RJ 10284 (A0302-012) February 18, 2003
Computer Science

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

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

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 $\texttt{FRC}_p$ 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 $\texttt{FRC}_p$ 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 $\mathtt{FRC}_p$, 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 *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 WS (working set) [3] and WSClock [4]. If the request stream is drawn from the so-called LRU Stack Depth Distribution (SDD), then LRU is the optimal policy [5]. 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 [5, 6] 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 [7] 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`($\lambda = 10^{-7}$): 734, `LRFU`($\lambda = 10^{-3}$): 418, and `LRFU`($\lambda = 0.99$): 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 workloads. The second drawback of `LRU-2` persists even in `2Q`.

Low Inter-reference Recency Set (`LIRS`) [9] is in the spirit of `2Q`. It maintains a variable size `LRU` stack (of potentially unbounded size) that serves as a cache directory. From this stack, it selects a few top pages depending 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 partitions 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 $\lambda > 0$, after t time units, updates $C(x) = 1 + 2^{-\lambda}C(x)$ if x is referenced and $C(x) = 2^{-\lambda}C(x)$ 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, Figure 7]. `ALRFU`, an adaptive `LRFU`, adjusts λ dynamically. `LRFU`'s two drawbacks: (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 λ, LRFU 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($2c$), which memorizes $2c$ pages and manages an imaginary cache of size $2c$, and a class $\Pi(c)$ of cache replacement policies. DBL($2c$) 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,

$0 \leq |L_1| + |L_2| \leq 2c$ and $0 \leq |L_1| \leq c$ 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 $\texttt{DBL}(2c)$, 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 $|L_1| + |L_2| \geq c$, 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 recenty 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 contains 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 $\texttt{LRU}(c)$ is in the class $\Pi(c)$. Conversely, for $0 < c' < c$, the most recent c pages need not always be in $\texttt{DBL}(2c')$.

4 Adaptive Replacement Cache

A *fixed replacement cache* $\texttt{FRC}_p(c)$ (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 ($|T_1| = p$ and $x \in B_2$), then replace the LRU page in T_1. (ii) If either $|T_1| < p$ or ($|T_1| = p$ 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 $\texttt{FRC}_p$, 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 B_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 IRM to 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.

ARC(c) INITIALIZE $T_1 = B_1 = T_2 = B_2 = \emptyset$, $p = 0$. 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. $x \notin L_1 \cup L_2$ (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.

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 serveral 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.

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 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, LRFU, and LIRS. ARC is competitive with MQ. In general, similar results hold for all the traces we examined.

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 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 MB	LRU	ARC	FRC_p 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 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) FRC_p with the best offline fixed choice of the parameter p.

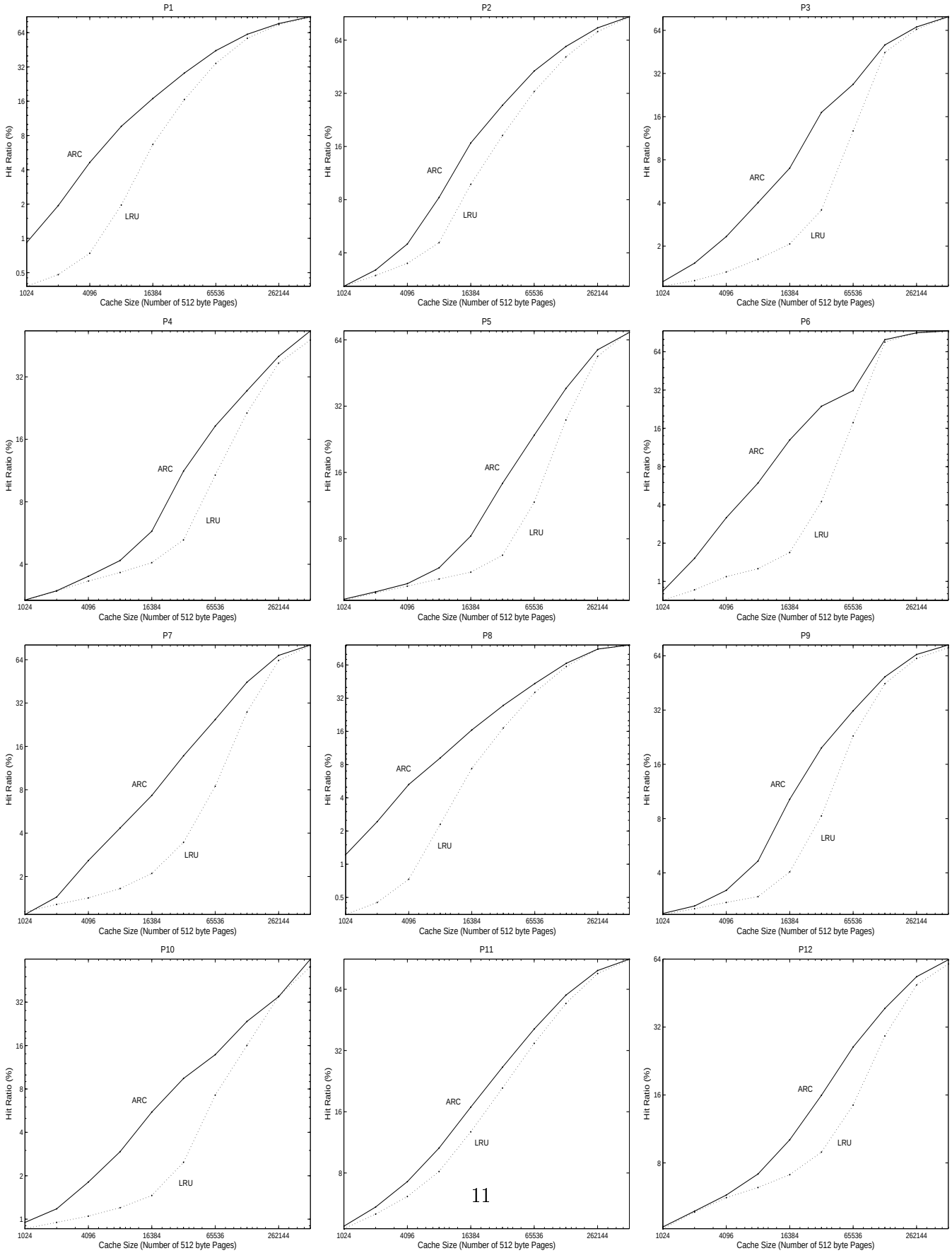

Figure 2: Hit ratios achieved by ARC and LRU in log-log scale.

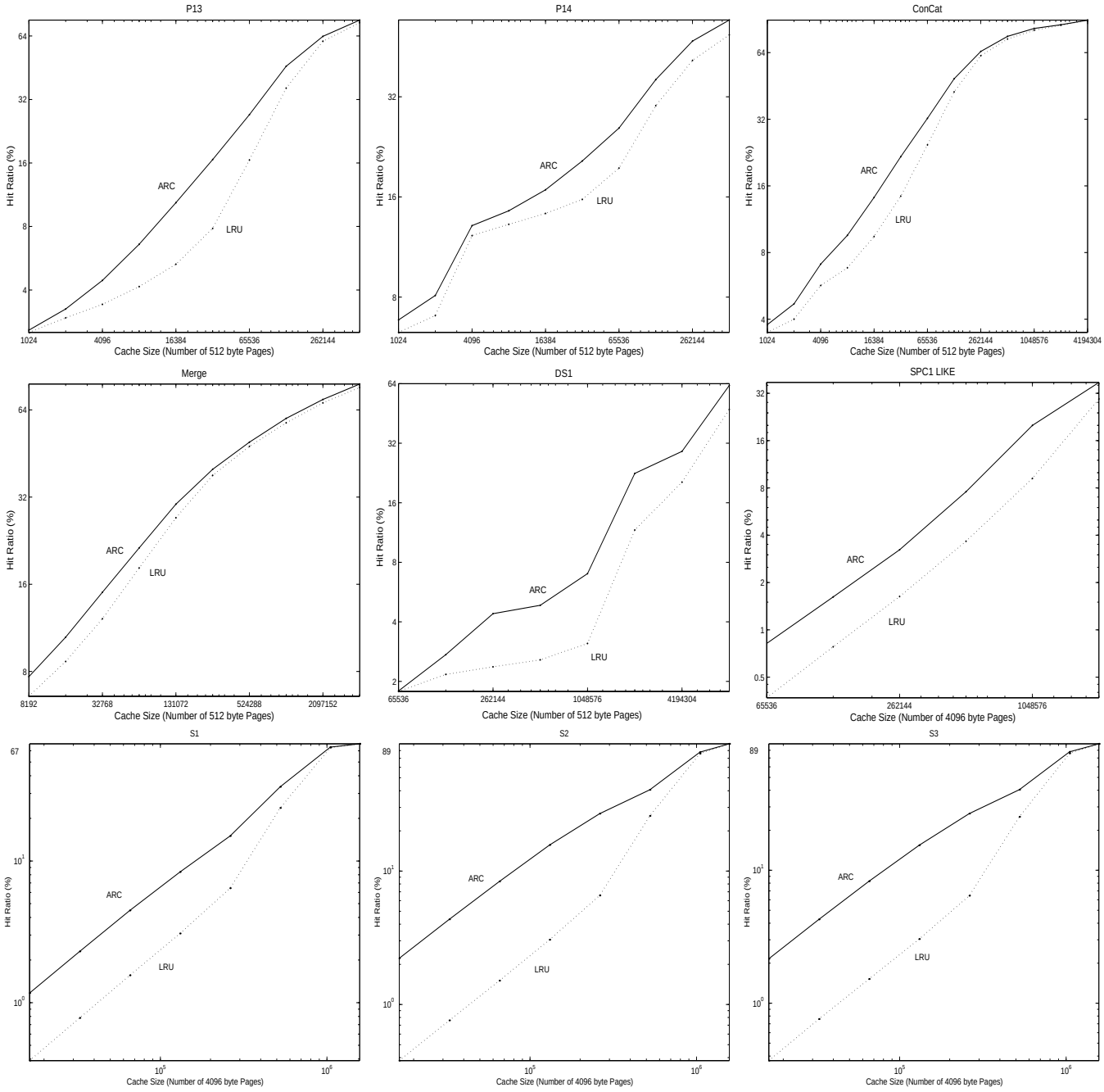

Figure 3: Hit ratios achieved by ARC and LRU in log-log scale

References

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.