Eviction Based Cache Placement for Storage Caches

Zhifeng Chen[†], Yuanyuan Zhou[†], and Kai Li[*]

[†]*Department of Computer Science*
University of Illinois at
Urbana-Champaign
1304 W. Springfield Ave
Urbana, IL 61801

[*]*Department of Computer Science*
35 Olden Street
Princeton University
Princeton, NJ 08544

Abstract

Most previous work on buffer cache management uses an access-based placement policy that places a data block into a buffer cache at the block's access time. This paper presents an *eviction-based placement* policy for a storage cache that usually sits in the lower level of a multi-level buffer cache hierarchy and thereby has different access patterns from upper levels. The main idea of the eviction-based placement policy is to delay a block's placement in the cache until it is evicted from the upper level. This paper also presents a method of using a client content tracking table to obtain eviction information from client buffer caches, which can avoid modifying client application source code.

We have evaluated the performance of this eviction-based placement by using both simulations with real-world workloads, and implementations on a storage system connected to a Microsoft SQL server database. Our simulation results show that the eviction-based cache placement has an up to 500% improvement on cache hit ratios over the commonly used access-based placement policy. Our evaluation results using OLTP workloads have demonstrated that the eviction-based cache placement has a speedup of 1.2 on OLTP transaction rates.

1 Introduction

With the ever-widening speed gap between processors and disks, and decreasing memory price, modern high-end storage systems typically have several or even tens of gigabytes of cache RAM [28]. The clients of a storage system, e.g. filers or database servers, also have large amount of devoted main memory for caching [30]. These buffer caches form a *multi-level buffer cache hierarchy* (See Figure 1). Though the aggregate size of this hierarchy is increasingly larger, the system might not deliver the expected performance commensurate to the aggregate cache size if these caches could not work together effectively. In this paper, we investigate a method to manage the multi-level buffer cache hierarchy effectively. Specifically, we focus on how to make better use of a storage server cache that coexists with large buffer caches of storage clients.

Previous studies [19, 31, 28] have shown that storage caches have different access patterns and thereby should be managed differently from caches at upper level. Accesses to storage caches usually exhibit weak temporal locality because accesses to storage caches are actually misses from upper level buffer caches. In other words, accesses made by applications are first filtered by upper level buffer caches before they reach storage caches. As a result, widely used locality-based cache replacement algorithms, such as Least Recently Used (LRU), do not perform well for storage caches. This has been observed by Muntz and Honeyman's as well as our previous study on file and storage server cache, respectively [19, 31].

Most previous work on file or storage buffer caches focused on cache replacement policies. Buffer cache management mainly consists of two components: replacement policy and placement (admission) policy. A replacement policy decides which block should be replaced to make space for a new block when the cache is full, while a placement policy decides when a block should be brought into a cache. The access-based placement policy has been widely used in most previous studies. This policy places a block into a cache at the time this block is accessed. The main motivation for such a placement is to maintain the *inclusion property* (any block that resides in an upper level buffer cache is also contained in a lower

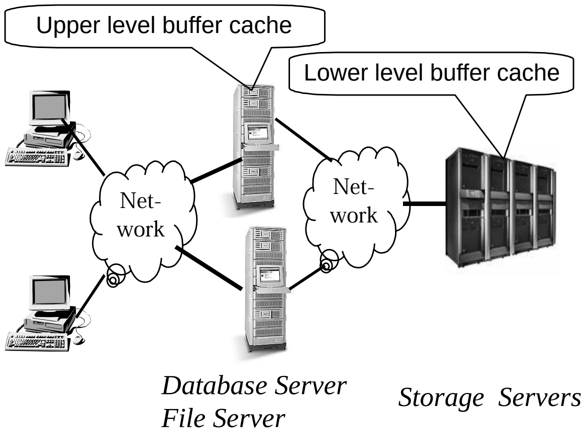


Figure 1: Storage caches in a multi-level buffer cache hierarchy.

level). This property is desirable when the upper level buffer cache is significantly smaller than the lower level one. For example, in a typical main memory hierarchy, the processor cache is several orders of magnitude smaller than the main memory. However, since a storage buffer cache usually has similar size to a storage client cache (e.g., a database or file server cache), maintaining this property at a storage cache has been shown unnecessary and can even hurt performance [19, 31, 28].

To make a storage cache *exclusive*, Wong and Wilkes have proposed an operation called *DEMOTE* to send data evicted from a client buffer to a disk array cache [28]. Their study made a very interesting observation about wasteful inclusiveness of storage caches. They also showed that *DEMOTE* can effectively improve the hit ratios of a storage cache in various simulated workloads. However, to implement this method in a real system, it requires modification to source code of client software, such as a database server, to explicitly utilize this new operation. Besides, in some production systems, the network between a storage client and a storage system can become a bottleneck because of the extra network traffic imposed by *DEMOTE* operations. For applications with intensive I/Os, *DEMOTE* operations can also increase the average miss penalty in a client cache due to waiting for free pages. In addition, the *DEMOTE* method has been evaluated only by simulations, so it is unclear how it would perform in a real system.

This paper generalizes the idea of an exclusive cache and presents an eviction-based cache placement policy for storage caches. Moreover, our method targets application domains where the *DEMOTE* approach is not readily applicable. In particular, we propose a method called client content tracking (CCT) table to estimate a client's eviction information. This method can avoid modifications to the client's software source code. We also propose letting the storage system decide when to fetch the evicted blocks from disks, a.k.a. *reload*, to avoid delaying demand access at the client side. In other word, our approach is transparent to applications. Since the decision is made by the storage servers, our method also enables more sophisticated optimizations such as eliminating unnecessary reloads or masking reloads using priority-based disk scheduling.

To evaluate the eviction-based placement policy, we have implemented it in both simulators and a storage system connected to a Microsoft SQL server database. Our simulation results with real-world workloads have shown this placement policy can significantly improve cache hit ratios by up to a factor of 5 over the commonly-used access-based placement. Our real system experimental results with OLTP workloads have demonstrated that the eviction-based placement can improve the transaction rate by 20%. We also compared the *DEMOTE* method with our scheme in a storage system. Our implementation results using OLTP workloads have shown that our scheme has a 20% higher transaction rate than the *DEMOTE* method when the client-storage interconnection has a limited bandwidth.

The remainder of this paper is organized as follows. Section 2 briefly describes cache placement policies and metrics to measure those policies. Section 3 presents the benefits of the eviction-based placement policy. Section 4 describes the CCT table to estimate eviction information from a client and Section 5 discusses ways to reduce the reload overheads introduced by the eviction-based placement. After we present the implementation results and the effects of optimizations in Section 6, we conclude the paper and point out the limitations of our study.

2 Cache Placement

A cache placement (admission) policy decides when a missed block should be fetched into a cache. For example, the commonly used access-based placement policy places a block into a cache at the time the block is accessed. The eviction-based placement policy presented in this paper fetches a block into a cache when this block is evicted from an upper level buffer cache.

We use a metric called *idle distance* to evaluate different cache placement policies. In this section, we describe *idle distance* and then use it to measure the access-based and eviction-based placement policies with three large real-world storage cache access traces.

2.1 Idle distance

To evaluate the effectiveness of different cache placement policies, we need a metric to measure the cost of keeping a block in a cache to generate a cache hit. The idle distance can well serve this purpose. For a reference to a block, its *idle distance* is defined as the period of time this block resides in the cache but is not being accessed. More specifically, for a *reference string* (a numbered sequence of temporally ordered accesses to a cache), we use sequence numbers to denote "time", and the time of the previous and current access to a block b as $prev(b)$ and $current(b)$, respectively. We then use $place(b)$ to denote the time b is put into the cache. The idle distance for the current reference to b is defined as $current(b) - max(prev(b), place(b))$, i.e., the time interval from the maximum of b's placement time and b's previous access time to the current access. During this time interval, b occupies a memory block but is not accessed.

A good cache placement policy should try to reduce idle distance to improve the efficiency of a buffer cache. An ideal policy would put a block into a cache right before it is accessed. But this is impossible unless the system has zero cost to load a missed block.

2.2 Access-based Placement

In the commonly-used access-based placement policy, the *idle distance* for a reference is equal to its *reuse distance*, which is the distance between the previous access and the current access to this block, i.e., $current(b) - prev(b)$. Since the access-based placement policy puts a missed block b into a cache right at its access time, $prev(b)$ equals $place(b)$. Therefore, the reuse distance for this reference is the same as its *idle distance*. Reuse distances have been used by many studies [18, 21, 1, 15] including our previous study [31] to examine the temporal locality in an access sequence.

Our previous study [31] has shown that accesses to storage caches have long reuse distances because accesses from applications have already been filtered through one or more levels of buffer caches before they arrive at storage caches. If a client cache of size k uses a locality based replacement policy like LRU, after a reference to a block, it takes at least k distinct references to evict this block from the client's buffer cache. Therefore, the next access to block b in the storage cache is separated by at least k distinct references in the reference sequence at the storage cache. This long reuse distance significantly limits the efficiency of commonly-used access-based placement at storage caches and other lower level buffer caches.

2.3 Eviction-based Placement

In the eviction-based placement policy, the *idle distance* for a reference is equal to its *eviction distance*. At a lower level cache like a storage cache, the eviction distance for a reference is defined as the distance between the current access and the last time it is evicted from a client buffer cache. In other words, if we use $evict(b)$ to denote the "time" when b was most recently evicted from a client buffer cache, the eviction distance for the current access to b is $current(b) - evict(b)$. Since the eviction-based placement policy fetches the block when it is evicted from a client, $place(b)$ equals $evict(b)$. Because an eviction from a client always happens after the previous access to the same block, $prev(b)$ is smaller than $evict(b)$, which implies $max(prev(b), place(b)) = evict(b)$. Therefore, the idle distance for a reference equals the eviction distance of this reference.

We use idle distance distributions to compare the two placement policies. An idle distance histogram shows the number of references for various distance values. Figure 2 compares idle distance distributions for both access-based placement policy (AC) and eviction-based placement policy (EV) using three real-world storage access traces including:

- **Auspex I/O Trace** is a disk I/O trace collected by filtering the Auspex file Server access trace [6] through an 8 MB NFS file server cache simulator.

- **MS-SQL-Large** is collected from a storage system connecting to a Microsoft SQL database server running the standard TPC-C benchmark [25, 16] for two hours. The TPC-C database contains 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The trace captures all I/O accesses from the Microsoft SQL server to the storage system. The trace ignores accesses to log disks. The Microsoft SQL server cache size is set to be the machine memory limit, 1 Gigabytes.

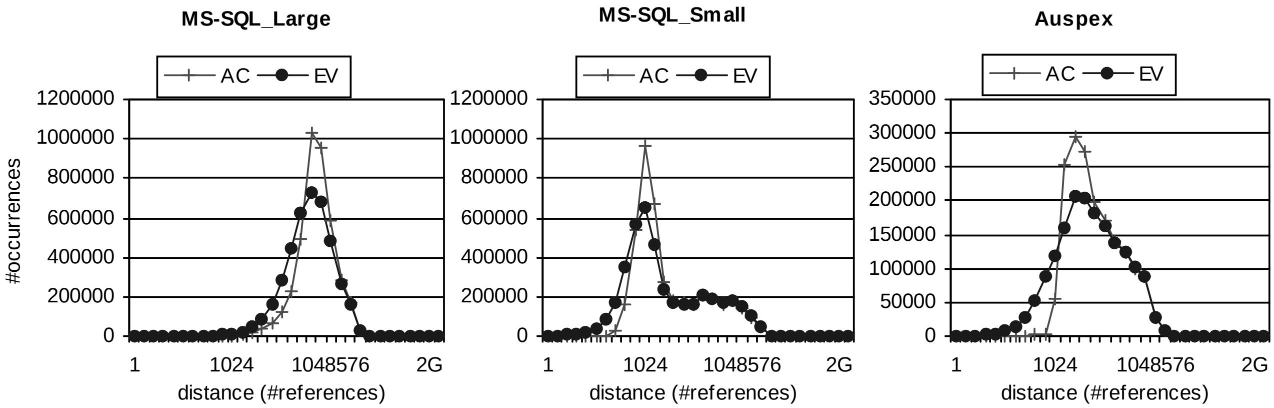


Figure 2: Idle distance distributions for both access-based placement policy (AC) and eviction-based placement policy (EV) with three storage access traces (Note: all figures are in logarithmic scales).

- **MS-SQL-Small** is collected with the same setup as the previous trace except the database buffer cache size is set to be 64 MBytes. We collected this trace in order to predict results with much larger databases.

As shown on Figure 2, all histogram curves are hill-shaped. Peak distance values, while different, are all relatively large and occur at distances greater than their client cache sizes. This indicates that most of accesses are far apart from previous accesses to the same blocks or previous evictions from clients, showing weaker temporal locality at storage caches.

Comparing the two curves, one can find out that eviction distances are shorter than reuse distances. Figure 2 shows there are fewer occurrences of EV at large distance values(or more occurrences at small distance values) than AC. For example, In the MS-SQL-Large trace, 3.0 million references in AC have idle distances greater than 262144, whereas only 2.3 million references in EV have idle distances greater than 262144. The main reason for this difference is very intuitive. Since a block b first needs to be fetched from a storage cache into a client buffer cache before being evicted from the client cache, $evict(b)$ is usually greater than $prev(b)$. As a result, the eviction distance $(current(b) - evict(b))$ is smaller than the reuse distance $(current(b) - prev(b))$. This implies that the eviction-based placement policy can utilize a storage cache more efficiently than the commonly used access-based placement policy.

3 Benefits of Eviction-based Placement

The eviction-based placement puts a block into a cache when this block is evicted from an upper level cache. This placement policy was first proposed in the victim cache design for hardware processor caches [14]. A victim cache, a small fully-associative cache between a processor cache and its refill path, is used to keep cache blocks that are recently evicted from the processor cache. It has been shown that a victim cache can significantly improve the processor cache performance.

Eviction-based placement is independent from cache replacement policies. Therefore, it can be combined with most replacement algorithms including LRU, Frequency Based Replacement (FBR) [22], 2Q [13], and Multi-Queue (MQ) [31].

To find out the effects of eviction-based placement on cache hit ratios of various replacement policies, we have built four trace-driven cache simulators that respectively use LRU, FBR, 2Q and MQ as the replacement policy. All cache simulators can run with two options: the original (access-based) placement policy and the eviction-based placement policy. Since our first goal is to find out the upper-bound of EV's improvement on hit ratios, we did not simulate disk accesses and network accesses. The extra overheads introduced by EV are discussed in detail in Section 5. These overheads are also reflected in our implementation results on a real system.

Figure 3 compares the hit ratios between the access-based and eviction-based placement policies for four different cache replacements with the MS-SQL-Large trace. LRU + EV means that the cache is managed using LRU as the replacement policy and

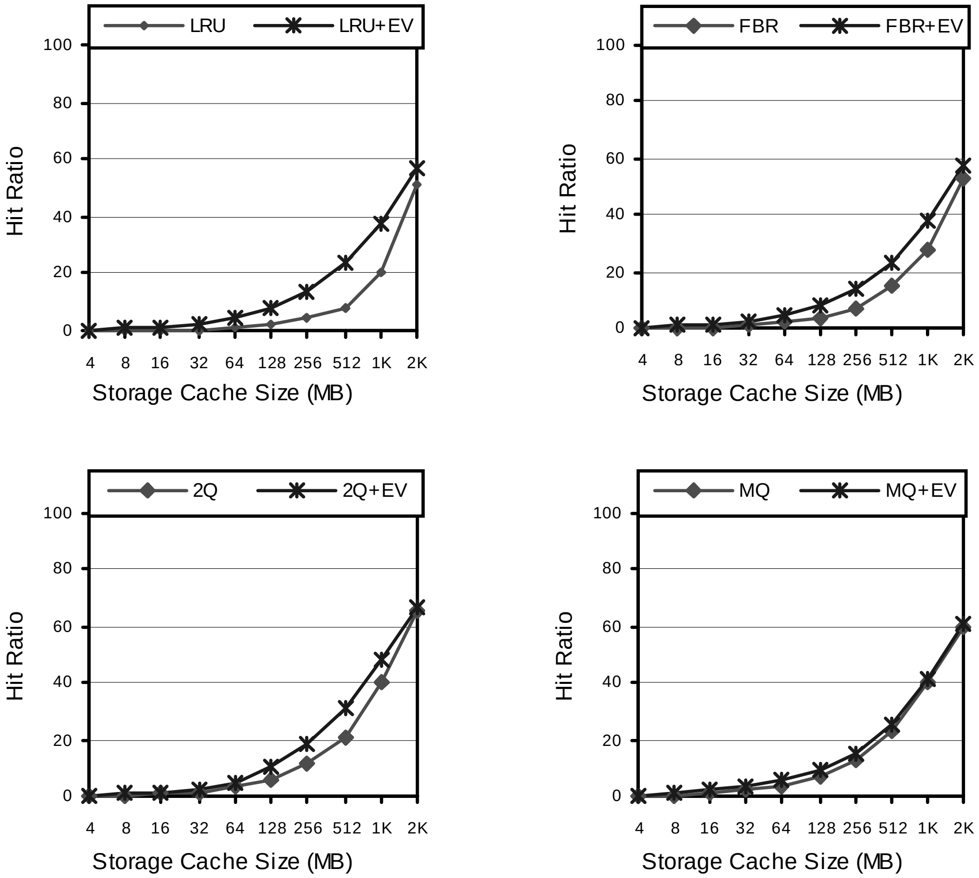


Figure 3: Benefits of eviction-based placement with MS-SQL-Large with different replacement algorithms.

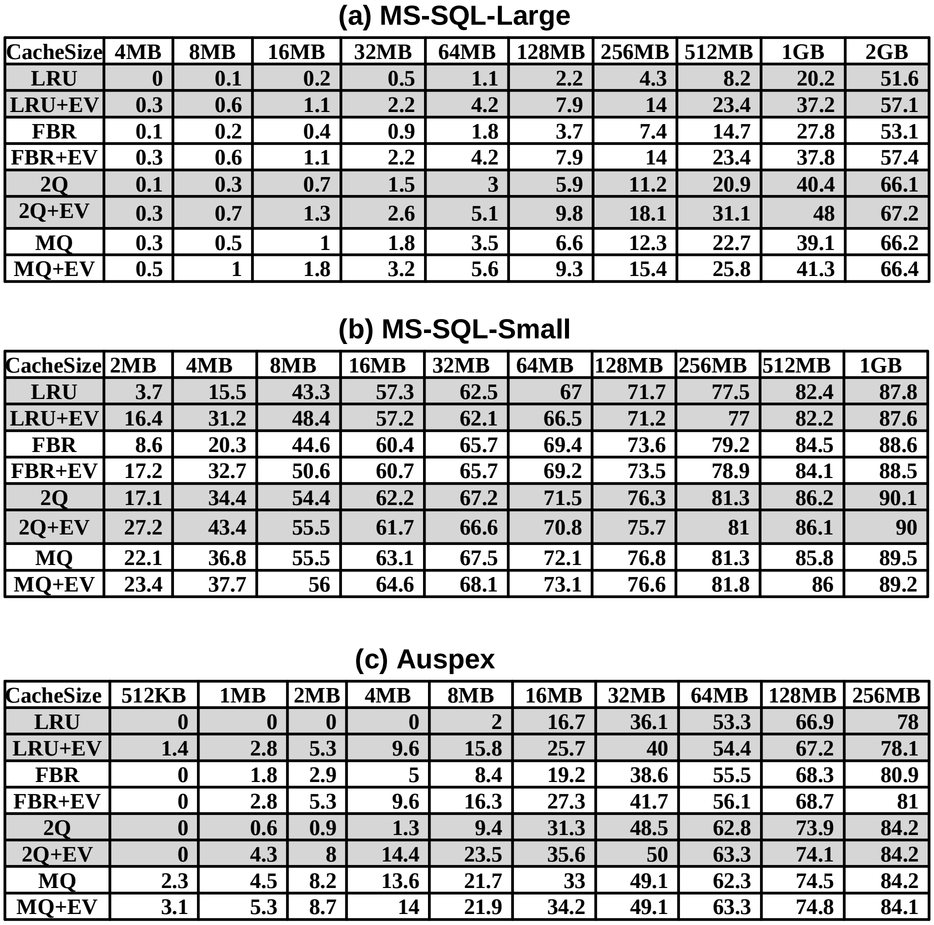
EV as the placement policy, and other abbreviations are similar.

As shown on Figure 3, the eviction-based placement always performs better than the access-based placement. In many cases the gap between these two is quite substantial. For example, when the LRU replacement policy is used, the eviction-based placement has 10% to 5 times higher hit ratios than the access-based placement. The improvements for FBR and 2Q are also significant, up to a factor of 2.

The effects of the eviction-based placement are different for various replacement algorithms. For example, in a 512 MBytes storage cache, the eviction-based placement outperforms the access-based placement by a factor of 2 for LRU, 49% for FBR, 59% for 2Q and only 15% for MQ. The eviction-based placement has the largest improvement on LRU than on the other three replacement algorithms because LRU replaces the block with the longest idle distance from the current time. The idle distance in the eviction-based placement equals the eviction distance, which is always smaller than the idle distance (reuse distance) in the access-based placement. As a result, some blocks that are evicted by LRU in the access-based placement can stay in the EV-based cache for a longer time to be hit again at next references.

The eviction-based placement has the least impact on MQ among all four replacement algorithms. Since MQ was designed based on the long idle distance access patterns at storage caches, it can selectively keep some frequently accessed blocks in a cache for a longer time. Because of this reason, delaying a block's placement time does not offer large benefit. Therefore, for MQ, EV only has 11-80% improvement over the access-based placement.

The gap between the eviction-based placement and the access-based placement is more pronounced for smaller cache sizes. For example, in the MS-SQL-Large trace with a 128 MBytes storage cache using the 2Q replacement policy, the eviction-based placement has a hit ratio of 9.8% whereas the access-based placement achieves a hit ratio of 5.9%. The gap is even larger for extremely smaller cache size



(a) MS-SQL-Large

CacheSize	4MB	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1GB	2GB
LRU	0	0.1	0.2	0.5	1.1	2.2	4.3	8.2	20.2	51.6
LRU+EV	0.3	0.6	1.1	2.2	4.2	7.9	14	23.4	37.2	57.1
FBR	0.1	0.2	0.4	0.9	1.8	3.7	7.4	14.7	27.8	53.1
FBR+EV	0.3	0.6	1.1	2.2	4.2	7.9	14	23.4	37.8	57.4
2Q	0.1	0.3	0.7	1.5	3	5.9	11.2	20.9	40.4	66.1
2Q+EV	0.3	0.7	1.3	2.6	5.1	9.8	18.1	31.1	48	67.2
MQ	0.3	0.5	1	1.8	3.5	6.6	12.3	22.7	39.1	66.2
MQ+EV	0.5	1	1.8	3.2	5.6	9.3	15.4	25.8	41.3	66.4

(b) MS-SQL-Small

CacheSize	2MB	4MB	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1GB
LRU	3.7	15.5	43.3	57.3	62.5	67	71.7	77.5	82.4	87.8
LRU+EV	16.4	31.2	48.4	57.2	62.1	66.5	71.2	77	82.2	87.6
FBR	8.6	20.3	44.6	60.4	65.7	69.4	73.6	79.2	84.5	88.6
FBR+EV	17.2	32.7	50.6	60.7	65.7	69.2	73.5	78.9	84.1	88.5
2Q	17.1	34.4	54.4	62.2	67.2	71.5	76.3	81.3	86.2	90.1
2Q+EV	27.2	43.4	55.5	61.7	66.6	70.8	75.7	81	86.1	90
MQ	22.1	36.8	55.5	63.1	67.5	72.1	76.8	81.3	85.8	89.5
MQ+EV	23.4	37.7	56	64.6	68.1	73.1	76.6	81.8	86	89.2

(c) Auspex

CacheSize	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB	256MB
LRU	0	0	0	0	2	16.7	36.1	53.3	66.9	78
LRU+EV	1.4	2.8	5.3	9.6	15.8	25.7	40	54.4	67.2	78.1
FBR	0	1.8	2.9	5	8.4	19.2	38.6	55.5	68.3	80.9
FBR+EV	0	2.8	5.3	9.6	16.3	27.3	41.7	56.1	68.7	81
2Q	0	0.6	0.9	1.3	9.4	31.3	48.5	62.8	73.9	84.2
2Q+EV	0	4.3	8	14.4	23.5	35.6	50	63.3	74.1	84.2
MQ	2.3	4.5	8.2	13.6	21.7	33	49.1	62.3	74.5	84.2
MQ+EV	3.1	5.3	8.7	14	21.9	34.2	49.1	63.3	74.8	84.1

Figure 4: Cache hit ratios for all three traces.

(4MBytes), although the hit ratios are so small that two curves in Figure 3 is indistinguishable. But with a 2 GBytes of storage cache, both placement policies have similar cache hit ratios. This can be explained using idle distances. Suppose a storage cache has k blocks. Accesses with idle distances smaller than k can usually hit in the cache, but most of the other accesses would generate cache misses. When k is smaller than the peak idle distance (the distance with most number of references) shown on an idle distance distribution histogram (Figure 2), more accesses have idle distances smaller than k in the eviction-based placement than in the access-based placement. As a result, the eviction-based placement performs better than the access-based placement. But this advantage of eviction-based placement decreases when k increases. As a result, the performance gap between these two also decreases.

Figure 4 shows the hit ratios for all three traces. The overall results for the other two traces are similar to those for MS-SQL-Large. For MQ-SQL-Small, the gap between the two placement policies disappears when the storage cache size is greater than 16 MBytes (2048 8 Kbytes-blocks). This is because the difference in the idle distance distribution between these two policies becomes invisible when the idle distance is greater than 2048 references(see Figure 2).

4 Obtaining Upper Level Eviction Information

Although the eviction-based placement has shown significant benefits over the access-based placement for storage caches, two challenging issues need to be addressed for the eviction-based placement to be used in real systems. The first is to obtain eviction information from client buffer caches. In the hardware victim cache example, when a processor cache evicts a block, it passes the block to the victim cache. However, in most software-managed buffer caches, the eviction information is usually not passed from

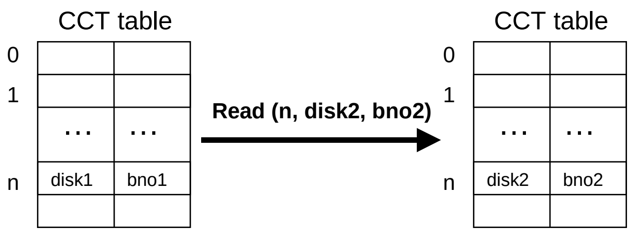


Figure 5: Client Content Tracking Table.

a client to a server. For example, a database buffer cache always silently evicts a clean page and only writes out dirty pages to its back-end storage systems.

Wong and Wilkes [28] have proposed an operation called *DEMOTE* for transferring data ejected from a client buffer cache to a disk array cache. Their approach is similar to the one used in victim caches. Since the current I/O interface between a client and a storage system does not include such an operation, this approach requires modification to client application such as a database server's source code. Therefore, this method is not applicable when the client application source code is not available.

In our study, we use a method that can successfully obtain the client eviction information without any modification to client source code. The main idea is to make use of the buffer address parameter in the I/O read/write interface and build a table to keep track of the contents of the client buffer cache. For example, in a standard I/O interface, a storage I/O read/write call passes at least the following input arguments: disk ID, disk offset, length and buffer address. The buffer address parameter indicates the virtual memory address to store/load the data.

Each entry in the client content tracking (CCT) table records the current disk block $(diskID, blockNo)$ that resides in each memory location of the client buffer cache. The size of the content table is extensible, i.e., it can grow or shrink dynamically based on the buffer addresses it has seen. Since only 16 bytes are needed for each cache block (of size 8 KBytes in our experiments), the content table does not require too much memory space. For example, if a client uses a 4 GBytes buffer cache, the total memory space needed for a CCT is only 8 MBytes, thereby imposing memory overhead of only 0.2%.

Figure 5 shows a CCT table and how it changes after a read request from a client application. At every read/write operation, CCT is consulted to find out which disk block was previously put in the given client memory address. If the old disk block is different from the currently accessed disk block, the old disk block must have been evicted from the client to make space for the new block. Then this eviction information is passed to the storage system. The corresponding CCT entry is modified to point to the currently accessed disk block.

There are two possible places in an I/O subsystem to implement the CCT table: the client side and the storage server side. In our study, we decided to implement it on the client side because it is easier to support clients that use multiple independent storage systems. More specifically, we implement the CCT table in a filter device driver. Since every I/O operation needs to pass through this filter driver, the CCT table can accurately keep track of client buffer cache content. The filter driver can pass eviction information (block numbers) to a corresponding storage node via piggy-backing on read/write messages to that node. Since the driver controls every read/write messages to the storage nodes, it can always find a message to the corresponding node in the send queue to bundle with the eviction information. In this way, no additional message is needed. Because the eviction information is just a few bytes, the additional delay is negligible.

5 Reducing Reload Overhead

The second challenge with the eviction-based placement is to reduce the reload overhead. Since a block's placement into a storage cache is postponed from its access time to the time when it is evicted from the client, the block needs to be reloaded from either clients or disks. As a result, it can increase the network or disk traffic, which can significantly offset the benefits of improved cache hit ratios of the eviction-based placement policy.

The *DEMOTE* mechanism proposed in [28] relies on clients to send an evicted block back to storage systems, even if the block is not dirty. Besides the burden on developers to modify the client software, this method also introduces three performance overheads, which may cancel out the benefits of exclusive caching for some workloads.

- Increased network traffic. *DEMOTE* operations can significantly increase the network traffic from clients to storage systems. As we know, most of the client buffer caches (for example database server buffers) usually try to evict clean pages first before evicting any dirty pages to avoid extra disk write-backs and consistency operations (undo-log logging). In an

OLTP workload, the read traffic is usually 2-3 times larger than the write traffic. If every read request to the storage cache incurs a *DEMOTE* operation, the resulting client-to-server traffic is almost doubled. In a system where the client-storage network is a bottleneck, the *DEMOTE* operations can significantly degrade the system throughput. This has also been pointed out as a limitation of the *DEMOTE* method by the authors themselves [28]. Our implementation results on a storage system also validate this limitation.

- Increased access time. When the buffer cache misses on a client are too bursty to mask the *DEMOTE* overheads, a currently missed block in a client buffer cache may have to wait for a *DEMOTE* operation to finish in order to get a free buffer block before sending a read request to the storage server. Consequently, the average access time will increase in such a case. For example, suppose an application repeats reading sequential blocks from 0 to n in a loop as in a table join operation, where n is larger than the number of blocks in a client buffer cache. Every access would be delayed because it needs to wait for a free block, which is only available after an evicted clean block is sent back to the storage cache.

- Limited flexibility for optimizations. Since a client buffer cache evicts a clean block to make space for a new block, the evicted block needs to be sent to the storage cache before being replaced. Due to this constraint, the time window to demote a block to the storage cache is very short, not enough to perform any effective scheduling or batching optimizations.

In our study, we propose to reload (prefetch) evicted blocks from disks to a storage cache. The first motivation for taking this approach is that the disk bandwidth is usually less utilized than storage area network bandwidth because real-world configurations typically put many disks (for example 60-100 SCSI disks) in a storage server[30]. With an average seek time of 5-6 ms, a modern SCSI hard drive can provide over 1MBps bandwidth for a traffic of random 8-KByte block accesses. Thus, without any caching at the storage server, a medium disk array, say 100 disks, can readily saturate a 1Gbps client-storage interconnection. Moreover, a storage server cache can also filter some of the data access traffic. For instance, if a storage cache has a hit ratio of 50%, only half of the network traffic will go to disks. In this case, using 50 disks per array can saturate a 1Gbps client-storage interconnection. On the other hand, in some environment where the SAN bandwidth is larger than the aggregate disk bandwidth, *DEMOTE* can be a better alternative to relieve the bottleneck of the disks. The second motivation is to avoid delaying demand requests on clients. By pushing reloads to storage systems, client demand requests can proceed without interference by any *DEMOTE* operations.

The third motivation is that one can easily reduce reloading overheads using the following two methods:

(1) Eliminating unnecessary reloads. Many reloads in the eviction-based placement are unnecessary. In most cache studies, the rule of thumb is that a large percentage of accesses are made to a small percentage of blocks. This means that most of the blocks (*cold blocks*) are accessed only once or twice in a long period of time. When these blocks are evicted from a client buffer cache, it is unnecessary to reload them from disks. Reloading these blocks can actually degrade the storage cache hit ratios because they can pollute a storage cache. Unfortunately, information on future accesses is usually not available in real systems. In our implementations, we speculate about cold blocks based on the number of previous accesses. In other words, our storage cache does not reload blocks that have been accessed fewer than the *reload_threshold* number of times. This is based on the observation that frequently accessed blocks are more likely to be accessed again in a near future. Many other previous studies [20, 13, 15, 31] were also based on this observation.

(2) Masking reload overheads through disk scheduling. To avoid reloads delaying demand disk requests, we give higher priority to demand accesses and lower priority to reloads. We treat reloads in a similar way to prefetching hints since it is perfectly OK if a reload operation is not performed. Given such flexibility, our storage system puts reload operations in a separate task queue and only issues them when there is no ongoing demand request competing for the same disk. Many previous work such as Freeblock scheduling [17] and other scheduling algorithms [8, 23, 11, 29, 2] can easily apply here to mask reload overheads. For example, the reload overheads can be hidden using the Freeblock scheduling that exploits the free bandwidth of disk rotational delay.

6 Evaluation on Real Systems

We implement the eviction-based placement in a storage system using a commercial database server (Microsoft SQL server) as a storage client. The evaluation is conducted using real world OLTP workloads. The goal of our experiments is to answer the following questions.

- How much can the eviction-based placement improve cache hit ratios in real systems?
- What is the overall impact of eviction-based placement on the application performance?
- What are the effects of optimizations for reducing reload overheads?
- What are the tradeoffs between our method and the *DEMOTE* approach [28]?

In this section, we first briefly describe our experimental platform. We then present the performance results, discuss the effects of optimizations and compare our method with the *DEMOTE* method.

6.1 Experimental Platform

We conduct our experiments in a configuration similar to our previous experiments [30]. It consists of three PCs, each of which has dual 933MHz Pentium III Coppermine processors with 256 KBytes L2 cache and 1 GBytes main memory. One PC runs the storage server software, one runs Microsoft SQL server 2000 Enterprise edition, and the last one runs a TPC-C benchmark engine [16] that sends transaction requests to the Microsoft SQL server. The TPC-C benchmark is provided by Microsoft. All PCs use Windows 2000 Advanced Server as operating systems. The TPC-C benchmark requires restoring the database to its initial state before each run to avoid performance discrepancy caused by enlarged database sizes from previous runs. To shorten our experiment execution time, we shrink the number of TPC-C warehouses to 10. The Microsoft SQL server cache size is configured to be 256 MBytes. We run the TPC-C benchmark for 30 minutes in each experiment.

The storage server connects to the database server via a Virtual Interface (VI) network [26] provided by Emulex cLAN network cards. The peak VI bandwidth is about 113 MBps and the one-way latency for a short message is 5.5 μs. The storage server machine has five Ultra66 IDE disks. The total storage capacity is 200 GBytes. The storage buffer cache size is configured to be 256 Mbytes. The storage system employs a write-through cache policy. We have implemented both MQ and LRU as the storage cache replacement algorithms. The parameters of the MQ algorithm are set according to our previous study [31]. Our previous study [30] also gives detailed description of the architecture.

6.2 Results Overview

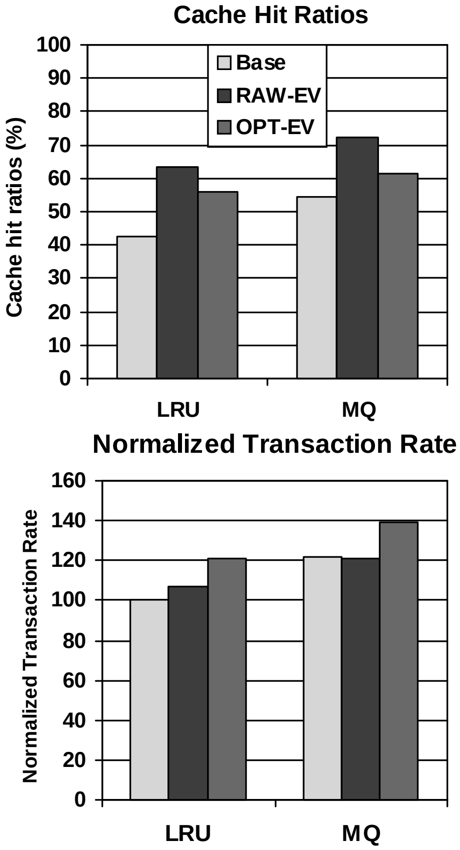


Figure 6: Storage cache hit ratios and normalized transaction rates. All transaction rates are normalized to the ones achieved using the access-based placement and LRU replacement for the storage cache.

Figure 6 compares the storage cache hit ratios and normalized transaction rates for the access-based and the eviction-based placements. We present the results for both LRU and MQ replacements. In these two figures, the base means the access-based placement; RAW-EV means the eviction-based placement without any optimizations; OPT-EV means the eviction-based placement with optimizations to reduce reload overhead.

The raw eviction-based placement has the highest storage cache hit ratios. EV can improve LRU's hit

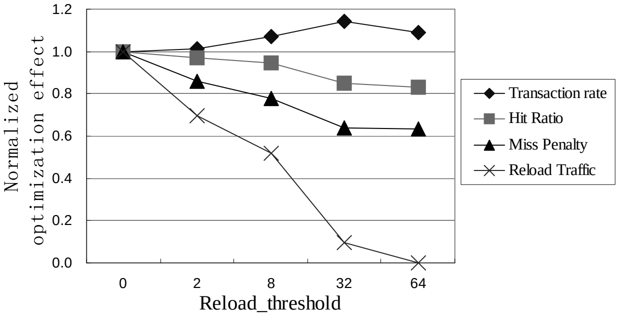


Figure 7: Effects of optimizations for reducing reload overhead.

ratio by a factor of 1.49, and MQ by a factor of 1.32. Similar to the simulation results, EV's improvement on storage cache hit ratios is more pronounced for LRU than for MQ because MQ can better tolerate long idle distances by selectively keeping frequently accessed blocks in a storage cache for a longer time.

Unfortunately, RAW-EV's substantial improvement on cache hit ratios does not fully translate into TPC-C performance. For example, for LRU, RAW-EV only outperforms the access-based placement by 7%. For MQ, RAW-EV does not have any improvement at all. The main reason is the high overheads for reloading data from disks. For reasons explained in Section 5, the reload overheads significantly offset the benefit of improved cache hit ratios. For MQ, the overheads are so large that they totally cancel out the 32% improvement in cache hit ratios.

However, after reducing the reload overheads by eliminating unnecessary reloads and prioritizing demand requests over reloads, the optimized EV can achieve much higher transaction rates. For example, for LRU, EV improves the transaction rate of the base case by a factor of 1.21. For MQ, EV has a speedup of 1.13 over the access-based placement. The effects of optimizations are discussed in detail in the next subsection.

6.3 Effects of Optimizations

To understand the effects of optimizations for reducing reload overheads, we have examined the impact of these optimizations on cache hit ratios, average response time (average miss penalty) of demand disk requests, reload traffic and application transaction rate by varying the *reload_threshold* value. Figure 7 plot these impacts for LRU. All numbers are, respectively, normalized to the ones achieved using RAW-EV. For example, when the *reload_threshold* is 32, the reload traffic is substantially reduced by OPT-EV to only 0.1 of that with RAW-EV. As a result, the miss penalty decreases to 0.65 of RAW-EV's. Unfortunately, OPT-EV also has a lower cache hit ratio, 0.82 of RAW-EV. Overall, the transaction rate with the optimized verson has a factor of 1.13 improvement over RAW-EV when the *reload_threshold* is 32.

When the *reload_threshold* value increases, the number of reloads is significantly reduced, leading to less contention on disks. As a result, the average disk response time for demand requests also decreases. For example, by simply eliminating reloads of blocks that have been previously accessed only once (*reload_threshold* is 2), the reload traffic is reduced by 31%, and the average miss penalty for demand requests is reduce by 14%. The impact on miss penalty is less because some of reloads in RAW-EV are performed when disks are idle, as a result of priority-based scheduling.

However, reducing the number of reloads also has a negative impact. It decreases storage cache hit ratios. For example, increasing the *reload_threshold* value from 0 to 64, the storage cache hit ratio is reduced by 15%. Combining the gain (decrease in disk traffic) and the loss (decrease in cache hit ratios) into the formula: $AverageAccessTime = HitTime * HitRatio + MissPenalty * (1 - HitRatio)$, the impact on application performance varies. The performance peaks when the threshold value is equal to 32.

Notice that our results can be further improved if a more sophisticated priority-based disk scheduling algorithm such as Freeblock scheduling [17] is used to mask reload overheads. We expect the performance improvement with such scheduling algorithm should be similar to the improvement in storage cache hit ratios.

6.4 Comparison with DEMOTE

We also evaluate the tradeoffs between of our method and the *DEMOTE* approach [28]. To do this, we also implement the *DEMOTE* operation in our system. When a clean block is evicted from the storage client (Microsoft SQL server buffer cache in our configuration), the filter driver sends ("demotes") this block to the storage server. Since the database working set size is relative small, we vary the database-storage network bandwidth in a range from 40MB/s to 113MB/s. Since the VI network in our platform can provide 113 MB/s user-to-user bandwidth, we have to run a simple ping-pong VI test program on the side to generate network traffic to utilize 1/3 or 2/3 of the VI bandwidth. The test

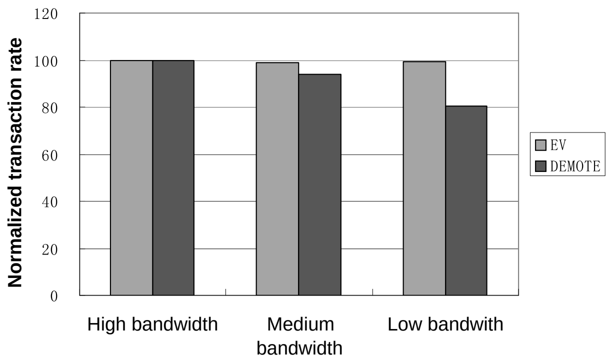


Figure 8: Normalized transaction rates of the EV and *DEMOTE* placement policies under three different configurations of network bandwidth. All transaction rates are normalized to their corresponding rate with 113MB/s network bandwidth.

program is very simple and introduces little processor overhead.

Figure 8 compares the performance of *DEMOTE* and EV under three different configurations of network bandwidth. When the available client-storage network bandwidth is high compared to the client workloads, *DEMOTE* and EV perform similarly. However, when the network bandwidth is low, EV outperforms *DEMOTE* by 20%, even though both approaches have similar cache hit ratios. This is because EV does not impose extra network traffic, whereas *DEMOTE* can potentially double the traffic. This result of *DEMOTE* in a real system matches the simulation result. These results indicate that EV would be a better alternative when the client-storage network has limited bandwidth.

7 Related Work

A large body of literature has examined the cache management problem. However, most previous work has focused on varying cache replacement policy with a fixed placement policy—access-based placement policy. The classic buffer replacement algorithms include the Least Recently Used (LRU) [9, 5], First in First Out (FIFO), Most Recently Used (MRU) and Least Frequently Used (LFU). Recently proposed algorithms include FBR [22], LRU-k [20], 2Q [13], LFRU [15], MQ [31], LIRS [12], and DEMOTE [28], just to name a few. These algorithms have shown performance improvement over the widely used LRU algorithm for evaluated workloads. In the spectrum of off-line algorithms, Belady's OPT algorithm and WORST algorithm [3, 18] are widely used to derive a lower and upper bound on the cache miss ratio for replacement algorithms.

Our work is motivated by previous studies [7, 10, 19, 31, 27, 28, 4]. Dan, Dias and Yu conducted a theoretical analysis of hierarchical buffering in a shared database environment [7]. Franklin, Carey and Livny also explored global memory management in database systems [10]. Muntz and Honeyman investigated multi-level caching in a distributed file system, showing that server caches have poor hit ratios [19]. Willick, Eager and Bunt have demonstrated that the Frequency Based Replacement (FBR) algorithm performs better for file server caches than locality based replacement algorithms such as LRU [27]. Cao and Irani showed that GreedyDualSize replacement algorithm performs better than other known policies for a web cache [4]. Storage caches had been shown to exhibit different access pattern and thereby should be managed differently from other single level buffer caches [31].

The eviction-based placement was first proposed in the hardware victim cache work [14] for hardware processor caches. However, few software managed buffer caches have used eviction-based placement because upper level buffer caches usually do not provide any eviction information to lower level caches. Wong and Wilkes proposed a *DEMOTE* operation to transfer data evicted from the client buffer to the storage cache [28]. This work has made a very good observation, i.e., storage caches should be made exclusive. Their simulation evaluation has shown promising results. But their approach has some limitations as described in previous sections. This study generalizes their approach and proposes alternatives to address those limitations. Moreover, we also evaluate the *DEMOTE* method in a storage system.

Many past studies have used metrics such as LRU stack distance [18], marginal distribution of stack distances [1], distance string models [24], interreference gap (IRG) model [21] or temporal distance distribution [31] to analyze the temporal locality of programs. In our study, we use idle distance distributions to measure the effects of cache placement policies.

8 Conclusions

This paper presents an eviction-based cache placement policy to manage storage caches. This placement policy puts a block into a storage cache when it is evicted from a client buffer cache. We have also described a method of using a client content tracking table to obtain eviction information from client buffer caches without modifying client applications. To reduce the reloading overheads introduced by the eviction-based placement, we have discussed two techniques, eliminating unnecessary reloads and masking reloads using priority-based disk scheduling.

Our simulation results of real-world workloads show that the eviction-based cache placement has 10% to 500% higher cache hit ratios than the access-based placement policy for four different cache replacement algorithms. Our implementation results on a storage system connected to Microsoft SQL server with OLTP workloads have demonstrated that the eviction-based cache placement can improve the application transaction rate by 20%. We also compare our method with *DEMOTE* in a storage system. Our implementation results show that our method has a 20% higher transaction rate than the *DEMOTE* method when the client-storage network has limited bandwidth.

This paper has several limitations. First, we have only used some simple techniques to reduce reloading overheads. We are currently implementing the Freeblock scheduling [17] to mask reloading overheads. Second, we have not done theoretical analysis on the eviction-based placement policy. Some theoretical analysis would be useful to better understand the characteristics of different cache placement policies. Third, we have studied only two types of storage workloads: one is a database OLTP workload and the other is a file system workload. It is interesting to see how well the eviction-based placement would work for other workloads. Even though this paper focuses on storage cache management, the techniques presented in this paper can easily apply to other lower level buffer cache management.

9 Acknowledgement

The authors are grateful to our shepherd Marvin Theimer for his invaluable feedback in preparing the final version of the paper. We would also like to thank anonymous reviewers for their detailed comments.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, Dec. 1996.

[2] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 55–65. ACM Press, 2002.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

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

[6] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 150–160, 1994.

[7] A. Dan, D. M. Dias, and P. S. Yu. Analytical modelling of a hierarchical buffer for a data sharing environment. *ACM SIGMETRICS Performance Evaluation Review*, 19(1):156–167, May 1991.

[8] P. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joint Computer Conference*, volume 31, pages 9–21, Washington, D.C., 1967. Thompson Book Co.

[9] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[10] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *International Conference On Very Large Data Bases (VLDB '92)*, pages 596–609, San Mateo, Ca., USA, Aug. 1992. Morgan Kaufmann Publishers, Inc.

[11] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard Company, February 1991.

[12] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the international conference on Measurement and modeling of computer systems*, pages 31–42. ACM Press, 2002.

[13] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 439–450, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–373. ACM Press, May 1990.

[15] D. Lee, J. Choi, J.-H. Kim, S. L. Min, Y. Cho, C. S. Kim, and S. H. Noh. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 134–143, New York, May 1–4 1999. ACM Press.

[16] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):22–31, June 1993.

[17] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *Symposium on Operating Systems Design and Implementation*, 2000.

[18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[19] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 305–314, Berkeley, CA, USA, Jan. 1991. Usenix Association.

[20] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, Washington, D.C., 26–28 May 1993.

[21] B. G. V. Phalke. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 291–300, New York, NY, USA, May 1995. ACM Press.

[22] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142. ACM Press, 1990.

[23] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.

[24] J. R. Spirn. Distance string models for program behavior. *Computer*, 9(11):14–20, Nov. 1976.

[25] Transaction Processing Performance Council. *TPC Benchmark C*. Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, CA 95112-6311, May 1991.

[26] Virtual interface architecture specication version 1.0. VI-Architecture Organization, 1997.

[27] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *13th International Conference on Distributed Computing Systems*, Pittsburgh Hilton, Pittsburgh, PA, May 1993. IEEE. University of Saskatchewan, Canada.

[28] 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.

[29] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251. ACM Press, 1994.

[30] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI communication for database storage. In *the 29th International Symposium on Computer Architecture (ISCA)*, May 2002.

[31] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Annual Technical Conference*, June 2001.