paper we will experimentally compare several common approaches to page replacement from the literature. In particular we focus on OPT, LRU, LFU, CLOCK and ARC. To compare the performance of cache algorithms, requests to objects in memory are needed. Such request log are often called *trace* files or *traces*. We will be using a set of real world traces from a big software vendor in our experiments.

2 Cache replacement algorithms

We now give an overview af cache algorithms, starting with an optimal one and working towards more practical ones.

2.1 Belady's algorithm

L. A. Belady described an optimal cache algorithm [3] (OPT) in 1966. When the cache is full and a new element must be inserted, OPT replaces the element that will not get a cache request for the longest period of time in the future. In practice cache sequences arrive in an online fashion and we cannot know the future requests. Therefore OPT cannot be used in practice but it is an important baseline with which to compare other algorithms.

2.2 Least Recently Used

One of the first cache replacement policies to be studied, dating back to at least 1965 [4], is the Least Recently Used algorithm (LRU) which replaces the element in the cache that was least recently used.

LRU handles some workloads well because recently used data are often reused in the near future. This algorithm is based on a similar idea as to OPT by using the requests to elements to determine which elements to keep in the cache, but LRU is an online algorithm as opposed to the offline nature of OPT.

LRU is usually implemented with a linked list and it therefore has a big drawback because moving elements to the most recently used position in the linked list at every request is expensive.

Another drawback of LRU is that many workloads use some elements more frequently than others and LRU does not make use of frequency information at all. LRU is also vulnerable to a scan of data, i.e., a sequence of requests to elements that are not requested again. So a scan may replace all the elements in the cache regardless of whether the elements will be used again or not.

2.3 Least Frequently Used

Another one of the first studied algorithms in caching is LFU which stands for Least Frequently Used and it dates back to at least 1971 [4]. LFU is not vulnerable to scans of requests and captures the frequency of workloads.

However, implementing LFU requires keeping track of the request frequency of each element in the cache. Usually this is done with some number of bits for each element and the number of bits limits how accurately the frequency is monitored. Regardless of the number of bits, keeping track of which element has the lowest frequency requires a priority queue and hence LFU has logarithmic complexity for all operations.

2.4 CLOCK

Introduced in 1968 [5] by F. J. Corbato, the CLOCK algorithm arranges cache elements in a circle and captures the recency of a workload like LRU with much less effort.

Every element has an associated bit called the recently used bit, which is set every time an element is accessed. The clock data structure has one "hand". When an element needs to be evicted from the cache, we check whether the recently used bit is set on the element E to which the hand points. If the recently used bit is not set on E, we replace E with the new element. However if the recently bit is set on E, we unset the bit on E and advance the hand to the next element. We repeat this until we find an element that does not have the recently used bit set. In the worst case the hand must traverse an entire circle and remove the element to which it pointed originally.

CLOCK handles recency like LRU without requiring locks in parallel systems. CLOCK also handles more requests per time unit because it does not move elements to a new position in a list at every request.

2.5 Adaptive Replacement Cache

The Adaptive Replacement Cache (ARC) algorithm introduced in 2003 [6] provides good performance on workloads where the access pattern is based on recency and frequency. To achieve this performance ARC combines LRU and LFU and is furthermore resistant to scans. It also adapts in real time to the recency or frequency access pattern of the workload.

ARC uses two lists L_1 and L_2. L_1 stores elements that have been seen only once recently but L_2 stores elements that have been seen at least twice, recently. It is useful to think of L_1 as the LRU list and L_2 as the LFU list. ARC then adaptively changes the number of elements stored in the cache from L_1 and L_2. This is done to meet the access pattern of the workload. The elements in L_1 and L_2 that are not in the cache are said to be in the *ghost cache*.

Since L_2 contains elements that have been seen at least twice recently it does not have logarithmic complexity on each request like LFU. Both L_1 and L_2 suffer from the same problem as LRU, every action requires a reordering of the elements in the list.

To address this issue another similar algorithm called Clock with Adaptive Replacement (CAR) [7] was proposed in 2004. It uses the ingenious solution from the CLOCK algorithm of using circular lists to lower the computational complexity. Both ARC and CAR are patented by IBM [8, 9].

2.6 Low Inter-reference Recency Set

Introduced in 2002, the Low Inter-reference Recency Set algorithm (LIRS) [10] is similar to LRU but does not use recency as a measure to evict elements but rather the distance between the last request and the second last request to an element. This distance is called the *reuse distance* of an element and LIRS evicts the element with the largest reuse distance. If an element has only been requested once, the reuse distance is defined to be infinite. LIRS is used [11] in the popular MySQL open source database.

For the same reason as why CLOCK was proposed to speed up LRU and CAR was proposed to speed up ARC, an algorithm called CLOCK-Pro was introduced in 2005 [12]. CLOCK-Pro is based on LIRS but uses circular lists. The CLOCK-Pro algorithm has been used in the NetBSD operating system [11] and in the Linux kernel [12].

3 Related work

In spite of the simplicity of CLOCK, it is claimed by S. Bansal and D.S. Modha [7] that the performance of CLOCK is close to that of LRU.

Results presented in [13] by the authors of ARC, N. Megiddo and D.S. Modha, show that ARC has better performance than LIRS. However, results presented by the authors of CLOCK-Pro in [12] show that CLOCK-Pro performs better than CAR on most traces. This discrepancy shows clearly that more research on the performance of those algorithms is required. On the other hand, all cache algorithms are based on different ideas and it is therefore logical to see different results on different traces

4 Implementation and simulations

In order to compare the performance of cache algorithms, they have to be implemented first.

When writing this paper the algorithms LFU, LRU, OPT, CLOCK were implemented by the author in Python [14]. The ARC implementation found in [15] was modified and all these implementations are available at `https://github.com/trauzti/cache`. The CLOCK implementation uses a list but LRU is implemented with a linked list. On the other hand, OPT and LFU are implemented with a heap which gives them a logarithmic complexity in the cache size on each request. The ARC implementation is not as fast and was the bottleneck in the simulations.

The papers on LIRS and CLOCK-Pro do not contain any pseudo-code so those algorithms were not implemented. To run simulations on those algorithms we obtained trace files from a large software vendor. These trace files were obtained by profiling real production systems. A short description of the trace files is as follows.

- **P1** is a 40MB file with 135,294 unique requests and 2,558,345 requests in total.
- **P4** is a 19MB file with 211,760 unique requests and 967,169 requests in total.
- **bank** is a 27MB file with 441,332 unique requests and 1,235,633 requests in total.
- **disk** is a 13MB file with 229,861 unique requests and 583,388 requests in total.

The algorithms LFU, LRU, OPT, CLOCK and ARC where run on these trace files with different cache sizes. The cache sizes used were 5, 10, 25, 50, 75, 100, 125 and 250. The size of ARC's ghost cache was set to be equal to the cache size.

The experiments were performed on a quad core computer running Ubuntu 12.04. To run the experiments in parallel, a script was used which can be found at `https://github.com/trauzti/cache/blob/master/run`.

5 Simulation results

The results from the simulations can be seen on Figures 1a, 1b, 1c and 1d. These figures show the hit

ratio as a function of the cache size. A short summary of the results for each algorithm is given below.

- **LRU:** LRU is better than LFU on traces **P1**, **P4** and **bank**.
- **CLOCK:** LRU and CLOCK have almost identical performance as claimed in [7]. This is noteworthy in light of CLOCK being only a one bit approximation to LRU and thus computationally much less demanding.
- **LFU:** LFU's performance is worst on all traces except on **disk** where LRU had the worst performance. The **disk** trace contains a lot of scans so this shows the scan-resistance of LFU and the scan vulnerability in LRU.
- **ARC:** ARC outperforms LRU, CLOCK and LFU in all cases. This is the same result as obtained by the authors of ARC in [13].
- **OPT:** The other algorithms are not close to the performance of OPT especially when considering trace **P4**. Hence, there is a massive opportunity for improvement.

6 Conclusions

The paper provided an introduction to some much studied cache algorithms. With the plethora of cache algorithms out there and contradictory claims on which algorithm is the best, it is worthwhile to run an independent performance analysis of these algorithms. We verified that CLOCK has similar performance to LRU and our results show that ARC consistently outperforms other algorithms with respect to hit ratio. There is still a great opportunity for improvement as ARC is not close to OPT.

The authors of CLOCK-Pro presented different results than from the authors of the competing algorithm CAR. So in future work we plan to assess the performance of LIRS and CLOCK-Pro as well, despite there being no pseudo-code provided in the respective papers.

7 Acknowledgments

Ymir Vigfusson (`http://www.ymsir.com`) supported the writing of this paper by providing the trace files used in the simulations.

References

[1] J. Dean, "Software engineering advice from building large-scale distributed systems" `http://research.google.com/people/jeff/stanford-295-talk.pdf`, 2007.

[2] `https://gist.github.com/2864150`. [Online; accessed 21-November-2012].

[3] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer" *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[4] N. Megiddo and D. S. Modha, "Adaptive Replacement Cache" `http://www-vlsi.stanford.edu/smart_memories/protected/meetings/spring2004/arc-fast.pdf`, 2003. [Online; accessed 21-November-2012].

[5] F. J. Corbato, "A paging experiment with the multics system" *MIT Project MAC Report MAC-M-384*, May 1968.

[6] N. Megiddo and D. Modha, "ARC: A self-tuning, low overhead replacement cache" in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 115–130, 2003.

[7] S. Bansal and D. Modha, "CAR: Clock with adaptive replacement" in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 187–200, 2004.

[8] N. Megiddo and D. Modha, "System and method for implementing an adaptive replacement cache policy" February 7 2006. US Patent 6,996,676.

[9] S. Bansal and D. Modha, "Method and system of clock with adaptive cache replacement and temporal filtering" September 30 2004. US Patent App. 10/955,201.

[10] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance" in *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, pp. 31–42, ACM, 2002.

[11] `http://www.ece.eng.wayne.edu/~sjiang/`. [Online; accessed 18-November-2012].

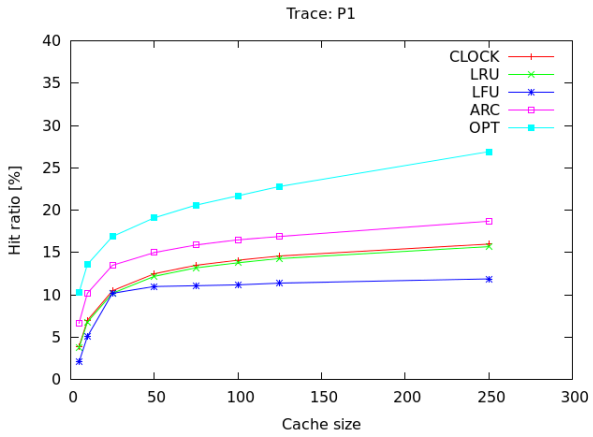
[12] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: an effective improvement of the CLOCK replacement" in *Proceedings of the annual conference on*

USENIX Annual Technical Conference, pp. 35–35, 2005.

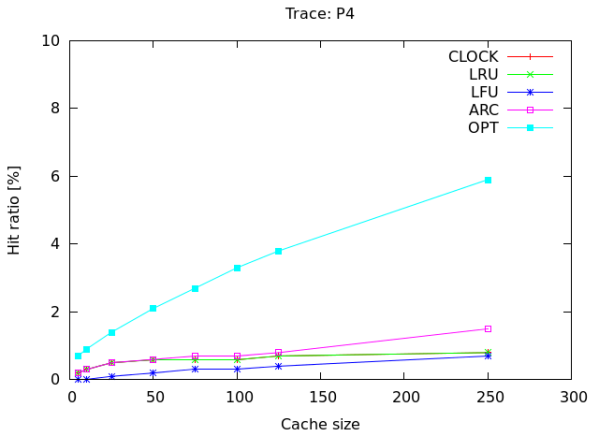
[13] N. Megiddo and D. Modha, "Outperforming LRU with an adaptive replacement cache algorithm" *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[14] G. van Rossum, "Python Programming Language – Official Website" `http://www.python.org/`, November 2012. [Online; accessed 18-November-2012].

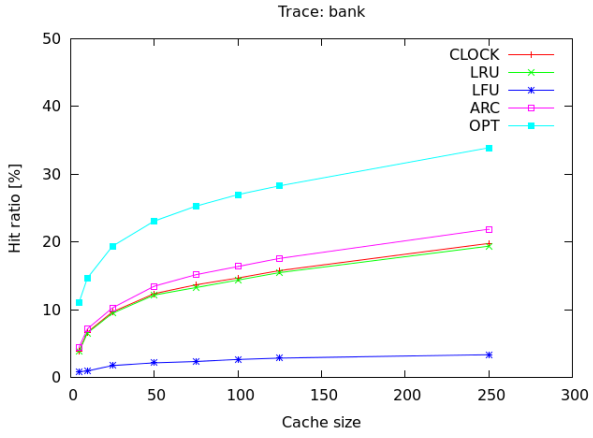
[15] E. Casteleijn, "Adaptive Replacement Cache in Python (Python recipe)" `http://code.activestate.com/recipes/576532/`, November 2012. [Online; accessed 16-November-2012].



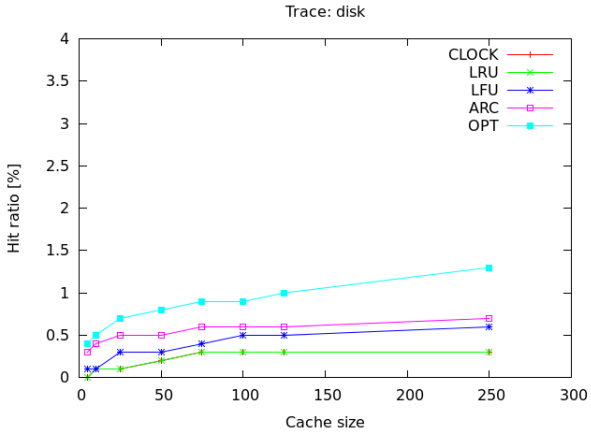
(a) Performance with trace: **P1**



(b) Performance with trace: **P4**



(c) Performance with trace: **bank**



(d) Performance with trace: **disk**

Figure 1: Results from simulations