IBM Research Report

A Simple Adaptive Cache Algorithm Outperforms LRU

Nimrod Megiddo, Dharmendra S. Modha
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
A Simple Adaptive Cache Algorithm Outperforms LRU

Nimrod Megiddo and Dharmendra S. Modha
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
Email: {megiddo,dmodha}@almaden.ibm.com

February 16, 2003

Abstract

Discarding the least recently used (LRU) item 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 pre-tune parameters. This article presents a new self-tuning, low overhead algorithm, namely, Adaptive Replacement Cache (ARC). It responds online to changing access patterns, continually balances between the recency and frequency features of the workload, and demonstrates that adaptation eliminates the need for workload specific pre-tuning. Like LRU, ARC can be easily implemented and its running time per request is essentially independent 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. ARC leads to substantial performance gains over LRU for a wide range of cache sizes. For example, on 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 benchmark, at 4GB cache, LRU delivers a hit ratio of 9.19% while ARC achieves a hit ratio of 20%.

1 Introduction

Caching is a fundamental metaphor in modern computing. It is widely used in storage systems [1], databases, web servers, middleware, processors, file systems, disk drives, RAID controllers, operating systems, and in varied and numerous other applications such as data compression and list updating [2]. Substantial progress in caching algorithms could affect the entire modern computational stack.
To benefit from caching, a storage system must include at least two kinds of memory: a cache and an auxiliary storage. Cache is faster than auxiliary storage, but is also more expensive. Because of cost, the cache size is usually a fraction of the auxiliary memory size.

Both memories handle uniformly sized items called pages. Requests for pages are first directed to the cache and, if the page is not in the cache, then to the auxiliary memory. In the latter case, a copy is “paged in” into the cache. This is called “demand paging,” and it rules out pre-fetching pages from the auxiliary memory into the cache. If the cache is full, then before a new page can be paged in, one of the pages currently in the cache must be “paged out.” A replacement policy determines which page is evicted. A commonly used criterion for evaluating a replacement policy is its hit ratio—the frequency at which a page is found in the cache. Of course, the implementation overhead should not exceed the anticipated time savings.

ARC is a policy with a high hit ratio and low overhead. Online adaptation is likely to have benefits for real-life workloads due to richness and variability with time. These may contain long sequential I/Os or moving hot spots, changing frequency and scale of temporal locality, and fluctuation between stable, repeating access patterns and ones with transient clustered references. A non-adaptive replacement policy would not work well for such access patterns.

The basic idea behind ARC is to maintain two LRU lists of pages, $L_1$ and $L_2$. $L_1$ maintains pages, which have been seen only once “recently”, while $L_2$ maintains pages, which have been seen at least twice “recently”. Pages on these lists are not necessarily present in the cache. The items that have been seen twice within a short time interval may be are thought of as having “high-frequency”. Hence, we think of $L_1$ as capturing “recency” while $L_2$ as capturing “frequency”. If the cache can hold $c$ pages, we endeavor to keep these two lists to roughly the same size, $c$. Together the two lists comprise a cache directory holding at most $2c$ pages. ARC keeps in cache a variable number of most recent pages from $L_1$ and from $L_2$. The precise number of pages from each list is adapted continually. To contrast the adaptive approach with a non-adaptive one, suppose $\text{FRC}_p$ is a fixed replacement policy, which 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 $\text{FRC}_p$ except that it may vary $p$ adaptively. Below we describe a learning rule, which allows ARC to adapt to a variable workload quickly and effectively.

Most algorithms use recency and frequency as predictors of the likelihood of pages of being used in the near future. ARC acts as a filter to detect and track temporal locality. If at some time recency (resp. frequency) becomes
important, then ARC will detect the change, and adapt accordingly.

We demonstrate that ARC works as well as the policy FRC\textsubscript{p}, even when the latter chooses the best (fixed) \( p \) with hindsight with respect to the particular workload! Surprisingly, ARC—which is completely online—delivers performance comparable to LRU-2, 2Q, LRFU, and LIRS—even when these policies choose with hindsight the best fixed values of their tuning parameters.

ARC can be implemented as two LRU lists. Thus, it is as easy to implement as LRU and its time overhead per request is independent of the cache size. Its space overhead is only marginally greater than that of 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 \textit{low overhead}. In contrast, the time overhead of LRU-2 and LRFU grows logarithmically with the cache size. As a result, in all our simulations, LRU-2 was twice slower than ARC and LRU, while LRFU can be 50 slower than ARC and LRU. ARC is “scan-resistant” in the sense that 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.

2 Prior Work

LRU always replaces the least recently used page. It has been known for a long time and various approximations and improvements thereto abound; two of the most important related algorithms are \textit{WS} (working set) \cite{buz} and \textit{WSClock} \cite{buz}. If the request stream is drawn from the so-called LRU Stack Depth Distribution (SDD), then LRU is the optimal policy \cite{buz}. LRU is simple to implement and responds well to deviations from the underlying SDD model. While SDD captures “recency”, it does not capture “frequency”.

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

LRU-2 \cite{buz} constitutes a significant practical progress. It approximates LFU, while eliminating the latter’s lack of adaptivity. It memorizes for each cache page the times of its two most recent occurrences, and replaces the page with the least second-most-recent occurrence. Under the IRM, LRU-2
has the maximum expected hit ratio of any online algorithm, which knows at
most two most recent references to each page [7], and it works well on several
traces [8]. It still has two practical drawbacks [8]: (i) It has logarithmic
complexity since it uses a priority queue. (ii) It has to tune the parameter
Correlated Information Period (CIP).

Logarithmic complexity is a severe practical drawback. 2Q is an improved
method with constant complexity [8]. It is similar to LRU-2 except that
it uses a simple LRU list instead of a priority queue. ARC's computational
overhead is similar to 2Q. Both are better than LRU-2, e.g., with a 16384 page
cache (512 bytes per page), the overhead in seconds for different policies is
as LRU: 13, ARC: 16, 2Q: 19, LRU-2: 30, MQ: 36, LRFU (λ = 10⁻⁷): 734,
LRFU (λ = 10⁻³): 418, and LRFU (λ = 0.99): 30.

The choice of CIP crucially affects LRU-2’s performance. No single a
priori fixed choice works uniformly well across various cache sizes and work-
loads. The second drawback of LRU-2 persists even in 2Q.

Low Inter-reference Recency Set (LIRS) [9] is in the spirit of 2Q. It main-
tains a variable size LRU stack (of potentially unbounded size) that serves
as a cache directory. From this stack, it selects a few top pages depend-
ing upon two parameters which crucially affect its performance. A certain
choice works well for stable IRM workloads and certain other for SDD ones.
LIRS's “stack pruning” operation implies only average rather than worst-
case constant-time overhead.

Frequency-based replacement (FBR) [10] maintains an LRU list, but par-
titions it into sections: new, middle and old, and moves pages between them.
It also maintains frequency counts for individual pages. The idea of factoring
out locality is 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 least reference count is replaced. The drawbacks of FBR are its need to
rescale the reference counts periodically and its tunable parameters.

Least Recently/Frequently Used (LRFU) subsumes LRU and LFU [11]. It
assigns a value C(x) = 0 to every page x and, depending on a parameter
λ > 0, after t time units, updates C(x) = 1 + 2⁻λC(x) if x is referenced and
C(x) = 2⁻λC(x) otherwise. This is similar to exponential smoothing, which
is a known 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 [11, Fig-
ure 7]. ALRFU, an adaptive LRFU, adjusts λ dynamically. LRFU’s two draw-
backs: (i) Both LRFU and ALRFU require a tunable parameter for controlling
correlated references [11, Figure 8]. (ii) The complexity of LRFU fluctuates
between constant and logarithmic. The required calculations make the practical complexity significantly higher than that of even LRU-2. For small $\lambda$, LR$\Phi$U can be 50 times slower than LRU and ARC. This can potentially wipe out the benefit of a high hit ratio.

Multi-queue replacement policy MQ [12] 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 algorithm also maintains a history buffer. On a 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. To estimate the parameter lifeTime, MQ makes an assumption that the distribution of temporal distances between consecutive accesses to a single page possesses a certain “hill” shape. ARC makes no such assumption, 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.

3 A Class of Replacement Policies

Let $c$ be the cache size in pages. We first introduce a policy DBL(2$c$), which memorizes 2$c$ pages and manages an imaginary cache of size 2$c$, and a class $\Pi(c)$ of cache replacement policies. DBL(2$c$) maintains two LRU lists: $L_1$, containing pages, which have been seen only once recently and $L_2$, containing pages, which 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 policy is: if $L_1$ contains exactly $c$ pages, then replace the LRU page in $L_1$; otherwise, replace the LRU page in $L_2$. Initially, $L_1 = L_2 = \emptyset$. If a requested page is in the cache, it is moved to the top of $L_2$; otherwise, it is placed at the top of $L_1$. In the latter case, if $|L_1| = c$, then the LRU member of $L_1$ is removed, and if $|L_1| < c$ and $|L_1| + |L_2| = 2c$, then the LRU member of $L_2$ is removed. Thus,
0 ≤ |L1| + |L2| ≤ 2c and 0 ≤ |L1| ≤ c throughout.

We propose a class \( \Pi(c) \) of policies, which track all the 2c items that would be present in a cache of size 2c managed by \( \text{DBL}(2c) \), but at most c are actually kept in cache. Thus, \( L_1 \) is partitioned into \( T_1 \) (containing the top or most recent pages in \( L_1 \)) and \( B_1 \) (containing the bottom or least recent pages in \( L_1 \)), and, similarly, \( L_2 \) is partitioned into top \( T_2 \) and bottom \( B_2 \), subject to the following conditions: (i) If \( |L_1| + |L_2| < c \), then \( B_1 = B_2 = \emptyset \). (ii) If \( |L_1| + |L_2| ≥ c \), then \( |T_1| + |T_2| = c \). (iii) For \( i = 1, 2 \), either \( T_i \) or \( B_i \) is empty, or the LRU page in \( T_i \) is more recent than the most recently used (MRU) page in \( B_i \). (iv) Throughout, \( T_1 \cup T_2 \) contains exactly those pages, which would be in cache under a policy in the class. The pages in \( T_1 \) and \( T_2 \) are in the cache directory and in the cache, but the pages in \( B_1 \) and \( B_2 \) are only in the cache directory and not in the cache. Once the cache directory has 2c pages, from then on, \( T_1 \cup T_2 \) and \( B_1 \cup B_2 \) will both contain exactly \( c \) pages. The policy 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 \( \Pi(c) \). Conversely, for \( 0 < c' < c \), the most recent \( c \) pages need not always be in \( \text{DBL}(2c') \).

4 Adaptive Replacement Cache

A fixed replacement cache \( \text{FRC}_p(c) \) (with a tunable parameter \( p \), \( 0 \leq p \leq c \)) in the class \( \Pi(c) \) attempts to keep in cache the \( p \) most recent pages from \( L_1 \) and the \( c - p \) most recent pages in \( L_2 \). Denote by \( x \) the requested page. (i) If either \( |T_1| > p \) or \((|T_1| = p \text{ and } x \in B_2)\), then replace the LRU page in \( T_1 \). (ii) If either \( |T_1| < p \) or \((|T_1| = p \text{ and } x \in B_1)\), then replace the LRU page in \( T_2 \). Roughly speaking, \( p \) is the current target size for the list \( T_1 \). ARC behaves like \( \text{FRC}_p \), except that \( p \) changes adaptively. The complete policy ARC is described in Figure 1.

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 \( T_2 \). These increases are effected by the continual updates of \( p \). The amount of change in \( p \) is very important. The “learning rates” depend on the relative sizes of \( B_1 \) and \( B_2 \). ARC attempts to keep \( T_1 \) and \( T_2 \) (resp. \( T_2 \) and \( B_1 \) ) to roughly the same size. On a hit in \( B_1 \), \( p \) is incremented by \( \max\{|B_2|/|B_1|, 1\} \) but not to exceed \( c \). Similarly, on a hit in \( B_2 \), \( p \) is decremented by \( \max\{|B_1|/|B_2|, 1\} \) but is never allowed to be less than 0. We demonstrate below that the compound effect of a number of changes in \( p \) is quite profound. ARC never stops adapting, so it always responds to workload changes from \( \text{IRM} \) to \( \text{SDD} \) and vice versa.
The LRU $c$ pages are always contained in $L_1 \cup L_2 = T_1 \cup T_2 \cup B_1 \cup B_2$, and, hence, LRU cannot experience cache hits unbeknownst to ARC, but ARC can (and often does) enjoy cache hits unbeknownst to LRU.

\[
\text{ARC}(c) \quad \text{INITIALIZE} \quad T_1 = B_1 = T_2 = B_2 = \emptyset, \quad p = 0. \quad x \text{- requested page.}
\]

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

Case II. $x \in B_1$ (a miss in \text{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 \text{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. $x \notin L_1 \cup L_2$ (a miss in DBL(2c) and \text{ARC}(c)):

\begin{itemize}
  \item \text{case (i)} \quad |L_1| = c:
    \begin{itemize}
      \item \text{If} \quad |T_1| < c \text{ then delete the LRU page of } B_1 \text{ and REPLACE}(p).
      \item \text{else} \quad \text{delete LRU page of } T_1 \text{ and remove it from the cache.}
    \end{itemize}
  \item \text{case (ii)} \quad |L_1| < c \text{ and } |L_1| + |L_2| \geq c:
    \begin{itemize}
      \item \text{if} \quad |L_1| + |L_2| = 2c \text{ then delete the LRU page of } B_2.
      \item \text{REPLACE}(p).
    \end{itemize}
\end{itemize}

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

\begin{flushleft}
\text{Subroutine REPLACE($p$)}
\end{flushleft}

\begin{itemize}
  \item \text{if} \quad (|T_1| \geq 1) \text{ and } ((x \in B_2 \text{ and } |T_1| = p) \text{ or } (|T_1| > p)) \text{ then move the LRU page of } T_1 \text{ to the top of } B_1 \text{ and remove it from the cache.}
  \item \text{else} \quad \text{move the LRU page in } T_2 \text{ to the top of } B_2 \text{ and remove it from the cache.}
\end{itemize}

Figure 1: Adaptive Replacement Cache

If a page is not in $L_1 \cup L_2$, it is put 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 $L_1$, so 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 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.


5 Experimental Results

We compared the performance of various algorithms on various traces: OLTP [8, 11] contains a one-hour’s worth of references to a CODASYL database. P1-P14 were each collected over several months from Windows NT workstations [13]. ConCat was obtained by concatenating the traces P1-P14, while Merge(P) was obtained by merging them. DS1 is a 7-day trace taken off a database server. The page size for all these traces was 512 Bytes. We captured a trace of the SPC1 (Storage Performance Council) synthetic benchmark, which contains long sequential scans in addition to random accesses. 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 over several 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.

Here, all hit ratios are cold start and reported in percentages.

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

Table 1: OLTP results. ARC outperforms LRU, LFU, FBR, LIRS, and MQ, and performs as well as LRU-2, 2Q, and LRFU with their respective offline best parameter values.

In Table 1, we compare ARC to several algorithms on OLTP. The tunable parameters for FBR and LIRS were set as in their original papers. The tunable parameters of LRU-2, 2Q, and LRFU were selected offline for the best result for each cache size. ARC requires no user-specified parameters, MQ was tuned online as in [12]. The LFU, FBR, LRU-2, 2Q, LRFU, and MIN are exactly the same as those in [11]. Similar results were found for the DB2 and the SPRITE file system traces in [11].

In Table 2 we compare ARC on the traces P8 and P12, to LRU, MQ, 2Q, LRU-2, LRFU, and LIRS, where the tunable parameters MQ were set online as in [12] and tunables of other algorithms were chosen offline to optimize for each cache size and each workload. ARC outperforms LRU and performs
close to 2Q, LRU-2, LRU, and LIRS. ARC is competitive with MQ. In general, similar results hold for all the traces we examined.

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

Table 2: A comparison of ARC with various cache algorithms on P8 and P12.

LRU is the single most widely used cache replacement policy. We now plot the hit ratio of ARC versus LRU in Figures 2 and 3. ARC substantially outperforms LRU on virtually all traces and for all cache sizes. In Table 3, we present an at-a-glance comparison of ARC with LRU for all the traces with a practically relevant cache size. The trace SPC1 contains long sequential scans interspersed with random requests. Due to scan-resistance, ARC outperforms LRU.

We now present the most surprising and intriguing of our results. In can be seen in Table 3 that ARC which tunes itself performs as well as the FRC_{p} with the best offline selection of the parameter p that is optimized for each cache size and each workload. This result holds for all the traces.

When the adaptation parameter p is close to zero, ARC can be deemed
emphasizing the contents of the $L_2$, and when it is close to the cache size, ARC emphasizes the contents of $L_1$. The parameter $p$ fluctuates and sometimes actually reaches these extremes, ARC can fluctuate from frequency to recency and then back, all within a single workload.

6 Conclusions

We have presented a new self-tuning, low overhead, scan-resistant cache replacement policy ARC that seems to outperform LRU. Our results show that there is considerable room for performance improvement in modern caches by using adaptation in cache replacement policy.

| Workload | c space | LRU | ARC | FRCp
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MB</td>
<td></td>
<td></td>
<td>OFFLINE</td>
</tr>
<tr>
<td>P1</td>
<td>32768</td>
<td>16</td>
<td>16.55</td>
<td>28.26</td>
</tr>
<tr>
<td>P2</td>
<td>32768</td>
<td>16</td>
<td>18.47</td>
<td>27.38</td>
</tr>
<tr>
<td>P3</td>
<td>32768</td>
<td>16</td>
<td>3.57</td>
<td>17.12</td>
</tr>
<tr>
<td>P4</td>
<td>32768</td>
<td>16</td>
<td>5.24</td>
<td>11.24</td>
</tr>
<tr>
<td>P5</td>
<td>32768</td>
<td>16</td>
<td>6.73</td>
<td>14.27</td>
</tr>
<tr>
<td>P6</td>
<td>32768</td>
<td>16</td>
<td>4.24</td>
<td>23.84</td>
</tr>
<tr>
<td>P7</td>
<td>32768</td>
<td>16</td>
<td>3.45</td>
<td>13.77</td>
</tr>
<tr>
<td>P8</td>
<td>32768</td>
<td>16</td>
<td>17.18</td>
<td>27.51</td>
</tr>
<tr>
<td>P9</td>
<td>32768</td>
<td>16</td>
<td>8.28</td>
<td>19.73</td>
</tr>
<tr>
<td>P10</td>
<td>32768</td>
<td>16</td>
<td>2.48</td>
<td>9.46</td>
</tr>
<tr>
<td>P11</td>
<td>32768</td>
<td>16</td>
<td>20.92</td>
<td>26.48</td>
</tr>
<tr>
<td>P12</td>
<td>32768</td>
<td>16</td>
<td>8.93</td>
<td>15.94</td>
</tr>
<tr>
<td>P13</td>
<td>32768</td>
<td>16</td>
<td>7.83</td>
<td>16.60</td>
</tr>
<tr>
<td>P14</td>
<td>32768</td>
<td>16</td>
<td>15.73</td>
<td>20.52</td>
</tr>
<tr>
<td>Concat</td>
<td>32768</td>
<td>16</td>
<td>14.38</td>
<td>21.67</td>
</tr>
<tr>
<td>Merge(P)</td>
<td>262144</td>
<td>128</td>
<td>38.05</td>
<td>39.91</td>
</tr>
<tr>
<td>DS1</td>
<td>2097152</td>
<td>1024</td>
<td>11.65</td>
<td>22.52</td>
</tr>
<tr>
<td>SPC1</td>
<td>1048576</td>
<td>4096</td>
<td>9.19</td>
<td>20.00</td>
</tr>
<tr>
<td>S1</td>
<td>524288</td>
<td>2048</td>
<td>23.71</td>
<td>33.43</td>
</tr>
<tr>
<td>S2</td>
<td>524288</td>
<td>2048</td>
<td>25.91</td>
<td>40.68</td>
</tr>
<tr>
<td>S3</td>
<td>524288</td>
<td>2048</td>
<td>25.26</td>
<td>40.44</td>
</tr>
<tr>
<td>Merge(S)</td>
<td>1048576</td>
<td>4096</td>
<td>27.62</td>
<td>40.44</td>
</tr>
</tbody>
</table>

Table 3: At-a-glance comparison of LRU and ARC for various workloads. It can be seen that ARC outperforms LRU—sometimes quite dramatically. Also, ARC which is online performs very close to (and sometimes even better than) FRCp with the best offline fixed choice of the parameter $p$. 
Figure 2: Hit ratios achieved by ARC and LRU in log-log scale.
Figure 3: Hit ratios achieved by ARC and LRU in log-log scale
References


