Red: An Efficient Replacement Algorithm Based on REsident Distance for Exclusive Storage Caches

Yingjie Zhao, Nong Xiao, Fang Liu
Department of Computer Science, National University of Defense Technology
Changsha, China
hyperseymour@163.com, xiao-n@vip.sina.com, liufang@nudt.edu.cn

***Abstract*—This paper presents our replacement algorithm named RED for storage caches. RED is exclusive. It can eliminate the duplications between a storage cache and its client cache. RED is high performance. A new criterion *Resident Distance* is proposed for making an efficient replacement decision instead of Recency and Frequency. Moreover, RED is non-intrusive to a storage client. It does not need to change client software and could be used in a real-life system. Previous work on the management of a storage cache can attain one or two of above benefits, but not all of them. We have evaluated the performance of RED by using simulations with both synthetic and real-life traces. The simulation results show that RED significantly outperforms LRU, ARC, MQ, and is better than DEMOTE, PROMOTE for a wide range of cache sizes.**

Keywords: exclusive caching; replacement algorithm; resident distance

I. INTRODUCTION

In general, storages and their clients (such as web servers, database systems) are physically detached to reduce the cost of management and maintenance in today's distributed computing environments. To enable the quick re-reference of data blocks, both of them typically have gigabytes even terabytes of buffer caches. Data requests from an application must travel through a client buffer before they reach a storage cache, which incurs redundancy of blocks and weak temporal locality [2, 3, 12, 14, 15] in a storage buffer cache.

In practice, storages and their clients are usually managed by independent cache policies, so they often duplicate the same blocks along their retrieval route. The duplication in a storage cache, however, is approximately useless and results in a waste of expensive cache space, since data requests are firstly satisfied by a client cache. To address this issue, exclusive caching [14, 15] has been proposed to eliminate redundant blocks and expected to deliver the performance commensurate with the aggregating cache sizes of a client-storage cache hierarchy. For example, Wong and Wilkes [14] suggest a storage cache to discard the block that has been sent to a client and to buffer the block that has been ejected by a client in their DEMOTE technique. Exclusive caching shows significant performance gain over inclusive caching [16] which has duplications. Unfortunately, most exclusive caching policies require extensions to communication protocols and changes to client software. Furthermore, exclusive caching is a multi-level collaboration technique in essence, and how to better utilize a storage cache remains an unresolved problem.

Locality-based policies work because of capturing access pattern. However, they perform poorly in a storage cache [1, 9], especially when exclusive caching is applied for a multi-level collaboration. For exclusive caching like DEMOTE, nearly all the blocks in a storage cache are originated from the demotion of an upper client cache, so it is hard to capture access pattern directly through the blocks that a storage cache holds. Previous work usually uses FIFO or LRU [5] policies to manage a storage cache, but they are obviously just better than nothing. A direct way of obtaining access pattern is to explicitly retrieve from a client. It seems promising, but leads to high communication cost and modifications to client software, which is undesirable for some real-life systems.

In this paper, we present our replacement policy for storage caches, in which *Resident Distance* is proposed as a new criterion to capture access pattern of an application, so we call this policy RED. *Resident Distance* is the amount of time that a block stays in a client cache. It can be used to distinguish locality strengths among blocks and make a replacement decision. A block with high *Resident Distance* should remain in the storage longer than one with low *Resident Distance* when they are reloaded into a storage cache after being ejected by a client. We follow Wong and Wilkes's work [14] to attain exclusion property, and we extend the Client Content Tracking Table in the paper [3] to avoid modifications to a client.

The rest of this paper is organized as follows. In Section 2, we introduce our replacement algorithm RED. In Section 3, we present our experiment setting, followed by simulation results in Section 4. In Section 5, we briefly discuss the related work. Finally, in Section 6, we conclude this paper.

II. RED

A. Attain Exclusion Property

We are inspired by DEMOTE [14] and Chen's eviction-based placement [3] to give an approach making a storage cache exclusive: (1) put the most recently used block to the position of being discarded, and drop it on the next replacement, since it will be buffered in a client, (2) detect the blocks ejected by a client, and reload them from storage disks. We follow Chen' work [3] to use the Client Content Tracking Table (CCT)

*This work is partially supported by NSFC60736013, NSFC 60903040, NCET-08-0145 and 863- 2006AA01A106 of China.

978-1-4244-7153-9/10/$26.00 ©2010 IEEE

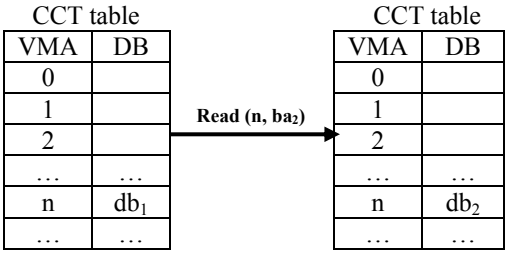

Figure 1. After receiving a read request Read (n, db_2), we can infer that the block db_1 has been ejected by a client cache.

for detecting: (1) when a block is ejected by a client cache, and (2) which block is ejected. The Client Content Tracking Table is also used in the uCache [2]. Fig. 1 shows how to keep track of the contents of a client cache by using the Client Content Tracking Table. Each entry in the CCT table records a virtual memory address (VMA) of a client cache and the disk block (DB) that resides at the VMA location. Both of these fields can be easily retrieved from the standard I/O interface. Upon receiving a read request issued from a client, the storage cache lookups the CCT table to check which disk block locates at the given client's virtual memory address. If the old block is different from the requested block, we can infer that the old block must have been ejected by the client cache, so we should reload it into the storage cache from disks. The reload overhead can be ignored due to various masking mechanism [3].

Compared with other exclusive caching, such as centralized -controlled, client-directed and hint-based policies [12, 13, 14, 15], the approach presented in this paper is smart and totally transparent to client applications. The cost of implementation is restricted within a storage server, which is important for real-life systems.

B. Resident Distance

We use *Resident Distance* (RD) to distinguish the blocks with low re-reference probability from those with high probability, instead of commonly used Recency and Frequency [4] due to weak locality in a storage cache. *Resident Distance* is defined as the time difference between when a block is passed to a client and when it is evicted by the client. Fig. 2 shows the *Resident Distance* Table (RD table) that records the RD values. When a block is about to be passed to a client, we record the time to the T_p field in the RD table. If we infer the block is ejected by the client on receiving a request at the time T, we reload it to the storage cache, and assign $\{T\text{-}T_p\}$ to its RD field.

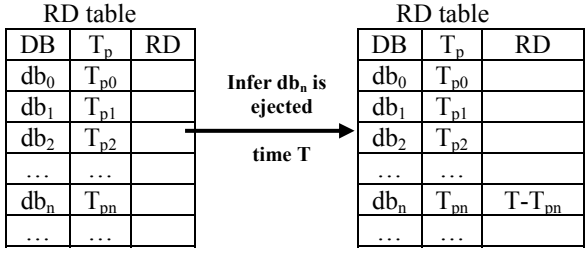

Figure 2. If we know the block db_n is passed to a client at the time T_{pn}, and we infer that the block db_n is ejected by the client at the time T, we assign $\{T\text{-}T_{pn}\}$ to its RD field.

Resident Distance denotes a relative long-time history access pattern, so it is better than Recency as a quantified criterion to distinguish blocks in a storage cache. Moreover, it can be easily obtained without changes to client software. It is also simple to use. When a free cache slot is needed, we just discard the block with the lowest RD value to make space.

C. Data Structure

Making a replacement decision requires ranking blocks. Ranking usually has the logarithmic time complexity. However, the LRU policy uses a LRU stack reducing the time complexity to a constant value. We build a similar data structure called a RD stack, shown in the Fig. 3. The RD stack contains M entries, each of which represents a block either in a client cache or in a storage cache. M is the total number of all the blocks in both caches. When a copy of a block is about to be passed to a client, we promote its entry to the top of the RD stack. When the client copy is ejected, we mark the entry and leave its position unchanged. So the position in the RD stack can approximately denote *Resident Distance*. To differentiate the blocks in the RD stack, RED sticks a boolean bit called *homeFlag* to each entry. The *homeFlag* is set to TRUE if a block is from a client cache, otherwise is set to FALSE. The RD stack is partitioned into two sub stacks. One is a SG stack, containing the blocks from a storage cache. The other is a CT stack, containing the blocks from a client cache. Hence, each block has a pair of entries: one is in the RD stack, the other either in the SG stack or in the CT stack. In fact, the SG stack and the CT stack can serve as the index to operate the RD stack. Since the entries in the SG stack are ordered by *Resident Distance*, when replacement occurs, RED can immediately discard the entry on the top of the SG stack, and then discard the pair entry in the RD stack by the mapping. When receiving a read request issued from a client, RED can also easily find the pairs of entries representing the victim block in the CT and RD stack.

D. Detailed Implementation

Fig. 4 shows the program flow chart of the RED algorithm. Procedures used in this figure are explained below. For simplicity, we assume that all the misses and hits to a storage cache are incurred by I/O requests issued from a client in the following discussion.

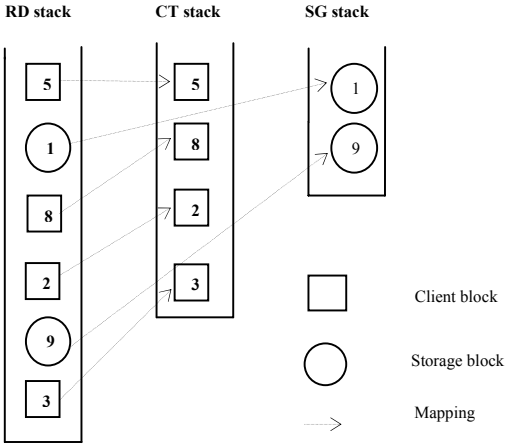

Figure 3. Data structure of the RED algorithm.

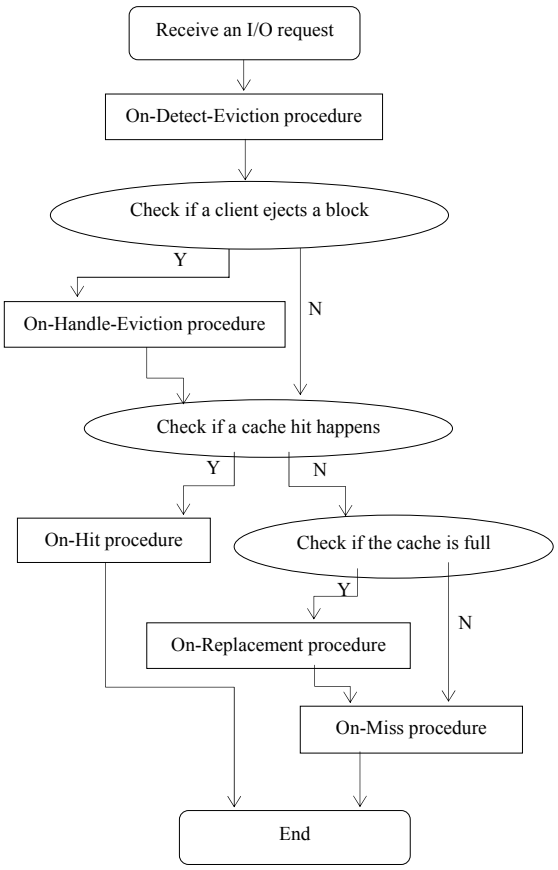

Figure 4. The program flow chart of the RED algorithm.

On-Detect-Eviction procedure: This procedure starts when a storage cache receives an I/O request issued from a client. We firstly parse the request to obtain the currently accessed disk block address (DB) and the corresponding virtual memory address (VMA) of a client, and then lookup the CCT table to check whether another block has resided at the given virtual memory address. If a different block does not exist at that location, a new record (VMA, DB) representing the new block is created and inserted into the CCT table, and then we can check if the requested block is in the storage cache. Otherwise, we update the CCT table by using the new block address to replace the old block address, and then we can call the *On-Handle-Eviction* procedure.

On-Handle-Eviction procedure: This procedure starts when a block is detected to be ejected by a client through the *On-Detect-Eviction* procedure. We can firstly find the entry representing the block in the CT stack, and then find the pair entry in the RD stack by the mapping between the CT stack and the RD stack. If the block is already buffered in the storage cache, that is to say- it is duplicated in both caches, we simply discard the pair entries. If the block is not buffered, we load the block into the storage cache, change the *homeFlag* bit of the pair entries to False, and then move the entry from the CT stack to the SG stack to indicate that the block has been moved from the client cache to the storage cache. Finally, the pair entries are modified to point to the cache location where the block is loaded.

On-Hit procedure: This procedure starts when a cache hit occurs in a storage cache. We send a copy of the requested block to a client directly from the storage cache. The existing entries representing the block are promoted to the top of the RD and SG stack. After that, we create new entries to represent the copy, and place them on the top of the RD and CT stack.

On-Replacement procedure: This procedure starts when a cache miss occurs in a storage cache that is full. We remove the entry on the top of the SG stack and the corresponding entry on the RD stack, and then discard the block it represents.

On-Miss procedure: This procedure starts when a cache miss occurs in a storage cache that has free cache space. We load the requested block into a free cache slot, and then send its copy to the client. After that, we create entries separately representing the block in the storage cache and its copy in the client cache. These entries are placed on the top of the corresponding stacks.

In a single-client single-storage model, according to the conditions whether the storage cache and the client cache are full, we can define the following four different cases:

- NF: neither cache is full.
- SFCN: the storage cache is full, but the client is not.
- CFSN: the client cache is full, but the storage is not.
- AF: both caches are full.

Initially, both the storage cache and the client cache are empty. The cache hierarchy is in the NF case until one of them is full. In the NF case, no block is ejected, so the contents of the storage cache and the client cache are the same. Since read requests that reach the storage are misses from the client, these requests also incur misses in the storage cache, so the *On-Miss* procedure is carried out immediately after the *On-Detect-Eviction* procedure completes.

If the client cache is larger than the storage cache, after a short-period running, there must be a time when the storage cache is full and the client cache is not. At that time, the cache hierarchy is in the SFCN case. In this case, no block is ejected by the client cache. Hence, the blocks in the storage cache are a subset of the blocks in the client cache. For the same reason as in the NF case, there are no hits occurring in the storage cache. Because the storage cache is full, the *On-Replacement* procedure must be carried out to make space after the *On-Detect-Eviction* procedure completes, and then the *On-Miss* procedure can start.

Otherwise, if the client cache is smaller, the client cache will be full prior to the storage cache. At the time when the client cache is full and the storage cache is not, the cache hierarchy is in the CFSN case. In this case, because the client cache is full, it needs to eject an old block to make space after retrieving a new block from the storage, so the *On-Handle-Eviction* procedure needs to be carried out after the *On-Detect-Eviction* procedure completes. After that, both the *On-Hit* procedure and the *On-Miss* procedure may start to handle the I/O requests depending on whether the requested block is in the storage cache. However, since the storage cache is not full, the *On-Replacement* procedure will not be used in this case.

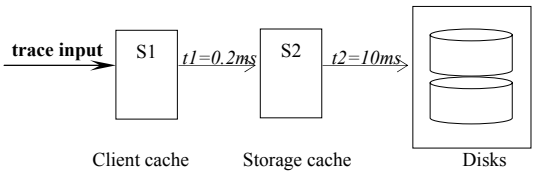

Figure 5. A single-client two-level cache hierarchy.

After a warm-up process, both the storage cache and the client cache are full. The cache hierarchy is in the AF case, which is very representative for a running system. In this case, all the five procedures may start to accomplish I/O requests.

III. EXPERIMENTS SETTING

A. Cache Model

Since exclusive caching policies do not always perform well in the multi-client systems, especially when the clients share significant amounts of blocks, it is usually used in the single-client systems or a proportion of multi-client systems in which each client accesses disjoint workloads. In this paper, we test our algorithm in a single-client two-level cache hierarchy, and we believe the simulation results can be extended to multi-level systems.

Fig. 5 shows the cache hierarchy and its configurations: the average network latency between the client and the storage is 0.2ms, the average disk latency is 10ms, and here we ignore the network latency between the storage and the disks.

B. Traces

We use both synthetic and real-life workloads to evaluate the performance of our cache replacement algorithm.

Zipf: we use a trace with a Zipf-like distribution, where the frequency of a READ for block i is proportional to $1/i^{\alpha}$, for α close to 1. This approximates many common access patterns, such as file access in web servers: a few are accessed frequently, others not. We follow previous work [13, 14, 15] to choose this workload for evaluating multi-level caching algorithms.

TPC-H: this trace is collected by running the TPC-H database benchmark, which is a decision support system defined by Transaction Processing Council (TPC). It contains long sequential and little random access, which is also used in previous work [13, 14].

T1-T2: these traces are collected over several months from a real-life high performance storage system, which is called the China National Grid system (CNGrid). Several large-scale applications are currently deployed in the CNGrid, such as National Meteorological Grid, Spatial Information Grid and Remote Sensing Data Processing system. These traces have been used in the paper [20].

C. Competitive Policies

We have implemented LRU, ARC [11], MQ, Demote and Promote in our cache simulator. LRU and ARC are widely-used and powerful single level algorithms; MQ is a smart second level algorithm; Demote and Promote are two typical multi-level collaboration policies.

We use LRU to manage the client cache. In the Demote and Promote policies, LRU is also used to manage the storage cache. We ignore the write commands in all the traces for simplicity since a write cache is typical managed using a different police from a read cache.

All the above algorithms except Promote have the same hit ratio in the client cache, so the hit ratio in the storage cache can represents their performances. We use the average response time as the metric when compared with Promote, since its hit ratio in the client cache is different.

IV. SIMULATION RESULTS

We change the size of the client cache from 20K to 180K while keeping the total size of the cache hierarchy equal to 200K blocks. The hit ratio in the storage cache on different polices is plot in Fig. 6. When the storage cache is enlarged, the hit ratio increases simultaneously. Depending on the hit ratio curve, we can divide these policies into three groups. The first group includes LRU and ARC, which are inclusive caching. When the storage cache is increased from 0K to 100K blocks, there are almost no hits in the storage cache. The second group contains only one policy MQ, which is designed for the second-level buffer cache. Essentially, it is an inclusive policy, however, it holds more blocks with high access frequency, which is different from the client cache, so it performs better than the policies in the first group. Demote and RED belongs to the third group. They are exclusive, and achieve good performance in the simulations. As we expected, RED performs better than Demote, about 10-15% when the storage cache is comparable with the client cache in size.

In Fig. 7, we present the average response time to compare RED with PROMOTE, since their hit ratio in the client cache is different. The average response time increases along with the enlargement of the storage cache, because the number of hits in the client cache decreases simultaneously. We observe that RED performs slightly better than Promote.

V. RELATED WORK

Single level cache algorithms have been studied extensively, such as LRU, LFU, LRFU [17], LRU-K [7], 2Q [8], LIRS [10], and ARC. However, such policies perform poorly when the cache level is beyond one. Zhou et al. proposes a Multi-Queue [9] algorithm for the second level buffer cache. Based on the observation that frequently used blocks is more important than recently used blocks for a storage server [18, 19], they use N queues to distinguish differently referenced blocks, where Q_i maintains blocks which are referenced at least 2^i times and no more than 2^{i+1}-1 times. The drawback is that the multi-queue duplicates the same blocks both at the storage side and at the client side. Recent researchers devote themselves to diminish such duplications among cache hierarchies. Bairavasundaram and Sivathanu infer contents of the client cache by monitoring update information of the file system metadata in their X-RAY [6] algorithm, which is constructed to manage RAID controller

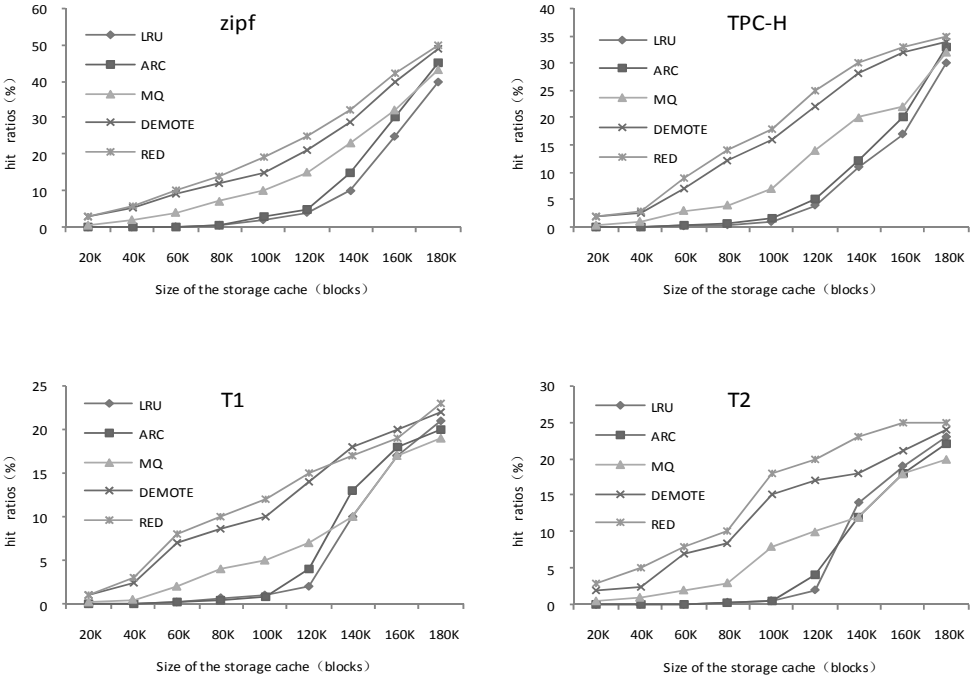

Figure 6. The hit ratio in the storage cache. The total size of the cache hierarchy is 200K block.

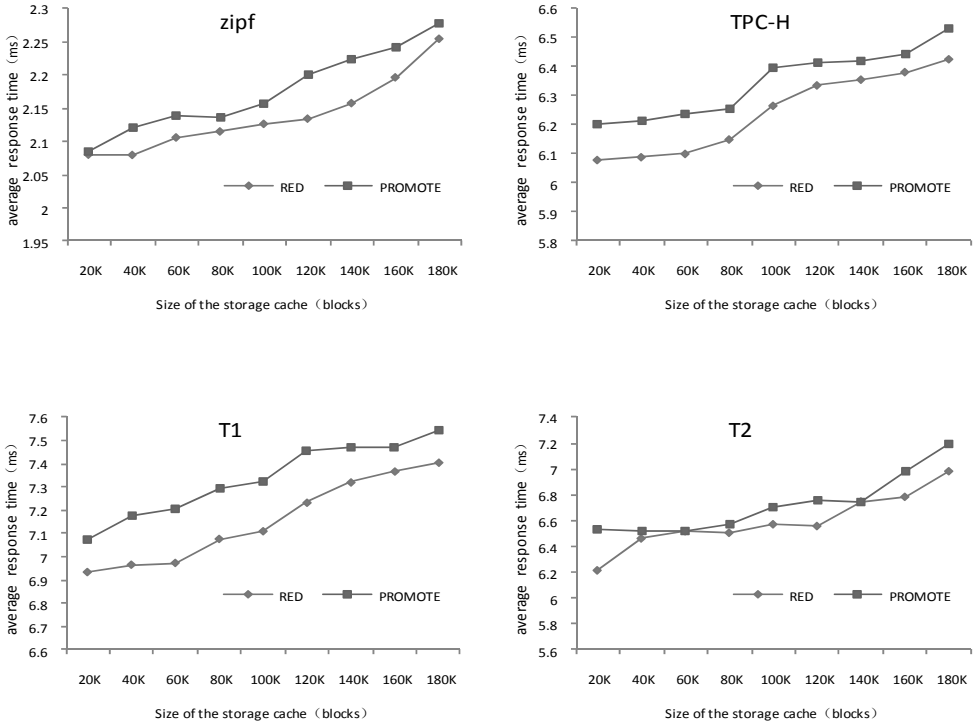

Figure 7. The average response time seen by the client. The total size of the cache hierarchy is 200K block.

caches. However, this work suffers from low guess accuracy. The Karma [13] policy presented by Yadgar and Factor uses specific application hints to make allocation and replacement decisions among all the cache levels. Although the database applications can benefit a lot from using this method, since the Karma is hint-based, it is hard to be applied to other kinds of applications. Jiang and Zhang propose a client-directed cache replacement algorithm called ULC [12], in which the client has the knowledge of all the caches, and controls the movement of blocks between low and high level caches by issuing Retrieve and Demote commands. The client cache has to keep track of the contents of all the caches, so the algorithm is very invasive and costly in the implementations. To enable the collaboration of independent policies among cache hierarchies, Wong and Wilkes suggest a general technique called DEMOTE [14], which can be applied to existing replacement policies to maintain exclusivity property. The main idea is before a client ejects a block, it first returns the block to the storage cache. This approach induces extra network traffic, so the performance gain may decline when the network bandwidth is limited. Chen et al. propose an eviction based cache placement policy [3] to eliminate such network traffic, in which a Client Content Tracking Table is used to monitor whether a client has discarded a block. Once such a block is found, the storage will reload it directly from storage disks. He et al. propose the uCache [2] by combining exclusive caching with cooperative caching, so it can achieve good performance even used in high-correlated multi-clients systems. Gill uses an adaptive probabilistic filtering mechanism to conduct the placement of blocks among cache hierarchies in the PROMOTE [15] policy, which provides exclusivity with low network traffic. In PROMOTE, the more a block is referenced, the faster it is promoted to a client cache, so more hits are accumulated in a client cache. However, implementing PROMOTE requires modifications to both storage and client caches, which is undesirable in some real-life systems.

VI. CONCLUSION

We have proposed *Resident Distance*, a new distinguished criterion for making a replacement decision; we have presented RED, an efficient replacement algorithm to enforce the cache hierarchy exclusive without modifications to client software; we have demonstrated its performance using trace-driven simulations. In our tests, RED significantly outperforms LRU, MQ, ARC, and is better than DEMOTE, PROMOTE in most cases.

REFERENCES

[1] D. Muntz and P. Honeyman, “Multi-level caching in distributed file systems - or - your cache ain’t nuthin’ but trash”, in Proc. of the USENIX Winter Conf., pages 305–313, 1992.

[2] Xubin He, Li Ou, Martha Kosa, Stephen Scott, and Christian Engelmann, “A Unified Cache for High Performance Cluster Storage Systems”, International Journal of High Performance Computing and Networking, Vol. 5, No. 1, 2007, pp. 97-109.

[3] Z. Chen, Y. Zhou, and K. Li, “Eviction-based cache placement for storage caches”, in Proceedings of the 2003 USENIX Annual Technical Conference, pages 269–282, 2003.

[4] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement”, in Proc. ACM SIGMET-RICS Conf., pages 134–142, 1990.

[5] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies”, IBM Sys. J., 9(2):78–117, 1970.

[6] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C.Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, “X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs”, in Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA ’04), Munich, Germany, June 2004.

[7] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K page replacement algorithm for database disk buffering”, in Proc. ACM SIGMOD Conf., pages 297–306, 1993.

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

[9] 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, pages 91–104, June 2001.

[10] 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., pages 31-42, 2002.

[11] N. Megiddo and D.S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache”, in Proc. Second USENIX Conf. File and Storage Technologies, pages 115-130, Mar. 2003.

[12] S. Jiang and X. Zhang, “ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches”,in Proc. ICDCS Conf., pages 168–177, 2004.

[13] M. Factor, A. Schuster, and G. Yadgar, “Karma: Know-it-all replacement for a multilevel cache”, in Proc. FAST Conf., pages 169-184, 2007.

[14] T. M. Wong and J. Wilkes, “My cache or yours? Making storage more exclusive”, in Proc. of the USENIX Annual Technical Conference, pages 161-175, 2002.

[15] B. S. Gill, “On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions”, in Proc. FAST Conf., pages 49-65, 2008.

[16] J.-L. Baer and W.-H. Wang, “On the inclusion properties for multi-level cache hierarchies”, in ISCA ’88: Proceedings of the 15th Annual International Symposium on Computer architecture, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[17] 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, 50(12):1352–1360, 2001.

[18] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt, “Disk cache replacement policies for network fileservers”, in International Conference on Distributed Computing Systems, pages 2–11, 1993.

[19] Kevin W. Froese and Richard B. Bunt, “The effect of client caching on file server workloads”, in HICSS (1), pages 150–159,1996.

[20] Rui Chu, Nong Xiao, Yongzhen Zhuang, Yunhao Liu, Xicheng Lu, “A distributed paging RAM grid system for wide-area memory sharing,” in Proc. IPDPS Conf. pages 10-17, 2006.