CAR: Clock with Adaptive Replacement

Sorav Bansal[†] **and Dharmendra S. Modha**[‡]
[†]Stanford University, [‡]IBM Almaden Research Center
Emails: sbansal@stanford.edu, dmodha@us.ibm.com

***Abstract*— CLOCK is a classical cache replacement policy dating back to 1968 that was proposed as a low-complexity approximation to LRU. On every 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 eliminates this *lock contention*, and, hence, can support high concurrency and high throughput environments such as virtual memory (for example, Multics, UNIX, BSD, AIX) and databases (for example, DB2). Unfortunately, CLOCK is still plagued by disadvantages of LRU such as disregard for "frequency", susceptibility to scans, and low performance.

As our main contribution, we propose a simple and elegant new algorithm, namely, CLOCK with Adaptive Replacement (CAR), that has several 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 inspired 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. As our second contribution, we introduce another novel algorithm, namely, CAR with Temporal filtering (CART), that has all the advantages of CAR, but, in addition, uses a certain temporal filter to distill pages with long-term utility from those with only short-term utility.

I. Introduction

A. Caching and Demand Paging

Modern computational infrastructure is rich in examples of memory hierarchies where a fast, but expensive main ("cache") memory is placed in front of a cheap, but slow auxiliary memory. Caching algorithms 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 application. For concreteness, 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 quickly resulting in 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". Usually, latency on a cache miss is significantly higher than that on a cache hit. Hence, caching algorithms focus on improving the hit ratio.

Historically, the assumption of "demand paging" has been used to study cache algorithms. Under demand paging, a page is brought in from the auxiliary memory to the cache only on a cache miss. In other words, demand paging precludes speculatively pre-fetching pages. Under demand paging, the only question of interest is: When the cache is full, and a new page must be inserted in the cache, which page should be replaced? The best, offline cache replacement policy is Belady's MIN that replaces the page that is used farthest in the future [1]. Of course, in practice, we are only interested in online cache replacement policies that do not demand any prior knowledge of the workload.

B. LRU: Advantages and Disadvantages

A popular online policy imitates MIN by replacing the least recently used (LRU) page. So far, LRU and its variants are amongst the most popular replacement policies [2], [3], [4]. The advantages of LRU are that it is extremely simple to implement, has constant time and space overhead, and captures "recency" or "clustered locality of reference" that is common to many workloads. In fact, under a certain Stack Depth Distribution (SDD) assumption for workloads, LRU is the optimal cache replacement policy [5].

The algorithm LRU has many disadvantages:

D1 On every hit to a cache page it must be moved to the most recently used (MRU) position. In an asynchronous computing environment where multiple threads may be trying to move pages to the MRU position, the MRU position is protected by a lock to ensure consistency and correctness. This lock typically leads to a great amount of contention, since all cache hits are serialized behind this lock. Such contention is often unacceptable in high performance and high throughput environments such as virtual memory, databases, file systems, and storage controllers.

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, p. 282]. More generally, 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 Corbató (who later went on to win the ACM Turing Award) introduced CLOCK [6] as a one-bit approximation to LRU:

> "In the Multics system a paging algorithm has been developed that has the implementation ease and low overhead of the FIFO strategy and is an approximation to the LRU strategy. In fact, the algorithm can be viewed as a particular member of a class of algorithms which embody for each page a shift register memory length of k. At one limit of $k = 0$, the algorithm becomes FIFO; at the other limit as $k \to \infty$, the algorithm is LRU. The current Multics system is using the value of $k = 1$, ..."

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. For this reason, variants of CLOCK have been widely used in Multics [6], DB2 [7], BSD [8], AIX, and VAX/VMS [9]. The importance of CLOCK is further underscored by the fact that major textbooks on operating systems teach it [3], [4].

D. Adaptive Replacement Cache

A recent breakthrough generalization of LRU, namely, Adaptive Replacement Cache (ARC), removes disadvantages D3 and D4 of LRU [10], [11]. The algorithm 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 [10, Table V].

E. Our Contribution

To summarize, CLOCK removes disadvantages D1 and D2 of LRU, while ARC removes disadvantages D3 and D4 of LRU. In this paper, as our main contribution, 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, T_1 and T_2, where T_1 contains pages with "recency" or "short-term utility" and T_2 contains pages with "frequency" or "long-term utility". New pages are first inserted in T_1 and graduate to T_2 upon passing a certain test of long-term utility. By using a certain precise history mechanism that remembers recently evicted pages from T_1 and T_2, 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. Furthermore, like ARC, the algorithm CAR is self-tuning and requires no user-specified magic parameters.

The algorithms ARC and CAR consider two consecutive hits to a page as a test of its long-term utility. At upper levels of memory hierarchy, for example, virtual memory, databases, and file systems, we often observe two or more successive references to the same page fairly quickly. Such quick successive hits are not a guarantee of long-term utility of a pages. Inspired by the "locality filtering" principle in [12], we introduce another novel algorithm, namely, CAR with Temporal filtering (CART), that has all the advantages of CAR, but, imposes a more stringent test to demarcate between pages with long-term utility from those with only short-term utility.

We expect that CAR is more suitable for disk, RAID, storage controllers, whereas CART may be more suited to virtual memory, databases, and file systems.

F. Outline of the Paper

In Section II, we briefly review relevant prior art. In Sections III and IV, we present the new algorithms CAR and CART, respectively. In Section V, we present results of trace driven simulations. Finally, in Section VI, we present some discussions and conclusions.

II. Prior Work

For a detail bibliography of caching and paging work prior to 1990, see [13], [14].

A. LRU and LFU: Related Work

The Independent Reference Model (IRM) captures the notion of frequencies of page references. Under the IRM, the requests at different times are stochastically independent. LFU replaces the least frequently used page and is optimal under the IRM [5], [15] but has several drawbacks: (i) Its running time per request is logarithmic in the cache size. (ii) It is oblivious to recent history. (iii) It does not adapt well to variable access patterns; it accumulates stale pages with past high frequency counts, which may no longer be useful.

The last fifteen years have seen development of a number of novel caching algorithms that have attempted to combine "recency" (LRU) and "frequency" (LFU) with the intent of removing one or more disadvantages of LRU. Chronologically, FBR [12], LRU-2 [16], 2Q [17], LRFU [18], [19], MQ [20], and LIRS [21] have been proposed. For a detailed overview of these algorithms, see [19], [20], [10]. It turns out, however, that each of these algorithms leaves something to be desired, see [10]. The cache replacement policy ARC [10] seems to eliminate essentially all drawbacks of the above mentioned policies, is self-tuning, low overhead, scan-resistant, and has performance similar to or better than LRU, LFU, FBR, LRU-2, 2Q, MQ, LRFU, and LIRS–even when some of these policies are allowed to select the best, offline values for their tunable parameters–without any need for pre-tuning or user-specified magic parameters.

Finally, all of the above cited policies, including ARC, use LRU as the building block, and, hence, continue to suffer from drawbacks D1 and D2 of LRU.

B. CLOCK: Related Work

As already mentioned, the algorithm CLOCK was developed specifically for low-overhead, low-lock-contention environment.

Perhaps the oldest algorithm along these lines was First-In First-Out (FIFO) [3] that simply maintains a list of all pages in the cache such that *head* of the list is the oldest arrival and *tail* of the list is the most recent arrival. FIFO was used in DEC's VAX/VMS [9]; however, due to much lower performance than LRU, FIFO in its original form is seldom used today.

Second chance (SC) [3] is a simple, but extremely effective enhancement to FIFO, where a page reference bit is maintained with each page in the cache while maintaining the pages in a FIFO queue. When a page arrives in the cache, it is appended to the tail of the queue and its reference bit set to zero. Upon a page hit, the page reference bit is set to one. Whenever a page must be replaced, the policy examines the page at the head of the FIFO queue and replaces it if its page reference bit is zero otherwise the page is moved to the tail and its page reference bit is reset to zero. In the latter case, the replacement policy reexamines the new page at the head of the queue, until a replacement candidate with page reference bit of zero is found.

A key deficiency of SC is that it keeps moving pages from the head of the queue to the tail. This movement makes it somewhat inefficient. CLOCK is functionally identical to SC except that by using a circular queue instead of FIFO it eliminates the need to move a page from the head to the tail [3], [4], [6]. Besides its simplicity, the performance of CLOCK is quite comparable to LRU [22], [23], [24].

While CLOCK respects "recency", it does not take "frequency" into account. A generalized version, namely, GCLOCK, associates a counter with each page that is initialized to a certain value. On a page hit, the counter is incremented. On a page miss, the rotating clock hand sweeps through the clock decrementing counters until a page with a count of zero is found [24]. A analytical and empirical study of GCLOCK [25] showed that "its performance can be either better or worse than LRU". A fundamental disadvantage of GCLOCK is that it requires counter increment on every page hit which makes it infeasible for virtual memory.

There are several variants of CLOCK, for example, the two-handed clock [9], [26] is used by SUN's Solaris. Also, [6] considered multi-bit variants of CLOCK as finer approximations to LRU.

III. CAR

A. ARC: A Brief Review

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: L_1 and L_2. The first list contains pages that have been seen only once recently, while the latter contains pages that have been seen at least twice recently. The list L_1 is thought of as "recency" and L_2 as "frequency". A more precise interpretation would have been to think of L_1 as "short-term utility" and L_2 as "long-term utility". The replacement policy

for managing DBL is: Replace the LRU page in L_1, if $|L_1| = c$; otherwise, replace the LRU page in L_2. The policy ARC builds on DBL by carefully selecting c pages from the $2c$ pages in DBL. The basic idea is to divide L_1 into top T_1 and bottom B_1 and to divide L_2 into top T_2 and bottom B_2. The pages in T_1 and T_2 are in the cache and in the cache directory, while the history pages in B_1 and B_2 are in the cache directory but not in the cache. The pages evicted from T_1 (resp. T_2) are put on the history list B_1 (resp. B_2). The algorithm sets a target size p for the list T_1. The replacement policy is simple: Replace the LRU page in T_1, if $|T_1| \geq p$; otherwise, replace the LRU page in T_2. 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 B_1 is observed; similarly, decrease p, if a hit in the history B_2 is observed. This completes our brief description of ARC.

B. CAR

Our policy CAR is inspired by ARC. Hence, 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 concreteness, let c denote the cache size in pages. The policy CAR maintains four doubly linked lists: T_1, T_2, B_1, and B_2. The lists T_1 and T_2 contain the pages in cache, while the lists B_1 and B_2 maintain history information about the recently evicted pages. For each page in the cache, that is, in T_1 or T_2, we will maintain a page reference bit that can be set to either one or zero. Let T_1^0 denote the pages in T_1 with a page reference bit of zero and let T_1^1 denote the pages in T_1 with a page reference bit of one. The lists T_1^0 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_1^0 and each history page in B_1 has either been requested exactly once since its most recent removal from $T_1 \cup T_2 \cup B_1 \cup B_2$ or it was requested only once (since inception) and was never removed from $T_1 \cup T_2 \cup B_1 \cup B_2$.

Each page in T_1^1, each page in T_2, and each history page in B_2 has either been requested more than once since its most recent removal from $T_1 \cup T_2 \cup B_1 \cup B_2$, or was requested more than once and was never removed from $T_1 \cup T_2 \cup B_1 \cup B_2$.

Intuitively, $T_1^0 \cup B_1$ contains pages that have been seen exactly once recently whereas $T_1^1 \cup T_2 \cup B_2$ contains pages that have been seen at least twice recently. We roughly think of $T_1^0 \cup B_1$ as "recency" or "short-term utility" and $T_1^1 \cup T_2 \cup B_2$ as "frequency" or "long-term utility".

In the algorithm in Figure 2, for a more transparent exposition, we will think of the lists T_1 and T_2 as second chance lists. However, SC and CLOCK are the same algorithm that have slightly different implementations. So, in an actual implementation, the reader may wish to use CLOCK so as to reduce the overhead somewhat. Figure 1 depicts T_1 and T_2 as CLOCKs. The policy ARC employs a strict LRU ordering on the lists T_1 and T_2 whereas CAR uses a one-bit approximation to LRU, that is, SC. The lists B_1 and B_2 are simple LRU lists.

We impose the following invariants on these lists:

- I1 $0 \leq |T_1| + |T_2| \leq c$.
- I2 $0 \leq |T_1| + |B_1| \leq c$.
- I3 $0 \leq |T_2| + |B_2| \leq 2c$.
- I4 $0 \leq |T_1| + |T_2| + |B_1| + |B_2| \leq 2c$.
- I5 If $|T_1| + |T_2| < c$, then $B_1 \cup B_2$ is empty.
- I6 If $|T_1| + |B_1| + |T_2| + |B_2| \geq c$, then $|T_1| + |T_2| = c$.
- I7 Due to demand paging, once the cache is full, it remains full from then on.

The idea of maintaining extra history pages is not new, see, for example, [16], [17], [19], [20], [21], [10]. We will use the extra history information contained in lists B_1 and B_2 to guide a continual adaptive process that keeps readjusting the sizes of the lists T_1 and T_2. For this purpose, we will maintain a *target size* p for the list T_1. By implication, the target size for the list T_2 will be $c - p$. The extra history leads to a negligible space overhead.

The list T_1 may contain pages that are marked either one or zero. Suppose we start scanning the list T_1 from the head towards the tail, until a page marked as zero is encountered; let T_1' denote all the pages seen by such a scan, until a page with a page reference bit of zero is encountered. The list T_1' does not need to be constructed, it is defined with the sole goal of stating our cache replacement policy.

The cache replacement policy CAR is simple:

> If $T_1 \setminus T_1'$ contains p or more pages, then remove a page from T_1, else remove a page from $T_1' \cup T_2$.

For a better approximation to ARC, the cache replacement policy should have been: If T_1^0 contains p or more pages, then remove a page from T_1^0, else remove a page from $T_1^1 \cup T_2$. However, this would require maintaining the list T_1^0, which seems to entail a much higher overhead on a hit. Hence, we eschew the precision, and

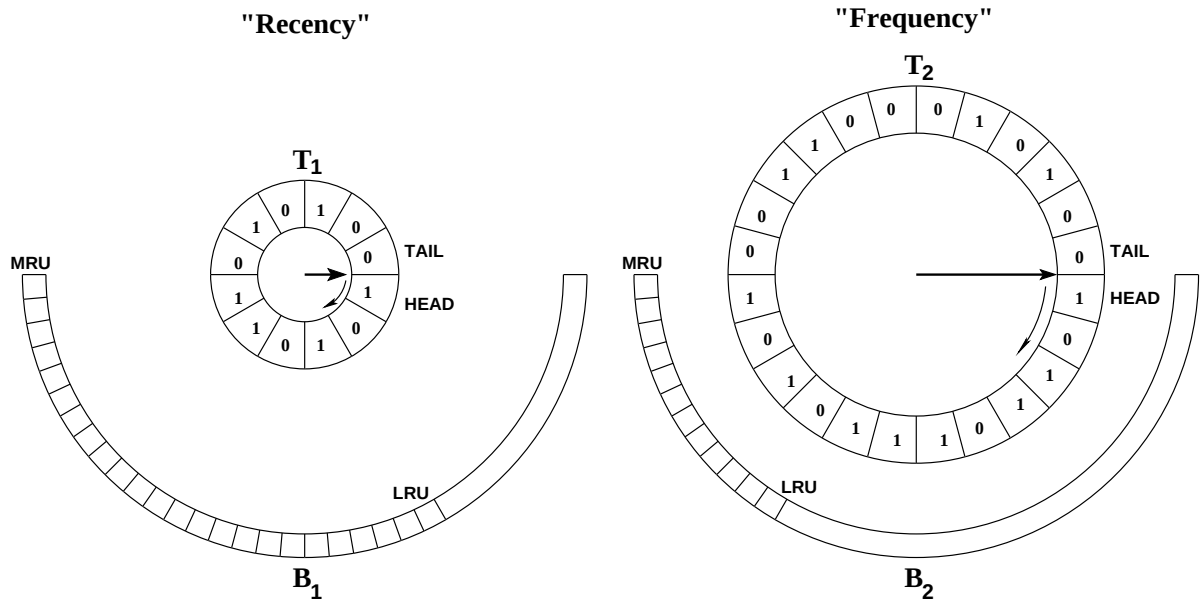

Fig. 1. A visual description of CAR. The CLOCKS T_1 and T_2 contain those pages that are in the cache and the lists B_1 and B_2 contain history pages that were recently evicted from the cache. The CLOCK T_1 captures "recency" while the CLOCK T_2 captures "frequency." The lists B_1 and B_2 are simple LRU lists. Pages evicted from T_1 are placed on B_1, and those evicted from T_2 are placed on B_2. The algorithm strives to keep B_1 to roughly the same size as T_2 and B_2 to roughly the same size as T_1. The algorithm also limits $|T_1| + |B_1|$ from exceeding the cache size. The sizes of the CLOCKs T_1 and T_2 are adapted continuously in response to a varying workload. Whenever a hit in B_1 is observed, the target size of T_1 is incremented; similarly, whenever a hit in B_2 is observed, the target size of T_1 is decremented. The new pages are inserted in either T_1 or T_2 immediately behind the clock hands which are shown to rotate clockwise. The page reference bit of new pages is set to 0. Upon a cache hit to any page in $T_1 \cup T_2$, the page reference bit associated with the page is simply set to 1. Whenever the T_1 clock hand encounters a page with a page reference bit of 1, the clock hand moves the page behind the T_2 clock hand and resets the page reference bit to 0. Whenever the T_1 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B_1. Whenever the T_2 clock hand encounters a page with a page reference bit of 1, the page reference bit is reset to 0. Whenever the T_2 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B_2.

go ahead with the above approximate policy where T_1' is used as an approximation to T_1^1.

The cache history replacement policy is simple as well:

> If $|T_1| + |B_1|$ contains exactly c pages, then remove a history page from B_1, else remove a history page from B_2.

Once again, for a better approximation to ARC, the cache history replacement policy should have been: If $|T_1^0| + |B_1|$ contains exactly c pages, then remove a history page from B_1, else remove a history page from B_2. However, this would require maintaining the size of T_1^0 which would require additional processing on a hit, defeating the very purpose of avoiding lock contention.

We now examine the algorithm in Figure 2 in detail.

Line 1 checks whether there is a hit, and if so, then line 2 simply sets the page reference bit to one. Observe that there is no MRU operation akin to LRU or ARC involved. Hence, cache hits are not serialized behind a lock and virtually no overhead is involved. The key insight is that the MRU operation is delayed until a replacement must be done (lines 29 and 36).

Line 3 checks for a cache miss, and if so, then line 4 checks if the cache is full, and if so, then line 5 carries out the cache replacement by deleting a page from either T_1 or T_2. We will dissect the cache replacement policy "replace()" in detail a little bit later.

If there is a cache miss (line 3), then lines 6-10 examine whether a cache history needs to be replaced. In particular, (line 6) if the requested page is totally new, that is, not in B_1 or B_2, and $|T_1| + |B_1| = c$ then (line 7) a page in B_1 is discarded, (line 8) else if the page is totally new and the cache history is completely full, then (line 9) a page in B_2 is discarded.

Finally, if there is a cache miss (line 3), then lines 12-20 carry out movements between the lists and also

carry out the adaptation of the target size for T_1. In particular, (line 12) if the requested page is totally new, then (line 13) insert it at the tail of T_1 and set its page reference bit to zero, (line 14) else if the requested page is in B_1, then (line 15) we increase the target size for the list T_1 and (line 16) insert the requested page at the tail of T_2 and set its page reference bit to zero, and, finally, (line 17) if the requested page is in B_2, then (line 18) we decrease the target size for the list T_1 and (line 19) insert the requested page at the tail of T_2 and set its page reference bit to zero.

Our adaptation rule is essentially the same as that in ARC. The role of the adaptation is to "invest" in the list that is most likely to give the highest hit per additional page invested.

We now examine the cache replacement policy (lines 22-39) in detail. The cache replacement policy can only replace a page with a page reference bit of zero. So, line 22 declares that no such suitable victim page to replace is yet found, and lines 23-39 keep looping until they find such a page.

If the size of the list T_1 is at least p and it is not empty (line 24), then the policy examines the head of T_1 as a replacement candidate. If the page reference bit of the page at the head is zero (line 25), then we have found the desired page (line 26), we now demote it from the cache and move it to the MRU position in B_1 (line 27). Else (line 28) if the page reference bit of the page at the head is one, then we reset the page reference bit to one and move the page to the tail of T_2 (line 29).

On the other hand, (line 31) if the size of the list T_1 is less than p, then the policy examines the page at the head of T_2 as a replacement candidate. If the page reference bit of the head page is zero (line 32), then we have found the desired page (line 33), and we now demote it from the cache and move it to the MRU position in B_1 (line 34). Else (line 35) if the page reference bit of the head page is one, then we reset the page reference bit to zero and move the page to the tail of T_2 (line 36).

Observe that while no MRU operation is needed during a hit, if a page has been accessed and its page reference bit is set to one, then during replacement such pages will be moved to the tail end of T_2 (lines 29 and 36). In other words, CAR approximates ARC by performing a delayed and approximate MRU operation during cache replacement.

While we have alluded to a multi-threaded environment to motivate CAR, for simplicity and brevity, our final algorithm is decidedly single-threaded. A true, real-life implementation of CAR will actually be based on a non-demand-paging framework that uses a free buffer pool of pre-determined size.

Observe that while cache hits are not serialized, like CLOCK, cache misses are still serialized behind a global lock to ensure correctness and consistency of the lists T_1, T_2, B_1, and B_2. This miss serialization can be somewhat mitigated by a free buffer pool.

Our discussion of CAR is now complete.

IV. CART

A limitation of ARC and CAR is that two consecutive hits are used as a test to promote a page from "recency" or "short-term utility" to "frequency" or "long-term utility". At upper level of memory hierarchy, we often observe two or more successive references to the same page fairly quickly. Such quick successive hits are known as "correlated references" [12] and are typically not a guarantee of long-term utility of a pages, and, hence, such pages can cause cache pollution–thus reducing performance. The motivation behind CART is to create a temporal filter that imposes a more stringent test for promotion from "short-term utility" to "long-term utility". The basic idea is to maintain a *temporal locality window* such that pages that are re-requested within the window are of short-term utility whereas pages that are re-requested outside the window are of long-term utility. Furthermore, the temporal locality window is itself an adaptable parameter of the algorithm.

The basic idea is to maintain four lists, namely, T_1, T_2, B_1, and B_2 as before. The pages in T_1 and T_2 are in the cache whereas the pages in B_1 and B_2 are only in the cache history. For simplicity, we will assume that T_1 and T_2 are implemented as Second Chance lists, but, in practice, they would be implemented as CLOCKs. The lists B_1 and B_2 are simple LRU lists. While we have used the same notation for the four lists, they will now be provided with a totally different meaning than that in either ARC or CAR.

Analogous to the invariants I1–I7 that were imposed on CAR, we now impose the same invariants on CART except that I2 and I3 are replaced, respectively, by:

I2' $\quad 0 \leq |T_2| + |B_2| \leq c.$
I3' $\quad 0 \leq |T_1| + |B_1| \leq 2c.$

As for CAR and CLOCK, for each page in $T_1 \cup T_2$ we will maintain a page reference bit. In addition, each page is marked with a *filter* bit to indicate whether it has *long-term utility* (say, "L") or only *short-term utility* (say, "S"). No operation on this bit will be required during a cache hit. We now detail manipulation and use of the filter bit. Denote by x a requested page.

1) Every page in T_2 and B_2 must be marked as "L".
2) Every page in B_1 must be marked as "S".
3) A page in T_1 could be marked as "S" or "L".
4) A head page in T_1 can only be replaced if its page reference bit is set to 0 and its filter bit is set to "S".

INITIALIZATION: Set $p = 0$ and set the lists T_1, B_1, T_2, and B_2 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 T1 as: 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 T1 is 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 T2 is 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 T_1 in lines 16 and 19. The **final key point** is that the algorithm requires no magic, tunable parameters as input.

5) If the head page in T_1 is of type "L", then it is moved to the tail position in T_2 and its page reference bit is set to zero.
6) If the head page in T_1 is of type "S" and has page reference bit set to 1, then it is moved to the tail position in T_1 and its page reference bit is set to zero.
7) A head page in T_2 can only be replaced if its page reference bit is set to 0.
8) If the head page in T_2 has page reference bit set to 1, then it is moved to the tail position in T_1 and its page reference bit is set to zero.
9) If $x \notin T_1 \cup B_1 \cup T_2 \cup B_2$, then set its type to "S."
10) If $x \in T_1$ and $|T_1| \geq |B_1|$, change its type to "L."
11) If $x \in T_2 \cup B_2$, then leave the type of x unchanged.
12) If $x \in B_1$, then x must be of type "S", change its type to "L."

When a page is removed from the cache directory, that is, from the set $T_1 \cup B_1 \cup T_2 \cup B_2$, its type is forgotten. In other words, a totally new page is put in T_1 and initially granted the status of "S", and this status is not upgraded upon successive hits to the page in T_1, but only upgraded to "L" if the page is eventually demoted from the cache and a cache hit is observed to the page while it is in the history list B_1. This rule ensures that there are two references to the page that are temporally separated by at least the length of the list T_1. Hence, the length of the list T_1 is the temporal locality window. The intent of the policy is to ensure that the $|T_1|$ pages in the list T_1 are the most recently used $|T_1|$ pages. Of course, this can only be done approximately given the limitation of CLOCK. Another source of approximation arises from the fact that a page in T_2, upon a hit, cannot immediately be moved to T_1.

While, at first sight, the algorithm appears very technical, the key insight is very simple: The list T_1 contains $|T_1|$ pages either of type "S" or "L", and is an approximate representation of "recency". The list T_2 contains remaining pages of type "L" that may have "long-term utility". In other words, T_2 attempts to capture useful pages which a simple recency based criterion may not capture.

We will adapt the temporal locality window, namely, the size of the list T_1, in a workload-dependent, adaptive, online fashion. Let p denote the target size for the list T_1. When p is set to the cache size c, the policy CART will coincide with the policy LRU.

The policy CART decides which list to delete from according to the rule in lines 36-40 of Figure 3. We also maintain a second parameter q which is the target size for the list B_1. The replacement rule for the cache history is described in lines 6-10 of Figure 3.

Let counters n_S and n_L denote the number of pages in the cache that have their filter bit set to "S" and "L", respectively. Clearly, $0 \leq n_S + n_L \leq c$, and, once the cache is full, $n_S + n_L = c$. The algorithm attempts to keep $n_S + |B_1|$ and $n_L + |B_2|$ to roughly c pages each.

The complete policy CART is described in Figure 3. We now examine the algorithm in detail.

Line 1 checks for a hit, and if so, line 2 simply sets the page reference bit to one. This operation is exactly similar to that of CLOCK and CAR and gets rid of the the need to perform MRU processing on a hit.

Line 3 checks for a cache miss, and if so, then line 4 checks if the cache is full, and if so, then line 5 carries out the cache replacement by deleting a page from either T_1 or T_2. We dissect the cache replacement policy "replace()" in detail later.

If there is a cache miss (line 3), then lines 6-10 examine whether a cache history needs to be replaced. In particular, (line 6) if the requested page is totally new, that is, not in B_1 or B_2, $|B_1| + |B_2| = c + 1$, and B_1 exceeds its target, then (line 7) a page in B_1 is discarded, (line 8) else if the page is totally new and the cache history is completely full, then (line 9) a page in B_2 is discarded.

Finally, if there is a cache miss (line 3), then lines 12-21 carry out movements between the lists and also carry out the adaptation of the target size for T_1. In particular, (line 12) if the requested page is totally new, then (line 13) insert it at the tail of T_1, set its page reference bit to zero, set the filter bit to "S", and increment the counter n_S by 1. (Line 14) Else if the requested page is in B_1, then (line 15) we increase the target size for the list T_1 (increase the temporal window) and insert the requested page at the tail end of T_1 and (line 16) set its page reference bit to zero, and, more importantly, also changes its filter bit to "L". Finally, (line 17) if the requested page is in B_2, then (line 18) we decrease the target size for the list T_1 and insert the requested page at the tail end of T_1, (line 19) set its page reference bit to zero, and (line 20) update the target q for the list B_1.

The essence of the adaptation rule is: On a hit in B_1, it favors increasing the size of T_1, and, on a hit in B_2, it favors decreasing the size of T_1.

Now, we describe the "replace()" procedure. (Lines 23-26) While the page reference bit of the head page in T_2 is 1, then move the page to the tail position in T_1, and also update the target q to control the size of B_1. In other words, these lines capture the movement from T_2 to T_1. When this while loop terminates, either T_2 is empty, or the page reference bit of the head page in T_2 is set to 0, and, hence, can be removed from the cache if desired.

(Line 27-35) While the filter bit of the head page in T_1 is "L" or the page reference bit of the head page in T_1 is 1, keep moving these pages. When this while loop

terminates, either T_1 will be empty, or the head page in T_1 has its filter bit set to "S" and page reference bit set to 0, and, hence, can be removed from the cache if desired. (Lines 28-30) If the page reference bit of the head page in T_1 is 1, then make it the tail page in T_1. At the same time, if B_1 is very small or T_1 is larger than its target, then relax the temporal filtering constraint and set the filter bit to "L". (Lines 31-33) If the page reference bit is set to 0 but the filter bit is set to "L", then move the page to the tail position in T_2. Also, change the target B_1.

(Lines 36-40) These lines represent our cache replacement policy. If T_1 contains at least p pages and is not empty, then remove the head page in T_1, else remove the head page in T_2.

Our discussion of CART is now complete.

V. Experimental Results

In this section, we will focus our experimental simulations to compare LRU, CLOCK, ARC, CAR, and CART.

A. Traces

Table I summarizes various traces that we used in this paper. These traces are the same as those in [10, Section V.A], and, for brevity, we refer the reader there for their description. These traces capture disk accesses by databases, web servers, NT workstations, and a synthetic benchmark for storage controllers. All traces have been filtered by up-stream caches, and, hence, are representative of workloads seen by storage controllers, disks, or RAID controllers.

Trace Name	Number of Requests	Unique Pages
P1	32055473	2311485
P2	12729495	913347
P3	3912296	762543
P4	19776090	5146832
P5	22937097	3403835
P6	12672123	773770
P7	14521148	1619941
P8	42243785	977545
P9	10533489	1369543
P10	33400528	5679543
P11	141528425	4579339
P12	13208930	3153310
P13	15629738	2497353
P14	114990968	13814927
ConCat	490139585	47003313
Merge(P)	490139585	47003313
DS1	43704979	10516352
SPC1	41351279	6050363
S1	3995316	1309698
S2	17253074	1693344
S3	16407702	1689882
Merge (S)	37656092	4692924

TABLE I. A summary of various traces used in this paper. Number of unique pages in a trace is termed its "footprint".

For all traces, we only considered the read requests. All hit ratios reported in this paper are *cold start*. We will report hit ratios in percentages (%).

B. Results

In Table II, we compare LRU, CLOCK, ARC, CAR, and CART for the traces SPC1 and Merge(S) for various cache sizes. It can be clearly seen that CLOCK has performance very similar to LRU, and CAR/CART have performance very similar to ARC. Furthermore, CAR/CART substantially outperform CLOCK.

SPC1

c (pages)	LRU	CLOCK	ARC	CAR	CART
65536	0.37	0.37	0.82	0.84	0.90
131072	0.78	0.77	1.62	1.66	1.78
262144	1.63	1.63	3.23	3.29	3.56
524288	3.66	3.64	7.56	7.62	8.52
1048576	9.19	9.31	20.00	20.00	21.90

Merge(S)

c (pages)	LRU	CLOCK	ARC	CAR	CART
16384	0.20	0.20	1.04	1.03	1.10
32768	0.40	0.40	2.08	2.07	2.20
65536	0.79	0.79	4.07	4.05	4.27
131072	1.59	1.58	7.78	7.76	8.20
262144	3.23	3.27	14.30	14.25	15.07
524288	8.06	8.66	24.34	24.47	26.12
1048576	27.62	29.04	40.44	41.00	41.83
1572864	50.86	52.24	57.19	57.92	57.64
2097152	68.68	69.50	71.41	71.71	71.77
4194304	87.30	87.26	87.26	87.26	87.26

TABLE II. A comparison of hit ratios of LRU, CLOCK, ARC, CAR, and CART on the traces SPC1 and Merge(S). All hit ratios are reported in percentages. The page size is 4 KBytes for both traces. The largest cache simulated for SPC1 was 4 GBytes and that for Merge(S) was 16 GBytes. It can be seen that LRU and CLOCK have similar performance, while ARC, CAR, and CART also have similar performance. It can be seen that ARC/CAR/CART outperform LRU/CLOCK.

In Figures 4 and 5, we graphically compare the hit-ratios of CAR to CLOCK for all of our traces. The performance of CAR was very close to ARC and CART and the performance of CLOCK was very similar to LRU, and, hence, to avoid clutter, LRU, ARC, and CART are not plotted. It can be clearly seen that across a wide variety of workloads and cache sizes CAR outperforms CLOCK–sometimes quite dramatically.

Finally, in Table III, we produce an at-a-glance-summary of LRU, CLOCK, ARC, CAR, and CART for various traces and cache sizes. Once again, the same conclusions as above are seen to hold: ARC, CAR, and CART outperform LRU and CLOCK, ARC, CAR, and CART have a very similar performance, and CLOCK has performance very similar to LRU.

INITIALIZATION: Set $p = 0$, $q = 0$, $n_S = n_L = 0$, and set the lists T_1, B_1, T_2, and B_2 to empty.

```
CART( 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()
            /* history replacement */
6:          if ((x ∉ B1 ∪ B2) and (|B1| + |B2| = c + 1) and ((|B1| > max{0, q}) or (B2 is empty))) then
7:              Remove the bottom page in B1 from the history.
8:          elseif ((x ∉ B1 ∪ B2) and (|B1| + |B2| = c + 1)) then
9:              Remove the bottom page in B2 from the history.
10:         endif
11:     endif
        /* history 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, set filter bit of x to "S", and ns = ns + 1.
        /* history hit */
14:     elseif (x is in B1) then
15:         Adapt: Increase the target size for the list T1 as: p = min{p + max{1, ns/|B1|}, c}. Move x to the tail of T1.
16:         Set the page reference bit of x to 0. Set nL = nL + 1. Set type of x to "L".
        /* history hit */
17:     else /* x must be in B2 */
18:         Adapt: Decrease the target size for the list T1 as: p = max{p − max{1, nL/|B2|}, 0}. Move x to the tail of T1.
19:         Set the page reference bit of x to 0. Set nL = nL + 1.
20:         if (|T2| + |B2| + |T1| − ns ≥ c) then, Set target q = min(q + 1, 2c − |T1|), endif
21:     endif
22: endif
```

```
replace()
23: while (the page reference bit of the head page in T2 is 1)) then
24:     Move the head page in T2 to tail position in T1. Set the page reference bit to 0.
25:     if (|T2| + |B2| + |T1| − ns ≥ c) then, Set target q = min(q + 1, 2c − |T1|), endif
26: endwhile
    /* The following while loop should stop, if T1 is empty */
27: while ((the filter bit of the head page in T1 is "L") or (the page reference bit of the head page in T1 is 1))
28:     if ((the page reference bit of the head page in T1 is 1)
29:         Move the head page in T1 to tail position in T1. Set the page reference bit to 0.
30:         if ((|T1| ≥ min(p + 1, |B1|)) and (the filter bit of the moved page is "S")) then,
                set type of x to "L", ns = ns − 1, and nL = nL + 1.
            endif
31:     else
32:         Move the head page in T1 to tail position in T2. Set the page reference bit to 0.
33:         Set q = max(q − 1, c − |T1|).
34:     endif
35: endwhile
36: if (|T1| >= max(1, p)) then
37:     Demote the head page in T1 and make it the MRU page in B1. ns = ns − 1.
38: else
39:     Demote the head page in T2 and make it the MRU page in B2. nL = nL − 1.
40: endif
```

Fig. 3. Algorithm for Clock with Adaptive Replacement and Temporal Filtering. 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 history.

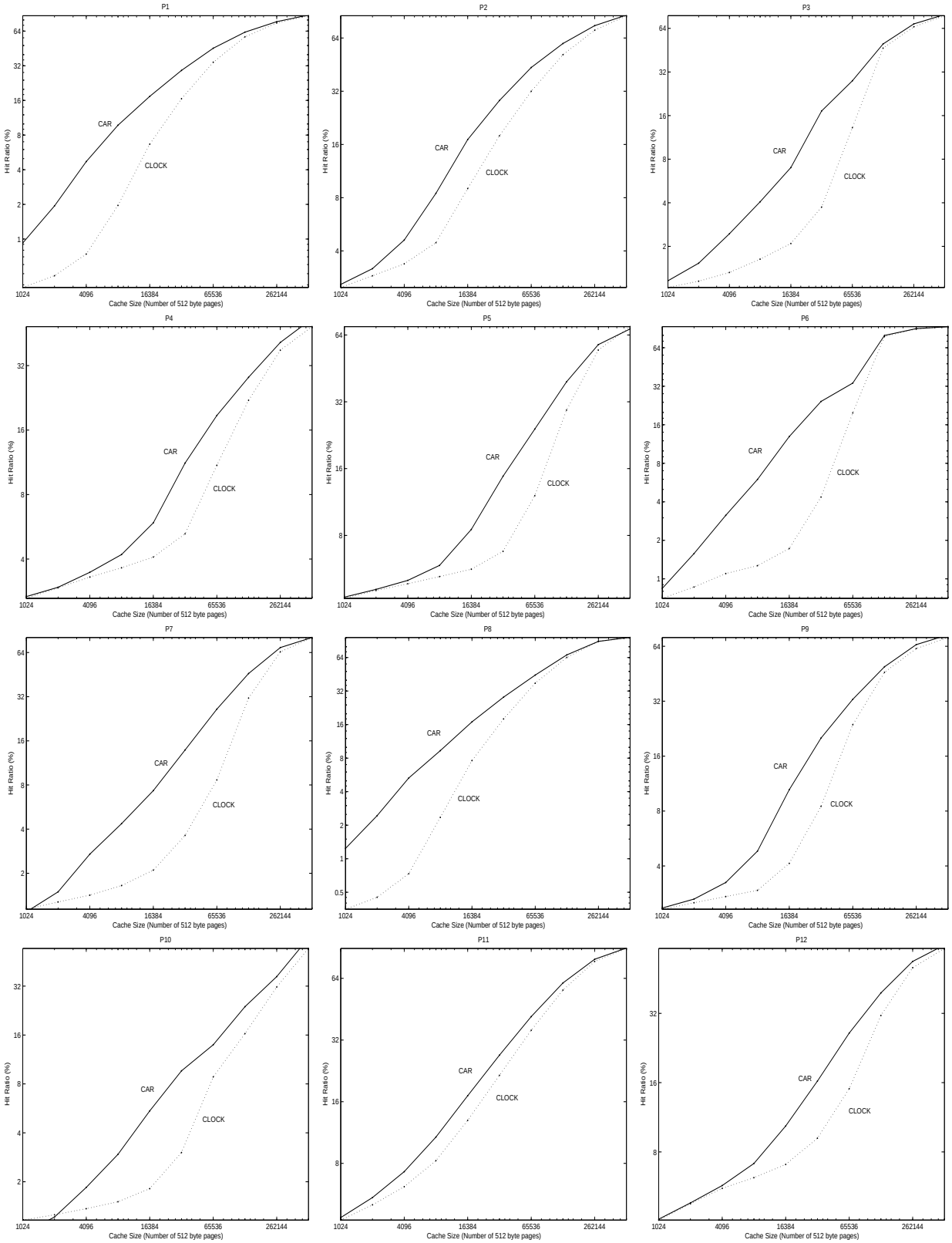

Fig. 4. A plot of hit ratios (in percentages) achieved by CAR and CLOCK. Both the x- and y-axes use logarithmic scale.

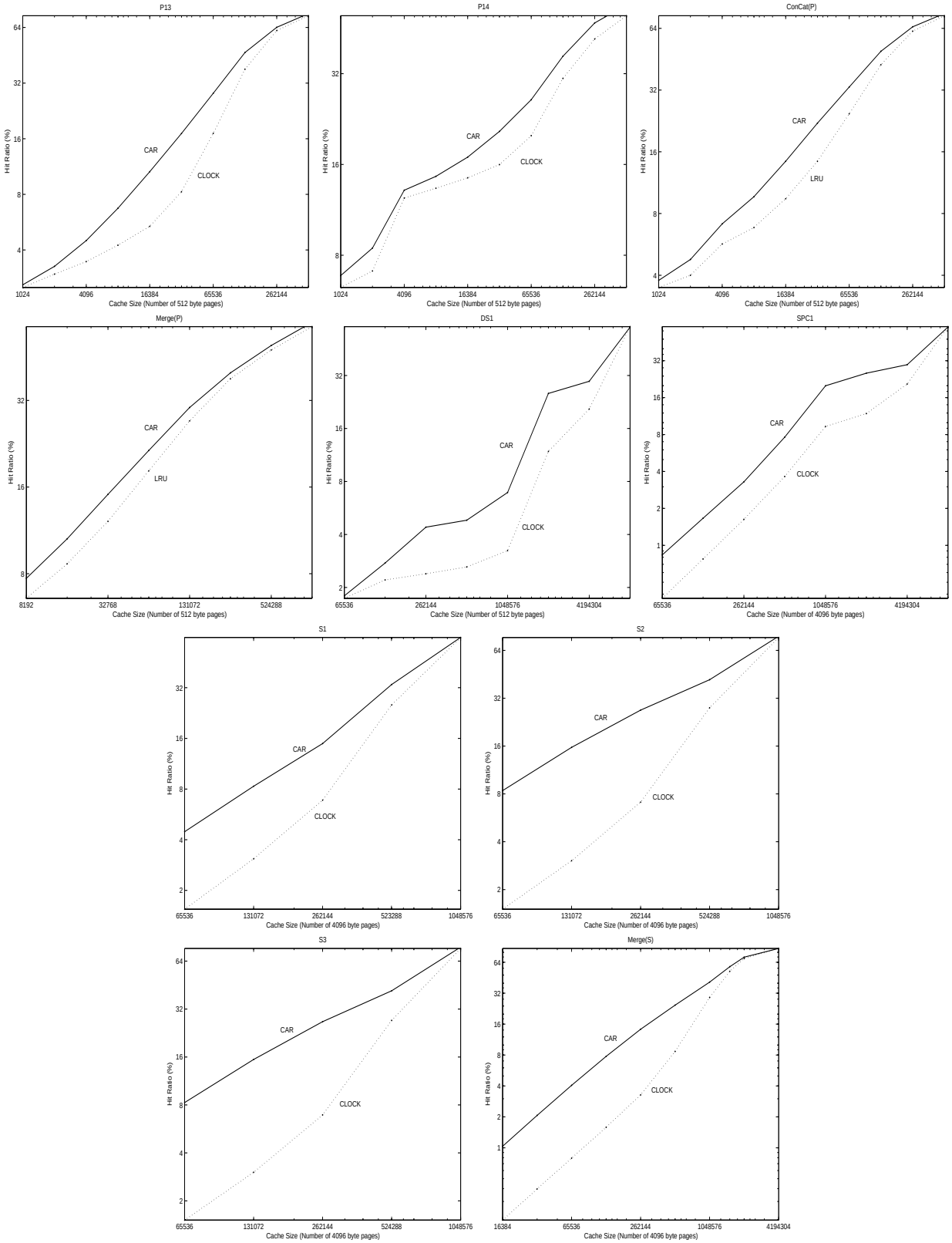

Fig. 5. A plot of hit ratios (in percentages) achieved by CAR and CLOCK. Both the x- and y-axes use logarithmic scale.

Workload	c (pages)	space (MB)	LRU	CLOCK	ARC	CAR	CART
P1	32768	16	16.55	17.34	28.26	29.17	29.83
P2	32768	16	18.47	17.91	27.38	28.38	28.63
P3	32768	16	3.57	3.74	17.12	17.21	17.54
P4	32768	16	5.24	5.25	11.24	11.22	9.25
P5	32768	16	6.73	6.78	14.27	14.78	14.77
P6	32768	16	4.24	4.36	23.84	24.34	24.53
P7	32768	16	3.45	3.62	13.77	13.86	14.79
P8	32768	16	17.18	17.99	27.51	28.21	28.97
P9	32768	16	8.28	8.48	19.73	20.09	20.75
P10	32768	16	2.48	3.02	9.46	9.63	9.71
P11	32768	16	20.92	21.51	26.48	26.99	27.26
P12	32768	16	8.93	9.18	15.94	16.25	16.41
P13	32768	16	7.83	8.26	16.60	17.09	17.74
P14	32768	16	15.73	15.98	20.52	20.59	20.63
ConCat	32768	16	14.38	14.79	21.67	22.06	22.24
Merge(P)	262144	128	38.05	38.60	39.91	39.90	40.12
DS1	2097152	1024	11.65	11.86	22.52	25.31	21.12
SPC1	1048576	4096	9.19	9.31	20.00	20.00	21.91
S1	524288	2048	23.71	25.26	33.43	33.42	33.62
S2	524288	2048	25.91	27.84	40.68	41.86	42.10
S3	524288	2048	25.26	27.13	40.44	41.67	41.87
Merge(S)	1048576	4096	27.62	29.04	40.44	41.01	41.83

TABLE III. At-a-glance comparison of hit ratios of LRU, CLOCK, ARC, CAR, and CART for various workloads. All hit ratios are reported in percentages. It can be seen that LRU and CLOCK have similar performance, while ARC, CAR, and CART also have similar performance. It can be seen that ARC, CAR, and CART outperform LRU and CLOCK–sometimes quite dramatically.

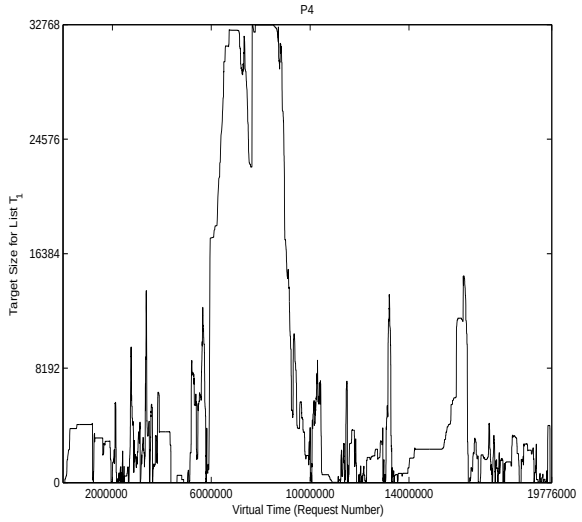

Fig. 6. A plot of the adaptation parameter p (the target size for list T_1) versus the virtual time for the algorithm CAR. The trace is P4, the cache size is 32768 pages, and the page size is 512 bytes.

VI. 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. In other words, adaptation really matters! Also, it can be shown that CAR performs as well as its offline counterpart which is allowed to select the best, offline, fixed value of p chosen specifically for a given workload and a cache size. In other words, adaptation really works! See Figure 6 for a graphical demonstration of how p fluctuates. The self-tuning nature of CAR makes it very attractive for deployment in environments where no *a priori* knowledge of the workloads is available.

CAR **is scan-resistant**. A scan is any sequence of one-time use requests. Such requests will be put on top of the list T_1 and will eventually exit from the cache without polluting the high-quality pages in T_2. Moreover, in presence of scans, there will be relatively fewer hits in B_1 as compared to B_2. Hence, our adaptation rule will tend to further increase the size of T_2 at the expense of T_1, thus further decreasing the residency time of scan in even T_1.

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.

CART **has temporal filtering**. The algorithm CART has all the above advantages of CAR, but, in addition, it employs a much stricter and more precise criterion to distinguish pages with short-term utility from those with long-term utility.

It should be clear from Section II-A that a large number of attempts have been made to improve upon LRU. In contrast, relatively few attempts have been made to improve upon CLOCK–the most recent being in 1978! We believe that this is due to severe constraints imposed by CLOCK on how much processing can be done on a hit and its removal of the single global lock. Genuine new insights were required to invent novel, effective algorithms that improve upon CLOCK. We hope that CAR and CART represents two such fundamental insights and that they will be seriously considered by cache designers.

ACKNOWLEDGMENT

We are grateful to Bruce Lindsay and Honesty Young for suggesting that we look at lock contention, to Frank Schmuck for pointing out bugs in our previous attempts, and to Pawan Goyal for urging us to publish this work. We are grateful to our manager, Moidin Mohiuddin, for his constant support and encouragement during this work. The second author is grateful to Nimrod Megiddo for his collaboration on ARC. We are grateful to Bruce McNutt and Renu Tewari for the SPC1 trace, to Windsor Hsu for traces P1 through P14, to Ruth Azevedo for the trace DS1, and to Ken Bates and Bruce McNutt for traces S1-S3. We are indebted to Binny Gill for drawing the beautiful and precise Figure 1.

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*. Englewood 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. Corbató, "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. Quarterman, *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 management 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," *Computer 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 performance 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 algorithm 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 performance," 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 management 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 generalized 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, 1996.