Second-Level Buffer Cache Management

Yuanyuan Zhou, *Member*, *IEEE*, Zhifeng Chen, and Kai Li, *Senior Member*, *IEEE*

Abstract—Buffer caches are commonly used in servers to reduce the number of slow disk accesses or network messages. These buffer caches form a multilevel buffer cache hierarchy. In such a hierarchy, second-level buffer caches have different access patterns from first-level buffer caches because accesses to a second-level are actually *misses* from a first-level. Therefore, commonly used cache management algorithms such as the Least Recently Used (LRU) replacement algorithm that work well for single-level buffer caches may not work well for second-level. This paper investigates multiple approaches to effectively manage second-level buffer caches. In particular, it reports our research results in 1) second-level buffer cache access pattern characterization, 2) a new local algorithm called Multi-Queue (MQ) that performs better than nine tested alternative algorithms for second-level buffer caches, 3) a set of global algorithms that manage a multilevel buffer cache hierarchy globally and significantly improve second-level buffer cache hit ratios over corresponding local algorithms, and 4) implementation and evaluation of these algorithms in a real storage system connected with commercial database servers (Microsoft SQL Server and Oracle) running industrial-strength online transaction processing benchmarks.

Index Terms—Cache memories, storage hierarchy, storage management.

1 Introduction

Technology trends have continued to increase the access-time gap between processor and disk. To bridge the gap, main-memory buffer caches are used along data retrieving paths to avoid slow disk accesses. Besides client buffer caches, requested data can also be cached at a Web browser, a Web proxy server, a Web server, an application server, a database server, and a file server. In addition, many modern storage systems also use large volatile or nonvolatile memory as buffer caches to speed-up I/O accesses [14], [18]. For example, the EMC Symmetrix Storage System has up to 256 Gbytes large software-managed buffer caches [14].

These buffer caches form a multilevel buffer cache hierarchy. Fig. 1 shows an example of such a hierarchy. In this example, a database server buffer cache or a file server buffer cache serves as a first-level (L1) buffer cache, whereas a storage server buffer cache serves as a second-level (L2) buffer cache. Note that L1/L2 buffer caches are very *different* from L1/L2 processor caches, which are usually set-associative and managed by hardware, whereas L1/L2 buffer caches are main-memory buffers distributed in multiple machines and managed by different software.

Second-level buffer caches such as storage caches have different access patterns from single level buffer caches because accesses to an L2 buffer cache are *misses* from an L1 buffer cache. L1 buffer caches, such as database buffer caches, typically employ a locality-based algorithm, such as the Least Recently Used (LRU) replacement algorithm, so that recently accessed blocks are kept in an L1. As a result, accesses to an L2 buffer cache exhibit poorer temporal locality than those to an L1. This implies that a replacement algorithm such as LRU, which works well for L1 buffer caches, may not perform well for L2 buffer caches. Though the aggregate cache size of the hierarchy is increasingly larger, the system might not deliver the expected performance commensurate to the aggregate cache size if these buffer caches could not work together effectively.

Multilevel caching has been previously studied by Muntz and Honeyman [29] in the context of distributed file systems. Their study shows that L2 buffer caches have poor hit ratios. They concluded that the poor hit ratios are due to poor data sharing among clients. This study did not characterize the behavior of accesses to lower-level buffer caches, but raised the question whether the algorithms that work well for client or single level buffer caches can effectively reduce misses for second-level. Another study conducted by Willick et al. demonstrated that the Frequency-Based Replacement (FBR) algorithm performs better for a back-end disk caches than locality-based replacement algorithms such as LRU [47], but this study did not study disk cache access patterns to understand their results.

These studies motivate research to address the following questions:

1. What are the access patterns at an L2 buffer cache? Does it have good temporal locality? How is it different from single level buffer caches?
2. How do recently proposed single-level cache replacement algorithms, such as LRU-k [31], Least Frequently Recently Used (LFRU) [24], Two Queues (2Q) [20], ARC [28], LIRS [19], and DEMOTE [48], perform for second level?
3. If most existing algorithms do not perform well for L2, how do we design a new cache management algorithm to improve L2 buffer cache performance?
4. How do we manage a multilevel buffer cache hierarchy globally? Is a global buffer cache management beneficial compared to local management?

This paper addresses the above problems. More specifically, it has the following contributions:

- *Y. Zhou and Z. Chen are with the Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801. E-mail: {yyzhou, zchen9}@cs.uiuc.edu.*
- *K. Li is with the Department of Computer Science, Princeton University, Princeton, NJ 08540. E-mail: li@cs.princeton.edu.*

Manuscript received 15 Aug. 2003; revised 24 Nov. 2003; accepted 27 Nov. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0139-0803.

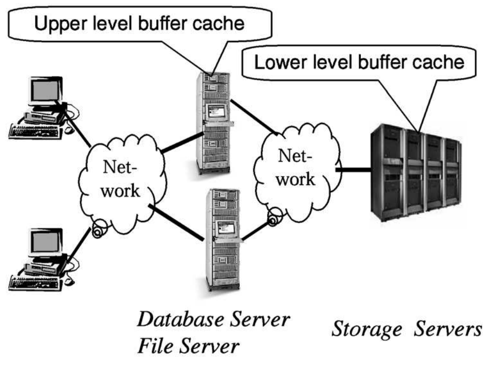

Fig. 1. An example of a multilevel buffer cache hierarchy.

- We have studied L2 buffer cache access patterns using several large real-system traces. The access pattern analysis shows that accesses to an L2 have poor temporal locality and some blocks are accessed more frequently than others.
- We present a new local L2 cache replacement algorithm called *Multi-Queue (MQ)* that outperforms nine tested alternatives including LRU, Most Recently Used (MRU), Least Frequently Used (LFU), FBR [34], LRU-2, LFRU [24], 2Q [20], LIRS [19], and ARC [28].
- We present a set of global algorithms that manage a multilevel buffer cache hierarchy globally and significantly improves corresponding local algorithms for L2 buffer caches. In addition, we have investigated two methods to implement the global algorithm, one of which requires modification to the L1 software and the other does not have such a requirement.
- We implement and evaluate the new algorithms and previous algorithms in a real storage system that is connected to commercial database servers (Microsoft SQL Server and Oracle) running online transaction processing (OLTP) benchmarks. Our implementation results show that MQ can provide similar transaction rate as LRU with a twice as large storage cache (L2 buffer cache). The global algorithms can further improve the transaction rate by 20 percent.

The rest of this paper is organized as follows: The next section briefly describes the background work on cache management. Section 3 describes the L2 buffer cache traces used in this study. Section 4 discusses L2 buffer cache access patterns. Section 5 presents the MQ replacement algorithm and Section 6 presents the global algorithm. Sections 7 and 8 present the simulation and implementation results. Related work is discussed in Section 9. Finally, Section 10 concludes this paper.

2 BACKGROUND

Much research has been conducted on buffer cache management. Most buffer cache replacement algorithms were designed for single-level buffer caches. In our study, we evaluate nine online replacement algorithms originally designed for single-level buffer caches. These algorithms include:

OPT (Optimal) algorithm [5], [27] is an offline replacement algorithm and gives the best possible cache hit ratios assuming demand-based access-policy (data is fetched into the cache when it is accessed). This algorithm assumes that the entire access sequence is available ahead of time and, therefore, cannot be used online.

LRU (Least Recently Used) has been used widely for buffer cache management [7], [12], [11], [39], [2]. When the cache is full, it replaces the block that is the least recently used. It is designed to take advantage of the temporal locality exhibited in accesses.

MRU (Most Recently Used) is also called the Fetch-and-Discard replacement algorithm [7], [12]. Instead of replacing the least recently used block like LRU, MRU replaces the most recently used block. It was originally designed to deal with situations like sequential scans.

LFU (Least Frequently Used) replaces the block that is least frequently used. The *frequency* of a block is its reference count. The frequency reflects the approximated probability of the block to be accessed again in the future. The "aged" version of LFU performs better than the original LFU because recent access history predicts more precisely future access patterns.

FBR (Frequency Based Replacement) algorithm was proposed by Robinson and Devarakonda [34]. It considers both recency and frequency to capture the benefit of both LRU and LFU. It divides LRU queue into three sections: new, middle, and old. It does not increment reference counts in the new section and replaces least frequent blocks in the old section. This algorithm requires parameter tuning to adjust the section sizes. So far, no online adaptive scheme has been proposed to dynamically tune these parameters.

LRU-k (Least kth-to-last Reference) algorithm was first proposed by O'Neil et al. for database systems [31]. It replaces the block with the least recent Kth-to-last access. LRU-1 is same as the classical LRU. When K is large, it discriminates the frequent and infrequent blocks. When K is small, it removes cold blocks (blocks accessed only once) quickly.

LFRU (Least Frequently Recently Used) algorithm was proposed by Lee et al. in 1999 to cover a spectrum of replacement algorithms that include LRU at one end and LFU at the other [24]. It endeavors to replace blocks that are the least frequently used and not recently used. It associates a value, called Combined Recency and Frequency (CRF), with each block. It replaces the block with the minimum CRF value. LFRU can implement LRU and LFU with different parameters. It also needs parameter tuning and no dynamic scheme has been proposed.

2Q (Two queue) algorithm was proposed by Johnson and Shasha in 1994 [20]. The algorithm utilizes one FIFO queue $A1_{in}$ and two LRU lists, $A1_{out}$ and A_m. It places a block in $A1_{in}$ on the first access and promotes the block to A_m on the second access. It replaces a block in $A1_{in}$ and put the block's identifier in $A1_{out}$ if $A1_{in}$ has more than a fixed number of blocks. Otherwise, it replaces a block in A_m.

LIRS (Low Inter-Reference Recency Set) was proposed by Jiang and Zhang in 2002 [19]. It uses Inter-Reference Recency (IRR) history instead of just access recency for making a replacement decision. Blocks with smaller IRR values are favored than those with larger IRR values.

TABLE 1
Characteristics of the Six Traces Used in the Study

	L1 Buffer Cache Size (MB)	# Reads (millions)	# Writes (millions)	# First Level
Oracle Miss Trace-128M	128	7.3	4.3	single
Oracle Miss Trace-16M	16	3.8	2.0	single
MS-SQL-Small	64	4.52	2.5	single
MS-SQL-Large	1024	5.07	3.0	single
HP Disk Trace	30	0.2	0.3	multiple
Auspex Server Trace	8 per client	1.8	0.8	multiple

Oracle Miss Trace-128M has more reads/writes than Oracle Miss Trace-16M because the former executes more transactions. Both traces are collected by running the TPC-C benchmark for two hours. Similarly, MS-SQL-Large has more reads/writes than MS-SQL-Small.

ARC (Adaptive Replacement Cache) was proposed in 2003 [28]. It combines recency and frequency by using two lists and dynamically adjusts their sizes depending which factor is more important.

3 Traces

To study L2 buffer cache access patterns and evaluate caching algorithms and policies, we use six L2 buffer cache access traces. These traces are chosen to represent different types of workloads. All traces contained only misses from one or multiple L1 buffer caches that use LRU or its variations as their replacement algorithms. We collect OLTP I/O traces from both Oracle and Microsoft SQL Server because these two database servers are very different internally, using different caching schemes and software architecture. In our study, we use 8 Kbytes as the cache block size for our access pattern analysis and our experimental evaluation of various algorithms. We have examined other block sizes and the results are similar.

Table 1 shows the characteristics of the six traces. Since L1 buffer cache sizes clearly affect an L2 cache's performance, we set the L1 buffer cache (database server cache) sizes for the two Oracle traces and two MS-SQL traces to represent typical configurations in real systems. However, we could not change the first level buffer cache sizes of the other two traces because they were obtained from other sources.

Oracle Miss Trace-128M and **Oracle Miss Trace-16M** are collected from a storage server connecting to an Oracle 8i database front-end running the standard TPC-C benchmark [44], [25] for about two hours. The Oracle buffer cache replacement algorithm is similar to LRU [32]. The TPC-C database contains 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The traces capture all I/O accesses from the Oracle Server to the storage server. The traces ignore all accesses to log disks. The first trace is collected by setting the Oracle buffer cache to be 128 MBytes, whereas the latter is collected with 16 MBytes of Oracle buffer cache.

MS-SQL-Large and **MS-SQL-Small** are collected on a different hardware platform running Microsoft SQL Server 2000. Similar to Oracle traces, the SQL Server runs TPC-C benchmark with 256 warehouses. In MS-SQL-Large, we set the SQL Server cache size to the maximum available amount of the platform, 1 Gbyte. To predict results for larger workloads, we reduce the Microsoft SQL-server cache size to 64 MBytes and collect another trace, MS-SQL-Small.

HP Disk Trace was collected at Hewlett-Packard Laboratories in 1992 [36], [35]. It captured all low-level disk I/O performed by the system. We used the trace gathered on Cello, which is a timesharing system used by a group of researchers at HP Lab to do simulations, compilation, editing, and e-mail. We have also tried other HP disk trace files and the results are similar.

Auspex Server Trace was an NFS file system activity trace on an Auspex file server in 1993 at UC Berkeley [9]. The system included 237 clients spread over four Ethernets, each of which connected directly to the central server. The trace covers seven days of activities.

4 Second-Level Buffer Cache Access Patterns

4.1 Temporal Locality

We first study the temporal locality of L2 buffer cache accesses. Previous studies have shown that L1 buffer cache accesses exhibit a high degree of temporal locality. An accessed block exhibits temporal locality if it is likely to be accessed again in the near future. The LRU replacement algorithm, typically used in L1 buffer caches, is designed to take advantage of temporal locality. Thus, blocks with a high degree of temporal locality are likely to remain in an L1 buffer cache. But, accesses to an L2 level buffer cache are misses from an L1. Do second-level buffer cache accesses exhibit temporal locality similar to those of an L1 buffer cache?

We use *reuse distance* histograms to observe the temporal locality of the traces. A *reference sequence* (or *reference string*) is a numbered sequence of temporally ordered accesses to a cache. A trace is essentially such a reference sequence. The *reuse distance* is the number of distinct accesses between two accesses to the same block in the reference sequence. It is similar to the interreference gap defined by a previous study [33]. For example, in the reference sequence *ABCDBAX*, the reuse distance from A_1 to A_6 is 4 and the reuse distance from B_2 to B_5 is 3. A reuse distance histogram shows the number of *correlated accesses* (accesses to the same block) for various reuse distances.

Fig. 2 compares an L1 and L2 buffer cache's temporal locality using reuse distance histograms. The L1 trace is collected at an Auspex client, while the L2 trace is captured at the Auspex Server. Each Auspex client uses an 8 MByte cache. The data in the figure shows the histograms by grouping reuse distances by powers of two. Distances that are not power of two are rounded up to the nearest power of two. Significantly, for the L1 buffer cache, 74 percent of the correlated references have a reuse distance less than or equal to 16. This indicates good temporal locality. On the

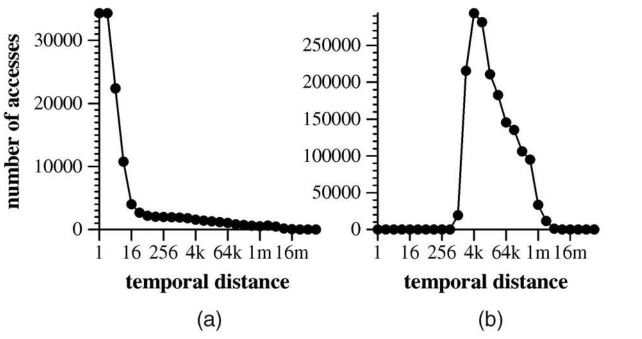

Fig. 2. Temporal locality of L1 and L2 buffer cache accesses using reuse distance histograms. (a) Auspex client trace and (b) Auspex server trace.

contrary, 99 percent of the correlated accesses to the L2 buffer cache have a reuse distance of 512 or greater, exhibiting weaker temporal locality.

Fig. 3 shows the reuse distance histograms of four L2 buffer cache access traces. These traces exhibit two common patterns. First, all histogram curves are hill-shaped. Second, peak reuse distance values, while different, are all relatively large and occur at distances greater than their L1 buffer cache sizes (see Table 1). This access behavior at L2 caches is expectable. If an L1 buffer cache of size k uses a locality-based replacement policy, after a reference to a block, it takes at least k references to evict this block from the L1 buffer cache. Thus, subsequent accesses to the L2 cache should be separated by at least k noncorrelated references in the L2 buffer cache reference sequence. Therefore, the "hill" regions starts after k in Fig. 3.

A good replacement algorithm for L2 caches should retain blocks that reside in the "hill" portion of the histogram for a longer period of time. In this paper, "time" means logical time, measured by the number of references. For example, initially, time is 0; after accesses ABC, time is 3. We call the beginning, peak, and end of this "hill" region *minimal distance* (or $minDist$), *peak distance* (or $peakDist$), and *maximal distance* (or $maxDist$), respectively. $minDist$ depends more on first level buffer cache sizes, whereas $peakDist$ and $maxDist$ depend more on workload characteristics. We picked $minDist$ and $maxDist$ for each trace by examining the histogram figure for simplicity. Since the reuse distance values in the "hill" are relatively large, a good replacement algorithm should keep most blocks in this region for at least $minDist$ time. It is clear that, when the number of blocks in an L2 cache is less than the $minDist$ of a given workload, the LRU replacement algorithm tends to perform poorly, because most blocks do not stay in an L2 buffer cache long enough for subsequent correlated accesses.

4.2 Access Frequency

We have also examined the behavior of L2 buffer cache accesses in terms of frequency. While it is clear that L2 buffer cache accesses represent misses from L1 buffer caches, the distribution of access frequencies among blocks remains uncertain. Past studies [13], [40] have shown that blocks are typically referenced unevenly: A few blocks are hot (frequently accessed), some blocks are warm, and most blocks are cold (infrequently accessed). Is this also true for L2 buffer caches?

Our hypothesis is that both hot and cold blocks will be referenced less frequently in an L2 buffer cache because hot blocks will stay in L1 buffer caches most of the time and cold blocks will be accessed infrequently by definition. If this hypothesis is true, the access frequency distributions at L2 caches should be uneven, though probably not as uneven as those at L1 buffer caches. A good L2 buffer cache replacement algorithm should be able to identify warm blocks and keep them in L2 caches for a longer period of time than others.

In order to understand the frequency distributions of reference sequences seen at L2 buffer caches, we examined the relationship between access distribution and block distribution for different frequencies. Similar to most cache studies, frequency here means the number of accesses. Fig. 4 shows, for a given frequency f, the percentage of total number of blocks accessed at least f times. It also shows the percentage of total accesses to those types of blocks. Notice that the number of blocks accessed at least i times includes blocks accessed at least j times ($j > i$). This explains why all the curves always decrease gradually. The access percentage curves decrease similarly for the same reason.

For all four traces, the access percentage curves decrease more slowly than the block percentage curves, indicating that a large percentage of accesses are to a small percentage of blocks. For example, in the Oracle Miss Trace-128M, around 60 percent accesses are made to less than 10 percent of blocks, each of which are accessed at least 16 times. This shows that the access frequency distribution among blocks at L2 buffer caches is uneven. In other words, a subset of blocks is accessed more frequently than others. Thus, if the

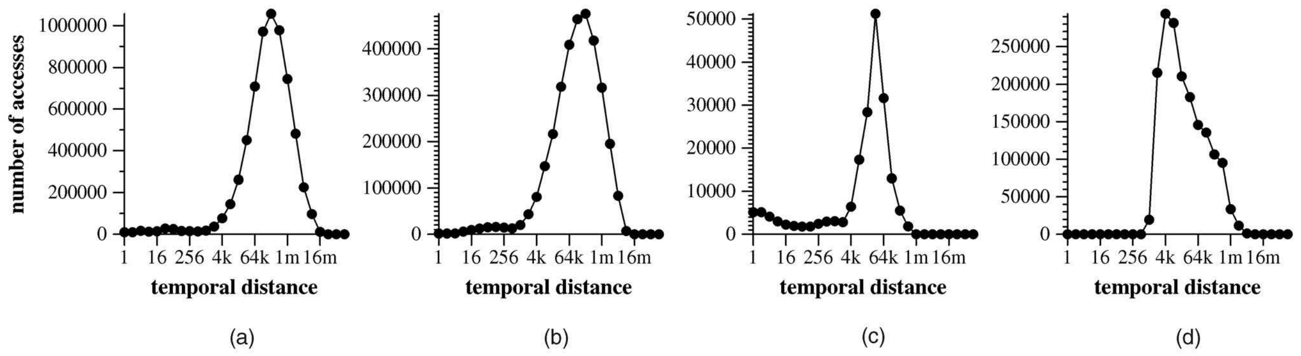

Fig. 3. Reuse distance histograms of L2 cache accesses for different traces. (Note: figures are in different scales.) (a) Oracle Miss Trace-128M, (b) Oracle Miss Trace-16M, (c) HP Disk Trace, and (d) Auspex server trace.

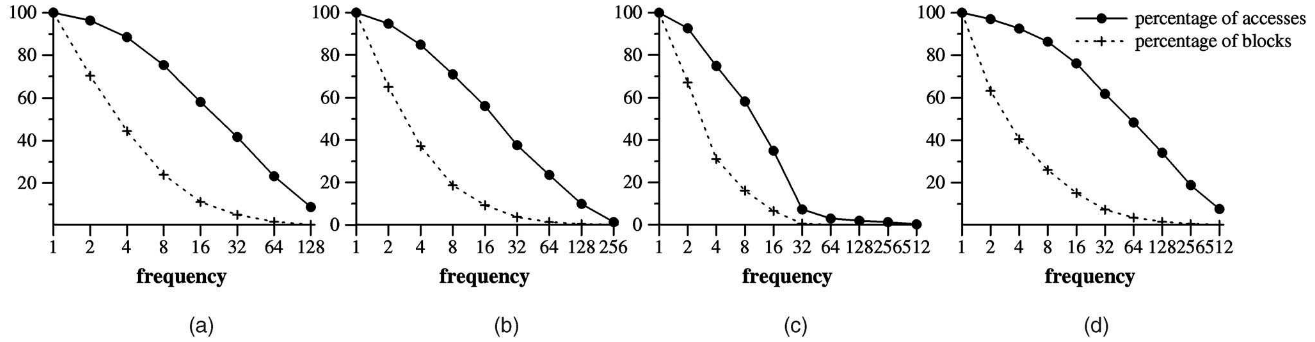

Fig. 4. Access and block distribution among different frequencies. A point (f, p_1) on the block percentage curve indicates that p_1 percent of the total number of blocks are accessed **at least** f times, while a point (f, p_2) on the access percentage curve represents that p_2 percent of the total number of accesses are to blocks accessed at least f times. (a) Oracle Miss Trace-128M, (b) Oracle Miss Trace-16M, (c) HP Disk Trace, and (d) Auspex server trace.

replacement algorithm can selectively keep those blocks for a long period of time, it will significantly reduce the number of misses, especially when the L2 buffer cache size is small.

Blocks that are accessed more frequently contribute more accesses and, therefore, should be given higher priorities. For example, in Fig. 4a, 70 percent of the blocks have been accessed at least twice and contribute to 95 percent of the accesses. However, only 40 percent of the blocks have been accessed at least four times, but contribute to 90 percent of the accesses. Therefore, there is a need to differentiate blocks accessed at least four times from those accessed only two or three times. In other words, the replacement algorithm needs to give different priorities for blocks with different frequencies.

5 Local L2 Buffer Cache Replacement Algorithm: Multi-Queue

Based on the special access patterns at L2 buffer caches, we design a new replacement algorithm, called Multi-Queue (MQ). This is a local cache replacement algorithm because it manages an L2 buffer cache without any information from first-level. The advantage with a local cache management scheme is that it does not require any modification to first-level software such as database servers (Fig. 1).

The main idea of this algorithm is to maintain blocks with different access frequencies for different periods of time in an L2 buffer cache such as a storage cache. To achieve this, it uses multiple LRU queues: $Q_0, \ldots, Q_{m-1}$, where m is a tunable parameter. Blocks in Q_j have a longer lifetime in the buffer cache than those in Q_i $(i < j)$. MQ also uses a history buffer Q_{out}, similarly to the 2Q algorithm [20], to remember access frequencies of recently evicted blocks for some period of time. Q_{out} only keeps block identifiers and their access frequencies. It is a FIFO queue of limited size.

On a cache hit to block b, b is first removed from the current LRU queue and then put at the tail of queue Q_k according to b's current access frequency. In other words, k is a function of the access frequency, $QueueNum(f)$. For example, for a given frequency f, $QueueNum(f)$ can be defined as $log_2 f$. So, the eighth access to a block that is already in an L2 buffer cache will promote this block from Q_2 to Q_3 according to this $QueueNum(f)$ function.

On a cache miss to block b, MQ evicts the head of the lowest nonempty queue from an L2 buffer cache in order to make room for b, i.e., MQ starts with the head of queue Q_0 when choosing victims for replacement. If Q_0 is empty, then MQ evicts the head block of Q_1 and so on. If block c is the victim, its identifier and current access frequency are inserted into the tail of the history buffer Q_{out}. If Q_{out} is full, the oldest identifier in Q_{out} will be deleted. If the requested block b is in Q_{out}, then it is loaded and its frequency f is set to be the remembered value plus 1 and then b's entry is removed from Q_{out}. If b is not in Q_{out}, it is loaded into the cache and its frequency is set to 1. Finally, block b is inserted into an LRU queue according to the value of $QueueNum(f)$.

MQ ages reference counts by demoting inactive blocks from higher to lower level queues. MQ does this by associating a value called $expireTime$ with each block in an L2 buffer cache. "Time" here refers to logical time, measured by the number of accesses. When a block stays in a queue for longer than a permitted period of time without any access, it is demoted to the next lower queue. This is easy to implement with LRU queues. When a block enters a queue, the block's $expireTime$ is set to be $currentTime + lifeTime$, where $lifeTime$, a tunable parameter, is the time that each block can be kept in a queue without any access. At each access, the $expireTime$ of each queue's head block is checked against the $currentTime$. If the former is less than the latter, it is moved to the tail of the next lower level queue and the block's $expireTime$ is reset. Fig. 5 gives a pseudocode outline for the MQ algorithm.

When m equals 1, the MQ algorithm is the LRU algorithm. When m equals 2, the MQ algorithm and the 2Q algorithm [20] both use two queues and a history buffer. However, MQ uses two LRU queues, while 2Q uses one FIFO and one LRU queue. MQ demotes blocks from Q_1 to Q_0 when their lifetime in Q_1 expires, while 2Q does not make this kind of adjustment. When a block in Q_1 (or A_m) is evicted in the 2Q algorithm, it is not put into the history buffer, whereas it is with MQ.

Like the 2Q algorithm, MQ has a time complexity of $O(1)$ because all queues are implemented using LRU lists and m is usually very small (less than 10). At each access, at most $m - 1$ head blocks are examined for possible demotion. MQ is faster in execution and also much simpler to implement than algorithms like FBR, LFRU, or LRU-K, which have a time complexity close to $O(log_2 n)$ (where n is the number of entries in the cache) and usually require a heap data structure for implementation.

```
/* Procedure to be invoked upon a reference
to block b */
if b is in cache{
   i = b.queue;
   remove b from queue Q[i];
}else{
   victim = EvictBlock();
   if b is in Qout {
      remove b from Qout;
   }else{
      b.reference = 0;
   }
   load b's data into victim's place;
}
b.reference ++;
b.queue = QueueNum(b.reference);
insert b to the tail of queue Q[k];
b.expireTime = currentTime + lifeTime;
Adjust();

EvictBlock(){
   i = the first non-empty queue number;
   victim = head of Q[i];
   remove victim from Q[i];
   if Qout is full
      remove the head from Qout;
   add victim's ID to the tail of Qout;
   return victim;
}

Adjust(){
   currentTime ++;
   for(k=1; k<m; k++){
      c = head of Q[k];
      if(c.expireTime < currentTime){
         move c to the tail of Q[k-1];
         c.expireTime = currentTime + lifeTime;
      }
   }
}
```

Fig. 5. MQ algorithm.

6 Global L2 Buffer Cache Management

The MQ algorithm described in the last section is a local L2 buffer cache replacement algorithm that does not require any information from first level. In this section, we present a global management scheme that exploits information from L1 buffer caches to effectively manage an L2 buffer cache. We also describe two approaches to implement this global scheme, one of which requires modification to first-level software and the other does not require such modification, but has less accurate information about first-level.

6.1 Main Idea

Intuitively, a multilevel buffer cache heirarchy is less efficient than a unified, single-level buffer cache with the same aggregate buffer cache size. For example, if an L1 buffer cache has S_1 blocks and an L2 buffer cache has S_2 blocks, the minimum number of misses from the two-level buffer cache hierarchy cannot be smaller than that from a unified single-level buffer cache with $S_1 + S_2$ blocks. This observation can be easily proved by contradiction.

Based on this observation, we propose a global cache management scheme to manage a multilevel hierarchy collaboratively. We use the LRU algorithm as an example even though this global scheme can also apply to other replacement algorithms. For simplicity, we first assume that there is only one L1 buffer cache and one L2 buffer cache. Section 6.3 will discuss how to extend it to multiple L1s and L2s.

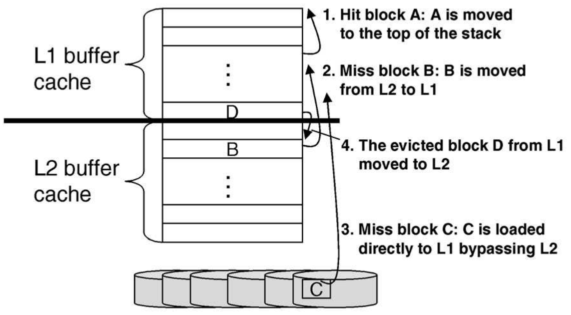
L1 buffer cache
L2 buffer cache
D
B
C
1. Hit block A: A is moved to the top of the stack
2. Miss block B: B is moved from L2 to L1
4. The evicted block D from L1 moved to L2
3. Miss block C: C is loaded directly to L1 bypassing L2

Fig. 6. Global LRU algorithm.

Fig. 6 shows a global LRU stack. The top of the LRU stack records blocks that are most recently used and the bottom of the stack records blocks least recently used. The top S_1 entries physically reside in the L1 buffer cache and the bottom S_2 entries reside in the L2 buffer cache. The global LRU algorithm works as follows:

- **Operation 1**: A hit to the L1 buffer cache is handled in the same way as the local algorithm: The accessed block is moved to the top of the global LRU stack.
- **Operation 2**: At a miss at L1, if the missed block is in the L2 buffer cache (lower half of the global LRU stack), the block is "moved" from L2 to L1 and is deleted in L2.
- **Operation 3**: If the missed block is not in L2, the block is loaded directly from disks to L1 bypassing L2. The bypassing process can be implemented using a temporary buffer at the second level.
- **Operation 4**: When L1 evicts a block, the evicted block is "shifted" to the top of L2's LRU stack partition.

With these operations, the two-level buffer cache hierarchy behaves in the same way as a local LRU algorithm managing a large single-level buffer cache. By following the same principle, this global LRU algorithm can be easily extended from two-level to multilevel.

The above four operations can also integrate with other replacement algorithms. Basically, the replacement decision can be made independently in an L1 and L2 using their local algorithms. To "globalize" an algorithm, one can simply add the four operations described above.

6.2 Design Issues

Two challenging issues need to be addressed for the global algorithm to be used in real systems. The first issue is to obtain replacement (eviction) information from L1 buffer caches in order to perform Operation 4. The second issue is where L2 should load blocks that are evicted from L1. This section discusses these two design issues and the trade offs between different solutions.

6.2.1 How to Obtain First-Level Replacement Information

In most software-managed buffer caches, the replacement (eviction) information is usually not passed from an upper level to a lower level. For example, a database buffer cache always silently evicts a clean page and only writes out dirty pages to its back-end storage systems.

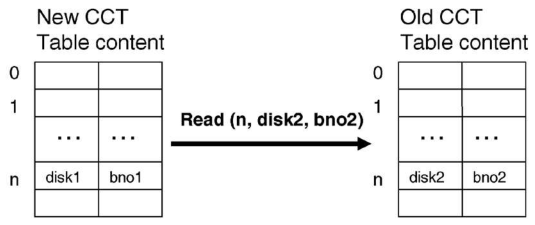

Fig. 7. Client Content Tracking (CCT) table (notation: (*disk*1, *bno*1) uniquely identifies a recently evicted block).

There are two approaches to obtain an L1's replacement information. The first approach is to modify an L1 software to pass the eviction information to an L2 software. In the example of database-storage buffer cache hierarchy, this approach requires modification to the database server source code to pass its buffer cache eviction information to the storage system using either explicit messages or pigging-back on ordinary requests. It is very similar to the DEMOTE approach investigated by Wong and Wilkes in their disk cache study [48]. The advantage with this approach is that an L2 can obtain very accurate eviction information from L1. The disadvantage is transparency. When the L1 software source code is unavailable (because it is usually developed by a different company from L2 software), this method is not applicable.

The second approach is to have L2 estimate an L1's replacement information. 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.

An L2 buffer cache can use a data structure called client content tracking (CCT) table to record current disk blocks $(diskID, blockNo)$ that reside in different memory location of an L1 buffer cache. The content table size grows 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 an L1 uses a 4 GBytes buffer cache, the memory space needed for a CCT is only 8 MByte and, therefore, imposing only 0.2 percent space overheads.

Fig. 7 shows a CCT table and how it changes after a read request from L1. At every read/write operation, CCT is consulted to find out which disk block was previously put in the given L1's memory address. If the old disk block is different from the currently accessed disk block, the old disk block must have been evicted from L1 to make space for the new block. Then, this eviction information is obtained by L2. The corresponding CCT entry is modified to point to the currently accessed disk block.

The advantage with the CCT approach is that it does not require modification to L1's software such as the database server's source code. However, this approach may not accurately capture an L1's eviction information. For example, if the L1 software moves blocks around in its buffer cache, this approach may capture the wrong eviction information (although this is not the case in both the Microsoft SQL Server and the Oracle Server used in our experiments).

Where to implement the CCT table depends on a particular system with second-level buffer caches. For example, in an I/O subsystem, there are two possible places: the I/O device driver and the storage server. In our experiments, we decided to implement it on the I/O device driver because it is easier to support storage clients (database servers) that use multiple storage systems. Since every I/O operation needs to pass through the I/O device driver, the CCT table can accurately keep track of an L1 buffer cache content and pass eviction information to the corresponding storage system.

6.2.2 Where to Load Blocks Evicted by L1

After an L2 buffer cache knows the L1's replacement information, L2 needs to load blocks evicted by L1. There are two methods to load these blocks: 1) The first method is to have the L1 software send the evicted blocks (even clean blocks) to L2. The DEMOTE mechanism proposed [48] takes this approach. The cleanest way to implement this method is to modify the L1 software because any external heuristic implementation may incorrectly send a block with unwanted updated back to L2. 2) The second method is to reload blocks (evicted by L1) from disks (third or lower level) to L2. This method does not require any modification to the L1 software.

One trade off between the two methods is the location of the extra traffic. The first method increases the amount of network traffic between L2 and L1, whereas the second method adds more traffic on disk accesses. In a real system, the disk bandwidth is usually less utilized than the L1-L2 network bandwidth. For example, in the database-storage hierarchy, real-world configurations typically put many disks (for example, 60-100 SCSI disks) in a storage server [50]. 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 database-storage interconnection. Moreover, a storage server cache (L2 buffer cache) can also filter some of the data access traffic. For instance, if a storage cache has a hit ratio of 50 percent, only one half of the network traffic will go to disks. In this case, using only 50 disks can saturate a 1Gbps database-storage interconnection. In such a system, the first method of loading evicted blocks from L1 can significantly degrade the system throughput. Our experimental results validate this limitation. On the other hand, in a system whose L1-L2 network bandwidth is larger than the aggregate disk bandwidth, the first method would be a better option.

Another trade off is the flexibility for optimizations. Since an L1 buffer cache evicts a clean block to make space for a new block, the first method needs to demote (send) the evicted block to an L2 buffer cache before replacing it. Due to this constraint, the time window to demote a block to the L2 buffer cache is very short, not enough to perform any effective scheduling or batching optimizations. In contrast, using the second method, any demand requests at L1 to L2 can proceed without interference.

However, the second method can also affect performance due to the additional disk traffic caused by reloading blocks (evicted by L1) from disks. To reduce the reload overhead, the following optimizations can be performed:

1. **Eliminating unnecessary reloads.** In most cache studies, the rule of thumb is that a large percentage

of accesses are made to 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 an L1 buffer cache, it is unnecessary to reload them from disks. Reloading these blocks can actually degrade an L2 buffer cache's hit ratios because they can pollute an L2 buffer cache. Unfortunately, information on future accesses is usually not available in real systems. In our implementations, we speculate cold blocks based on the number of previous accesses. In other words, L2 does not reload blocks that have been accessed fewer than $reload\_threshold$ number of times.

2. **Masking reload overheads through disk scheduling.** To avoid reloads delaying demand disk requests, demand requests can have higher priority than reloads. Reloads can be considered in a similar way to prefetching hints since it is perfectly OK if a reload operation is not performed. Given such flexibility, an L2 can put 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 works such as the freeblock scheduling [26], [3] can be easily applied here to mask reload overheads.

6.3 Extensions to Multiple L1 and L2 Buffer Caches

The global algorithm can be extended to manage multiple L1 and L2 buffer caches. First, if there are multiple L2 buffer caches in the hierarchy, the four operations of the global algorithm can still be performed with little change. The only extension is that, when an L1 evicts a block, this block should be reloaded by an L2 that is connected to the disk storing the block. If there are multiple choices, the decision can be made based on their workloads.

To support multiple L1 buffer caches is more complicated. If different L1s access different data (which is typically the case in parallel databases such as Microsoft SQL Server), one method is to divide an L2 buffer cache into multiple partitions, one for each L1. To effectively use the L2 buffer cache, a marginal benefit analysis similar to [22] can be used to dynamically determine the partition sizes. If multiple L1s share data, the global algorithm can be modified by integrating with a cooperative caching scheme for all L1s [8]. That is, if a block missed by an L1 buffer cache has a copy in another L1 buffer cache, the former can fetch the data directly from the latter. When an L1 evicts a block, the block is not reloaded into an L2 unless this is the last copy among all L1s.

7 Simulation Results

We have evaluated the local and global algorithms for second level buffer caches using both trace-driven simulations and implementations on a real storage system. First, we report our trace-driven simulation results in this section.

7.1 Evaluation of Local Algorithms

We have implemented MQ and eight existing replacement algorithms including LRU, MRU, LFU, LRU-2, FBR, LFRU, 2Q, ARC, LIRS, and offline OPT (optimal) algorithms in our L2 buffer cache simulator. The block size for all simulations is 8 KBytes. With experiments, we found out that using $log(f)$ function as our $QueueNum(f)$ function works very well for all traces. Our experiments also show that eight LRU queues are enough to separate the warmer blocks from the others. The history buffer Q_{out} size is set to be four times the number of blocks in the L2 buffer cache. Each entry of the history buffer occupies fewer than 32 bytes so that the memory requirement for the history buffer is quite small, less than 0.5 percent of the L2 buffer cache size. The *lifeTime* parameter is adjusted adaptively at running time. The main idea for dynamic *lifeTime* adjusting is to efficiently collect statistic information on the reuse distance distributions from access history. We will not discuss it further, but details can be found in [49].

The history buffer size for 2Q is one-half of the number of blocks in the cache as suggested by Johnson and Shasha in [20]. For fairness, we have extended the LFRU, LRU-2, FBR, and LFU algorithms to keep track of CRF values, second-to-last reference time, and access frequencies for recently evicted blocks, respectively. We have tuned the FBR and LFRU algorithms with several different parameters suggested by the authors and report the best results. We also implemented the offline optimal algorithm (OPT) to give a low-bound of any online local algorithms.

As with all cache studies, interesting effects can only be observed if the size of the working set exceeds the cache capacity. The two traces provided by other sources (HP Disk Trace and Auspex Server Trace) have relatively small working sets. To anticipate the current trends that working set sizes increase with user demands and new technologies, we chose to use smaller buffer cache sizes for these three traces. In most of experiments, we set the L2 buffer cache size to be larger than the L1 buffer cache size. However, this property does not always hold in real systems. For example, in many low-end or high-end configurations, storage systems such as the IBM Enterprise Storage Server have less than 1 Gigabyte of storage cache (second level buffer cache), while the front-end server, database, or file servers, typically have more than 4 Gigabytes of buffer cache (first level buffer cache). Because of this reason, we have also explored a few cases where the L2 buffer cache is equal to or smaller than the L1 buffer cache.

7.1.1 Overall Results

Table 2 shows that the MQ algorithm performs better than other online algorithms for L2 buffer caches. Its performance is robust for different workloads and cache sizes. MQ is substantially better than LRU. With the Oracle Miss Trace-128M, LRU's hit ratio is 30.9 percent for a 512 Mbyte storage cache (L2 buffer cache), whereas MQ's is 47.5 percent, a 53 percent improvement. For the same cache size, MQ has a 10 percent higher hit ratio than FBR. The main reason is that this algorithm can selectively keep a warm block in caches for a long period of time till the next access.

LRU does not perform well for the four L2 buffer cache access traces, though it works quite well for L1 buffer caches. This is because LRU does not keep blocks in the cache long enough. The LFU algorithm performs worse than LRU. The long reuse distance ($minDist$) at L2 buffer caches makes frequency values inaccurate. Of the eight online algorithms, the MRU algorithm has the worst performance. Although this algorithm can keep old blocks for a long time in L2 buffer caches, unfortunately, old blocks are not accessed frequently.

FBR, LFRU, and LRU-2 perform better than LRU, but always worse than MQ. The gap between these three algorithms and MQ is quite large in several cases. Although

TABLE 2
Hit Ratios in Percentage

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
64MB	21.6	14.0	12.0	8.4	10.1	8.1	6.1	5.9	2.6
128MB	30.3	21.7	20.0	14.6	16.3	14.1	10.1	10.8	3.7
256MB	41.8	33.0	30.0	24.2	24.3	23.5	17.6	18.7	5.8
512MB	56.1	47.5	43.5	37.8	39.5	38.2	30.9	31.1	9.9
1GB	70.7	62.1	62.1	55.8	58.8	57.2	53.0	47.6	17.9
2GB	82.0	76.3	76.8	75.2	76.8	75.8	74.5	65.1	33.7

(a)

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
16MB	16.5	10.0	7.5	4.4	5.1	4.2	4.1	3.2	1.9
32MB	22.7	15.2	12.4	9.0	10.0	7.2	6.3	6.0	2.3
64MB	30.8	22.9	16.2	15.5	19.0	12.6	11.4	11.0	3.1
128MB	40.8	32.3	32.5	25.2	26.8	21.5	19.9	19.1	4.7
256MB	52.4	44.1	43.8	38.4	36.0	34.0	32.2	30.3	7.9
512MB	63.9	57.4	57.8	53.7	50.2	49.5	47.7	44.5	14.3
1GB	72.6	69.2	69.1	68.1	67.0	66.0	64.8	61.1	26.7
2GB	83.8	80.1	80.0	79.7	80.1	79.5	79.2	76.1	50.1

(b)

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
16MB	36.5	22.0	20.6	20.5	20.2	12.9	14.5	20.2	12.6
32MB	49.9	36.4	36.3	30.0	29.6	24.8	22.1	29.6	16.9
64MB	65.8	54.2	53.8	47.4	44.5	47.6	41.4	43.6	22.7
128MB	77.2	68.9	69.2	65.1	65.0	64.0	62.5	57.3	32.9
256MB	81.2	78.5	78.3	78.5	78.0	76.8	77.8	71.5	51.0

(c)

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
8MB	32.4	21.7	9.4	8.4	8.4	13.1	2.0	6.9	0.8
16MB	45.2	33.0	31.3	19.2	18.7	26.5	16.7	13.8	1.8
32MB	57.7	47.1	46.9	38.7	38.3	40.3	36.1	20.7	3.5
64MB	68.7	59.3	59.5	55.5	55.0	53.4	53.3	25.7	7.3
128MB	77.9	70.5	70.4	68.3	67.3	64.9	66.9	35.4	13.9
256MB	86.4	81.3	80.8	80.9	78.8	76.3	78.0	60.6	26.3

(d)

(a) Oracle Miss Trace-128M, (b) Oracle Miss Trace-16M, (c) HP Disk Trace, and (d) Auspex Server Trace.

FBR and LFRU can overcome some of the LRU drawbacks by taking access frequency into account, it is difficult to choose the right combination of frequency and recency by tuning the parameters for these two algorithms. LRU-2 does not work well because it favors blocks with small reuse distances.

2Q performs better than other online algorithms except MQ. With a separate queue ($A1_{in}$) for blocks that have only been accessed once, 2Q can keep frequently accessed blocks in the A_m queue for a long period of time. However, when the L2 buffer cache size is small, 2Q performs worse than MQ. For example, with Oracle Miss Trace-128M, 2Q has a 4 percent lower hit ratio than MQ for a 512 MBytes cache. With Oracle Miss Trace-16M, the gap between MQ and 2Q is 6.7 percent for a 64 MBytes cache. This is because the lifetime of a block in the 2Q-managed L2 buffer cache is not long enough to keep the block resident for the next access.

7.1.2 Performance Analysis

To understand the performance results in more detail, we use reuse distance as a measure to analyze the algorithms. Since the traces in our study have similar access patterns, this section reports the analysis using the Oracle Miss Trace-128M trace as a representative.

The performance of a local replacement algorithm at L2 buffer caches primarily depends on how well they can satisfy the life time property. As we have observed from Section 4, accesses to L2 buffer caches tend to have long reuse distances. If the majority of accesses have reuse distances greater than D, a replacement algorithm that cannot keep blocks longer than D time is unlikely to perform well.

Our method to analyze the performance is to classify all accesses into two categories according to their reuse distances: $< C$ and $\geq C$, where C is the number of entries in the L2 buffer cache. Table 3 shows the number of hits and misses in the two categories for a 512 MBytes L2 buffer cache.

LRU has no miss in the left category because any access in this category is less than C references away from its previous access to the same block. The block being accessed should still remain in the cache since the buffer cache can hold C blocks. However, LRU has a large number of misses in the right category because any block that has not been accessed for more than C time can be evicted from the cache and, therefore, lead to a miss for the next access to this block. Since the right category dominates the total number of accesses (Fig. 3a), LRU does not perform well.

The 2Q, FBR, LFRU, and LRU2 algorithms reduce the number of misses in the right category by 15 to 25 percent because these algorithms can keep warm blocks in the cache longer than C time. However, in order to achieve this, the FBR, LFRU, and LRU2 algorithms have to sacrifice some blocks, which are kept in the cache for a short period of time. As a result, these three algorithms have some misses in the left category. But, the number of such misses is much smaller than the number of misses avoided in the right category. Overall, the three algorithms have fewer misses

TABLE 3
Oracle Miss Trace-128M Hits and Misses Distribution with a 512 MByte Buffer Cache

Algor-ithms	distance < 64k		distance ≥ 64k	
	#hits	#misses	#hits	#misses
MQ	1553k	293k	1919k	2646k
2Q	1846k	0	1330k	3235k
FBR	1611k	234k	1146k	3418k
LFRU	1412k	434k	1470k	3094k
LRU2	1606k	239k	1179k	3385k
LRU	1846k	0	407k	4157k

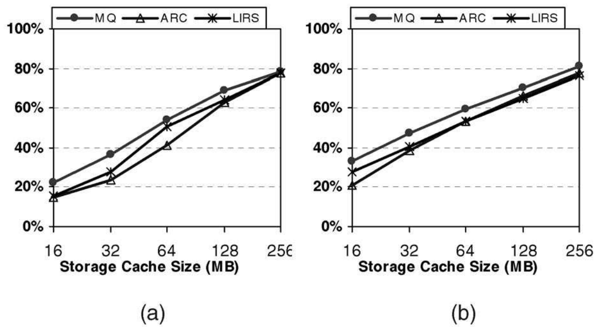

Fig. 8. Comparison with the ARC and LIRS algorithms. (a) HP Disk Trace and (b) Auspex Server Trace.

than LRU. Because the 2Q algorithm has no misses in the left category, it outperforms the FBR, LFRU, and LRU2 algorithms.

MQ significantly reduces the number of misses in the right category. As shown in Table 3, MQ has 2,646k misses in the right category, 36 percent fewer than LRU. Similarly to the FBR algorithm, MQ also has some misses in the left category. However, the number of such misses is so small that it contributes to only 10 percent of the total number of misses. Overall, the MQ algorithm performs better than others.

7.1.3 Comparison with Recently Proposed Algorithms

We have also compared MQ with two recently proposed algorithms, LIRS [19] and ARC [28]. The LIRS simulator is provided by the original author and the ARC algorithm is implemented by ourselves. Fig. 8 presents the results for two traces: the HP Disk Trace and the Auspex Server Trace.

Our results show that MQ outperforms both LIRS and ARC, especially with small L2 buffer caches. For example, for the HP Disk Trace with a 32MB storage cache, MQ gives a 36.4 percent hit ratio, whereas LIRS and ARC yield 27.6 percent and 23.4 percent hit ratios, respectively. Our results also show that LIRS is slight better than ARC.

7.2 Evaluation of Global Algorithms

To evaluate the global multilevel buffer cache management scheme, we have integrated the four operations of the global scheme into four trace-driven cache simulators that respectively use LRU, FBR, 2Q, and MQ as the local replacement policy. All cache simulators can run with two options: the pure local algorithm and the integrated global algorithm. The goal of our simulation evaluation is to find out the improvement of the global scheme on hit ratios, so we did not simulate disk accesses and network accesses. The extra overheads introduced by the global scheme are discussed in detail in Section 6.2.2. However, these overheads are reflected in our implementation results on a real system (see Section 8).

Fig. 9 compares the hit ratios between the pure-local and integrated-global algorithms for four different cache replacements with the MS-SQL-Large trace. GL-LRU denotes the global LRU algorithm, and other abbreviations are similar. Fig. 10 shows the hit ratios for all three traces. The overall results for the other two traces are similar.

As shown in Fig. 9, a global algorithm always performs better than its corresponding local algorithm. In many cases, the gap between these two is quite substantial. For example, GL-LRU has 10 percent to five times higher hit ratios than LRU. The improvements for FBR and 2Q are also significant, up to a factor of 2.

The effects of the global scheme are different for various baseline replacement algorithms. For example, with a 512 MByte L2 buffer cache, GL-FBR outperforms FBR by 49 percent and GL-2Q outperforms 2Q by 59 percent, but GL-MQ outperforms MQ by only 15 percent. The improvement of GL-MQ over MQ is relative small (11 to 80 percent) compared to other local algorithms because MQ is designed specifically for second-level buffer caches and it is already aware of the long reuse distance access pattern at L2.

The gap between the global and local algorithms is more pronounced for smaller cache sizes. For example, in the MS-SQL-Large trace with a 128 MByte L2 buffer cache, GL-2Q has a hit ratio of 9.8 percent, whereas 2Q achieves a hit ratio of 5.9 percent. But, with 2 GBytes of storage cache, they have similar cache hit ratios. The reason is that, when an L2 buffer cache is small, it is very wasteful to replicate some of L1s' blocks in an L2. A global algorithm completely eliminates replicas between L1 and L2 and, therefore, can much more effectively use an L2 buffer cache than its corresponding local algorithm. However, when an L2 buffer cache is reasonably large, the negative effect of block replication on cache hit ratios is reduced. Therefore, the difference between a global algorithm and a local algorithm becomes smaller. But, in all cases, a global algorithm is never worse than its corresponding local algorithm.

Among all local and global algorithms, GL-MQ can achieve the best hit ratios for L2 in most cases. In a few exceptions, GL-2Q has better cache hit ratios than GL-MQ, but the difference is small.

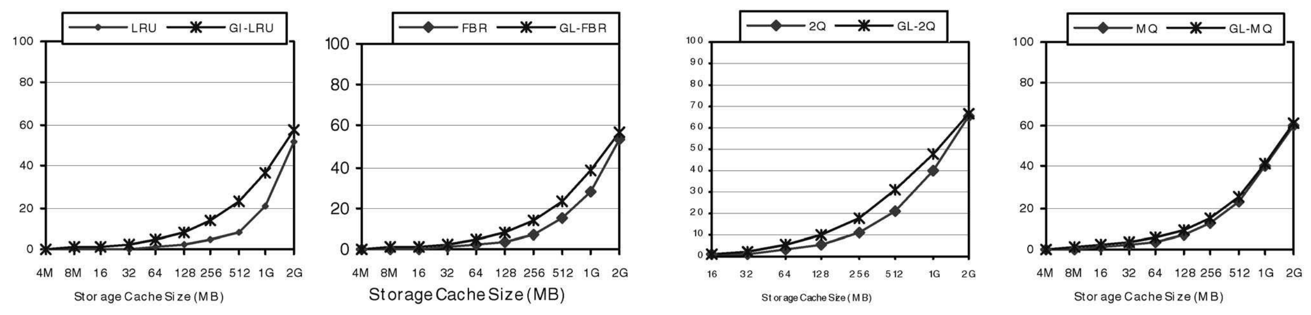

Fig. 9. Improvement of global algorithms (MS-SQL-Large trace).

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
GL-LRU	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
GL-FBR	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
GL-2Q	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
GL-MQ	23.4	37.7	56	64.6	68.1	73.1	76.6	81.8	86	89.2

(a)

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
GL-LRU	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
GL-FBR	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
GL-2Q	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
GL-MQ	3.1	5.3	8.7	14	21.9	34.2	49.1	63.3	74.8	84.1

(b)

Fig. 10. Cache hit ratios for (a) MS-SQL-Small and (b) Auspex.

8 Evaluation on Real System

To evaluate the local and global algorithms in a practical setting, we have implemented the MQ, LRU, GL-LRU, and GL-MQ in a storage system. The storage system is an industrial-strength system and has been tested by customers for more than six months [50]. We use OLTP workloads running on commercial database servers including Microsoft SQL Server and Oracle Servers to evaluate the effects of these caching algorithms on end application performance. We have also evaluated different approaches to implement global algorithms.

8.1 Experimental Testbeds

The storage system runs on a PC cluster similar to other clustered storage system [38]. We have used two hardware platforms in our experiments, one connected with Oracle Server and the other connect with Microsoft SQL Server. These two platforms have similar architecture. In each platform, the storage system manages multiple virtual volumes (virtual disks). A virtual volume can be implemented using a single or multiple physical disk partitions. Our previous study [50] also gives a detailed description of the architecture.

All PCs are connected together using a Giganet cLan network. Clients communicate with storage server nodes using the Virtual Interface (VI) communication model [45]. The peak communication bandwidth is about 100 to 113 MBytes/sec and the one-way latency for a short message is about 5 to 10 microseconds. Data transfer from the database's buffer to the storage buffer uses direct DMA without memory copying. Each PC runs Windows NT 4.0 or 2000 operating system. The interrupt time for incoming messages is 10 to 20 microseconds. The bandwidth of data transfers between disk and host memory is about 15 Mbytes/sec and the access latency for random read/writes is about 9 milliseconds. Each PC in our storage system has a large buffer cache to speed up I/O accesses. Logging disk data is not cached in the storage system. The storage system employs a write-through cache policy. But, blocks written by front-ends are still cached in the storage cache. We can also use a write-back policy, but this has little effects on our replacement algorithm evaluation.

The storage systems are connected to a database server running either Oracle 8i Server (Platform 1) or Microsoft SQL Server (Platform 2). Performance results are evaluated using the TPC-C benchmark [25]. The hardware and software setups are similar to those used for collecting the Oracle Miss Trace-128M or the MS-SQL-Large. The database server runs on a separate PC, serving as a client to the storage system. It accesses raw partitions directly. All raw I/O requests from the database server are forwarded to the storage system through Giganet. The parameters of the database server are well tuned to achieve the best TPC-C performance. Each test runs the TPC-C script on a database client machine for two hours. The database client also runs on a separate PC which connects to the database server through Fast Ethernet. It simulates 48 clients, each of which generates transactions to the database server. Our database contains 256 warehouses and occupies 100 GBytes disk space excluding logging disks.

We have implemented MQ, LRU, GL-MQ, and GL-LRU in the storage cache The parameters of the MQ algorithm are the same as described in the previous section. We have also evaluated two different approaches to implementing global algorithms. One approach modifies a database server (Microsoft SQL Server) source code to transfer evicted blocks to the storage cache, whereas the other approach uses the Client Content Tracking (CCT) table described in Section 6.2.1 to estimate L1 eviction information and reload data from disks.

Platform 1 is used to evaluate the MQ replacement algorithm and Platform 2 is used to evaluate the effects of global algorithms. Table 4 lists the difference between these two platforms.

8.2 Evaluation of MQ Replacement Algorithm

Fig. 11 shows the hit ratios of the storage buffer cache with the MQ and LRU replacement algorithms. The difference between the implementation and simulation results is less

TABLE 4
Difference of Two Experimental Platforms

Component	Platform1	Platform2
CPU	dual 400 MHz PII	dual 933 MHz PIII
L2	512 KB	2 MB
Memory	1 GB	1 GB
Database	Oracle 8i Server	Microsoft SQL Server 2000
OS	Windows NT 4.0	Windows 2000

than 10 percent, which validates our simulation study. The small difference is mainly caused by two factors. The first is that the timing is different in the real system due to concurrency. The second is the interaction between cache hit ratios and request rates. When the cache hit ratio increases, more I/O requests are forwarded to storage.

As shown in Fig. 11, MQ achieves much higher hit ratios than LRU. For a 512 MByte storage buffer cache, MQ has a 12.65 percent higher hit ratio than LRU. In fact, in order for LRU to achieve the same hit ratio as MQ, its cache size needs to be doubled. The hit ratio of MQ with a 128 MByte cache is slightly greater than that of LRU with a 256 MByte cache. The hit ratio of MQ with a 256 MBytes cache is about the same as LRU with a 512 MBytes cache.

Fig. 12 shows the end performance of the MQ and LRU algorithms. For all three buffer cache sizes, MQ improves the TPC-C end performance over LRU by 8 to 11 percent. Due to certain license problems, we are not allowed to report the absolute performance in terms of transaction rate. Therefore, all performance numbers are normalized to the transaction rate with a 128 MByte buffer cache using the LRU replacement algorithm. Because of the high hit ratios, the MQ algorithm improves the transaction rates over LRU by 8 percent, 12 percent, and 10 percent for 128 MByte, 256 MByte, and 512 MByte cache sizes, respectively.

Similar to the cache hit ratio improvement, using the MQ algorithm is equivalent to using LRU with a double sized cache. With a 128 MByte buffer cache, MQ increases the transaction rate by 8 percent. MQ with a 256 MByte cache achieves a similar transaction rate to LRU with a 512 MByte cache.

8.3 Evaluation of Global Algorithms

Fig. 13 compares the storage cache hit ratios and normalized transaction rates for local and global algorithms. RAW-GL means that it uses global algorithm without any optimizations to hide reload overheads; OPT-GL means that it uses a global algorithm with optimization to reduce reload overhead. The results for the global algorithms are presented using the second implementation that does not require any modification to the database server. At the end of this section, we will compare this implementation with the other implementation that modifies Microsoft SQL Server to send evicted data to the storage cache.

The raw global algorithms without optimization has the highest storage cache hit ratios. For example, RAW-GL-LRU can improve LRU's hit ratio by a factor of 1.49, and RAW-GL-MQ improves MQ by a factor of 1.32. Similar to the simulation results, global algorithms' improvement on storage cache hit ratios is more pronounced for LRU than for MQ.

Unfortunately, RAW-GL's improvement on cache hit ratios does not fully translate into improvement on end transaction rate. For example, RAW-GL-LRU only outperforms the access-based placement by 7 percent. RAW-GL-MQ does not have any improvement at all. The main reason is the high overheads for reloading data from disks, which significantly offsets the benefit of improved cache hit ratios. In RAW-GL-MQ, the overheads are so large that they cancel out the 32 percent improvement on cache hit ratios.

However, after reducing the reload overheads by eliminating unnecessary reloads and prioritizing demand requests over reloads, the optimized GL can achieve much higher transaction rates. For example, OPT-GL-LRU improves the transaction rate of LRU by 21 percent. OPT-GL-MQ has a speedup of 1.13 over MQ.

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. Fig. 14 plots these impacts for OPT-GL-LRU. All numbers are, respectively, normalized to the ones achieved using RAW-GL-LRU. When the

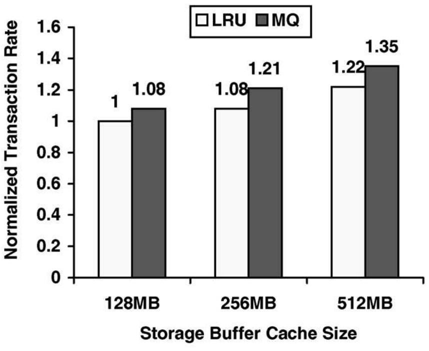

Fig. 12. Normalized TPC-C transaction rate with different storage cache sizes (all numbers are normalized to the one achieved by LRU with a 128 MByte storage cache).

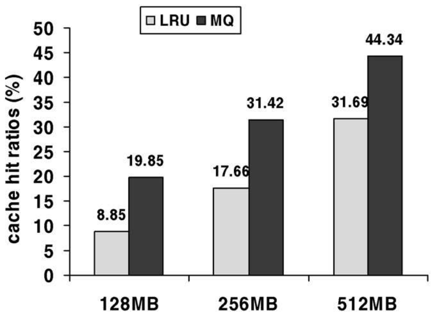

Fig. 11. Storage buffer cache hit ratios (%).

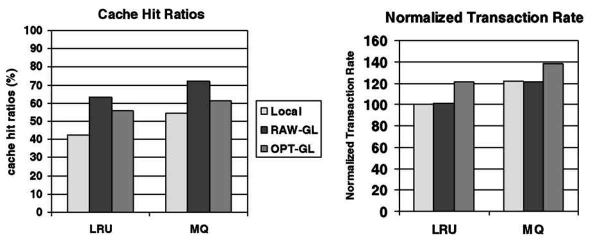

Fig. 13. Storage cache hit ratios and normalized transaction rates. All transaction rates are normalized to the ones achieved using local LRU.

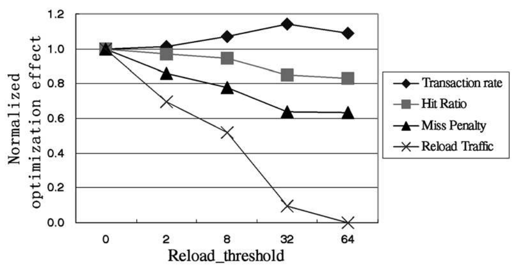

Fig. 14. Effects of optimizations for reducing reload overhead.

$reload\_threshold$ 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 becomes smaller.

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 percent. 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 32.

Implementation Trade Offs. To evaluate the trade offs between of the two different implementations of global algorithms, we also implement the other approach that sends evicted (even clean) blocks from Microsoft SQL Server's buffer cache to the storage cache. For convenience of description, we call this implementation the *demote* implementation. And, we call the default implementation that estimates L1's eviction information and reload evicted blocks from disks as the *reload* implementation.

Fig. 15 compares the performance of the two implementations under three different configurations of network bandwidth. 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 program is very simple and introduces little processor overheads.

When the available client-storage network bandwidth is high compared to the client workloads, the two implementations perform similarly. However, when the network bandwidth is low, the reload implementation outperforms the demote implementation by 20 percent, even though both approaches have similar cache hit ratios. This is because the former does not impose extra database-storage (L1-L2) network traffic, whereas the latter can potentially double the L1-L2 traffic. These results indicate that the reload implementation would be a better alternative when the L1-L2 network is heavily utilized.

9 Related Work

A large body of literature has examined cache replacement algorithms. Examples of buffer cache replacement algorithms include the LRU [12], [7], GCLOCK [42], [30], First in First Out (FIFO), MRU, LFU, Random, FBR [34], LRU-k [31], 2Q [20], LFRU [24], LIRS [19], and ARC [28]. In the spectrum of offline algorithms, Belady's OPT algorithm and WORST algorithm [5], [27] are widely used to derive a

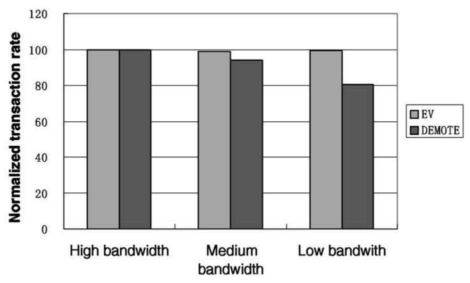

Fig. 15. Comparison of the two implementations of global algorithms with different L1-L2 bandwidth (all transaction rates are normalized to their corresponding rate with 113MB/s network bandwidth).

lower and upper bound on the cache miss rate. Other closely related works include Muntz and Honeyman's file server caching study [29] and Willick et al.'s disk cache study [47].

Cache replacement policies have been intensively studied in various contexts in the past, including processor caches [40], paged virtual memory systems [42], [6], and disk caches [41]. Although several studies [4], [46], [21] have explored two-level processor cache design issues, their conclusions do not apply to software-based L2 buffer cache designs because the former has more restrictions. Some analytical models of the storage hierarchies have been given in [17], [23].

Many past studies have used metrics such as LRU stack distance [27], marginal distribution of stack distances [1], or distance string models [43] to analyze the temporal locality of programs. Phalke recently proposed the interreference gap (IRG) model [33] to characterize temporal localities in program behavior. But, this model looks at each address separately. Therefore, it cannot capture global access behavior well.

Our work also builds upon many other previous studies [10], [16], [29], [51], [47], [48]. Dan et al. conducted a theoretical analysis of hierarchical buffering in a shared database environment [10]. Franklin et al. also explored global memory management in database systems [16]. Our work is also related to work on hardware victim caches [21], which is used to keep a few recently evicted blocks from set-associative processor caches.

Our work is related to but different from previous work on cooperative caching or global memory management [8], [15], [37] because those works focus on multiple buffer caches at the same level and managed by the same software (e.g., distributed file systems). Our work studies the multilevel buffer cache hierarchy.

10 Conclusion

This paper reported our study on second-level buffer cache management. More specifically, it presents a new local replacement algorithm called MQ and a global cache management scheme to effectively manage second-level buffer caches. These algorithms are based on our study of second-level buffer cache access patterns.

We have evaluated these algorithms using both simulations and implementations on a real storage system that is connected to commercial database servers (Microsoft SQL Server and Oracle Server) running OLTP benchmarks. Our results show that MQ performs better than nine existing local algorithms and that it is robust for different workloads and cache sizes. The implementation results of the TPC-C benchmark on a 100 GBytes database show that the MQ algorithm has much better hit ratios than LRU and improves the TPC-C transaction rate by 8 to 12 percent over LRU. For LRU to achieve a similar level of performance, the cache size needs to be doubled. Our results also show that the global algorithms can improve local algorithms' cache hit ratios by 10 to 500 percent for second-level buffer caches and improve the end application transaction rate by 20 percent. Even though our experiments are conducted on storage systems, we expect the main results are similar on other systems with second-level buffer caches that are not significantly larger than their first-level buffer caches.

Acknowledgments

The authors would like to thank James F. Philbin for the useful discussions on this work. They would also like to thank the anonymous reviewers and editors, especially Dr. Alok Choudhary, for the useful comments and feedback. This work was supported by the US National Science Foundation CCR-0305854 grant and IBM CAS Fellowship. The authors experiments were conducted on equipment provided through the IBM SUR-2003 grant.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing Reference Locality in the WWW," *Proc. IEEE Conf. Parallel and Distributed Information Systems,* Dec. 1996.

[2] O.I. Aven, E.G.I. Coffmann, and I.A. Kogan, *Stochastic Analysis of Computer Storage.* Amsterdam: Reidel, 1987.

[3] E. Bachmat and J. Schindler, "Analysis of Methods for Scheduling Low Priority Disk Drive Tasks," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems,* 2002.

[4] J.-L. Baer and W.-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Ann. Int'l Symp. Computer Architecture,* pp. 73-80, May-June 1988.

[5] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.,* vol. 5, no. 2, pp. 78-101, 1966.

[6] R. Carr and J. Hennessy, "Wsclock—A Simple and Efficient Algorithm for Virtual Memory Management," *Proc. Eighth Symp. Operating Systems Principles,* Dec. 1981.

[7] E. Coffman Jr. and P. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.

[8] M. Dahlin, R. Wang, T.E. Anderson, and D.A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *Operating Systems Design and Implementation,* pp. 267-280, 1994.

[9] M.D. Dahlin, C.J. Mather, R.Y. Wang, T.E. Anderson, and D.A. Patterson, "A Quantitative Analysis Scalability for Network File Systems," *Proc. SIGMETRICS,* 1994.

[10] A. Dan, D.M. Dias, and P.S. Yu, "Analytical Modelling of a Hierarchical Buffer for a Data Sharing Environment," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Rev.,* vol. 19, no. 1, May 1991.

[11] A. Dan and D. Towsley, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes," 1990.

[12] P.J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM,* vol. 11, no. 5, pp. 323-333, May 1968.

[13] P.J. Denning and S.C. Schwartz, "Properties of the Working-Set Model," *Comm. ACM,* vol. 15, no. 3, pp. 191-198, Mar. 1972.

[14] EMC Corp., Symmetrix 3000 and 5000 Enterprise Storage Systems, Product Description Guide, http://www.emc.com/products/product pdfs/pdg/symm_3_5_pdg.pdf, 1999.

[15] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy, and C.A. Thekkath, "Implementing Global Memory Management in a Workstation Cluster," *Proc. Symp. Operating Systems Principles,* pp. 201-212, 1995.

[16] M.J. Franklin, M.J. Carey, and M. Livny, "Global Memory Management in Client-Server DBMS Architectures," *Proc. Int'l Conf. Very Large Data Bases,* pp. 596-609, Aug. 1992.

[17] J. Gecsei, "Determining Hit Ratios for Multilevel Hierarchies," *IBM J. Research and Development,* vol. 18, no. 4, pp. 316-327, July 1974.

[18] IBM, IBM Enterprise Storage Server, www.storage.ibm.com/hardsoft/products/ess/ess.htm, IBM Corp., 1999.

[19] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. SIGMETRICS,* pp. 31-42, 2002.

[20] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. Very Large Databases Conf.,* pp. 439-450, 1995.

[21] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 364-373, May 1990.

[22] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Symp. Operating Systems Design and Implementation,* 2000.

[23] C. Lam and S.E. Madnick, "Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data," *ACM Trans. Database Systems,* vol. 4, no. 3, pp. 345-367, Sept. 1979.

[24] 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," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computing Systems, SIGMETRICS Performance Evaluation Rev.,* vol. 27, no. 1, pp. 134-143, May 1999.

[25] S.T. Leutenegger and D. Dias, "A Modeling Study of the TPC-C Benchmark," *SIGMOD Record,* vol. 22, no. 2, pp. 22-31, June 1993.

[26] 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," *Proc. Symp. Operating Systems Design and Implementation,* 2000.

[27] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.

[28] N. Megiddo and D.S. Modha, "Arc: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Conf. File and Storage Technologies,* 2003.

[29] D. Muntz and P. Honeyman, "Multi-Level Caching in Distributed File Systems-or-Your Cache Ain't Nuthin' but Trash," *Proc. Usenix Winter 1992 Technical Conf.,* pp. 305-314, Jan. 1991.

[30] V.F. Nicola, A. Dan, and D.M. Dias, "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing," *Proc. 1992 ACM SIGMETRICS and PERFORMANCE Int'l Conf. Measurement and Modeling of Computer Systems,* vol. 20, no. 1, p. 35, June 1992.

[31] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 297-306, May 1993.

[32] Oracle 8i Concepts, Oracle Corp., 1999.

[33] B.G.V. Phalke, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 291-300, May 1995.

[34] J. Robinson and M. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems,* 1990.

[35] C. Ruemmler and J. Wilkes, "A Trace-Driven Analysis of Disk Working Set Sizes," Technical Report HPL-OSR-93-23, Hewlett-Packard Laboratories, Palo Alto, Calif., Apr. 1993.

[36] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proc. Winter 1993 USENIX Conf.,* 1993.

[37] P. Sarkar and J. Hartman, "Efficient Cooperative Caching Using Hints," *Proc. Second ACM Symp. Operating Systems Design and Implementation,* 1996.

[38] R.A. Shillner and E.W. Felten, "Simplifying Distributed File Systems Using a Shared Logical Disk," Technical Report TR-524-96, CS Dept., Princeton Univ., 1996.

[39] D. Sleator and R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Comm. ACM,* vol. 28, no. 2, pp. 202-208, Feb. 1985.

[40] A.J. Smith, "Cache Memories," *ACM Computing Surveys,* vol. 14, no. 3, pp. 473-530, Sept. 1982.

[41] A.J. Smith, "Disk Cache—Miss Ratio Analysis and Design Considerations," *ACM Trans. Computer Systems,* vol. 3, pp. 161-203, 1985.

[42] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer," *Proc. Int'l Conf. Parallel Processing,* pp. 6-8, 1978.

[43] J.R. Spirn, “Distance String Models for Program Behavior,” *Computer,* vol. 9, no. 11, pp. 14-20, Nov. 1976.
[44] Transaction Processing Performance Council, TPC Benchmark C, Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, Calif. 95112-6311, May 1991.
[45] Virtual Interface Architecture Specication Version 1.0, VI-Architecture Organization, http://www.viarch.org/, 1997.
[46] W.-H. Wang, J.-L. Baer, and H.M. Levy, “Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy,” *Proc. Int'l Symp. Computer Architecture,* 1989.
[47] D.L. Willick, D.L. Eager, and R.B. Bunt, “Disk Cache Replacement Policies for Network Fileservers,” *Proc. 13th Int'l Conf. Distributed Computing Systems,* May 1993.
[48] T. Wong and J. Wilkes, “My Cache or Yours? Making Storage More Exclusive,” *Proc. USENIX Ann. Technical Conf.,* 2002.
[49] Y. Zhou, “Memory Management for Networked Servers,” PhD thesis, technical report, Computer Science Dept., Princeton Univ., Nov. 2000.
[50] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J.F. Philbin, and K. Li, “Experiences with VI Communication for Database Storage,” *Proc. Int'l Symp. Computer Architecture.,* May 2002.
[51] Y. Zhou, J.F. Philbin, and K. Li, “The Multi-Queue Replacement Algorithm for Second Level Buffer Caches,” *Proc. Usenix Technical Conf.,* June 2001.

Yuanyuan Zhou received the PhD and MA degrees from Princeton University and the BS degree from Beijing University, China. She is currently an assistant professor at the University of Illinois at Urbana/Champaign (UIUC). Prior to UIUC, she worked at the NEC Research Institute as a scientist from 2000 to 2002. She is a member of the IEEE and the IEEE Computer Society.

Zhifeng Chen received the MA degree from Princeton University and the BS degree from Fudan University, China. He is currently a PhD candidate at the University of Illinois at Urbana/Champaign (UIUC).

Kai Li received the PhD degree from Yale University. He is currently a Charles Fitzmorris Professor at Princeton University and CTO of DataDomain. He is a fellow of the ACM and a senior member of the IEEE and the IEEE Computer Society.