DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics

Cong Li
Intel Corporation
cong.li@intel.com

ABSTRACT

As one of the state-of-the-art policies for buffer cache replacement, Low Inter-Reference Recency Set (LIRS) uses Inter-Reference Recency (IRR) to predict future access behaviors of blocks. With a static allocation of most cache space to low IRR blocks, it does not perform well in some LRU-friendly workloads. Inspired by the idea of dynamic cache space partitioning from another state-of-the-art policy, Adaptive Replacement Cache (ARC), we propose a new Dynamic LIRS (DLIRS) policy. The new policy uses a simple mechanism to perform an approximated online estimation on how well IRR predicts future access behaviors, and then dynamically adapts the space allocation of low IRR blocks against high IRR blocks. Experiments are performed on traces from the UMass Trace Repository as well as a synthetic trace drawn from a stack depth distribution. While sometimes LIRS outperforms ARC with a significant margin and sometimes vice versa, the new DLIRS policy consistently performs close to the winner between ARC and LIRS in all the cases.

CCS CONCEPTS

• **Information systems** → *Storage management*; • **Theory of computation** → *Caching and paging algorithms*;

KEYWORDS

Buffer cache replacement, LIRS, ARC, dynamic cache space allocation

1 INTRODUCTION

Caching mechanisms are widely used in a variety of systems including, e.g., disk drives [11], file systems [9], middleware [4], databases [12], web servers [2], etc. Cache block replacement is one of the most fundamental problems in computing. Many algorithms, e.g., Least-Recently Used (LRU) [3], Least-Frequently Used (LFU) [3], 2Q [7], LRU-K [10], etc., have been proposed to determine the victim to be replaced given a new block to be loaded into the cache.

Low Inter-Reference Recency Set (LIRS) [6] is one of the state-of-the-art policies for cache replacement. It uses Inter-Reference Recency (IRR), defined as the number of unique blocks accessed between two consecutive accesses of a block, to predict the likelihood of its future access. LIRS efficiently discriminates high IRR (HIR) blocks from low IRR (LIR) blocks and statically devotes most of the cache space to LIR blocks. When the recent IRR is not a good access pattern predictor, e.g., in some workloads drawn from a stack depth distribution (SDD), LIRS would not perform well enough [8].

Adaptive Replacement Cache (ARC) [8] is another state-of-the-art policy. It maintains two lists, the recency list for blocks with one recent access and the frequency list for blocks with two or more recent accesses. The policy monitors accesses in the two lists and dynamically partitions the cache space within the two lists to balance access recency versus access frequency. ARC is able to adapt to SDD workloads by allocating most cache space to the recency list, but its frequency list is less capable in capturing repeated accesses with relatively long temporal distances than the fine-grain metric of IRR in LIRS.

In both LIRS and ARC, the algorithm manipulates the corresponding data structure whenever there is a cache hit. The manipulations need to be serialized, introducing a lock contention problem. Clock-style approximations of LIRS and ARC are proposed to resolve the lock contention problem [1, 5]. Overhead of LIRS can also be reduced using an asynchronous concurrency model [14].

Inspired by the idea of ARC, we propose a novel *Dynamic LIRS* (DLIRS) policy. The new policy monitors accesses to non-resident HIR blocks and accesses to HIR blocks demoted from LIR status for an approximated online estimation on the predictive capability of IRR. The information is then used to guide the dynamic allocation of the cache space to LIR blocks against HIR blocks. We perform empirical evaluation of the policies on traces from the UMass Trace Repository along

SYSTOR '18, June 4–7, 2018, HAIFA, Israel
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5849-1/18/06…$15.00
https://doi.org/10.1145/3211890.3211891

with a synthetic SDD trace with different cache space configurations. Experimental results show that while sometimes LIRS outperforms ARC with a significant margin and sometimes vice versa, the new DLIRS policy consistently performs close to the winner between ARC and LIRS in all the cases. Studies on two of the cases are carried out to demonstrate how the new policy overcomes the weaknesses of LIRS and ARC.

2 BACKGROUND

2.1 LIRS Policy

The policy of Low Inter-Reference Recency Set (LIRS) uses Inter-Reference Recency (IRR) as the fine-grained metric to predict future accesses to a block. IRR of a block is defined as the number of unique blocks accessed between two consecutive accesses of the block. The policy assumes that with a lower value of its most recent IRR, a block is more likely to be accessed in the future.

In the recommended setting in [6], LIRS devotes 99% of its cache space to low IRR (LIR) blocks and the remaining 1% to resident high IRR (HIR) blocks. It also uses shadow cache entries [13] to track the meta-data of other blocks accessed after the access of the least-recently used (LRU) LIR block cached. We refer to those blocks as non-resident HIR blocks. Figure 1(a) shows an example of the data structure used in LIRS. The LIRS stack S is used to maintain the list of blocks accessed after the access of the LRU LIR block, operating like an LRU stack. The HIR block list Q tracks all the resident HIR blocks. When a new block is first accessed, it goes to the most-recently used (MRU) positions of both S and Q as an HIR block. In Figure 1(a), when an HIR block A (no matter whether it is resident or not) in S gets accessed, its current IRR, d_1, is lower than the future IRR of the LRU LIR block B which will be greater than d_2. As a result, block A's status is promoted from HIR to LIR and it goes to the MRU position in S. Block B's status is demoted from LIR to HIR and it goes from S to Q. In this example, since block A's data needs to be fetched (due to its previous non-resident status), the data of the last resident HIR block C in Q is evicted and block C becomes a non-resident HIR. Since there is a constraint that the status of the LRU block in S must be LIR, block D is pruned. If a lot of new blocks arrive, S will be extended to an unacceptable length. Practically the stack is given a size limit. When the limit is exceeded, the last HIR blocks close to S's LRU position are removed.

It is straightforward to notice that there are a small number of resident HIR blocks close to the MRU position of S. After that S is interleaved with LIR blocks and non-resident HIR blocks. If those non-resident HIR blocks relatively close to the MRU position are accessed, it will result in cache misses which can be captured by a basic LRU policy with the same cache size. An example workload is a request stream drawn from a stack depth distribution (SDD) [3]. The distribution models the workload that blocks accessed are kept in a stack. A block with a smaller stack depth is more likely to be accessed than blocks with larger depths. This results in the pattern that new blocks are accessed, stay in small depths in the stack, and get accessed again shortly. In such cases, IRR is less capable in predicting the future access of a block, especially when a new block is first encountered with a lack of the IRR value.

2.2 ARC Policy

With a cache size of c, the policy of Adaptive Replacement Cache (ARC) maintains two lists of length c, the recency list for blocks accessed only once recently and the frequency list for blocks accessed twice or more than twice recently. Figure 1(b) shows the data structure used in ARC. If an entry in the recency list is accessed, it gets moved to the MRU position of the frequency list. In each of the lists, its MRU portion (T_1 or T_2) is devoted to real cache entries and its LRU portion (B_1 or B_2) is devoted to shadow cache entries. The sizes of T_1 and T_2 are constrained to $|T_1| + |T_2| = c$ and their *target* sizes are dynamically adjusted to balance recency with frequency. When a shadow entry in B_1 gets accessed, it indicates that if T_1 is grown by one entry, the access will likely become a hit with probability $\frac{1}{|B_1|}$. Similarly, when a shadow entry in B_2 gets accessed, it indicates that if T_2 is grown by one entry, the access will likely become a hit with probability $\frac{1}{|B_2|}$. As a result, when an entry in B_1 gets accessed, we compare $\frac{1}{|B_1|}$ with $\frac{1}{|B_2|}$ and grow the target size of T_1 by $\min\{1, \frac{|B_2|}{|B_1|}\}$. When an entry in B_2 gets accessed, we compare $\frac{1}{|B_2|}$ with $\frac{1}{|B_1|}$ and grow the target size of T_2 by $\min\{1, \frac{|B_1|}{|B_2|}\}$.

ARC learns the dynamic balance between recency and frequency. However, its frequency list contains less granular information to capture repeated accesses with relatively long temporal distances without a fine-grained metric like IRR.

2.3 A Glance at Comparing LIRS with ARC

We evaluate the hit ratios of LIRS and ARC using traces from the UMass Trace Repository along with a synthetic SDD trace. Different cache sizes are tried. The maximum number of shadow cache entries used by LIRS is configured to be the same as that in ARC. While the complete description is located in Section 4, Table 1 gives a quick glance at the cases where one algorithm outperforms the other with a relative hit ratio improvement of at least 10%. As we see from the results, neither LIRS nor ARC is the consistent winner. Each of them enjoys its strength in some cases and suffers from its weakness in some other cases.

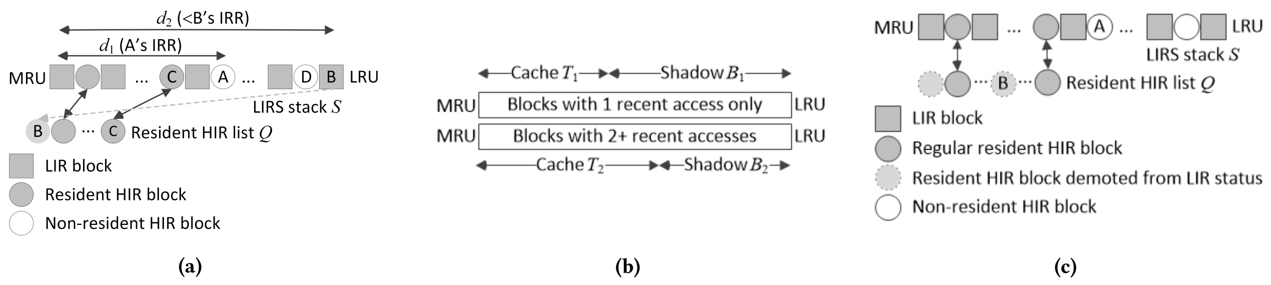


Figure 1: Data structure of (a) LIRS (the next access on block A); (b) ARC; and (c) DLIRS (the next access on either block A or block B leading to dynamics in cache space partitioning).

Table 1: LIRS vs. ARC on selected configurations.

Trace (cache size)	LIRS	ARC
Websearch1 (131072)	**13.08%**	8.37%
Websearch1 (262144)	**24.83%**	15.03%
Websearch1 (524288)	**41.01%**	33.02%
Websearch2 (524288)	**46.57%**	40.58%
Websearch3 (524288)	**46.55%**	40.36%
Financial1 (512)	15.83%	**23.34%**
Financial1 (1024)	19.36%	**26.06%**
Financial1 (2048)	25.45%	**29.39%**
SDD (256)	17.21%	**20.52%**
SDD (512)	31.57%	**36.91%**

3 IMPROVING LIRS WITH DYNAMICS

As the performance of LIRS under LRU-friendly workloads is impacted by the non-resident HIR blocks relatively close to the MRU position of the LIRS stack, it is natural to allocate more cache space to HIR blocks so that those non-resident blocks become resident. However, this is at the cost of reducing the number of LIR block entries, limiting the length of the LIRS stack, and therefore reducing the number of non-resident HIR blocks tracked, which at last deteriorates the capability in capturing repeated accesses with long temporal distances in other workloads. For example, when allocating 30% of the cache space to HIR blocks, the performance of LIRS on the configuration of 'Financial1 (1024)' would improve from 19.36% to 25.89%, but that on the configuration of 'WebSearch1 (131072)' would degrade from 13.08% to 10.68%. Inspired by the idea of ARC, we propose Dynamic LIRS (DLIRS), a new policy which dynamically allocates cache space to LIR blocks against HIR blocks in order to adapt to different types of workloads.

Figure 1(c) shows the data structure used in DLIRS. For each resident HIR block, we use an additional bit to track whether the block is demoted from LIR status. In the figure, if a non-resident HIR block A is accessed, it indicates that its IRR is less capable in predicting the access. Extending the cache space for resident HIR blocks may cover the cache miss. If a demoted block B is hit, it indicates a previous inappropriate demotion due to a smaller size allocated to LIR blocks. Extending the cache space for LIR blocks may keep the block in the LIRS stack.

Figure 2 shows the algorithm of DLIRS. Let $\tilde{L}$ denote the target number of LIR blocks and $\tilde{H_r}$ the target number of resident HIR blocks. The initialization of $\tilde{L}$ and $\tilde{H_r}$ follows the default setting of LIRS [6]. Let L and H_r denote the current number of LIR blocks and that of resident HIR blocks respectively. We introduce two new variables, H_n to track the current number of non-resident HIR blocks and H_d to track the current number of resident HIR blocks demoted from LIR status. Given the cache size of c and the same number of shadow entries, we have $\tilde{L} + \tilde{H_r} = c$, $L + H_r \leq c$, and $L + H_r + H_n \leq 2c$. If a non-resident HIR block is accessed, increasing the number of resident HIR blocks by one is likely to make the access a hit with probability $\frac{1}{H_n}$. If a demoted resident HIR block is hit, increasing the number of LIR blocks by one is likely to keep the block in its LIR status with probability $\frac{1}{H_d}$. In either case we compare the probability with its adversary and determine the actual size to grow, which is similar to ARC. Whenever possible, e.g., during LIR block demotion or resident HIR block data ejection, we bring the current number of LIR blocks and that of resident HIR blocks back to their target numbers.

Overhead: Comparing with the standard LIRS policy, the new DLIRS policy introduces two new variables and attaches a bit to each of the resident HIR blocks. The space overhead is minimum. The algorithm of DLIRS closely follows the workflow of LIRS. However, the new policy dynamically adjusts the target number of LIR blocks and that of resident HIR blocks. Additional operations are performed in order to bring the current number of the LIR blocks and the current number of the resident HIR blocks back to their target numbers (that is, the subroutine in Figure 2). During dynamic

```
Data structure
  $S$: LIRS stack
  $Q$: resident HIR block list
Variables
  $\tilde{L}$: target number of LIR blocks
  $\tilde{H_r}$: target number of resident HIR blocks
  $L$: current number of LIR blocks
  $H_r$: current number of resident HIR blocks
  $H_n$: current number of non-resident HIR blocks
  $H_d$: current number of resident HIR blocks demoted from LIR status
Input a request block $x$
  Case 1: $x$ is an LIR block
    Move $x$ to the MRU position of $S$
    If $x$ is the previous LRU block of $S$, prune LRU HIR blocks in $S$, and
      update $H_n$
  Case 2: $x$ is a non-resident HIR block in $S$
    $\tilde{H_r} \leftarrow \max\{c-1, \tilde{H_r} + \min\{1, \frac{H_d}{H_n}\}\}$, $\tilde{L} \leftarrow c - \tilde{H_r}$
    $H_n \leftarrow H_n - 1$, $L \leftarrow L + 1$
    Call BringCurrentBackToTarget
    Promote $x$'s status to LIR, fetch the data, and move it to the MRU
      position of $S$
  Case 3: $x$ is a resident HIR block in both $S$ and $Q$
    $H_r \leftarrow H_r - 1$, $L \leftarrow L + 1$
    Call BringCurrentBackToTarget
    Promote $x$'s status to LIR, remove it from $Q$, and move it to the
      MRU position of $S$
  Case 4: $x$ is a resident HIR block in $Q$, but not in $S$
    If $x$ is a demoted block, $\tilde{L} \leftarrow \max\{c-1, \tilde{L} + \min\{1, \frac{H_n}{H_d}\}\}$, $\tilde{H_r} \leftarrow$
      $c - \tilde{L}$, $H_d \leftarrow H_d - 1$, and clear $x$'s demotion bit
    Move $x$ to the MRU position of $S$ and $Q$
  Case 5: $x$ is a new block
    If $H_r = 0$ & $L < \tilde{L}$, $L \leftarrow L + 1$ and set $x$ to LIR status; otherwise
      set $x$ to HIR status, $H_r \leftarrow H_r + 1$, put $x$ in the MRU position of
      $Q$, and call BringCurrentBackToTarget
    Fetch the data and put $x$ in the MRU position of $S$
    If $L + H_r + H_n > 2c$, remove $L + H_r + H_n - 2c$ non-resident HIR
      blocks close to the LRU position of $S$ and update $H_n$
Subroutine BringCurrentBackToTarget
  If $L > \tilde{L}$, demote $L - \tilde{L}$ LRU LIR blocks to HIR status, move them
    from $S$ to the MRU position of $Q$, prune LRU HIR blocks in $S$, and
    update $L$, $H_r$, $H_n$, and $H_d$
  If $H_r > \tilde{H_r}$, eject the data of $H_r - \tilde{H_r}$ LRU resident HIR block in $Q$,
    and update $H_r$ as well as $H_d$ if applicable
```

Figure 2: Algorithm of DLIRS.

adjustment, the change of $\tilde{L}$ or $\tilde{H_r}$ in number of blocks is minor. The operations in memory are fast. Therefore the overall overhead in execution time is also minimum. Similar to LIRS, DLIRS does not address the lock contention problem.

4 EXPERIMENTS

To evaluate the performance of the policies in terms of cache hit ratio, we use the storage I/O traces from the UMass Trace Repository[1] which are composed of 3 search engine traces and 2 financial online transaction processing traces.

[1]http://traces.cs.umass.edu/index.php/Storage/Storage.

Table 2: Traces and configurations.

Trace	Requests	Unique blocks	Cache size	
			Min.	Max.
WebSearch1	3996451	1310273	2048	524288
WebSearch2	17253075	1693344	2048	524288
WebSearch3	16407703	1689882	2048	524288
Financial1	5561703	827801	512	131072
Financial2	13882742	827801	512	131072
SDD	1048576	7578	256	2048

Table 3: Hit ratios of LIRS, ARC, and DLIRS on selected configurations.

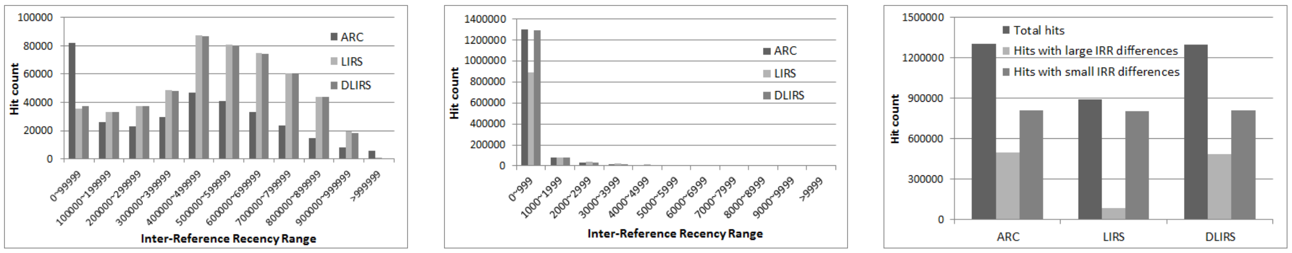
Trace (cache size)	LIRS	ARC	DLIRS
WebSearch1 (131072)	**13.08%**	8.37%	**13.02%**
WebSearch1 (262144)	**24.83%**	15.03%	**24.84%**
WebSearch1 (524288)	**41.01%**	33.02%	**41.01%**
WebSearch2 (524288)	**46.57%**	40.58%	**46.57%**
WebSearch3 (524288)	**46.55%**	40.36%	**46.55%**
Financial1 (512)	15.83%	**23.34%**	**23.20%**
Financial1 (1024)	19.36%	**26.06%**	**25.96%**
Financial1 (2048)	25.45%	**29.39%**	**29.23%**
SDD (256)	17.21%	**20.52%**	**20.37%**
SDD (512)	31.57%	**36.91%**	**36.35%**

To further evaluate how LIRS performs in LRU-friendly workloads, we create a synthetic SDD trace as follows. A stack is maintained to keep the blocks accessed recently. Each time a random stack depth is generated according to the cumulative distribution function of $P(N \leq n) = 1 - e^{-\lambda n}$ where λ is set to $\frac{\ln 2}{768}$.[2] If the depth falls out of the current depth of the stack, a new random block is requested and brought into the stack. Otherwise the block in the corresponding depth of the stack is requested.

Table 2 shows the characteristics of the traces as well as the different cache sizes used for evaluation.[3] We fix the number of shadow cache entries to be the same as the number of cache blocks.[4]

Table 3 highlights the performance of the policies on a selected set of configurations in which a significant performance difference is observed. Given a large cache size on the 3 search traces, LIRS outperforms ARC with a relative margin of at least 10%. Given a small cache size on the first financial trace and the synthetic SDD trace, ARC outperforms LIRS with a relative margin of at least 10%. In all the

[2]It makes a 50% hit ratio with an LRU policy of a cache size of 768.
[3]The maximum cache size for the 3 web search traces follows the setting in [8]. Note that in [8] LIRS has not been evaluated on the 3 traces.
[4]Refer to https://www.dropbox.com/s/5y1czqzioqx7wxs/Cache.zip (ZIP password: **DLIRS2018!**) for the code and the datasets.



(a) (b) (c)

Figure 3: Hit count histogram (a) with respect to IRR ranges on 'WebSearch1 (131072)'; (b) with respect to IRR ranges on 'Financial1 (1024)'; and (c) with respect to whether the previous IRR range of a hit is quite different from its current IRR range on 'Financial1 (1024)'.

10 cases highlighted, DLIRS performs close to the winner. DLIRS significantly outperforms ARC in the 5 configurations of the search traces. Meanwhile, it significantly improves LIRS in the 3 configurations of the first financial trace and the 2 configurations of the synthetic SDD trace.

In all the other cases not listed in the table, the performance of DLIRS is also consistent, following closely to that of the winner between ARC and LIRS in the individual case. The complete experimental results are plotted in Appendix, in which the performance of LRU is also given as the most basic baseline.

To empirically evaluate the execution time overhead, we examine the change in $\tilde{L}$ or $\tilde{H}_r$ during dynamic allocation of the cache space to LIR blocks against HIR blocks. As analyzed in Section 3, the change in number of blocks is the most significant contributor to the overhead in execution time. Throughout all the configurations, the average change in number of blocks ranges from 1.42 to 2.00, providing the empirical justification that the execution time overhead is minimum.

Case Study 1: For the configuration of 'WebSearch1 (131072)', we analyze each of the cache hits by the policies and extract the range of the IRR value of that hit. It is done by concatenating 10 LRU stacks and examining which stack the hit is located in. Figure 3(a) shows the hit count histogram with respect to different IRR ranges. As we see from the figure, LIRS generates more hits in high IRR ranges than ARC does. This indicates that IRR is the fine-grained metric capable of capturing repeated accesses with relatively long temporal distances. ARC is less capable in capturing such access patterns. DLIRS performs close to LIRS in terms of large hit counts in high IRR ranges. Over the run, the average space allocated to LIR blocks in DLIRS stablizes at 98.4% (versus the static setting of 99% in LIRS), demonstrating that it maintains the strength of LIRS in capturing those access patterns.

Case Study 2: Figure 3(b) displays the hit count histogram with respect to different IRR ranges for the configuration of 'Financial1 (1024)'. Most hits are concentrated on the lowest IRR range in which LIRS generates much less hits than ARC does. Figure 3(c) gives the insight by examining the current IRR range versus the previous IRR range for each of the hits. We categorize the hits into two categories. The first one is that the two ranges are close. The second category is that either the two ranges have a range difference greater than 2 or the previous IRR value even does not exists (that is, the hit is the first hit on the second access of the block). LIRS generates much less hits when either the previous IRR value is not a good predictor of the current IRR value or the previous IRR value does not exist. In this case, DLIRS performs close to ARC. Over the run, the average space allocated to LIR blocks in DLIRS drops to 66.2% (from the static setting of 99% in LIRS). It demonstrates that given a low predictive capability of IRR, DLIRS adapts to an LRU-style cache replacement with more cache space devoted to resident HIR blocks.

5 CONCLUSION

In this paper we have proposed a novel improvement to the LIRS cache replacement policy. The new DLIRS policy borrows the idea from ARC and dynamically allocates the cache space to LIR blocks against HIR blocks. Experimental results indicate that DLIRS overcomes the weaknesses of ARC and LIRS, and performs consistently close to the winner between the two state-of-the-art policies in different configurations.

ACKNOWLEDGEMENTS

We thank Jia Bao and Honesty C. Young for many useful discussions regarding the algorithms in this paper. We acknowledge the anonymous reviewers and our shepherd, Fred Douglis, for their valuable comments and criticisms to improve the paper.

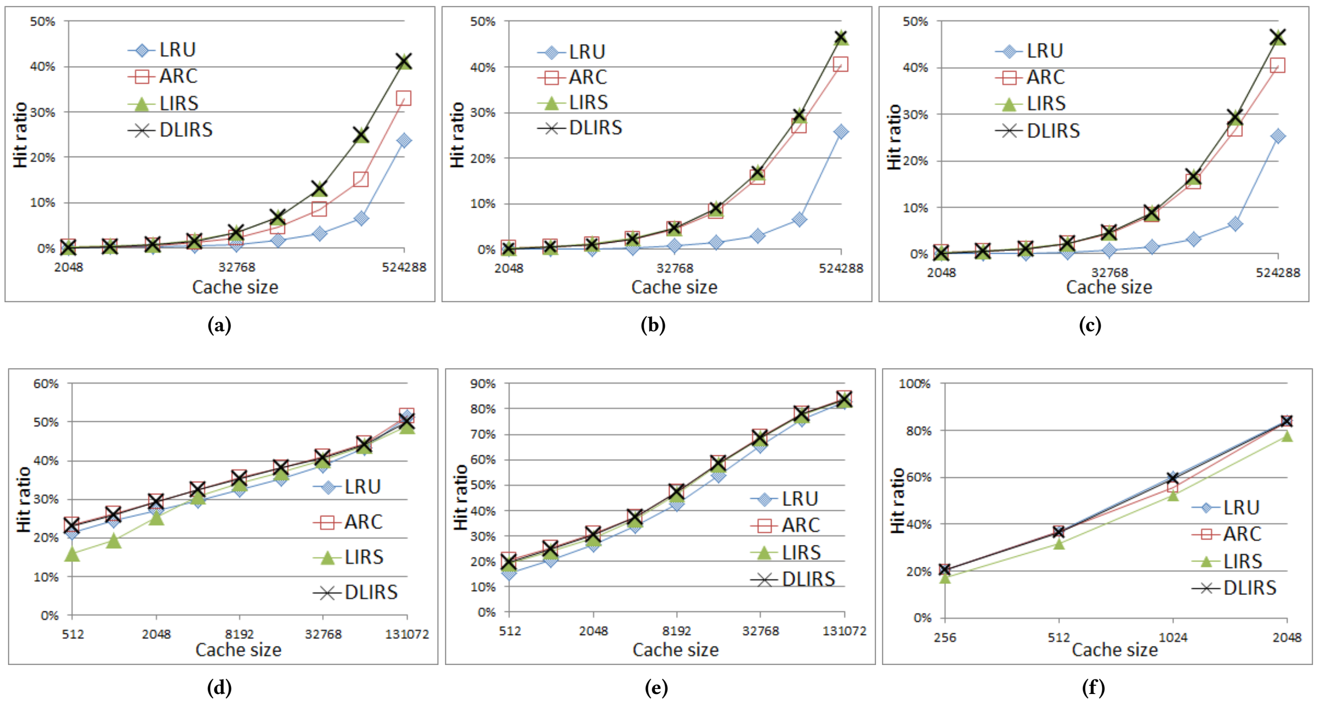


Figure 4: Complete experimental results on the traces of (a) WebSearch1; (b) WebSearch2; (c) WebSearch3; (d) Financial1; (e) Financial2; and (f) synthetic SDD trace.

APPENDIX: COMPLETE RESULTS

Figure 4 shows the complete experimental results of LRU, ARC, LIRS, and DLIRS on the 6 traces. DLIRS consistently performs close to the winner of the other policies in all the cases.

REFERENCES

[1] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*. 187–200.

[2] Pei Cao and Sandy Irani. 1997. Cost-aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems (USITS'97)*. 193–206.

[3] Jr. Edward G. Coffman and Peter J. Denning. 1973. *Operating Systems Theory*. Prentice Hall Professional Technical Reference.

[4] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. 2000. A Middleware System Which Intelligently Caches Query Results. In *Middleware (Lecture Notes in Computer Science)*, Vol. 1795. 24–44.

[5] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATEC '05)*. 323–336.

[6] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. 31–42.

[7] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. 439–450.

[8] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST '03)*.

[9] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. 1988. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 134–154.

[10] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. 297–306.

[11] Alan J. Smith. 1985. Disk Cache - Miss Ratio Analysis and Design Considerations. *ACM Trans. Comput. Syst.* 3, 3 (Aug. 1985), 161–203.

[12] James Z. Teng and Robert A. Gumaer. 1984. Managing IBM Database 2 Buffers to Maximize Performance. *IBM Systems Journal* 23, 2 (1984), 211–218.

[13] Theodore M. Wong and John Wilkes. 2002. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. 161–175.

[14] Yongrui Xu and Yongguo Han. 2011. ALIRS: A High Scalability and High Cache Hit Ratio Replacement Algorithm. In *2011 International Conference on Computational and Information Sciences (ICCIS '11)*. 66–70.