Outperforming LRU with an Adaptive Replacement Cache Algorithm

The self-tuning, low-overhead, scan-resistant adaptive replacement cache algorithm outperforms the least-recently-used algorithm by dynamically responding to changing access patterns and continually balancing between workload recency and frequency features.

Nimrod Megiddo

Dharmendra S. Modha
IBM Almaden Research Center

Caching, a fundamental metaphor in modern computing, finds wide application in storage systems,[1] databases, Web servers, middleware, processors, file systems, disk drives, redundant array of independent disks controllers, operating systems, and other applications such as data compression and list updating.[2] In a two-level memory hierarchy, a cache performs faster than auxiliary storage, but it is more expensive. Cost concerns thus usually limit cache size to a fraction of the auxiliary memory's size.

Both cache and auxiliary memory handle uniformly sized items called *pages*. Requests for pages go first to the cache. When a page is found in the cache, a *hit* occurs; otherwise, a cache *miss* happens, and the request goes to the auxiliary memory. In the latter case, a copy is *paged in* to the cache. This practice, called *demand paging*, rules out prefetching pages from the auxiliary memory into the cache. If the cache is full, before the system can page in a new page, it must page out one of the currently cached pages. A *replacement policy* determines which page is evicted.

A commonly used criterion for evaluating a replacement policy is its *hit ratio*—the frequency with which it finds a page in the cache. Of course, the replacement policy's implementation overhead should not exceed the anticipated time savings.

Discarding the least-recently-used page is the policy of choice in cache management. Until recently, attempts to outperform LRU in practice had not succeeded because of overhead issues and the need to pretune parameters. The *adaptive replacement cache* is a self-tuning, low-overhead algorithm that responds online to changing access patterns. ARC continually balances between the recency and frequency features of the workload, demonstrating that adaptation eliminates the need for the workload-specific pretuning that plagued many previous proposals to improve LRU.

ARC's online adaptation will likely have benefits for real-life workloads due to their richness and variability with time. These workloads can contain long sequential I/Os or moving hot spots, changing frequency and scale of temporal locality and fluctuating between stable, repeating access patterns and patterns with transient clustered references.

Like LRU, ARC is easy to implement, and its running time per request is essentially independent of the cache size. A real-life implementation revealed that ARC has a low space overhead—0.75 percent of the cache size. Also, unlike LRU, ARC is *scan-resistant* in that it allows one-time sequential requests to pass through without polluting the cache or flushing pages that have temporal locality. Likewise, ARC also effectively handles long periods of low temporal locality. ARC leads to sub-

stantial performance gains in terms of an improved hit ratio compared with LRU for a wide range of cache sizes.

ARC INTUITION

ARC maintains two LRU page lists: L_1 and L_2. <mark>L_1 maintains pages that have been seen only once, recently, while L_2 maintains pages that have been seen at least twice, recently.</mark> The algorithm actually caches only a fraction of the pages on these lists. The pages that have been seen twice within a short time may be thought of as having high frequency or as having longer term reuse potential. Hence, we say that <mark>L_1 captures *recency*, while L_2 captures *frequency*.</mark>

If the cache can hold c pages, we strive to keep these two lists to roughly the same size, c. Together, the two lists comprise a cache directory that holds at most $2c$ pages. ARC caches a variable number of most recent pages from both L_1 and L_2 such that the total number of cached pages is c. ARC continually adapts the precise number of pages from each list that are cached.

To contrast an adaptive approach with a nonadaptive approach, suppose FRC_p provides a fixed-replacement policy that attempts to keep in cache the p most recent pages from L_1 and the $c - p$ most recent pages in L_2. Thus, ARC behaves like FRC_p except that it can vary p adaptively. We introduce a learning rule that lets ARC adapt quickly and effectively to a variable workload.

Many algorithms use recency and frequency as predictors of the likelihood that pages will be reused in the future. ARC acts as an adaptive filter to detect and track temporal locality. If either recency or frequency becomes more important at some time, ARC will detect the change and adapt its investment in each of the two lists accordingly.

ARC works as well as the policy FRC_p, even when that policy uses hindsight to choose the best fixed p with respect to the particular workload and the cache size. Surprisingly, ARC, which operates completely online, delivers performance comparable to several state-of-the-art cache-replacement policies, even when, with hindsight, these policies choose the best fixed values for their tuning parameters. ARC matches LRU's ease of implementation, requiring only two LRU lists.

CACHE REPLACEMENT ALGORITHMS

Laszlo A. Belady's MIN[1,3] is an optimal, offline policy for replacing the page in the cache that has the greatest distance to its next occurrence. The LRU policy always replaces the least-recently-used

page. In use for decades, this policy has undergone numerous approximations and improvements. Three of the most important related algorithms are Clock,[4] WS (working set),[5] and WSClock.[6] If the request stream is drawn from the LRU *stack depth distribution*, LRU offers the optimal policy.[7] Simple to implement, LRU responds well to deviations from the underlying SDD model. While SDD captures recency, it does not capture frequency.[7]

The *independent reference model* captures the notion of page reference frequencies. Under IRM, requests received at different times are stochastically independent. LFU replaces the least-frequently-used page and is optimal under IRM,[7,8] but it has several drawbacks: LFU's running time per request is logarithmic in the cache size, it is oblivious to recent history, and it adapts poorly to variable access patterns by accumulating stale pages with past high-frequency counts, which may no longer be useful.

LRU-2[9] represents significant practical progress, approximating the original LFU but working adaptively. LRU-2 memorizes the times for each cache page's two most recent occurrences and replaces the page with the least second-most-recent occurrence. Under IRM, LRU-2 has the maximum expected hit ratio of any online algorithm, which knows at most the two most recent references to each page,[9] and it works well on several traces.[10] However, LRU-2 suffers from two practical drawbacks:[10] It uses a priority queue, which gives it logarithmic complexity, and it must tune the *parameter-correlated information period*.

Logarithmic complexity is a severe practical drawback that 2Q, an improved method with constant complexity, alleviates.[10] It resembles LRU-2, except that it uses a simple LRU list instead of a priority queue. ARC's low computational overhead resembles 2Q's. The choice of correlated information period crucially affects LRU-2's performance. No single a priori fixed choice works uniformly well across various cache sizes and workloads. This LRU-2 drawback persists even in 2Q.

The *low inter-reference recency set's* design[11] builds upon 2Q. LIRS maintains a variable size LRU stack of potentially unbounded size that serves as a cache directory. From this stack, LIRS selects a few top pages, depending on two parameters that crucially affect its performance: A certain choice works well for stable IRM workloads, while other choices work well for SDD workloads. Due to a certain stack pruning operation, LIRS has

> **ARC acts as an adaptive filter to detect and track temporal locality.**

average-case rather than worst-case constant-time overhead, which is a significant practical drawback.

Frequency-based replacement[12] maintains an LRU list but partitions it into three sections—new, middle, and old—and moves pages between them. FBR also maintains frequency counts for individual pages. The idea of *factoring out locality* works on the theory that if the hit page is stored in the new section, the reference count would not increment. On a cache miss, the system replaces the page in the old section that has the least-reference count. FBR's drawbacks include its need to rescale the reference counts periodically and its tunable parameters.

The *least-recently/frequently-used* (LRFU) policy subsumes LRU and LFU.[13] It assigns a value $C(x) = 0$ to every page x and, depending on a parameter $\lambda > 0$, after every cache access, updates $C(x) = 1 + 2^{-\lambda}C(x)$ if x is referenced and $C(x) = 2^{-\lambda}C(x)$ otherwise. This approach resembles the exponential smoothing statistical forecasting method. LRFU replaces the page with the least $C(x)$ value. As λ tends to 0, $C(x)$ tends to the number of occurrences of x and LRFU collapses to LFU. As λ tends to 1, $C(x)$ emphasizes recency and LRFU collapses to LRU. The performance depends crucially on λ.[13] ALRFU, an adaptive LRFU, adjusts λ dynamically.

LRFU has two drawbacks. First, both LRFU and ALRFU require a tunable parameter for controlling correlated references.[13] Second, LRFU's complexity fluctuates between constant and logarithmic. The required calculations make its practical complexity significantly higher than that of even LRU-2. For small λ, LRFU can be 50 times slower than LRU and ARC. This can potentially wipe out the benefit of a high hit ratio.

The *multiqueue* replacement policy[14] uses m queues, where for $0 \leq i \leq m - 1$, the ith queue contains pages that have been seen at least 2^i times but no more than $2^{i+1} - 1$ times recently. The MQ algorithm also maintains a history buffer. On a hit, the page frequency increments, the page is placed at the appropriate queue's most recently used (MRU) position, and the page's *expireTime* is set to *currentTime* + *lifeTime*, where *lifeTime* is a tunable parameter. On each access, the memory checks the *expireTime* for the LRU page in each queue and, if it is less than *currentTime*, moves the page to the next lower queue's MRU position.

To estimate the parameter *lifeTime*, MQ assumes that the distribution of temporal distances between consecutive accesses to a single page has a certain hill shape. Because ARC makes no such assumption, it will likely be robust under a wider range of workloads. Also, MQ will adjust to workload evolution when it can detect a measurable change in peak temporal distance, whereas ARC will track an evolving workload nimbly because it adapts continually. While MQ has constant-time overhead, it still needs to check LRU page time stamps for m queues on every request and hence has a higher overhead than LRU, ARC, and 2Q.

In contrast to the LRU-2, 2Q, LIRS, FBR, and LRFU algorithms—which all require offline selection of tunable parameters—our ARC replacement policy functions online and is completely self-tuning. Because ARC maintains no frequency counts, unlike LFU and FBR, it does not suffer from periodic rescaling requirements. Also, unlike LIRS, ARC does not require potentially unbounded space overhead. Finally, ARC, 2Q, LIRS, and FBR have constant-time implementation complexity while LFU, LRU-2, and LRFU have logarithmic implementation complexity.

CACHE AND HISTORY

Let c be the cache size in pages. We introduce a policy, DBL($2c$), that memorizes $2c$ pages and manages an imaginary cache of size $2c$, and also introduce a class II(c) of cache replacement policies.

DBL($2c$) maintains two LRU lists: L_1 that contains pages that have been seen recently only once and L_2 that contains pages that have been seen recently at least twice. More precisely, a page resides in L_1 if it 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 never removed from $L_1 \cup L_2$. Similarly, a page resides 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 policy functions as follows: If L_1 contains exactly c pages, replace the LRU page in L_1; otherwise, replace the LRU page in L_2. Initially, the lists are empty: $L_1 = L_2 = \varnothing$. If a requested page resides in $L_1 \cup L_2$, the policy moves it to the MRU position of L_2; otherwise, it moves to the MRU position of L_1. In the latter case, if $|L_1| = c$, then the policy removes the LRU member of L_1 and, if $|L_1| < c$ and $|L_1| + |L_2| = 2c$, the policy removes the LRU member of L_2. Thus, the constraints $0 \leq |L_1| + |L_2| \leq 2c$ and $0 \leq |L_1| \leq c$ on the list sizes are maintained throughout.

We propose a class II(c) of policies that track all the $2c$ items that would be present in a cache of size $2c$ managed by DBL($2c$), but at most c are actually kept in cache. Thus, L_1 is partitioned into

ARC(c)   INITIALIZE $T_1 = B_1 = T_2 = B_2 = 0$, $p = 0$. x - requested page.

Case I. $x \in T_1 \cup T_2$ (a hit in ARC(c) and DBL($2c$)): Move x to the top of T_2.

Case II. $x \in B_1$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \min\{c, p + \max\{|B_2|/|B_1|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case III. $x \in B_2$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \max\{0, p - \max\{|B_1|/|B_2|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case IV. $x \in L_1 \cup L_2$ (a miss in DBL($2c$) and ARC(c)):

case (i) $|L_1| = c$:
if $|T_1| < c$ then delete the LRU page of B_1 and REPLACE(p).
else delete LRU page of T_1 and remove it from the cache.
case (ii) $|L_1| < c$ and $|L_1| + |L_2| \geq c$:
if $|L_1| + |L_2| = 2c$ then delete the LRU page of B_2.
REPLACE(p).

Put x at the top of T_1 and place it in the cache.

Subroutine REPLACE(p)

if ($|T_1| \geq 1$) and (($x \in B_2$ and $|T_1| = p$) or ($|T_1| > p$)) then move the LRU page of T_1 to the top of B_1 and remove it from the cache.

else move the LRU page in T_2 to the top of B_2 and remove it from the cache.

- T_1, which contains the top or most-recent pages in L_1, and
- B_1, which contains the bottom or least-recent pages in L_1.

Similarly, L_2 is partitioned into top T_2 and bottom B_2, subject to the following conditions:

- If $|L_1| + |L_2| < c$, then $B_1 = B_2 = \varnothing$.
- If $|L_1| + |L_2| > c$-1, then $|T_1| + |T_2| = c$.
- For $i = 1, 2$, either T_i or B_i is empty or the LRU page in T_i is more recent than the MRU page in B_i.
- Throughout, $T_1 \cup T_2$ contains exactly those pages, which would be cached under a policy in the class.

The pages in T_1 and T_2 reside in the cache directory and in the cache, but the history pages in B_1 and B_2 reside only in the cache directory, not in the cache. Once the cache directory has $2c$ pages, $T_1 \cup T_2$ and $B_1 \cup B_2$ will both contain exactly c pages thenceforth. ARC will leverage the extra history information in $B_1 \cup B_2$ to effect a continual adaptation. It can be shown that the policy LRU(c) is in the class II(c). Conversely, for $0 < c' < c$, the most recent c pages do not always need to be in DBL($2c'$). This justifies the choice to maintain c history pages.

ADAPTIVE REPLACEMENT CACHE

A *fixed replacement cache* FRC$_p$(c)—with a tunable parameter p, $0 \leq p \leq c$, in the class II(c)—

attempts to keep in cache the p most recent pages from L_1 and the $c - p$ most recent pages in L_2. Use x to denote the requested page.

- If either $|T_1| > p$ or ($|T_1| = p$ and $x \in B_2$), replace the LRU page in T_1.
- If either $|T_1| < p$ or ($|T_1| = p$ and $x \in B_1$), replace the LRU page in T_2.

Roughly speaking, p is the current *target size* for the list T_1. ARC behaves like FRC$_p$, except that p changes adaptively. Figure 1 describes the complete ARC policy.

Intuitively, a hit in B_1 suggests an increase in the size of T_1, and a hit in B_2 suggests an increase in the size of T_2. The continual updates of p effect these increases. The amount of change in p is important. The learning rates depend on the relative sizes of B_1 and B_2. ARC attempts to keep T_1 and B_2 to roughly the same size and also T_2 and B_1 to roughly the same size.

On a hit in B_1, p increments by $\max\{|B_2|/|B_1|, 1\}$ but does not exceed c. Similarly, on a hit in B_2, p decrements by $\max\{|B_1|/|B_2|, 1\}$, but it never drops below zero. When taken together, numerous such small increments and decrements to p have a profound effect. ARC never stops adapting, so it always responds to workload changes from IRM to SDD and vice versa.

Because $L_1 \cup L_2 = T_1 \cup T_2 \cup B_1 \cup B_2$ always contains the LRU c pages, LRU cannot experience cache hits unbeknownst to ARC, but ARC can and often

Table 1. Comparison between ARC and other algorithms on an online transaction processing workload.

Cache (512-byte pages)	Online hit ratios (%)						Offline hit ratios (%)			
	ARC	LRU	LFU	FBR	LIRS	MQ	LRU-2	2Q	LRFU	MIN
1,000	38.93	32.83	27.98	36.96	34.80	37.86	39.30	40.48	40.52	53.61
2,000	46.08	42.47	35.21	43.98	42.51	44.10	45.82	46.53	46.11	60.40
5,000	55.25	53.65	44.76	53.53	47.14	54.39	54.78	55.70	56.73	68.27
10,000	61.87	60.70	52.15	62.32	60.35	61.08	62.42	62.58	63.54	73.02
15,000	65.40	64.63	56.22	65.66	63.99	64.81	65.22	65.82	67.06	75.13

Table 2. Comparison between ARC and other algorithms on trace P8.

Cache (512-byte pages)	Online hit ratios (%)			Offline hit ratios (%)			
	ARC	LRU	MQ	2Q	LRU-2	LRFU	LIRS
1,024	1.22	0.35	0.35	0.94	1.63	0.69	0.79
2,048	2.43	0.45	0.45	2.27	3.01	2.18	1.71
4,096	5.28	0.73	0.81	5.13	5.50	3.53	3.60
8,192	9.19	2.30	2.82	10.27	9.87	7.58	7.67
16,384	16.48	7.37	9.44	18.78	17.18	14.83	15.26
32,768	27.51	17.18	25.75	31.33	28.86	28.37	27.29
65,536	43.42	36.10	48.26	47.61	45.77	46.72	45.36
131,072	66.35	62.10	69.70	69.45	67.56	66.60	69.65
262,144	89.28	89.26	89.67	88.92	89.59	90.32	89.78
524,288	97.30	96.77	96.83	96.16	97.22	97.38	97.21

Table 3. Comparison between ARC and other algorithms on trace P12.

Cache (512-byte pages)	Online hit ratios (%)			Offline hit ratios (%)			
	ARC	LRU	MQ	2Q	LRU-2	LRFU	LIRS
1,024	4.16	4.09	4.08	4.13	4.07	4.09	4.08
2,048	4.89	4.84	4.83	4.89	4.83	4.84	4.83
4,096	5.76	5.61	5.61	5.76	5.81	5.61	5.61
8,192	7.14	6.22	6.23	7.52	7.54	7.29	6.61
16,384	10.12	7.09	7.11	11.05	10.67	11.01	9.29
32,768	15.94	8.93	9.56	16.89	16.36	16.35	15.15
65,536	26.09	14.43	20.82	27.46	25.79	25.35	25.65
131,072	38.68	29.21	35.76	41.09	39.58	39.78	40.37
262,144	53.47	49.11	51.56	53.31	53.43	54.56	53.65
524,288	63.56	60.91	61.35	61.64	63.15	63.13	63.89

L_1 L_2

does experience cache hits unbeknownst to LRU.

If a page is not in $L_1 \cup L_2$, the system places it at the top of L_1. From there, it makes its way to the LRU position in L_1; unless requested once again prior to being evicted from L_1, it never enters L_2.

Hence, a long sequence of read-once requests passes through L_1 without flushing out possibly important pages in L_2. In this sense, ARC is scan resistant. Arguably, when a scan begins, fewer hits occur in B_1 compared to B_2. Hence, by the effect of the learn-

Table 4. Comparison of ARC and LRU hit ratios (in percentages) for various workloads.

Workload	Cache (pages)	Cache (Mbytes)	LRU	ARC	FRC$_p$ (Offline)
P1	32,768	16	16.55	28.26	29.39
P2	32,768	16	18.47	27.38	27.61
P3	32,768	16	3.57	17.12	17.60
P4	32,768	16	5.24	11.24	9.11
P5	32,768	16	6.73	14.27	14.29
P6	32,768	16	4.24	23.84	22.62
P7	32,768	16	3.45	13.77	14.01
P8	32,768	16	17.18	27.51	28.92
P9	32,768	16	8.28	19.73	20.28
P10	32,768	16	2.48	9.46	9.63
P11	32,768	16	20.92	26.48	26.57
P12	32,768	16	8.93	15.94	15.97
P13	32,768	16	7.83	16.60	16.81
P14	32,768	16	15.73	20.52	20.55
ConCat	32,768	16	14.38	21.67	21.63
Merge(P)	262,144	128	38.05	39.91	39.40
DS1	2,097,152	1,024	11.65	22.52	18.72
SPC1-like	1,048,576	4,096	9.19	20.00	20.11
S1	524,288	2,048	23.71	33.43	34.00
S2	524,288	2,048	25.91	40.68	40.57
S3	524,288	2,048	25.26	40.44	40.29
Merge(S)	1,048,576	4,096	27.62	40.44	40.18

ing law, list T_2 will grow at the expense of list T_1. This further accentuates ARC's resistance to scans.

EXPERIMENTAL RESULTS

We compared the performance of various algorithms on various traces. OLTP[10,13] contains an hour's worth of references to a Codasyl database. We collected P1 through P14 over several months from Windows NT workstations,[15] obtained ConCat by concatenating traces P1 through P14, then merged them using time stamps on each request to obtain Merge(P). We took DS1, a seven-day trace, from a commercial database server. All these traces have a page size of 512 bytes.

We also captured a trace of the Storage Performance Council's SPC1-like synthetic benchmark, which contains long sequential scans in addition to random accesses and has a page size of 4 Kbytes.

Finally, we considered three traces—S1, S2, and S3—that perform disk-read accesses initiated by a large commercial search engine in response to various Web search requests over several hours. These traces have a page size of 4 Kbytes. We obtained the trace Merge(S) by merging the traces S1 through S3 using time stamps on each request. All hit ratios are cold starts and are reported in percentages.

Table 1 compares ARC's hit ratios to the hit ratios of several algorithms on the OLTP trace. We set the tunable parameters for FBR and LIRS according to their original descriptions. We selected the tunable parameters of LRU-2, 2Q, and LRFU offline for the best result for each cache size. ARC requires no user-specified parameters. We tuned MQ online.[14] The LFU, FBR, LRU-2, 2Q, LRFU, and MIN parameters exactly match those in the LRFU policy.[13]

ARC outperforms LRU, LFU, FBR, LIRS, and MQ. Further, it performs as well as LRU-2, 2Q, LRFU, and MIN with their respective offline best-parameter values. We found similar results for the DB2 and Sprite file system traces.[13]

Tables 2 and 3 compare ARC to LRU, MQ, 2Q, LRU-2, LRFU, and LIRS on the P8 and P12 traces, where the tunable parameters for MQ were set online[14] and the tunable parameters of other algorithms were chosen offline to be optimized for each cache size and workload. ARC outperforms LRU and performs nearly as well or competitively against 2Q, LRU-2, LRFU, LIRS, and MQ. In general, similar results hold for all the traces examined.[16]

Table 4 compares ARC with LRU for all traces with a practically relevant cache size. The SPC1-like trace contains long sequential scans inter-

Figure 2. ARC and LRU hit ratios (in percentages) versus cache size (in pages) in log-log scale for traces P6, SPC1-like, and Merge(S).
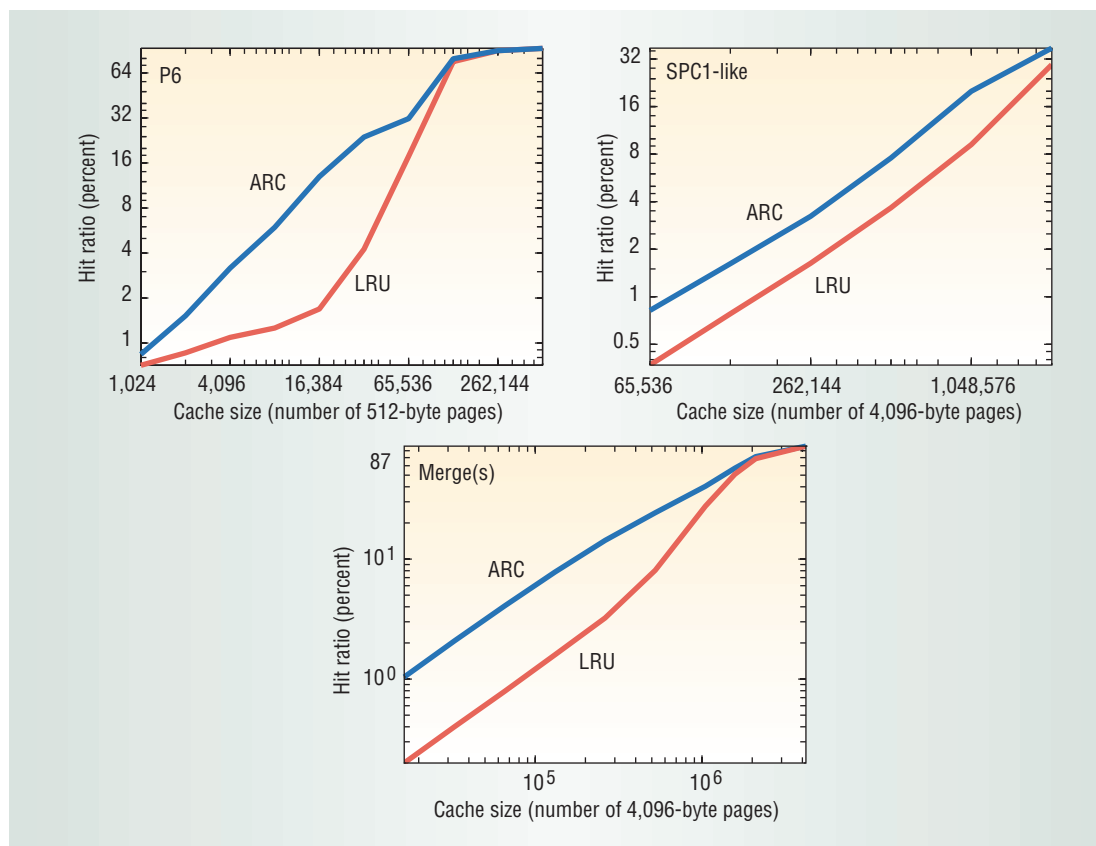
spersed with random requests. Due to scan resistance, ARC outperforms LRU, sometimes quite dramatically. ARC, working online, performs closely to and sometimes better than FRC$_p$ with the best offline fixed choice of the parameter p for all the traces.

When the adaptation parameter p approaches zero, ARC emphasizes the L_2's contents; when parameter p approaches the cache size, ARC emphasizes L_1's contents. Parameter p fluctuates and sometimes actually reaches these extremes. ARC can fluctuate from frequency to recency and back, all within a single workload.

Figure 2 compares the hit ratios for ARC against those for LRU for three traces: P6, SPC1-like, and Merge(S). ARC substantially outperforms LRU on virtually all traces and for all cache sizes.[16]

Our results show that the self-tuning, low-overhead, scan-resistant ARC cache-replacement policy outperforms LRU. Thus, using adaptation in a cache replacement policy can produce considerable performance improvements in modern caches. ◼

References

1. R.L. Mattson et al., "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.*, vol. 9, no. 2, 1970, pp. 78-117.
2. D.D. Sleator and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Comm. ACM*, vol. 28, no. 2, 1985, pp. 202-208.
3. L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems J.*, vol. 5, no. 2, 1966, pp. 78-101.
4. F.J. Corbato, "A Paging Experiment with the Multics System," *In Honor of P.M. Morse*, MIT Press, 1969, pp. 217-228.
5. P.J. Denning, "Working Sets Past and Present," *IEEE Trans. Software Eng.*, vol. 6, no. 1, 1980, pp. 64-84.
6. W.R. Carr and J.L. Hennessy, "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. 8th Symp. Operating System Principles*, ACM Press, 1981, pp. 87-95.
7. J.E.G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice Hall, 1973, p. 282.
8. A.V. Aho, P.J. Denning, and J.D. Ullman, "Principles of Optimal Page Replacement," *J. ACM*, vol. 18, no. 1, 1971, pp. 80-93.
9. E.J. O'Neil, P.E. O'Neil, and G. Weikum, "An Opti-

mality Proof of the LRU-K Page Replacement Algorithm," *J. ACM*, vol. 46, no. 1, 1999, pp. 92-112.

10. T. Johnson and D. Shasha, "2Q: A Low Overhead High-Performance Buffer Management Replacement Algorithm," *Proc. VLDB Conf.*, Morgan Kaufmann, 1994, pp. 297-306.

11. S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. ACM Sigmetrics Conf.*, ACM Press, 2002, pp. 31-42.

12. J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. ACM SIGMETRICS Conf.*, ACM Press, 1990, pp. 134-142.

13. D. Lee et al., "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Trans. Computers*, vol. 50, no. 12, 2001, pp. 1352-1360.

14. Y. Zhou and J.F. Philbin, "The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches," *Proc. Usenix Ann. Tech. Conf.* (Usenix 2001), Usenix, 2001, pp. 91-104.

15. W.W. Hsu, A.J. Smith, and H.C. Young, *The Automatic Improvement of Locality in Storage Systems*, tech. report, Computer Science Division, Univ. of California, Berkeley, 2001.

16. N. Megiddo and D.S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Usenix Conf. File and Storage Technologies* (FAST 2003), Usenix, 2003, pp. 115-130.

Nimrod Megiddo *is a research staff member at the IBM Almaden Research Center in San Jose, Calif. His research interests include optimization, algorithm design and analysis, game theory, and machine learning. Megiddo received a PhD in mathematics from the Hebrew University of Jerusalem. Contact him at megiddo@almaden.ibm.com.*

Dharmendra S. Modha *is a research staff member at the IBM Almaden Research Center in San Jose, Calif. His research interests include machine learning, information theory, and algorithms. Modha received a PhD in electrical and computer engineering from the University of California, San Diego. He is a senior member of the IEEE. Contact him at dmodha@almaden.ibm.com.*