Outperforming LRU with an Adaptive Replacement Cache Algorithm

Caching, a fundamental metaphor in modern computing, finds wide application in storage systems, 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. In a two-level memory hierarchy, a cache performs faster than auxiliary storage, but 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-
ststantial 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 pages lists: $L_1$ and $L_2$. $L_1$ maintains pages that have been seen only once, recently, while $L_2$ maintains pages that have been seen at least twice, recently. 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 $L_1$ captures recency, while $L_2$ captures frequency.

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 non-adaptive 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 adapt 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, WS (working set), and WSClock. If the request stream is drawn from the LRU stack depth distribution, LRU offers the optimal policy. Simple to implement, LRU responds well to deviations from the underlying SDD model. While SDD captures recency, it does not capture frequency.

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, 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, and it works well on several traces. However, LRU-2 suffers from two practical drawbacks: 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. 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
average-case rather than worst-case constant-time overhead, which is a significant practical drawback.

Frequency-based replacement\(^\text{12}\) 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.\(^\text{13}\) 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. LRUF replaces the page with the least \(C(x)\) value. As \(\lambda\) tends to 0, \(C(x)\) tends to the number of occurrences of \(x\) and LRFU collapses to LFU. As \(\lambda\) tends to 1, \(C(x)\) emphasizes recency and LRFU collapses to LRU. The performance depends crucially on \(\lambda\).\(^\text{13}\) ALRFU, an adaptive LRFU, adjusts \(\lambda\) dynamically.

LRU has two drawbacks. First, both LRFU and ALRFU require a tunable parameter for controlling correlated references.\(^\text{13}\) 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 \(\lambda\), LRFU can be 50 times slower than LRU and ARC. This can potentially wipe out the benefit of a high hit ratio.

The \textit{multiqueue} replacement policy\(^\text{14}\) uses \(m\) queues, where for \(0 \leq i \leq m - 1\), the \(i\)th 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 \textit{expireTime} is set to \textit{currentTime} + \textit{lifeTime}, where \textit{lifeTime} is a tunable parameter. On each access, the memory checks the \textit{expireTime} for the LRU page in each queue and, if it is less than \textit{currentTime}, moves the page to the next lower queue’s MRU position.

To estimate the parameter \textit{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(2\(c\)), that memorizes \(2c\) pages and manages an imaginary cache of size \(2c\), and also introduce a class \(\text{II}(c)\) of cache replacement policies.

DBL(2\(c\)) 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 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: \(L_1 = L_2 = \emptyset\). 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 \(\text{II}(c)\) of policies that track all the \(2c\) items that would be present in a cache of size \(2c\) managed by DBL(2\(c\)), but at most \(c\) are actually kept in cache. Thus, \(L_1\) is partitioned into
Once the cache directory has 2 resides only in the cache directory, not in the cache.

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. 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 \( p = \max(0, p - \max(|B_2|/|B_1|, 1)) \). REPLACE. 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. el se 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.

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

Subroutine REPLACE(p)
if \( (|T_1| \geq 1) \) and \((x \in B_2 \text{ and } |T_1| = p) \) 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 \( B_1 = B_2 = \emptyset \).
- 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 \( |T_1| > p \) or \(|T_1| = p \) and \( x \in B_2 \), replace the LRU page in \( T_1 \).
- If \( |T_1| < p \) 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 \( B_2 \) 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 contain the LRU \( c \) pages, LRU cannot experience cache hits unbeknownst to ARC, but ARC can and often
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 1. Comparison between ARC and other algorithms on an online transaction processing workload.

<table>
<thead>
<tr>
<th>Cache (512-byte pages)</th>
<th>Online hit ratios (%)</th>
<th>Offline hit ratios (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARC</td>
<td>LRU</td>
</tr>
<tr>
<td>1,000</td>
<td>38.93</td>
<td>32.83</td>
</tr>
<tr>
<td>2,000</td>
<td>46.08</td>
<td>42.47</td>
</tr>
<tr>
<td>5,000</td>
<td>55.25</td>
<td>53.65</td>
</tr>
<tr>
<td>10,000</td>
<td>61.87</td>
<td>60.70</td>
</tr>
<tr>
<td>15,000</td>
<td>65.40</td>
<td>64.63</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cache (512-byte pages)</th>
<th>Online hit ratios (%)</th>
<th>Offline hit ratios (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARC</td>
<td>LRU</td>
</tr>
<tr>
<td>1,024</td>
<td>1.22</td>
<td>0.35</td>
</tr>
<tr>
<td>2,048</td>
<td>2.43</td>
<td>0.45</td>
</tr>
<tr>
<td>4,096</td>
<td>5.28</td>
<td>0.73</td>
</tr>
<tr>
<td>8,192</td>
<td>9.19</td>
<td>2.30</td>
</tr>
<tr>
<td>16,384</td>
<td>16.48</td>
<td>7.37</td>
</tr>
<tr>
<td>32,768</td>
<td>27.51</td>
<td>17.18</td>
</tr>
<tr>
<td>65,536</td>
<td>43.42</td>
<td>36.10</td>
</tr>
<tr>
<td>131,072</td>
<td>66.35</td>
<td>62.10</td>
</tr>
<tr>
<td>262,144</td>
<td>89.28</td>
<td>89.26</td>
</tr>
<tr>
<td>524,288</td>
<td>97.30</td>
<td>96.77</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cache (512-byte pages)</th>
<th>Online hit ratios (%)</th>
<th>Offline hit ratios (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARC</td>
<td>LRU</td>
</tr>
<tr>
<td>1,024</td>
<td>4.16</td>
<td>4.09</td>
</tr>
<tr>
<td>2,048</td>
<td>4.89</td>
<td>4.84</td>
</tr>
<tr>
<td>4,096</td>
<td>5.76</td>
<td>5.61</td>
</tr>
<tr>
<td>8,192</td>
<td>7.14</td>
<td>6.22</td>
</tr>
<tr>
<td>16,384</td>
<td>10.12</td>
<td>7.09</td>
</tr>
<tr>
<td>32,768</td>
<td>15.94</td>
<td>8.93</td>
</tr>
<tr>
<td>65,536</td>
<td>26.09</td>
<td>14.43</td>
</tr>
<tr>
<td>131,072</td>
<td>38.68</td>
<td>29.21</td>
</tr>
<tr>
<td>262,144</td>
<td>53.47</td>
<td>49.11</td>
</tr>
<tr>
<td>524,288</td>
<td>63.56</td>
<td>60.91</td>
</tr>
</tbody>
</table>
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\textsuperscript{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,\textsuperscript{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.\textsuperscript{14}

The LFU, FBR, LRU-2, 2Q, LRFU, and MIN parameters exactly match those in the LRFU policy.\textsuperscript{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.\textsuperscript{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\textsuperscript{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.\textsuperscript{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-

<table>
<thead>
<tr>
<th>Workload</th>
<th>Cache (pages)</th>
<th>Cache (Mbytes)</th>
<th>LRU</th>
<th>ARC</th>
<th>FRC(_p) (Offline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>32,768</td>
<td>16</td>
<td>16.55</td>
<td>28.26</td>
<td>29.39</td>
</tr>
<tr>
<td>P2</td>
<td>32,768</td>
<td>16</td>
<td>18.47</td>
<td>27.38</td>
<td>27.61</td>
</tr>
<tr>
<td>P3</td>
<td>32,768</td>
<td>16</td>
<td>3.57</td>
<td>17.12</td>
<td>17.60</td>
</tr>
<tr>
<td>P4</td>
<td>32,768</td>
<td>16</td>
<td>5.24</td>
<td>11.24</td>
<td>9.11</td>
</tr>
<tr>
<td>P5</td>
<td>32,768</td>
<td>16</td>
<td>6.73</td>
<td>14.27</td>
<td>14.29</td>
</tr>
<tr>
<td>P6</td>
<td>32,768</td>
<td>16</td>
<td>4.24</td>
<td>23.84</td>
<td>22.62</td>
</tr>
<tr>
<td>P7</td>
<td>32,768</td>
<td>16</td>
<td>3.45</td>
<td>13.77</td>
<td>14.01</td>
</tr>
<tr>
<td>P8</td>
<td>32,768</td>
<td>16</td>
<td>17.18</td>
<td>27.51</td>
<td>28.92</td>
</tr>
<tr>
<td>P9</td>
<td>32,768</td>
<td>16</td>
<td>8.28</td>
<td>19.73</td>
<td>20.28</td>
</tr>
<tr>
<td>P10</td>
<td>32,768</td>
<td>16</td>
<td>2.48</td>
<td>9.46</td>
<td>9.63</td>
</tr>
<tr>
<td>P11</td>
<td>32,768</td>
<td>16</td>
<td>20.92</td>
<td>26.48</td>
<td>26.57</td>
</tr>
<tr>
<td>P12</td>
<td>32,768</td>
<td>16</td>
<td>8.93</td>
<td>15.94</td>
<td>15.97</td>
</tr>
<tr>
<td>P13</td>
<td>32,768</td>
<td>16</td>
<td>7.83</td>
<td>16.60</td>
<td>16.81</td>
</tr>
<tr>
<td>P14</td>
<td>32,768</td>
<td>16</td>
<td>15.73</td>
<td>20.52</td>
<td>20.55</td>
</tr>
<tr>
<td>ConCat</td>
<td>32,768</td>
<td>16</td>
<td>14.38</td>
<td>21.67</td>
<td>21.63</td>
</tr>
<tr>
<td>Merge(P)</td>
<td>262,144</td>
<td>128</td>
<td>38.05</td>
<td>39.91</td>
<td>39.40</td>
</tr>
<tr>
<td>DS1</td>
<td>2,097,152</td>
<td>1,024</td>
<td>11.65</td>
<td>22.52</td>
<td>18.72</td>
</tr>
<tr>
<td>SPC1-like</td>
<td>1,048,576</td>
<td>4,096</td>
<td>9.19</td>
<td>20.00</td>
<td>20.11</td>
</tr>
<tr>
<td>S1</td>
<td>524,288</td>
<td>2,048</td>
<td>23.71</td>
<td>33.43</td>
<td>34.00</td>
</tr>
<tr>
<td>S2</td>
<td>524,288</td>
<td>2,048</td>
<td>25.91</td>
<td>40.68</td>
<td>40.57</td>
</tr>
<tr>
<td>S3</td>
<td>524,288</td>
<td>2,048</td>
<td>25.26</td>
<td>40.44</td>
<td>40.29</td>
</tr>
<tr>
<td>Merge(S)</td>
<td>1,048,576</td>
<td>4,096</td>
<td>27.62</td>
<td>40.44</td>
<td>40.18</td>
</tr>
</tbody>
</table>
spersed with random requests. Due to scan resis-
tance, ARC outperforms LRU, sometimes quite
dramatically. ARC, working online, performs
closely to and sometimes better than FRC\textsubscript{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\textsubscript{2}'s contents; when
parameter \( p \) approaches the cache size, ARC
emphasizes L\textsubscript{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

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).

Our results show that the self-tuning, low-over-
head, scan-resistant ARC cache-replacement
policy outperforms LRU. Thus, using adap-
tation in a cache replacement policy can produce
considerable performance improvements in mod-
ern caches.

References
1. R.L. Mattson et al., “Evaluation Techniques for Stor-
age Hierarchies,” IBM Systems J., vol. 9, no. 2, 1970,
pp. 78-117.
4. F.J. Corbato, “A Paging Experiment with the Mult-
ics System,” In Honor of P.M. Morse, MIT Press,
Trans. Software Eng., vol. 6, no. 1, 1980, pp. 64-84.
and Effective Algorithm for Virtual Memory Man-
agement,” Proc. 8th Symp. Operating System Prin-
7. J.E.G. Coffman and P.J. Denning, Operating Systems
8. A.V. Aho, P.J. Denning, and J.D. Ullman, “Principles
1, 1971, pp. 80-93.

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.