ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE

USENIX File & Storage Technologies Conference (FAST), March 31, 2003, San Francisco, CA

Nimrod Megiddo and Dharmendra S. Modha
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

***Abstract*— We consider the problem of cache management in a demand paging scenario with uniform page sizes. We propose a new cache management policy, namely, Adaptive Replacement Cache (ARC), that has several advantages.

In response to evolving and changing access patterns, ARC *dynamically, adaptively, and continually* balances between the recency and frequency components in an *online* and *self-tuning* fashion. The policy ARC uses a learning rule to adaptively and continually revise its assumptions about the workload.

The policy ARC is *empirically universal*, that is, it empirically performs as well as a certain *fixed replacement policy*–even when the latter uses the best workload-specific tuning parameter that was selected in an offline fashion. Consequently, ARC works uniformly well across varied workloads and cache sizes without any need for workload specific *a priori* knowledge or tuning. Various policies such as LRU-2, 2Q, LRFU, and LIRS require user-defined parameters, and, unfortunately, no single choice works uniformly well across different workloads and cache sizes.

The policy ARC is simple-to-implement and, like LRU, has constant complexity per request. In comparison, policies LRU-2 and LRFU both require logarithmic time complexity in the cache size.

The policy ARC is *scan-resistant*: it allows one-time sequential requests to pass through without polluting the cache.

On 23 real-life traces drawn from numerous domains, ARC leads to substantial performance gains over LRU for a wide range of cache sizes. For example, for a SPC1 like synthetic benchmark, at 4GB cache, LRU delivers a hit ratio of 9.19% while ARC achieves a hit ratio of 20%.

I. INTRODUCTION

> "We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, J. von Neumann, in *Preliminary Discussion of the Logical Design of Electronic Computing Instrument*, Part I, Vol. I, Report prepared for U.S. Army Ord. Dept., 28 June 1946.

A. The Problem

Caching is one of the oldest and most fundamental metaphor in modern computing. It is widely used in storage systems (for example, IBM ESS or EMC Symmetrix), databases [1], web servers [2], middleware [3], processors [4], file systems [5], disk drives [6], RAID controllers [7], operating systems [8], and in varied and numerous other applications such as data compression [9] and list updating [10]. Any substantial progress in caching algorithms will affect the entire modern computational stack.

Consider a system consisting of two memory levels: *main* (or *cache*) and *auxiliary*. The cache is assumed to be significantly faster than the auxiliary memory, but is also significantly more expensive. Hence, the size of the cache memory is usually only a fraction of the size of the auxiliary memory. Both memories are managed in units of uniformly sized items known as *pages*. We assume that the cache receives a continuous *stream of requests* for pages. We assume a *demand paging* scenario where a page of memory is *paged in* the cache from the auxiliary memory only when a request is made for the page and the page is not already present in the cache. In particular, demand paging rules out pre-fetching. For a full cache, before a new page can be brought in one of the existing pages must be *paged out*. The victim page is selected using a cache *replacement policy*. Under demand paging model, the replacement policy is the only algorithm of interest. The most important metric for a cache replacement policy is its *hit rate*–the fraction of pages that can be served from the main memory. The *miss rate* is the fraction of pages that must be paged into the cache from the auxiliary memory. Another important metric for a cache replacement policy is its overhead which should be low.

The problem of cache management is to design a replacement policy that maximizes the hit rate measured over a very long trace subject to the important practical constraints of minimizing the computational and space overhead involved in implementing the policy.

B. Our Contributions

One of the main themes of this paper is to design a replacement policy with a high hit ratio while paying conscientious attention to its implementation complexity. Another equally important theme of this paper is that real-life workloads possess a great deal of richness and variation, and do not admit a one-size-fits-all characterization. They may contain long sequential I/Os or moving hot spots. The frequency and scale of temporal locality may also change with time. They may fluctuate between stable, repeating access patterns and access patterns with transient clustered references. No static, *a priori* fixed replacement policy will work well

over such access patterns. We seek a cache replacement policy that will adapt in an on-line, on-the-fly fashion to such dynamically evolving workloads.

We propose a new cache replacement policy, namely, Adaptive Replacement Cache (ARC). The basic idea behind ARC is to maintain two LRU lists of pages. One list, say L_1, contains pages that have been seen only once "recently", while the other list, say L_2, contains pages that have been seen at least twice "recently". The items that have been seen twice within a short time have a low inter-arrival rate, and, hence, are thought of as "high-frequency". Hence, we think of L_1 as capturing "recency" while L_2 as capturing "frequency". We endeavor to keep these two lists to roughly the same size, namely, the cache size c. Together the two lists remember exactly twice the number of pages that would fit in the cache. In other words, ARC maintains a *cache directory* that remembers twice as many pages as in the cache memory. At any time, ARC chooses a variable number of most recent pages to keep from L_1 and from L_2. The precise number of pages drawn from the list L_1 is a tunable parameter that is adaptively and continually tuned. Let FRC_p denote a *fixed replacement policy* that attempts to keep the p most recent pages in L_1 and $c-p$ most recent pages in L_2 in cache at all times. At any given time, the policy ARC behaves like FRC_p for some fixed p. However, ARC may behave like FRC_p at one time and like FRC_q, $p \neq q$, at some other time. The key new idea is to adaptively–in response to an evolving workload–decide how many top pages from each list to maintain in the cache at any given time. We achieve such on-line, on-the-fly adaptation by using a *learning rule* that allows ARC to track a workload quickly and effectively. The effect of the learning rule is to induce a "random walk" on the parameter p. Intuitively, by learning from the recent past, ARC attempts to keep those pages in the cache that have the greatest likelihood of being used in the near future. It acts as a filter to detect and track temporal locality. If during some part of the workload, recency (resp. frequency) becomes important, then ARC will detect the change, and configure itself to exploit the opportunity. We think of ARC as *dynamically, adaptively, and continually* balancing between recency and frequency–in an *online* and *self-tuning* fashion–in response to evolving and possibly changing access patterns.

We empirically demonstrate that ARC works as well as the policy FRC_p that assigns a fixed portion of the cache to recent pages and the remaining fixed portion to frequent pages–even when the latter uses the best fixed, offline workload and cache size dependent choice for the parameter p. In this sense, ARC is *empirically universal* [1]. Surprisingly, ARC–which is completely online–delivers performance comparable to LRU-2, 2Q, LRFU, and LIRS–even when these policies use the best tuning parameters that were selected in an offline fashion. The policy ARC also compares favorably to an online adaptive policy MQ.

To implement ARC, we need two LRU lists. Hence, ARC is no more difficult to implement than LRU, has constant-time complexity per request, and requires only marginally higher space overhead over LRU. In a real-life implementation, we found that the space overhead of ARC was 0.75% of the cache size. We say that ARC is *low overhead*. In contrast, LRU-2 and LRFU require logarithmic time complexity per request. As a result, in all our simulations, LRU-2 was a factor of 2 slower than ARC and LRU, while LRFU can be as much as a factor of 50 slower than ARC and LRU.

The policy ARC is *scan-resistant*, that is, it allows one-time-only sequential read requests to pass through the cache without flushing pages that have temporal locality. By the same argument, it effectively handles long periods of low temporal locality.

Finally, on a large number of real life workloads drawn from CODASYL, 14 workstation disk drives, a commercial ERP system, a SPC1 like [2] synthetic benchmark, as well as web search requests, we demonstrate that ARC substantially outperforms LRU. As anecdotal evidence, for a workstation disk drive workload, at 16MB cache, LRU delivers a hit ratio of 4.24% while ARC achieves a hit ratio of 23.82%, and, for a SPC1 like benchmark, at 4GB cache, LRU delivers a hit ratio of 9.19% while ARC achieves a hit ratio of 20%.

C. A Brief Outline of the Paper

In Section II, we briefly review relevant work and provide a context for ARC. In Section III, we introduce a class of replacement policies and show that this class contains LRU as a special case. In Section IV, we introduce the policy ARC. In Section V, we present experimental results on several workloads. Finally, in Section VI, we present conclusions.

II. Prior Work: A Brief Review

A. Offline Optimal

For an *a priori* known page reference stream, Belady's MIN that replaces the page that has the greatest forward distance is known to be optimal in terms of the hit ratio [12], [13]. The policy MIN provides an upper bound on the achievable hit ratio by any on-line policy.

B. Recency

The policy LRU always replaces the least recently used page [13]. It dates back at least to 1965 [14], and may in fact be older. Various approximations and improvements to LRU abound, see, for example, enhanced clock algorithm [15]. It is known that if

the workload or the request stream is drawn from a LRU Stack Depth Distribution (SDD), then LRU is the optimal policy [16]. LRU has several advantages, for example, it is simple to implement and responds well to changes in the underlying SDD model. However, while the SDD model captures "recency", it does not capture "frequency". To quote from [16, p. 282]: "The significance of this is, in the long run, that each page is equally likely to be referenced and that therefore the model is useful for treating the clustering effect of locality but not the nonuniform page referencing."

C. Frequency

The Independent Reference Model (IRM) provides a workload characterization that captures the notion of frequency. Specifically, IRM assumes that each page reference is drawn in an independent fashion from a fixed distribution over the set of all pages in the auxiliary memory. Under the IRM model, policy LFU that replaces the least frequently used page is known to be optimal [16], [17]. The LFU policy has several drawbacks: it requires logarithmic implementation complexity in cache size, pays almost no attention to recent history, and does not adapt well to changing access patterns since it accumulates stale pages with high frequency counts that may no longer be useful.

A relatively recent algorithm LRU-2 [18], [19] approximates LFU while eliminating its lack of adaptivity to the evolving distribution of page reference frequencies. This was a significant practical step forward. The basic idea is to remember, for each page, the last two times when it was requested, and to replace the page with the least recent penultimate reference. Under the IRM assumption, it is known that LRU-2 has the largest expected hit ratio of any on-line algorithm that knows at most two most recent references to each page [19]. The algorithm has been shown to work well on several traces [18], [20]. Nonetheless, LRU-2 still has two practical limitations [20]: (i) it needs to maintain a priority queue, and, hence, it requires logarithmic implementation complexity and (ii) it contains at one crucial tunable parameter, namely, *Correlated Information Period* (CIP), that roughly captures the amount of time a page that has only been seen once recently should be kept in the cache.

In practice, logarithmic implementation complexity is a severe overhead, see, Table I. This limitation was mitigated in 2Q [20] which reduces the implementation complexity to constant per request. The algorithm 2Q uses a simple LRU list instead of the priority queue used in LRU-2; otherwise, it is similar to LRU-2. Policy ARC has a computational overhead similar to 2Q and both are better than LRU-2, see, Table I.

Table II shows that the choice of the parameter CIP crucially affects performance of LRU-2. It can be seen that no single fixed *a priori* choice works uniformly well across across various cache sizes, and, hence, judicious selection of this parameter is crucial to achieving good performance. Furthermore, we have found that no single *a priori* choice works uniformly well across across various workloads and cache sizes that we examined. For example, a very small value for the CIP parameters work well for stable workloads drawn according to the IRM, while a larger value works well for workloads drawn according to the SDD. Indeed, it has been previously noted [20] that "it was difficult to model the tunables of the algorithm exactly." This underscores the need for on-line, on-the-fly adaptation.

Unfortunately, the second limitation of LRU-2 persists even in 2Q. The authors introduce two parameters (Kin and $Kout$) and note that "Fixing these parameters is potentially a tuning question . . ." [20]. The parameter Kin is essentially the same as the parameter CIP in LRU-2. Once again, it has been noted [21] that "Kin and $Kout$ are predetermined parameters in 2Q, which need to be carefully tuned, and are sensitive to types of workloads." Due to space limitation, we have shown Table II only for LRU-2, however, we have observed similar dependence of 2Q on the workload

c	LRU	ARC	2Q	LRU-2	LRFU λ		
					10^{-7}	10^{-3}	.99
1024	17	14	17	33	554	408	28
2048	12	14	17	27	599	451	28
4096	12	15	17	27	649	494	29
8192	12	16	18	28	694	537	29
16384	13	16	19	30	734	418	30
32768	14	17	18	31	716	420	31
65536	14	16	18	32	648	424	34
131072	14	15	16	32	533	432	39
262144	13	13	14	30	427	435	42
524288	12	13	13	27	263	443	45

TABLE I. A comparison of computational overhead of various cache algorithms on a trace P9 that was collected from a workstation running Windows NT by using Vtrace which captures disk requests. For more details of the trace, see Section V-A. The cache size c represents number of 512 byte pages. To obtain the numbers reported above, we assumed that a miss costs nothing more than a hit. This focuses the attention entirely on the "book-keeping" overhead of the cache algorithms. All timing numbers are in seconds, and were obtained by using the "clock()" subroutine in "time.h" of the GNU C compiler. It can be seen that the computational overhead of ARC and 2Q is essentially the same as that of LRU. It can also be seen that LRU-2 has roughly double the overhead of LRU, and that LRFU can have very large overhead when compared to LRU. The same general results hold for all the traces that we examined.

c	CIP/c 0.01	0.05	0.1	0.25	0.5	0.75	0.9	0.95	0.99
1024	0.87	1.01	1.41	3.03	3.77	4.00	4.07	4.07	4.07
2048	1.56	2.08	3.33	4.32	4.62	4.77	4.80	4.83	4.83
4096	2.94	4.45	5.16	5.64	5.81	5.79	5.70	5.65	5.61
8192	5.36	7.02	7.39	7.54	7.36	6.88	6.47	6.36	6.23
16384	9.53	10.55	10.67	10.47	9.47	8.38	7.60	7.28	7.15
32768	15.95	16.36	16.23	15.32	13.46	11.45	9.92	9.50	9.03
65536	25.79	25.66	25.12	23.64	21.33	18.26	15.82	14.99	14.58
131072	39.58	38.88	38.31	37.46	35.15	32.21	30.39	29.79	29.36
262144	53.43	53.04	52.99	52.09	51.73	49.42	48.73	49.20	49.11
524288	63.15	63.14	62.94	62.98	62.00	60.75	60.40	60.57	60.82

TABLE II. Hit ratios (in percentages) achieved by the algorithm LRU-2 on a trace P12 for various values of the tunable parameter CIP and various cache sizes. The trace P12 was collected from a workstation running Windows NT by using Vtrace which captures disk requests, for details of the trace, see Section V-A. The cache size c represents number of 512 byte pages.

and the cache size. After theoretically analyzing how to set parameter $Kout$, [20] concluded that "Since this formula requires an a priori estimate of the miss rate, it is of little use in practical tuning." They suggested $Kout = 0.5c$ "is almost always a good choice".

Another recent algorithm is Low Inter-reference Recency Set (LIRS) [21]. The algorithm maintains a variable size LRU stack whose LRU page is the L_{lirs}-th page that has been seen at least twice recently, where L_{lirs} is a parameter. From all the pages in the stack, the algorithm keeps all the L_{lirs} pages that have been seen at least twice recently in the cache as well as L_{hirs} pages that have been seen only once recently. The parameter L_{hirs} is similar to CIP of LRU-2 or Kin of 2Q. The authors suggest setting L_{hirs} to 1% of the cache size. This choice will work well for stable workloads drawn according to the IRM, but not for those LRU-friendly workloads drawn according to the SDD. Just as CIP affects LRU-2 and Kin affects 2Q, we have found that the parameter L_{hirs} crucially affects LIRS. A further limitation of LIRS is that it requires a certain "stack pruning" operation that in the worst case may have to touch a very large number of pages in the cache. This implies that overhead of LIRS is constant-time in the expected sense, and not in the worst case as for LRU. Finally, the LIRS stack may grow arbitrarily large, and, hence, it needs to be *a priori* limited.

Our idea of separating items that have been only seen once recently from those that have been seen at least twice recently is related to similar ideas in LRU-2, 2Q, and LIRS. However, the precise structure of our lists L_1 and L_2 and the self-tuning, adaptive nature of our algorithm have no analogue in these papers.

D. Recency and Frequency

Over last few years, interest has focussed on combining recency and frequency. Several papers have attempted to bridge the gap between LRU and LFU by combining recency and frequency in various ways. We shall mention two heuristic algorithms in this direction.

Frequency-based replacement (FBR) policy [22] maintains a LRU list, but divides it into three sections: new, middle, and old. For every page in cache, a counter is also maintained. On a cache hit, the hit page is moved to the MRU position in the new section; moreover, if the hit page was in the middle or the old section, then its reference count is incremented. The key idea known as *factoring out locality* was that if the hit page was in the new section then the reference count is not incremented. On a cache miss, the page in the old section with the smallest reference count is replaced. A limitation of the algorithm is that to prevent cache pollution due to stale pages with high reference count but no recent usage the algorithm must periodically resize (rescale) all the reference counts. The algorithm also has several tunable parameters, namely, the sizes of all three sections, and some other parameters C_{max} and A_{max} that control periodic resizing. Once again, much like in LRU-2 and 2Q, different values of the tunable parameters may be suitable for different workloads or for different cache sizes. The historical importance of FBR stems from the fact it was one of the earliest papers to combine recency and frequency. It has been shown that performance of FBR is similar to that of LRU-2, 2Q, and LRFU [23].

Recently, a class of policies, namely, Least Recently/Frequently Used (LRFU), that subsume LRU and LFU was studied [23]. Initially, assign a value $C(x) = 0$ to every page x, and, at every time t, update as:

$$C(x) = \begin{cases} 1 + 2^{-\lambda} C(x) & \text{if } x \text{ is referenced at time } t; \\ 2^{-\lambda} C(x) & \text{otherwise,} \end{cases}$$

where λ is a tunable parameter. With hindsight, we can easily recognize the update rule as a form of *exponential*

smoothing that is widely used in statistics. The LRFU policy is to replace the page with the smallest $C(x)$ value. Intuitively, as λ approaches 0, the C value is simply the number of occurrences of page x and LRFU collapses to LFU. As λ approaches 1, due to the exponential decay, the C value emphasizes recency and LRFU collapses to LRU. The performance of the algorithm depends crucially on the choice of λ, see [23, Figure 7]. A later adaptive version, namely, Adaptive LRFU (ALRFU) dynamically adjusts the parameter λ [24]. Still, LRFU has two fundamental limitations that hinder its use in practice: (i) LRFU and ALRFU both require an additional tunable parameter for controlling correlated references. The choice of this parameter matters, see [23, Figure 8]. (ii) The implementation complexity of LRFU fluctuates between constant per request to logarithmic in cache size per request. However, due to multiplications and exponentiations, its practical complexity is significantly higher than that of even LRU-2, see, Table I. It can be seen that, for small values of λ, LRFU can be as much as 50 times slower than LRU and ARC. Such overhead can potentially wipe out the entire benefit of a higher hit ratio.

E. Temporal Distance Distribution

Recently, [25] studied a multi-queue replacement policy MQ. The idea is to use m (typically, $m = 8$) LRU queues: $Q_0, \ldots, Q_{m-1}$, where Q_i contains pages that have been seen at least 2^i times but no more than $2^{i+1} - 1$ times recently. The algorithm also maintains a history buffer Q_{out}. On a page hit, the page frequency is incremented, the page is placed at the MRU position of the appropriate queue, and its $expireTime$ is set to $currentTime + lifeTime$, where $lifeTime$ is a tunable parameter. On each access, $expireTime$ for the LRU page in each queue is checked, and if it is less than $currentTime$, then the page is moved to the MRU position of the next lower queue. The optimal values of $lifeTime$ and the length of Q_{out} depend upon the workload and the cache size.

The algorithm MQ is designed for storage controllers. Due to up-stream caches, two consecutive accesses to a single page have a relatively long temporal distance. The algorithm assumes that the temporal distance possesses a "hill" shape. In this setting, the recommended value of the $lifeTime$ parameter is the peak temporal distance, and can be dynamically estimated for each workload.

The algorithm ARC makes no such stringent assumption about the shape of the temporal distance distribution, and, hence, is likely to be robust under a wider range of workloads. Also, MQ will adjust to workload evolution when a measurable change in peak temporal distance can be detected, whereas ARC will track an evolving workload nimbly since it adapts continually. While MQ has constant-time overhead, it still needs to check time stamps of LRU pages for m queues on every request, and, hence, has a higher overhead than LRU, ARC, and 2Q. For example, in the context of Table I, with $m = 8$, the overhead ranged from 36 to 54 seconds for various cache sizes.

F. Caching using Multiple Experts

Recently, [26] proposed a master-policy that simulates a number of caching policies (in particular, 12 policies) and, at any given time, adaptively and dynamically chooses one of the competing policies as the "winner" and switches to the winner. As they have noted, "rather than develop a new caching policy", they select the best policy amongst various competing policies. Since, ARC (or any of the other policies discussed above) can be used as one of the competing policies, the two approaches are entirely complementary. From a practical standpoint, a limitation of the master-policy is that it must simulate all competing policies, and, hence, requires high space and time overhead. Recently, [27] applied above ideas to distributed caching.

G. Ghost Caches

In this paper, we will maintain a larger cache directory than that is needed to support the underlying cache. Such directories are known as a *shadow cache* or as a *ghost cache*. Previously, ghost caches have been employed in a number of cache replacement algorithms such as 2Q, MQ, LRU-2, ALRFU, and LIRS to remember recently evicted cache pages. Most recently, while studying how to make a storage array's cache more exclusive of the client caches, [28] used ghost caches to simulate two LRU lists: one for disk-read blocks and the other for client-demoted blocks. The hits rates on the ghost LRU lists were then used to adaptively determine the suitable insertion points for each type of data in a LRU list.

H. Summary

In contrast to LRU-2, 2Q, LIRS, FBR, and LRFU which require offline selection of tunable parameters, our replacement policy ARC is online and is completely self-tuning. Most importantly, ARC is empirically universal. The policy ARC maintains no frequency counts, and, hence, unlike LFU and FBR, it does not suffer from periodic rescaling requirements. Also, unlike LIRS, the policy ARC does not require potentially unbounded space overhead. In contrast to MQ, the policy ARC may be useful for a wider range of workloads, adapts quickly to evolving workloads, and has less computational overhead. Finally, ARC, 2Q, LIRS, MQ, and FBR have constant-time implementation complexity while LFU, LRU-2, and LRFU have logarithmic implementation complexity.

III. A Class of Replacement Policies

Let c denote the cache size in number of pages. For a cache replacement policy π, we will write $\pi(c)$ when we want to emphasize the number of pages being managed by the policy.

We will first introduce a policy DBL$(2c)$ that will manage and remember twice the number of pages present in the cache. With respect to the policy DBL, we introduce a new class of cache replacement policies $\Pi(c)$.

A. Double Cache and a Replacement Policy

Suppose we have a cache that can hold $2c$ pages. We now describe a cache replacement policy DBL$(2c)$ to manage such a cache. We will use this construct to motivate the development of an adaptive cache replacement policy for a cache of size c.

The cache replacement policy DBL$(2c)$ maintains two variable-sized lists L_1 and L_2, the first containing pages that have been seen *only once recently* and the second containing pages that have been seen *at least twice recently*. Precisely, a page is in L_1 if has been requested exactly once since the last time it was removed from $L_1 \cup L_2$, or if it was requested only once and was never removed from $L_1 \cup L_2$. Similarly, a page is in L_2 if it has been requested more than once since the last time it was removed from $L_1 \cup L_2$, or was requested more than once and was never removed from $L_1 \cup L_2$. The replacement policy is:

> Replace the LRU page in L_1, if L_1 contains exactly c pages; otherwise, replace the LRU page in L_2.

The replacement policy attempts to equalize the sizes of two lists. We exhibit a complete algorithm that captures DBL in Figure 1 and pictorially illustrate its structure in Figure 2. The sizes of the two lists can fluctuate, but the algorithm ensures that following invariants will always hold:

$$0 \leq |L_1| + |L_2| \leq 2c,\ 0 \leq |L_1| \leq c,\ 0 \leq |L_2| \leq 2c.$$

B. A New Class of Policies

We now propose a new class of policies $\Pi(c)$. Intuitively, the proposed class will contain demand paging policies that track all $2c$ items that would have been in a cache of size $2c$ managed by DBL, but physically keep only (at most) c of those in the cache at any given time.

Let L_1 and L_2 be the lists associated with DBL$(2c)$. Let $\Pi(c)$ denote a class of demand paging cache replacement policies such that for every policy $\pi(c) \in \Pi(c)$ there exists a dynamic partition of list L_1 into a top portion T_1^π and a bottom portion B_1^π and a dynamic partition of list L_2 into a top portion T_2^π and a bottom portion B_2^π such that

---

DBL$(2c)$

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $\ell_1 = 0$, $\ell_2 = 0$, $L_1 = \emptyset$ and $L_2 = \emptyset$.

For every $t \geq 1$ and any x_t, one and only one of the following two cases must occur.

Case I: x_t is in L_1 or L_2.
A cache hit has occurred. Make x_t the MRU page in L_2.

Case II: x_t is neither in L_1 nor in L_2.
A cache miss has occurred. Now, one and only one of the two cases must occur.

Case A: L_1 has exactly c pages.
Delete the LRU page in L_1 to make room for the new page, and make x_t the MRU page in L_1.

Case B: L_1 has less than c pages.
1) If the cache is full, that is, $(|L_1| + |L_2|) = 2c$, then delete the LRU page in L_2 to make room for the new page.
2) Insert x_t as the MRU page in L_1.

Fig. 1. Algorithm for the cache replacement policy DBL that manages $2c$ pages in cache.

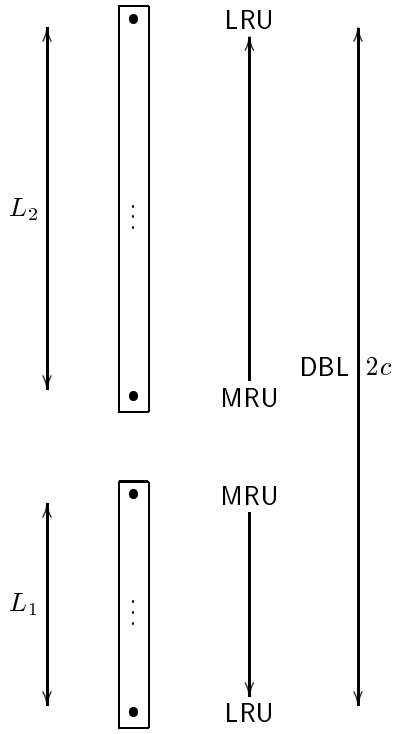

Fig. 2. General structure of the cache replacement policy DBL. The cache is partitioned into two LRU lists: L_1 and L_2. The former contains pages that have been seen only once "recently" while the latter contains pages that have been seen at least twice "recently". Visually, the list L_2 is inverted when compared to L_1. This allows us to think of the more recent items in both the lists as closer in the above diagram. The replacement policy deletes the LRU page in L_1, if L_1 contains exactly c pages; otherwise, it replaces the LRU page in L_2.

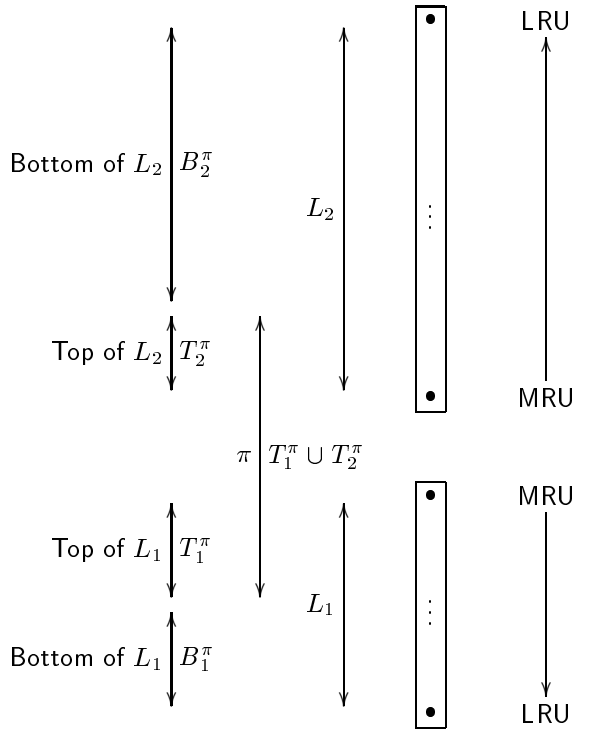

Fig. 3. General structure of a generic cache replacement policy $\pi(c) \in \Pi(c)$. The lists L_1 and L_2 are exactly as in Figure 2. The list L_1 is partitioned into a top portion T_1^π and a bottom portion B_1^π. Similarly, the list L_2 is partitioned into a top portion T_2^π and a bottom portion B_2^π. The policy $\pi(c)$ maintains pages in $T_1^\pi \cup T_2^\pi$ in cache.

A.1 The lists T_1^π and B_1^π are disjoint and, also, the lists T_2^π and B_2^π are disjoint, and

$$L_1 = T_1^\pi \cup B_1^\pi \text{ and } L_2 = T_2^\pi \cup B_2^\pi.$$

A.2 If $|L_1 \cup L_2| < c$, then both B_1^π and B_2^π are empty.

A.3 If $|L_1 \cup L_2| \geq c$, then, together, T_1^π and T_2^π contain exactly c pages.

A.4 Either T_1^π (resp. T_2^π) or B_1^π (resp. B_2^π) is empty, or the LRU page in T_1^π (resp. T_2^π) is more recent than the MRU page in B_1^π (resp. B_2^π).

A.5 For all traces and at each time, $T_1^\pi \cup T_2^\pi$ will contain exactly those pages that would be maintained in cache by the policy $\pi(c)$.

The generic structure of the policy π is depicted in Figure 3. It follows from Conditions A.1 and A.5 that the pages contained in a cache of size c managed by $\pi(c)$ will always be a subset of the pages managed by $\mathsf{DBL}(2c)$. Since any policy in $\Pi(c)$ must track all pages that would have been in $\mathsf{DBL}(2c)$, it would require essentially double the space overhead of LRU.

Remark III.1 Condition A.4 implies that if a page in L_1 (resp. L_2) is kept, then all pages in L_1 (resp. L_2) that are more recent than it must all be kept in the cache. Hence, the policy $\pi(c)$ can be thought of as "skimming the top (or most recent) few pages" in L_1 and L_2. Suppose that we are managing a cache with $\pi(c) \in \Pi(c)$, and also let us suppose that the cache is full, that is, $|T_1^\pi \cup T_2^\pi| = c$, then, it follows from Condition A.4 that, from now on, for any trace, on a cache miss, only two *actions* are available to the policy $\pi(c)$: (i) replace the LRU page in T_1^π or (ii) replace the LRU page in T_2^π.

C. LRU

We now show that the policy $\mathsf{LRU}(c)$ is contained in the class $\Pi(c)$. In particular, we show that the most recent c pages will always be contained in $\mathsf{DBL}(2c)$. To see this fact observe that $\mathsf{DBL}(2c)$ deletes either the LRU item in L_1 or the LRU item in L_2. In the first case, L_1 must contain exactly c items (see case II(A) in Figure 1). In the second case, L_2 must contain at least c items (see case II(B)(1) in Figure 1). Hence, DBL never deletes any of the most recently seen c pages and always contains all pages contained in a LRU cache with c items. It now follows that there must exist a dynamic partition of lists L_1 and L_2 into lists T_1^{LRU}, B_1^{LRU}, T_2^{LRU}, and B_2^{LRU}, such that the conditions A.1–A.5 hold. Hence, the policy $\mathsf{LRU}(c)$ is contained in the class $\Pi(c)$ as claimed.

Conversely, if we consider $\mathsf{DBL}(2c')$ for some positive integer $c' < c$, then the most recent c pages need not always be in $\mathsf{DBL}(2c')$. For example, consider the trace $1, 2, \ldots, c, 1, 2, \ldots, c, \ldots, 1, 2, \ldots, c, \ldots$. For this trace, hit ratio of $\mathsf{LRU}(c)$ approaches 1 as size of the trace increases, but hit ratio of $\mathsf{DBL}(2c')$, for any $c' < c$, is zero. The above remarks shed light on choice of $2c$ as the size of the cache directory DBL.

IV. Adaptive Replacement Cache

A. Fixed Replacement Cache

We now describe a *fixed replacement cache*, $\mathsf{FRC}_p(c)$, in the class $\Pi(c)$, where p is a tunable parameter p, $0 \leq p \leq c$. The policy $\mathsf{FRC}_p(c)$ will satisfy conditions A.1–A.5. For brevity, let us write $T_{1,p} \equiv T_1^{\mathsf{FRC}_p(c)}$, $T_{2,p} \equiv T_2^{\mathsf{FRC}_p(c)}$, $B_{1,p} \equiv B_1^{\mathsf{FRC}_p(c)}$, and $B_{2,p} \equiv B_2^{\mathsf{FRC}_p(c)}$.

The crucial additional condition that $\mathsf{FRC}_p(c)$ satisfies is that it will attempt to keep exactly p pages in the list $T_{1,p}$ and exactly $c - p$ pages in the list $T_{2,p}$. In other words, the policy $\mathsf{FRC}_p(c)$ attempts to keep exactly p top (most recent) pages from the list L_1 and $c - p$ top (most recent) pages from the list L_2 in the cache. Intuitively, the parameter p is the *target size* for

the list $T_{1,p}$. In light of Remark III.1, the replacement policy is:

B.1 If $|T_{1,p}| > p$, replace the LRU page in $T_{1,p}$.
B.2 If $|T_{1,p}| < p$, replace the LRU page in $T_{2,p}$.
B.3 If $|T_{1,p}| = p$ and the missed page is in $B_{1,p}$ (resp. $B_{2,p}$), replace the LRU page in $T_{2,p}$ (resp. $T_{1,p}$).

The last replacement decision is somewhat arbitrary, and can be made differently if desired. The subroutine REPLACE in Figure 4 is consistent with B.1–B.3.

B. The Policy

We now introduce an adaptive replacement policy ARC(c) in the class $\Pi(c)$. At any time, the behavior of the policy ARC is completely described once a certain *adaptation parameter* $p \in [0, c]$ is known. For a given value of p, ARC will behave exactly like FRC$_p$. But, ARC differs from FRC in that it does not use a single fixed value for the parameter p over the entire workload. The policy ARC continuously adapts and tunes p in response to an observed workload.

Let T_1^{ARC}, B_1^{ARC}, T_2^{ARC}, and B_2^{ARC} denote a dynamic partition of L_1 and L_2 corresponding to ARC. For brevity, we will write $T_1 \equiv T_1^{\mathsf{ARC}}$, $T_2 \equiv T_2^{\mathsf{ARC}}$, $B_1 \equiv B_1^{\mathsf{ARC}}$, and $B_2 \equiv B_2^{\mathsf{ARC}}$.

The policy ARC dynamically decides, in response to an observed workload, which item to replace at any given time. Specifically, in light of Remark III.1, on a cache miss, ARC adaptively decides whether to replace the LRU page in T_1 or to replace the LRU page in T_2 depending upon the value of the adaptation parameter p at that time. The intuition behind the parameter p is that it is the *target size* for the list T_1. When p is close to 0 (resp. c), the algorithm can be thought of as favoring L_2 (resp. L_1).

We now exhibit the complete policy ARC in Figure 4. If the two "ADAPTATION" steps are removed, and the parameter p is *a priori* fixed to a given value, then the resulting algorithm is exactly FRC$_p$.

The algorithm in Figure 4 also implicitly simulates the lists $L_1 = T_1 \cup B_1$ and $L_2 = T_2 \cup B_2$. The lists L_1 and L_2 obtained in these fashion are identical to the lists in Figure 1. Specifically, Cases I, II, and III in Figure 4 correspond to Case I in Figure 1. Similarly, Case IV in Figure 4 corresponds to Case II in Figure 1.

C. Learning

The policy continually revises the parameter p. The fundamental intuition behind learning is the following: if there is a hit in B_1 then we should increase the size of T_1, and if there is a hit in B_2 then we should increase the size of T_2. Hence, on a hit in B_1, we increase p which is the target size of T_1, and on a hit in B_2, we decrease p. When we increase (resp. decrease) p, we implicitly decrease (resp. increase) $c - p$ which is the target size of T_2. The precise magnitude of the revision in p is also very important. The quantities δ_1 and δ_2 control the magnitude of revision. These quantities are termed as the *learning rates*. The learning rates depend upon the sizes of the lists B_1 and B_2. On a hit in B_1, we increment p by 1, if the size of B_1 is at least the size of B_2; otherwise, we increment p by $|B_2|/|B_1|$. All increments to p are subject to a cap of c. Thus, smaller the size of B_1, the larger the increment. Similarly, On a hit in B_2, we decrement p by 1, if the size of B_2 is at least the size of B_1; otherwise, we decrement p by $|B_1|/|B_2|$. All decrements to p are subject to a minimum of 0. Thus, smaller the size of B_2, the larger the decrement. The idea is to "invest" in the list that is performing the best. The compound effect of a number of such small increments and decrements to p is quite profound as we will demonstrate in the next section. We can roughly think of the learning rule as inducing a "random walk" on the parameter p.

Observe that ARC never becomes complacent and never stops adapting. Thus, if the workload were to suddenly change from a very stable IRM generated to a transient SDD generated one or vice versa, then ARC will track this change and adapt itself accordingly to exploit the new opportunity. The algorithm continuously revises how to invest in L_1 and L_2 according to the recent past.

D. Scan-Resistant

Observe that a totally new page, that is, a page that is not in $L_1 \cup L_2$, is always put at the MRU position in L_1. From there it gradually makes it way to the LRU position in L_1. It never affects L_2 unless it is used once again before it is evicted from L_1. Hence, a long sequence of one-time-only reads will pass through L_1 without flushing out possibly important pages in L_2. In this sense, ARC is *scan-resistant*; it will only flush out pages in T_1 and never flush out pages in T_2. Furthermore, when a scan begins, arguably, less hits will be encountered in B_1 compared to B_2, and, hence, by the effect of the learning law, the list T_2 will grow at the expense of the list T_1. This further accentuates the resistance of ARC to scans.

E. Extra History

In addition to the c pages in the cache, the algorithm ARC remembers c recently evicted pages. An interesting question is whether we can incorporate more history information in ARC to further improve it. Let us suppose that we are allowed to remember k extra pages. We now demonstrate how to carry out this step. Define a LRU list Z that contains pages that are discarded from the list L_1 in Case IV(A) of the algorithm in Figure 4.

ARC(c)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists T_1, B_1, T_2, and B_2 to empty.

For every $t \geq 1$ and any x_t, one and only one of the following four cases must occur.

Case I: x_t is in T_1 or T_2. A cache hit has occurred in ARC(c) and DBL($2c$).
Move x_t to MRU position in T_2.

Case II: x_t is in B_1. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_1 to the MRU position in T_2 (also fetch x_t to the cache).

Case III: x_t is in B_2. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_2 to the MRU position in T_2 (also fetch x_t to the cache).

Case IV: x_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC(c) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly c pages.
If ($|T_1| < c$)
Delete LRU page in B_1. REPLACE(x_t, p).
else
Here B_1 is empty. Delete LRU page in T_1 (also remove it from the cache).
endif
Case B: $L_1 = T_1 \cup B_1$ has less than c pages.
If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
Delete LRU page in B_2, if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
REPLACE(x_t, p).
endif
Finally, fetch x_t to the cache and move it to MRU position in T_1.

Subroutine REPLACE(x_t, p)
If (($|T_1|$ is not empty) and (($|T_1|$ exceeds the target p) or (x_t is in B_2 and $|T_1| = p$)))
Delete the LRU page in T_1 (also remove it from the cache), and move it to MRU position in B_1.
else
Delete the LRU page in T_2 (also remove it from the cache), and move it to MRU position in B_2.
endif

Fig. 4. Algorithm for Adaptive Replacement Cache. This algorithm is completely self-contained, and can directly be used as a basis for an implementation. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache directory. ARC corresponds to $T_1 \cup T_2$ and DBL corresponds to $T_1 \cup T_2 \cup B_1 \cup B_2$.

Pages that are discarded from the list L_2 are not put on the Z list. The list Z will have a variable time-dependent size. At any time, Z is the longest list such that (a) $|Z| \leq k$ and (b) the least recent page in Z is more recent than the least recent page in the list L_2. The Z list is related to the LIRS stack in [21].

The Z list can be constructed and used as follows. Whenever the LRU page of L_1 is discarded in Case IV(A) of the Figure 4, make the discarded page the MRU page in the list Z. Discard the LRU page in Z, if $|Z| > k$. Now, whenever the LRU page of L_2 is discarded in Case IV(B) of the Figure 4, ensure that the least recent page in Z is more recent than the *new* least recent page in the list L_2; otherwise, discard pages in the Z list until this condition is satisfied. This latter step may have to discard arbitrarily large number of pages from Z, and, hence the resulting algorithm is constant-time in an expected sense only. Finally, on a hit in the list Z, move the hit page to the top of the list L_2. No adaptation takes place on a hit in Z. We refer to the resulting algorithm as ARC(c, k). In our experiments, we will focus on ARC(c)= ARC($c, 0$).

V. Experimental Results

A. Traces

Table III summarizes various traces that we used in this paper. 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
OLTP	914145	186880
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 like	41351279	6050363
S1	3995316	1309698
S2	17253074	1693344
S3	16407702	1689882
Merge (S)	37656092	4692924

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

The trace OLTP has been used in [18], [20], [23]. It contains references to a CODASYL database for a one-hour period. The traces P1–P14 were collected from workstations running Windows NT by using Vtrace which captures disk operations through the use of device filters. The traces were gathered over several months, see [29]. The page size for these traces was 512 bytes. The trace ConCat was obtained by concatenating the traces P1–P14. Similarly, the trace Merge(P) was obtained by merging the traces P1–P14 using time stamps on each of the requests. The idea was to synthesize a trace that may resemble a workload seen by a small storage controller. The trace DS1 was taken off a database server running at a commercial site running an ERP application on top of a commercial database. The trace is seven days long, see [30]. We captured a trace of the SPC1 like synthetic benchmark that contains long sequential scans in addition to random accesses. For precise description of the mathematical algorithms used to generate the benchmark, see [31], [32], [33]. The page size for this trace was 4 KBytes. Finally, we consider three traces S1, S2, and S3 that were disk read accesses initiated by a large commercial search engine in response to various web search requests. The trace S1 was captured over a period of an hour, S2 was captured over roughly four hours, and S3 was captured over roughly six hours. The page size for these traces was 4 KBytes. The trace Merge(S) was obtained by merging the traces S1–S3 using time stamps on each of the requests.

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. OLTP

For this well-studied trace, in Table IV, we compare ARC to a number of algorithms. The tunable parameters for FBR and LIRS were set as in their original papers. The tunable parameters for LRU-2, 2Q, and LRFU were selected in an offline fashion by trying different parameters and selecting the best result for each cache size. The parameter $lifeTime$ of MQ was dynamically estimated, and the history size was set to the cache size resulting in a space overhead comparable to ARC. The algorithm ARC is self-tuning, and requires no user-specified parameters. It can be seen that ARC outperforms all online algorithms, and is comparable to offline algorithms LRU-2, 2Q, and LRFU. The LFU, FBR, LRU-2, 2Q, LRFU, and MIN numbers are exactly the same as those reported in [23].

C. Two Traces: P8 and P12

For two traces P8 and P12, in Table V, we display a comparison of hit ratios achieved by ARC with those of LRU, 2Q, LRU-2, LRFU, and LIRS. The tunable parameters for 2Q, LRU-2, LRFU, and LIRS were selected in an offline fashion by trying different parameters and selecting the best result for each trace and each cache size. It can be seen that ARC outperforms LRU and performs close to 2Q, LRU-2, LRFU, and LIRS even when these algorithms use the best offline parameters. While, for brevity, we have exhibited results on only two traces, the same general results continue to hold for all the traces that we examined [3].

D. ARC *and* 2Q

In Table V, we compared ARC with 2Q where the latter used the best fixed, offline choice of its tunable parameter Kin. In Table VI, we compare ARC to 2Q where the latter is also online and is forced to use "reasonable" values of its tunable parameters, specifically, $Kin = 0.3c$ and $Kout = 0.5c$ [20]. It can be seen that ARC outperforms 2Q.

OLTP

c	LRU	ARC	FBR	LFU	LIRS	MQ	LRU-2	2Q	LRFU	MIN
	ONLINE						OFFLINE			
1000	32.83	38.93	36.96	27.98	34.80	37.86	39.30	40.48	40.52	53.61
2000	42.47	46.08	43.98	35.21	42.51	44.10	45.82	46.53	46.11	60.40
5000	53.65	55.25	53.53	44.76	47.14	54.39	54.78	55.70	56.73	68.27
10000	60.70	61.87	62.32	52.15	60.35	61.08	62.42	62.58	63.54	73.02
15000	64.63	65.40	65.66	56.22	63.99	64.81	65.22	65.82	67.06	75.13

TABLE IV. A comparison of ARC hit ratios with those of various cache algorithms on the OLTP trace. All hit ratios are reported as percentages. It can be seen that ARC outperforms LRU, LFU, FBR, LIRS, and MQ and performs as well as LRU-2, 2Q, and LRFU even when these algorithms use the best offline parameters.

P8

c	LRU	MQ	ARC	2Q	LRU-2	LRFU	LIRS
	ONLINE			OFFLINE			
1024	0.35	0.35	1.22	0.94	1.63	0.69	0.79
2048	0.45	0.45	2.43	2.27	3.01	2.18	1.71
4096	0.73	0.81	5.28	5.13	5.50	3.53	3.60
8192	2.30	2.82	9.19	10.27	9.87	7.58	7.67
16384	7.37	9.44	16.48	18.78	17.18	14.83	15.26
32768	17.18	25.75	27.51	31.33	28.86	28.37	27.29
65536	36.10	48.26	43.42	47.61	45.77	46.72	45.36
131072	62.10	69.70	66.35	69.45	67.56	66.60	69.65
262144	89.26	89.67	89.28	88.92	89.59	90.32	89.78
524288	96.77	96.83	97.30	96.16	97.22	97.38	97.21

P12

c	LRU	MQ	ARC	2Q	LRU-2	LRFU	LIRS
	ONLINE			OFFLINE			
1024	4.09	4.08	4.16	4.13	4.07	4.09	4.08
2048	4.84	4.83	4.89	4.89	4.83	4.84	4.83
4096	5.61	5.61	5.76	5.76	5.81	5.61	5.61
8192	6.22	6.23	7.14	7.52	7.54	7.29	6.61
16384	7.09	7.11	10.12	11.05	10.67	11.01	9.29
32768	8.93	9.56	15.94	16.89	16.36	16.35	15.15
65536	14.43	20.82	26.09	27.46	25.79	25.35	25.65
131072	29.21	35.76	38.68	41.09	39.58	39.78	40.37
262144	49.11	51.56	53.47	53.31	53.43	54.56	53.65
524288	60.91	61.35	63.56	61.64	63.15	63.13	63.89

TABLE V. A comparison of ARC hit ratios with those of various cache algorithms on the traces P8 and P12. All hit ratios are reported in percentages. It can be seen that ARC outperforms LRU and performs close to 2Q, LRU-2, LRFU, and LIRS even when these algorithms use the best offline parameters. On the trace P8, ARC outperforms MQ for some cache sizes, while MQ outperforms ARC for some cache sizes. On the trace P12, ARC uniformly outperforms MQ.

E. ARC *and* MQ

It can be seen from Table V that, for the trace P8, ARC outperforms MQ for some cache sizes, while MQ outperforms ARC for some cache sizes. Furthermore, it can be seen that ARC uniformly outperforms MQ, for the trace P12.

The workloads SPC1 and Merge(S) both represent requests to a storage controller. In Table VI, we compare hit ratios of LRU, MQ, and ARC for these workloads. It can be seen that MQ outperforms LRU, while ARC outperforms both MQ and LRU. These results are quite surprising since the algorithm MQ is designed especially for storage servers.

We will show in Figure 7 that ARC can quickly track an evolving workload, namely, P4, that fluctuates from one extreme of recency to the other of frequency. For the trace P4, in Table VII, we compare hit ratios of LRU, MQ, and ARC. It can be clearly seen that ARC outperforms the other two algorithms. LRU is designed for recency while MQ is designed for workloads with stable temporal distance distributions, and, hence, by design, neither can meaningfully track this workload.

Taken together, Tables V, VI and VII imply that ARC is likely to be effective under a wider range of workloads than MQ. Also, while both have constant-time complexity, ARC has a smaller constant, and, hence, less overhead. Finally, adaptation in ARC requires tuning a single scalar, while adaptation in MQ requires maintaining a histogram of observed temporal distances.

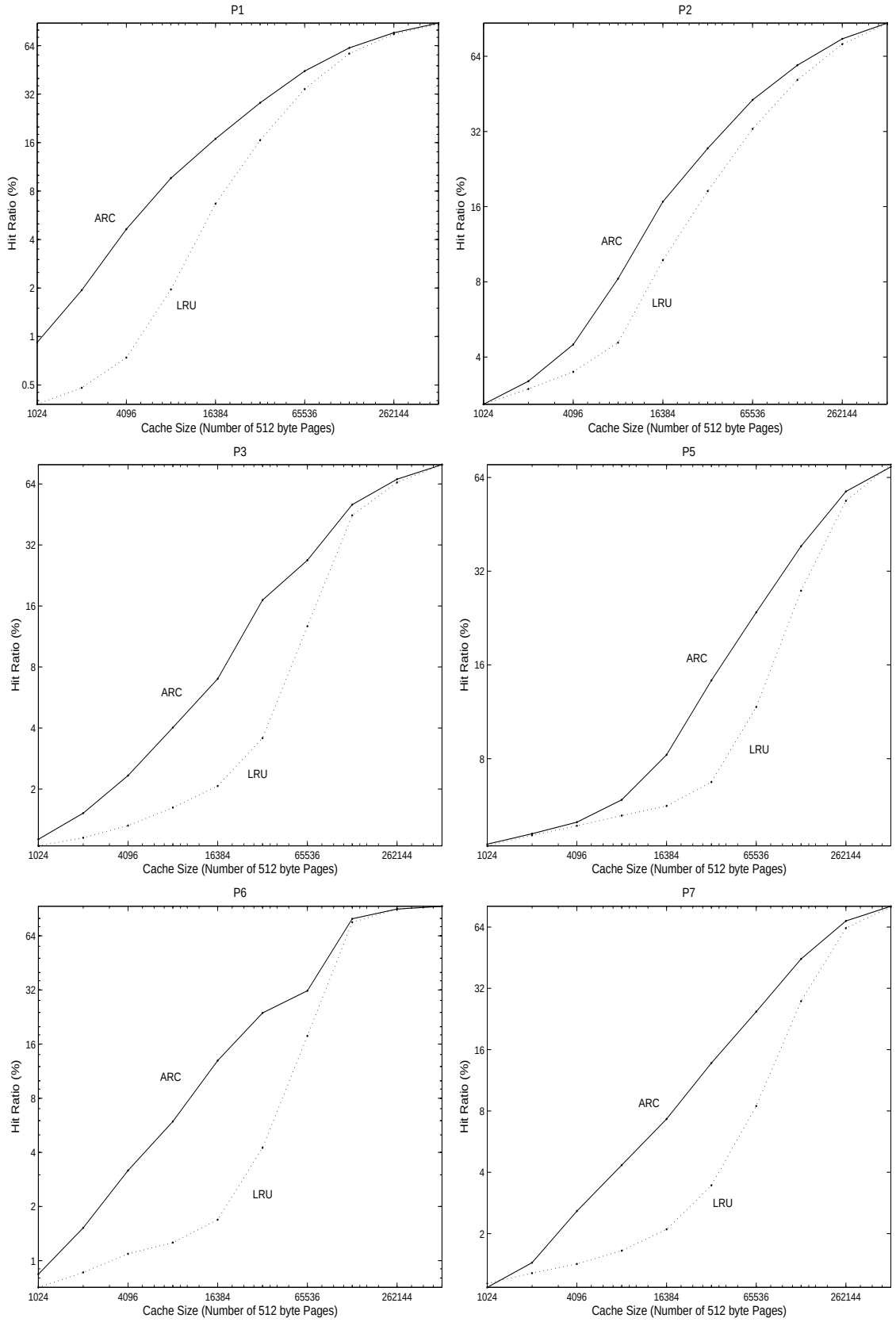

Fig. 5. A plot of hit ratios (in percentages) achieved by ARC and LRU. Both the x- and y-axes use logarithmic scale.

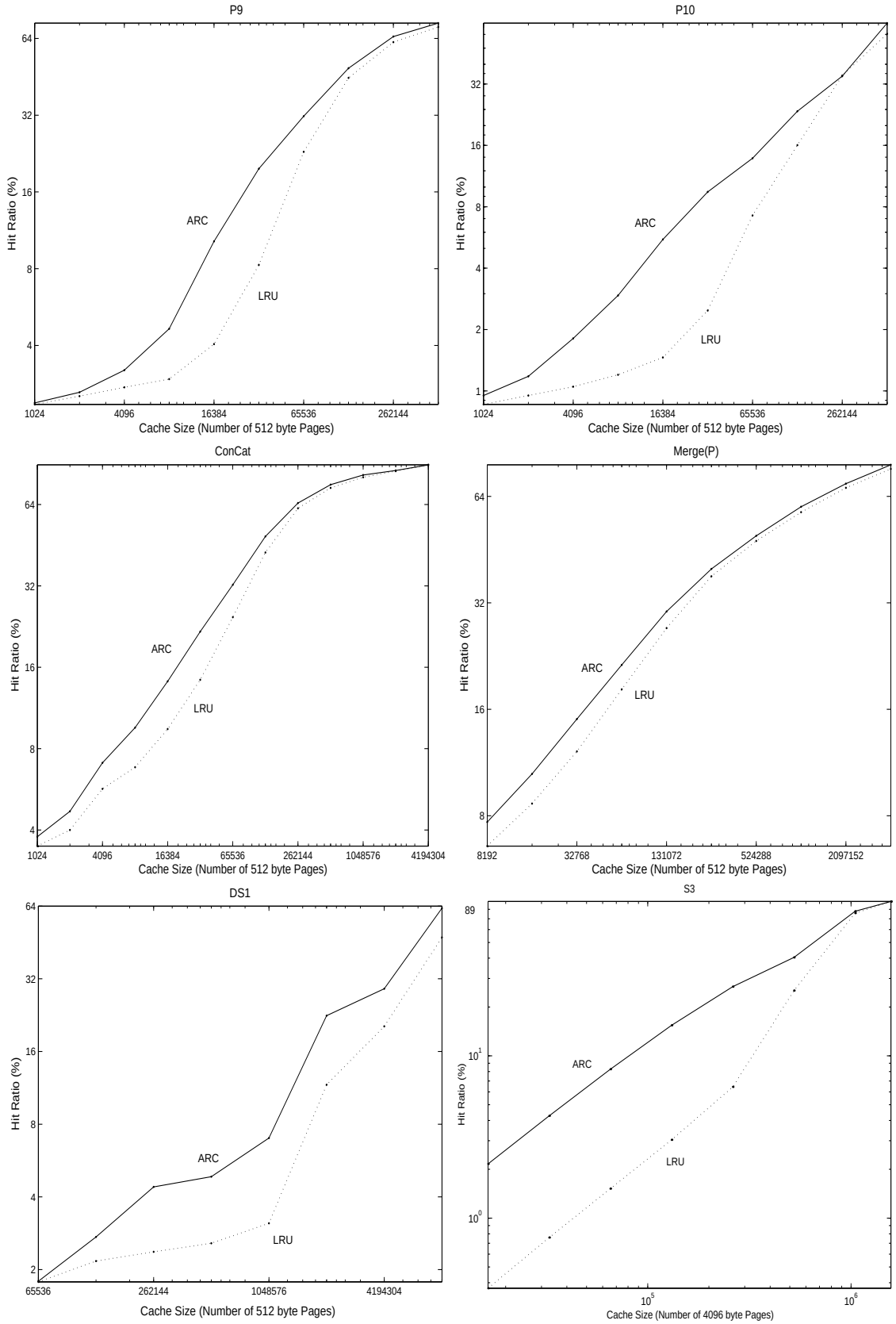

Fig. 6. A plot of hit ratios (in percentages) achieved by ARC and LRU. Both the x- and y-axes use logarithmic scale.

SPC1 like

c	LRU	MQ	2Q	ARC
	ONLINE			
65536	0.37	0.37	0.66	0.82
131072	0.78	0.77	1.31	1.62
262144	1.63	1.65	2.59	3.23
524288	3.66	3.72	6.34	7.56
1048576	9.19	14.96	17.88	20.00

Merge(S)

c	LRU	MQ	2Q	ARC
	ONLINE			
16384	0.20	0.20	0.73	1.04
32768	0.40	0.40	1.47	2.08
65536	0.79	0.79	2.85	4.07
131072	1.59	1.59	5.52	7.78
262144	3.23	4.04	10.36	14.30
524288	8.06	14.89	18.89	24.34
1048576	27.62	40.13	35.39	40.44
1572864	50.86	56.49	53.19	57.19
2097152	68.68	70.45	67.36	71.41
4194304	87.30	87.29	86.22	87.26

TABLE VI. A comparison of hit ratios of LRU, MQ, 2Q, and ARC on the traces SPC1 like and Merge(S). All hit ratios are reported in percentages. The algorithm 2Q is forced to use "reasonable" values of its tunable parameters, specifically, $Kin = 0.3c$ and $Kout = 0.5c$. The page size is 4 KBytes for both traces. The largest cache simulated for SPC1 like was 4 GBytes and that for Merge(S) was 16 GBytes.

P4

c	LRU	MQ	ARC
	ONLINE		
1024	2.68	2.67	2.69
2048	2.97	2.96	2.98
4096	3.32	3.31	3.50
8192	3.65	3.65	4.17
16384	4.07	4.08	5.77
32768	5.24	5.21	11.24
65536	10.76	12.24	18.53
131072	21.43	24.54	27.42
262144	37.28	38.97	40.18
524288	48.19	49.65	53.37

TABLE VII. A comparison of hit ratios of LRU, MQ, and ARC on the trace P4. All hit ratios are reported as percentages. It can be seen that ARC outperforms the other two algorithms.

F. ARC *and* LRU

We now focus on LRU which is the single most widely used cache replacement policy. For all the traces listed in Section V-A, we now plot the hit ratio of ARC versus LRU in Figures 5 and 6. The traces P4, P8, P12, SPC1 like, and Merge(S) are not plotted since they have been discussed in various tables. The traces S1 and S2 are not plotted since their results are virtually identical to S3 and Merge(S). The traces P11, P13, and P14 are not plotted for space consideration; for these traces, ARC was uniformly better than LRU. It can be clearly seen that ARC substantially outperforms LRU on virtually all the traces and for all cache sizes. In Table VIII, we present an at-a-glance comparison of ARC with LRU for all the traces–where for each trace we selected a practically relevant cache size. The trace SPC1 contains long sequential scans interspersed with random requests. It can be seen that even for this trace ARC, due to its scan-resistance, continues to outperform LRU.

Workload	c	space MB	LRU	ARC	FRC OFFLINE
P1	32768	16	16.55	28.26	29.39
P2	32768	16	18.47	27.38	27.61
P3	32768	16	3.57	17.12	17.60
P4	32768	16	5.24	11.24	9.11
P5	32768	16	6.73	14.27	14.29
P6	32768	16	4.24	23.84	22.62
P7	32768	16	3.45	13.77	14.01
P8	32768	16	17.18	27.51	28.92
P9	32768	16	8.28	19.73	20.28
P10	32768	16	2.48	9.46	9.63
P11	32768	16	20.92	26.48	26.57
P12	32768	16	8.93	15.94	15.97
P13	32768	16	7.83	16.60	16.81
P14	32768	16	15.73	20.52	20.55
ConCat	32768	16	14.38	21.67	21.63
Merge(P)	262144	128	38.05	39.91	39.40
DS1	2097152	1024	11.65	22.52	18.72
SPC1	1048576	4096	9.19	20.00	20.11
S1	524288	2048	23.71	33.43	34.00
S2	524288	2048	25.91	40.68	40.57
S3	524288	2048	25.26	40.44	40.29
Merge(S)	1048576	4096	27.62	40.44	40.18

TABLE VIII. At-a-glance comparison of hit ratios of LRU and ARC for various workloads. All hit ratios are reported in percentages. It can be seen that ARC outperforms LRU–sometimes quite dramatically. Also, ARC which is online performs very close to FRC with the best fixed, offline choice of the parameter p.

G. ARC *is Self-Tuning and Empirically Universal*

We now present the most surprising and intriguing of our results. In Table VIII, *it can be seen that* ARC*, which tunes itself, performs as well as (and sometimes even better than) the policy* FRC_p *with the best fixed, offline selection of the parameter* p*.* This result holds for all the traces. In this sense, ARC is empirically universal.

It can be seen in Table VIII that ARC can sometimes outperform offline optimal FRC_p. This happens, for example, for the trace P4. For this trace, Figure 7 shows that the parameter p fluctuates over its entire range. Since ARC is adaptive, it tracks the variation in the workload by dynamically tuning p. In contrast, FRC_p must use a single, fixed choice of p throughout the entire workload; the offline optimal value was $p = 0.1c$. Hence, the performance benefit of ARC.

On the other hand, ARC can also be slightly worse than offline optimal FRC_p. This happens, for example, for the trace P8. Throughout this trace, ARC maintains a

very small value of the parameter p, that is, it favors frequency over the entire workload. In this case, arguably, there exists a small range of optimal parameters. Now, due to its constant adaptation, ARC will never lock into a single, fixed parameter, but rather keeps fluctuating around the optimal choice. This fluctuations cost ARC slightly over offline optimal FRC_p–in terms of the hit ratio. Hence, for stable workloads, we expect ARC to be slightly worse than offline optimal FRC_p. Nonetheless, the latter is a theoretical construct that is not available in practice. Moreover, even for stable workloads, the value of best fixed, offline parameter p depends on the workload and the cache size. The policy ARC provides a reasonable, online approximation to offline optimal FRC_p without requiring any *a priori* workload-specific or cache size-specific tuning.

H. A Closer Examination of Adaptation in ARC

We now study the adaptation parameter p more closely. For the trace P4 and for cache size $c = 32768$ pages, in Figure 7, we plot the parameter p versus the virtual time (or the number of requests). When p is close to zero, ARC can be thought of as emphasizing the contents of the list L_2, and, when p is close to the cache size, 32768, ARC can be thought of as emphasizing the contents of the list L_1. It can be seen that the parameter p keeps fluctuating between these two extremes. Also, it can be seen that it touches both the extremes. As a result, ARC continually adapts and reconfigures itself. Quite dramatically, the policy ARC can fluctuate from frequency to recency and then back all within a single workload. Moreover, such fluctuations may occur as many times as dictated by the nature of the workload without any *a priori* knowledge or offline tuning. Poetically, at any time, p dances to the tune being played by the workload. It is this continuous, unrelenting adaptation in response to the changing and evolving workload that allows ARC to outperform LRU.

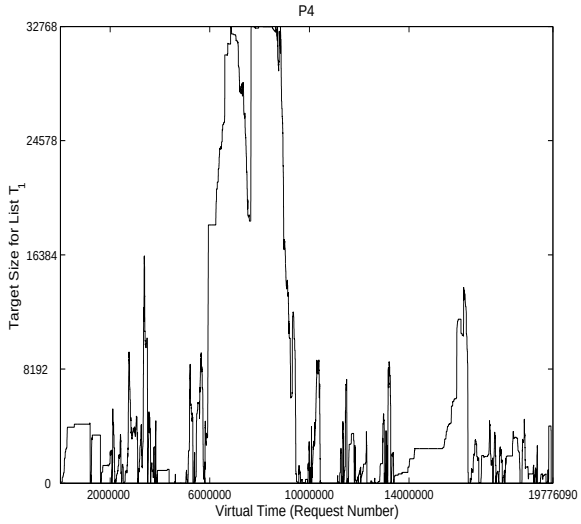

Fig. 7. A plot of the adaptation parameter p (the target size for list T_1) versus the virtual time for the trace P4. The cache size was 32768 pages. The page size was 512 bytes.

VI. Conclusions

We have reviewed various recent cache replacement algorithms, namely, LRU-2, 2Q, LRFU, LIRS. Performance of these algorithms depends crucially on their respective tunable parameters. Hence, no single universal rule of thumb can be used to *a priori* select the tunables of these algorithms. In addition, we have demonstrated that the computational overhead of LRU-2 and LRFU makes them practically less attractive.

We have presented a new cache replacement policy ARC that is online and self-tuning. We have empirically demonstrated that ARC adapts to a given workload dynamically. We have empirically demonstrated that ARC performs as well as (and sometimes even better than) FRC_p with the best offline choice of the parameter p for each workload and cache size. Similarly, we have also shown that ARC which is online performs as well as LRU-2, 2Q, LRFU, and LIRS–even when these algorithms use the best offline values for their respective tuning parameters. We have demonstrated that ARC outperforms 2Q when the latter is forced to be online and use "reasonable" values of its tunable parameters. We have demonstrated that performance of ARC is comparable to and often better than MQ even in the specific domain for which the latter was designed. Moreover, in contrast to MQ, we have shown that ARC is robust for a wider range of workloads and has less overhead. We have demonstrated that ARC has overhead comparable to that of LRU, and, hence, is a low overhead policy. We have argued that ARC is scan-resistant. Finally, and most importantly, we have shown that ARC substantially outperforms LRU virtually uniformly across numerous different workloads and at various different cache sizes.

Our results show that there is considerable room for performance improvement in modern caches by using adaptation in cache replacement policy. We hope that ARC will be seriously considered by cache designers as a suitable alternative to LRU.

Endnotes

1. Our use of *universal* is motivated by a similar use in data compression [11] where a coding scheme is termed universal if–even without any knowledge of the source that generated the string–it asymptotically compresses a given string as well as a coding scheme that knows the statistics of the generating source.

2. We use the legally correct term "SPC1 like", since, as per the benchmark specification, to use the term "SPC1" we must also quote other performance numbers that are not of interest here.

3. Due to extremely large overhead and numerical instabilities, we were not able to simulate LRFU and LRU-2 for larger traces such as ConCat and Merge(P) and for larger cache sizes beyond 512 MBytes.

ACKNOWLEDGMENT

We are grateful to Jai Menon for holding out the contrarian belief that cache replacement policies can be improved. The second author is grateful to his managers, Moidin Mohiuddin and Bill Tetzlaff, for their constant support and encouragement during this work. We are grateful to Brent Beardsley, Pawan Goyal, Robert Morris, William Tetzlaff, Renu Tewari, and Honesty Young for valuable discussions and suggestions. 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, to Gerhard Weikum for the CODASYL trace, to Ken Bates and Bruce McNutt for traces S1-S3, to Song Jiang for sharing his LIRS simulator, and to Sang Lyul Min and Donghee Lee for sharing their LRFU simulator and for various numbers reported in Table IV. We would like to thank Peter Chen, our FAST shepherd, and anonymous referees for valuable suggestions that greatly improved this paper.

REFERENCES

[1] J. Z. Teng and R. A. Gumaer, "Managing IBM database 2 buffers to maximize performance," *IBM Sys. J.*, vol. 23, no. 2, pp. 211–218, 1984.

[2] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. USENIX Symp. Internet Technologies and Systems, Monterey, CA*, 1997.

[3] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou, "A middleware system which intelligently caches query results," in *Middleware 2000*, vol. LNCS 1795, pp. 24–44, 2000.

[4] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache performance of the SPEC benchmark suite," Tech. Rep. CS-TR-1991-1049, University of California, Berkeley, 1991.

[5] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134–154, 1988.

[6] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 161–203, 1985.

[7] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, 1994.

[8] M. J. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[9] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Comm. ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[10] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Comm. ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[12] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Sys. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[14] P. J. Denning, "Working sets past and present," *IEEE Trans. Software Engineeing*, vol. SE-6, no. 1, pp. 64–84, 1980.

[15] 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.

[16] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[17] 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.

[18] 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.

[19] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-K page replacement algorithm," *J. ACM*, vol. 46, no. 1, pp. 92–112, 1999.

[20] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. VLDB Conf.*, pp. 297–306, 1994.

[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] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–142, 1990.

[23] 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.

[24] 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.

[25] 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.

[26] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari, "Adaptive caching by refetching," in *NIPS*, 2002.

[27] I. Ari, A. Amer, R. Gramarcy, E. Miller, S. Brandt, and D. Long, "ACME: Adaptive caching using multiple experts," in *Proceedings of the Workshop on Distributed Data and Structures (WDAS), Carleton Scientific*, 2002.

[28] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proc. USENIX Annual Tech. Conf. (USENIX 2002), Monterey, CA*, pp. 161–175, June 2002.

[29] W. W. Hsu, A. J. Smith, and H. C. Young, "The automatic improvement of locality in storage systems." Tech. Rep., Computer Science Division, Univ. California, Berkeley, Nov. 2001.

[30] W. W. Hsu, A. J. Smith, and H. C. Young, "Characteristics of I/O traffic in personal computer and server workloads." Tech. Rep., Computer Science Division, Univ. California, Berkeley, July 2001.

[31] B. McNutt and S. A. Johnson, "A standard test of I/O cache," in *Proc. the Computer Measurements Group's 2001 International Conference*, 2001.

[32] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.

[33] B. McNutt, *The Fractal Structure of Data Reference: Applications to the Memory Hierarchy*. Boston, MA: Kluwer Academic Publishers, 2000.