Volume 4, Issue 9, September 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

CAR: Clock with Adaptive Replacement

Indu Bala
CSE, MRIU,
India

Abstract— CAR is a cache replacement policy dating back to 1968 .It was discribed as a low-complexity approximation to LRU. In each cache hit, the policy LRU needs to move the accessed item to the most recently used position, at which point, to ensure consistency and correctness, it serializes cache hits behind a single global lock. CLOCK removes this lock contention, and, hence, can support high concurrency and high throughput environments for example vir-tual memory (for example, Multics,) and databases (for example, DB2).CLOCK and LRU have own disadvantages, here we are removing by the CAR. As a result we are getting high performance .
As our main focus, to propose a simple and elegant new algorithm, namely, CLOCK with Adaptive Replacement (CAR), it has some advantages over CLOCK: (i) it is scan-resistant; (ii) it is self-tuning and it adaptively and dynamically captures the "recency" and "frequency" features of a workload; (iii) it uses essentially the same primitives as CLOCK, and, hence, is low-complexity and amenable to a high-concurrency implementation; and (iv) it outperforms CLOCK across a wide-range of cache sizes and workloads. The algorithm CAR is accepted by the Adaptive Replacement Cache (ARC) algorithm, and inherits virtually all advantages of ARC including its high performance, but does not serialize cache hits behind a single global lock.

Keywords— Caching, LRU, CLOCK, ARC, CAR

I. INTRODUCTION

A. CACHING AND DEMAND PAGING

Modern computational infrastructure is rich in exam- ples of memory hierarchies where a fast, but expensive main ("cache") memory is placed in front of a cheap, but slow auxiliary memory. Caching algorithms used to manage the contents of the cache so as to improve the overall performance. In particular, cache algorithms are of tremendous interest in databases (for example, DB2), virtual memory management in operating systems (for example, LINUX), storage systems (for example, IBM ESS, EMC Symmetrix, Hitachi Lightning), etc., where cache is RAM and the auxiliary memory is a disk subsystem. In this paper, we study the generic cache replacement problem and will not concentrate on any specific appli- cation. For correctness, we assume that both the cache and the auxiliary memory are managed in discrete, uniformly-sized units called "pages". If a requested page is present in the cache, then it can be served as a "cache hit". On the other hand, if a requested page is not present in the cache, then it must be fetched from the auxiliary memory resulting in a "cache miss". Mostly, latency on a cache miss is significantly higher than that on a cache hit. Hence, caching algorithms focus on improving the hit ratio.

B. LRU: Advantages and Disadvantages

The main advantages are:It is simple to implement. It captures recency feature and is a locality of reference.
and disadvantages:
D1 On every hit to a cache page it must be moved to the most recently used (MRU) position.
D2 In a virtual memory setting, the overhead of moving a page to the MRU position–on every page hit–is unacceptable [3].
D3 While LRU captures the "recency" features of a workload , it does not capture and exploit the "frequency" features of a workload [5]. Usually, if some pages are often re-requested, but the temporal distance between consecutive requests is larger than the cache size, then LRU cannot take advantage of such pages with "long-term utility".
D4 LRU can be easily polluted by a scan, that is, by a sequence of one-time use only page requests leading to lower performance.

C. CLOCK

Frank Corbat´o (who later went on to win the ACM Turing Award) introduced CLOCK [6] as a one-bit approximation to LRU:
CLOCK removes disadvantages D1 and D2 of LRU. The algorithm CLOCK maintains a "page reference bit" with every page. When a page is first brought into the cache, its page reference bit is set to zero. The pages in the cache are organized as a circular buffer known as a clock. On a hit to a page, its page reference bit is set to one. Replacement is done by moving a clock hand through the circular buffer. The clock hand can only replace a page with page reference bit set to zero. However, while the clock hand is traversing to find the victim page, if it encounters a page with page

reference bit of one, then it resets the bit to zero. Since, on a page hit, there is no need to move the page to the MRU position, no serialization of hits occurs. Moreover, in virtual memory applications, the page reference bit can be turned on by the hardware. Furthermore, performance of CLOCK is usually quite comparable to LRU [3], [4].

D. Adaptive Replacement Cache

Now a days developers efforting to remove disadvantages of LRU, namely, Adaptive Replacement Cache (ARC), removes disadvantages D3 and D4 of LRU [10], [11]. The algo-rithm ARC is scan-resistant, exploits both the recency and the frequency features of the workload in a self- tuning fashion, has low space and time complexity, and outperforms LRU across a wide range of workloads and cache sizes. Furthermore, ARC which is self-tuning has performance comparable to a number of recent, state- of-the-art policies even when these policies are allowed the best, offline values for their tunable parameters .

E. Our Contribution

To elaborate, CLOCK removes disadvantages D1 and D2 of LRU, while ARC removes disadvantages D3 and D4 of LRU. In this paper, as our main focus, we present a simple new algorithm, namely, Clock with Adaptive Replacement (CAR), that removes all four disadvantages D1, D2, D3, and D4 of LRU. The basic idea is to maintain two clocks, say, T1 and T2, where T1 contains pages with "recency" or "short-term utility" and T2 contains pages with "frequency" or "long- term utility". New pages are first inserted in T1 and graduate to T2 upon passing a certain test of long-term utility. By using a certain precise history mechanism that remembers recently evicted pages from T1 and T2, we adaptively determine the sizes of these lists in a data-driven fashion. Using extensive trace-driven simulations, we demonstrate that CAR has performance comparable to ARC, and substantially outperforms both LRU and CLOCK. Most probably, like ARC, the algorithm CAR is self-tuning and requires no user-specified magic parameters.

II. EXISTING CLOCK WITH ADAPTIVE REPLACEMENT ALGORITHM

A. ADAPTIVE REPLACEMENT CACHE

Suppose that the cache can hold c pages. The policy ARC maintains a cache directory that contains 2c pages–c pages in the cache and c history pages. The cache directory of ARC, which was referred to as DBL in [10], maintains two lists: L1 and L2. The first list con- tains pages that have been seen only once recently, while the latter contains pages that have been seen at least twice recently. The list L1 is thought of as "recency" and L2 as "frequency". A more precise interpretation would have been to think of L1 as "short-term utility" and L2 as "long-term utility". The replacement policy for managing DBL is: Replace the LRU page in L1, if $|L1| = c$; otherwise, replace the LRU page in L2. The policy ARC builds on DBL by carefully selecting c pages from the 2c pages in DBL. The basic idea is to divide L1 into top T1 and bottom B1 and to divide L2 into top T2 and bottom B2. The pages in T1 and T2 are in the cache and in the cache directory, while the history pages in B1 and B2 are in the cache directory but not in the cache. The pages evicted from T1 (resp. T2) are put on the history list B1 (resp. B2). The algorithm sets a target size p for the list T1. The replacement policy is simple: Replace the LRU page in T1, if $|T1| \geq p$; otherwise, replace the LRU page in T2. The adaptation comes from the fact that the target size p is continuously varied in response to an observed workload. The adaptation rule is also simple: Increase p, if a hit in the history B1 is observed; similarly, decrease p, if a hit in the history B2 is observed. This completes our brief description of ARC.

B. CLOCK WITH ADAPTIVE REPLACEMENT

Our policy CAR is inspired by ARC. Finally, for the sake of consistency, we have chosen to use the same notation as that in [10] so as to facilitate an easy comparison of similarities and differences between the two policies. For a visual description of CAR, see Figure 1, and for a complete algorithmic specification, see Figure 2. We now explain the intuition behind the algorithm. For correctness, let c denote the cache size in pages. The policy CAR manage four doubly linked lists: T1, T2, B1, and B2. The lists T1 and T2 contain the pages in cache, while the lists B1 and B2 maintain history information about the recently evicted pages. For each page in the cache, that is, in T1 or T2, we will maintain a page reference bit that can be set to either one or zero. Let T^0_1 denote the pages in T1 with a page reference bit of zero and let T^1_1 denote the pages in T1 with a page reference bit of one. The lists T^0_1 and T^1_1 are introduced for expository reasons only–they will not be required explicitly in our algorithm. Not maintaining either of these lists or their sizes was a key insight that allowed us to simplify ARC to CAR. The precise definition of the four lists is as follows. Each page in T^0_1 and each history page in B1 has either been requested exactly once since its most recent removal from T1 ∪ T2 ∪ B1 ∪ B2 or it was requested only once (since inception) and was never removed from T1 ∪ T2 ∪ B1 ∪ B2. Each page in T^1_1 , each page in T2, and each history page in B2 has either been requested more than once since its most recent removal from T1∪T2∪B1∪B2, or was requested more than once and was never removed from T1 ∪ T2 ∪ B1 ∪ B2.

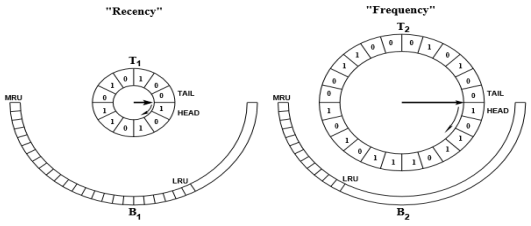

Fig.1 A visual description of CAR.

CLOCK, cache misses are still serialized behind a global lock to ensure correctness and consistency of the lists T1, T2, B1, and B2. This miss serialization can be somewhat mitigated by a free buffer pool. Our discussion of CAR is now complete.

INITIALIZATION: Set p = 0 and set the lists T1, B1, T2, and B2 to empty.
CAR(x)
INPUT: The requested page x.
1: if (x is in T1∪ T2) then /* cache hit */
2: Set the page reference bit for x to one.
3: else /* cache miss */
4: if (|T1| + |T2| = c) then
/* cache full, replace a page from cache */
5: replace()
/* cache directory replacement */
6: if ((x is not in B1∪ B2) and (|T1| + |B1| = c)) then
7: Discard the LRU page in B1.
8: elseif ((|T1| + |T2| + |B1| + |B2| = 2c) and (x is not in B1∪ B2)) then
9: Discard the LRU page in B2.
10: endif
11: endif
/* cache directory miss */
12: if (x is not in B1∪ B2) then
13: Insert x at the tail of T1. Set the page reference bit of x to 0.
/* cache directory hit */
14: elseif (x is in B1) then
15: Adapt: Increase the target size for the list T1as: p = min{p + max{1,|B2|/|B1|},c}
16: Move x at the tail of T2. Set the page reference bit of x to 0.
/* cache directory hit */
17: else /* x must be in B2*/
18: Adapt: Decrease the target size for the list T1 as: p = max{p − max{1,|B1|/|B2|},0}
19: Move x at the tail of T2. Set the page reference bit of x to 0.
20: endif
21: endif
replace()
22: found = 0
23: repeat
24: if (|T1| >= max(1,p)) then 25: if (the page reference bit of head page in T1is 0) then
26: found = 1;
27: Demote the head page in T1 and make it the MRU page in B1.
28: else
29: Set the page reference bit of head page in T1 to 0, and make it the tail page in T2.
30: endif
31: else
32: if (the page reference bit of head page in T2is 0), then
33: found = 1;
34: Demote the head page in T2 and make it the MRU page in B2.
35: else
36: Set the page reference bit of head page in T2 to 0, and make it the tail page in T2.
37: endif
38: endif
39: until (found)

Fig. 2. Algorithm for Clock with Adaptive Replacement. This algorithm is self-contained. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache directory. The first key point of the above algorithm is the simplicity of line 2, where cache hits are not serialized behind a lock and virtually no overhead is involved. The second key point is the continual adaptation of the target size of the list T1 in lines 16 and 19. The final key point is that the algorithm requires no magic, tunable parameters as input.

III. CONCLUSIONS

In this paper, by combining ideas and best features from CLOCK and ARC we have introduced a policy CAR that removes disadvantages D1, D2, D3 and D4 of LRU. CAR removes the cache hit serialization problem of LRU and ARC. CAR has a very low overhead on cache hits. CAR is self-tuning. The policy CAR requires no tunable, magic parameters. It has one tunable parameter p that balances between recency and frequency. The policy adaptively tunes this parameter– in response to an evolving workload–so as to increase the hit-ratio. A closer examination of the parameter p shows that it can fluctuate from recency (p = c) to frequency (p = 0) and back– all within a single workload. CAR is scan-resistant. A

scan is any sequence of one-time use requests. Such requests will be put on top of the list T1 and will eventually exit from the cache without polluting the high-quality pages in T2. More- over, in presence of scans, there will be relatively fewer hits in B1 as compared to B2. Finally, our adaptation rule will tend to further increase the size of T2 at the expense of T1, thus further decreasing the residency time of scan in even T1. CAR is high-performance. CAR outperforms LRU and CLOCK on a wide variety of traces and cache sizes, and has performance very comparable to ARC. CAR has low space overhead, typically, less that 1%. CAR is simple to implement. Please see Figure 2.

REFERENCES

[1] L. A. Belady, "A study of replacement algorithms for virtual storage computers," IBM Sys. J., vol. 5, no. 2, pp. 78–101, 1966.
[2] M. J. Bach, The Design of the UNIX Operating System. Engle- wood Cliffs, NJ: Prentice-Hall, 1986.
[3] A. S. Tanenbaum and A. S. Woodhull, Operating Systems: Design and Implementation. Prentice-Hall, 1997.
[4] A. Silberschatz and P. B. Galvin, Operating System Concepts. Reading, MA: Addison-Wesley, 1995.
[5] J. E. G. Coffman and P. J. Denning, Operating Systems Theory. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[6] F. J. Corbat´o, "A paging experiment with the multics system," in In Honor of P. M. Morse, pp. 217–228, MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.
[7] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita, "Starburst mid-flight: As the dust clears," IEEE Trans. Knowledge and Data Engineering, vol. 2, no. 1, pp. 143– 160, 1990.
[8] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarter- man, The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, 1996.
[9] H. Levy and P. H. Lipman, "Virtual memory management in the VAX/VMS operating system," IEEE Computer, pp. 35–41, March 1982.
[10] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA, pp. 115–130, 2003.
[11] N. Megiddo and D. S. Modha, "One up on LRU," ;login – The Magazine of the USENIX Association, vol. 28, pp. 7–11, August 2003.
[12] J. T. Robinson and M. V. Devarakonda, "Data cache man- agement using frequency-based replacement," in Proc. ACM SIGMETRICS Conf., pp. 134–142, 1990.
[13] A. J. Smith, "Bibliography on paging and related topics," Operating Systems Review, vol. 12, pp. 39–56, 1978.
[14] A. J. Smith, "Second bibliography for cache memories," Com- puter Architecture News, vol. 19, no. 4, 1991.
[15] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," J. ACM, vol. 18, no. 1, pp. 80–93, 1971.
[16] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in Proc. ACM SIGMOD Conf., pp. 297–306, 1993.
[17] T. Johnson and D. Shasha, "2Q: A low overhead high per- formance buffer management replacement algorithm," in Proc. VLDB Conf., pp. 297–306, 1994.
[18] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in Proc. ACM SIGMETRICS Conf., pp. 134–143, 1999.
[19] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2001.
[20] Y. Zhou and J. F. Philbin, "The multi-queue replacement algo- rithm for second level buffer caches," in Proc. USENIX Annual Tech. Conf. (USENIX 2001), Boston, MA, pp. 91–104, June 2001.
[21] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache perfor- mance," in Proc. ACM SIGMETRICS Conf., 2002.
[22] W. R. Carr and J. L. Hennessy, "WSClock – a simple and effective algorithm for virtual memory management," in Proc. Eighth Symp. Operating System Principles, pp. 87–95, 1981.
[23] H. T. Chou and D. J. DeWitt, "An evaluation of buffer manage- ment strategies for relational database systems," in Proceedings of the 11th International Conference on Very Large Databases, Stockholm, Sweden, pp. 127–141, 1985.
[24] A. J. Smith, "Sequentiality and prefetching in database systems," ACM Trans. Database Systems, vol. 3, no. 3, pp. 223–247, 1978.
[25] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the gener- alized clock buffer replacement scheme for database transaction processing," in ACM SIGMETRICS, pp. 35–46, 1992.
[26] U. Vahalia, UNIX Internals: The New Frontiers. Prentice Hall,

