The Multi-Queue Replacement Algorithm for Second Level Buffer Caches

Yuanyuan Zhou and James F. Philbin
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540
Kai Li
Computer Science Department, Princeton University, Princeton, NJ 08544

Abstract

This paper reports our research results that improve second level buffer cache performance. Several previous studies have shown that a good single level cache replacement algorithm such as LRU does not work well with second level buffer caches. Second level buffer caches have different access pattern from first level buffer caches because *Accesses* to second level buffer caches are actually *misses* from first level buffer caches.

The paper presents our study of second level buffer cache access patterns using four large traces from various servers. We also introduce a new second level buffer cache replacement algorithm called Multi-Queue (MQ). Our trace-driven simulation results show that MQ performs better than all seven tested alternatives. Our implementation on a real storage system validates these results.

1 Introduction

Servers such as file servers, storage servers, and web servers play a critical role in today's distributed, multiple-tier computing environments. In addition to providing clients with key services and maintaining data consistency and integrity, a server usually improves performance by using a large buffer to cache data. For example, both EMC Symmetric Storage Systems and IBM Enterprise Storage Server deploy large software-managed caches to speed up I/O accesses [7, 8].

Figure 1 shows two typical scenarios of networked clients and servers. A client can be either an end user program such as a file client (Figure 1a), or a front-tier server such as a database server (Figure 1b). A server buffer cache thus serves as a second level buffer cache in a multi-level caching hierarchy. In order to distinguish server buffer caches from traditional single level buffer caches, we call a server buffer cache a *second level buffer cache*. In contrast, we call a client cache or a front-tier server cache as a *first level buffer cache*.

Second level buffer caches have different access pattern from single level buffer caches because *accesses* to a second level buffer cache are *misses* from a first level buffer cache. First level buffer caches typically employ an LRU replacement algorithm, so that recently accessed blocks will be kept in the cache. As a result, accesses to a second buffer cache exhibit poorer temporal locality than those to a first level buffer cache; a replacement algorithm such as LRU, which works well for single level buffer caches, may not perform well for second level buffer caches.

Muntz and Honeyman [28] investigated multi-level caching in a distributed file system, showing that server 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 server buffer caches, but raised the question whether the algorithms that work well for client or single level buffer caches can effectively reduce misses for server caches. 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, which works well for client caches [43]. However, several key related questions still remain open.

- Do other server workloads have access patterns similar to file servers?
- How do recently proposed client cache replacement algorithms such as LRU-k [15], Least Frequently Recently Used (LFRU) [14] and Two Queues (2Q) [23] perform for server caches?

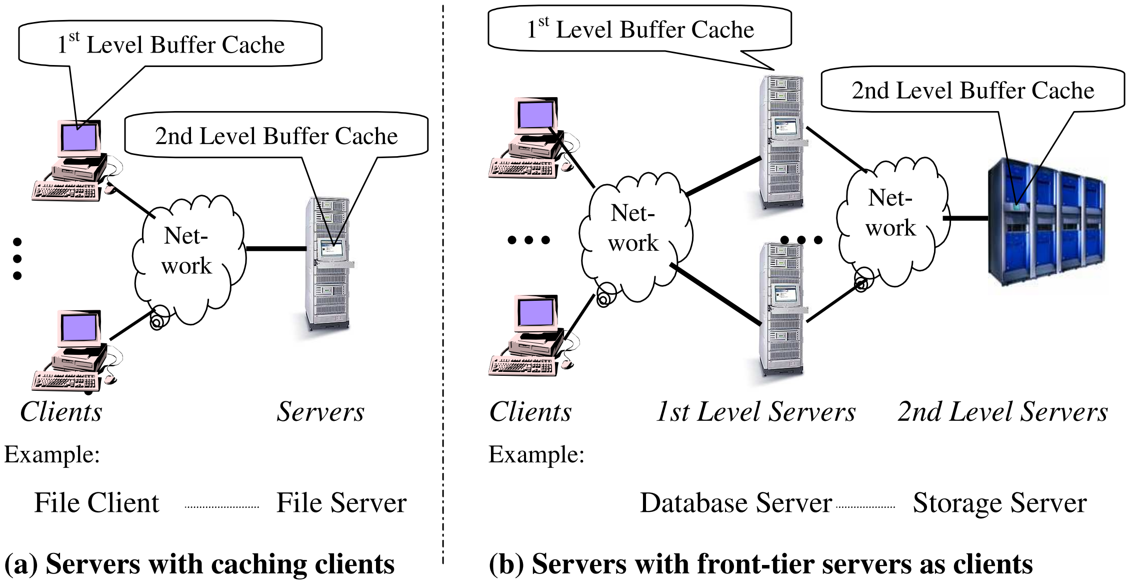


Figure 1: Second level buffer caches.

- What properties should a good server buffer cache replacement algorithm have?
- Is there an algorithm that performs better than FBR for server buffer caches?

This paper reports the results of our study of these questions. We first studied second level buffer cache access patterns using four large traces from file servers, disk subsystems and database back-end storage servers. Our analysis shows that a good second level buffer cache replacement algorithm should have three properties: minimal life time, frequency-based priority and temporal frequency. Finally, we introduce a new algorithm called Multi-Queue. Our trace-driven simulation results show that the new algorithm performs better than LRU, MRU, LFU, FBR, LRU-2, LFRU and 2Q, and that it is robust for different workloads and cache sizes. Our result also reveals that the 2Q algorithm, which does not perform as well as others for single level buffer caches, has higher hit ratios than all tested alternatives except Multi-Queue for second level buffer caches.

To further validate our results, we have implemented the Multi-Queue and LRU algorithms on a storage server system with the Oracle 8i Enterprise Server as the client. Our results using the TPC-C benchmark on a 100 GBytes database demonstrate that the Multi-Queue algorithm improves the transaction rate by 8-11% over LRU. To achieve the same improvement with LRU requires doubling the cache size of the storage server.

2 Methodology

The goal of our study is to improve the second level buffer cache performance. In this section, we briefly describe the existing algorithms tested in our evaluation and present the four traces used in our study.

2.1 Algorithms

We evaluate seven existing on-line replacement algorithms that were original designed for client/single level buffer caches.

The **Least Recently Used (LRU)** algorithm has been widely employed for buffer cache management [9, 6]. It replaces the block in the cache which has not been used for the longest period of time. It is based on the observation that blocks which have been referenced in the recent past will likely be referenced again in the near future. However, for second level buffer caches, this observation is no longer present, or exists to much lesser extent. That is the reason why LRU does not perform well for file server caches in Willick, Eager and Bunt's study [43]. It is interesting to see whether this is also true for other workloads such as database back-end storage servers. The time complexity of this algorithm is $O(1)$.

The **Most Recently Used (MRU)** algorithm is also called Fetch-and-Discard replacement algorithm [9, 6]. This algorithm is used to deal with the case such as sequential scanning access pattern, where most of recently accessed pages are not reused in the near future. Blocks that are recently accessed in a second level buffer cache will likely stay in a first level buffer cache for a period of time, so they won't be reused in the second level buffer cache in the near future. Does this mean MRU is likely to perform well for second level buffer caches? Willick, Eager and Bunt did not evaluate this algorithm in their study [43]. The time complexity of this algorithm is $O(1)$.

The **Least Frequently Used (LFU)** algorithm is another classic replacement algorithm. It replaces the block that is least frequently used. The motivation for this algorithm is that some blocks are accessed more frequently than others so that the reference counts can be used as an estimate of the probability of a block being referenced. The "aged" LFU usually performs better than the original LFU because the former gives different weight to the recent references and very old references. In [43], "Aged" LFU always performs better than LRU for the file server workload, except when the client cache is small compared to the second level buffer cache. Our results show this is true for two traces, but for the other four, LFU performs worse than LRU because they have some temporal locality. The time complexity of this algorithm is $O(log(n))$.

The **Frequency Based Replacement(FBR)** algorithm was originally proposed by Robinson and Devarakonda [31] within a context of a stand-alone system. It is a hybrid replacement policy combining both LRU and LFU algorithms in order to capture the benefits of both algorithms. It maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. The time complexity of this algorithm ranges from $O(1)$ to $O(log_2 n)$ depending on the section sizes. The algorithm also requires parameter tuning to adjust the section sizes. So far, no on-line adaptive scheme has been proposed. In Willick, Eager and Bunt's file server cache study (1992) [43], FBR is the best algorithm among all tested ones including LRU, LFU, MIN, and RAND.

The **Least kth-to-last Reference (LRU-k)** algorithm was first proposed by O' Neil, et.al for database systems [15] in 1993. It bases its replacement decision on the time of the K^{th}-to-last reference of the block, i.e., the reference density observed during the past K references. When K is large, it can discriminate well between frequently and infrequently referenced blocks. When K is small, it can remove cold blocks quickly since such blocks would have a wider span between the current time and the Kth-to-last reference time. The time complexity of this algorithm is $O(log(n))$.

The **Least Frequently Recently Used (LFRU)** 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 [14]. 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. The algorithm replaces the block with the minimum CRF value. Each reference to a block contributes to its CRF value. A reference's contribution is determined by a weighting function $F(x)$ where x is the time span from the reference to the current time. By changing the parameters of the weighting function, LFRU can implement either LRU or LFU. The time complexity of this algorithm is between $O(1)$ and $O(log(n))$. It also requires parameter tuning and no dynamic scheme has been proposed.

The **Two Queue (2Q)** algorithm was first proposed for database systems by Johnson and Shasha in 1994 [23]. The motivation is to removed cold blocks quickly. It achieves this by using one FIFO queue $A1_{in}$ and two LRU lists, $A1_{out}$ and A_m. When first accessed, a block is placed in $A1_{in}$; when a block is evicted from $A1_{in}$, its identifier is inserted into $A1_{out}$. An access to a block in $A1_{out}$ promotes this block to A_m. The time complexity of the 2Q algorithm is $O(1)$. The authors have proposed a scheme to select the input parameters: $A1_{in}$ and $A1_{out}$ sizes. Lee and et. al. showed that 2Q does not perform as well as others for single level buffer caches [14]. However, our results show that 2Q in most cases performs better than tested alternatives except the new algorithm for second level buffer caches.

2.2 Traces

We have collected four server buffer cache traces from file servers, disk subsystems and database back-end storage servers. These traces are chosen to represent different types of workloads. All traces contained only misses from one or multiple client buffer caches that use LRU or its variations as their replacement algorithms. The block sizes for these traces are 8 Kbytes.

Table 1 shows the characteristics of the four traces.

	First Level Cache Size (MBytes)	# Reads (millions)	# Writes (millions)	# Clients or # First level Servers
Oracle Miss Trace-128M	128	7.3	4.3	single
Oracle Miss Trace-16M	16	3.8	2.0	single
HP Disk Trace	30	0.2	0.3	multiple
Auspex Server Trace	8 per client	1.8	0.8	multiple

Table 1: Characteristics of the four traces used in the study.

The first level buffer cache size clearly affects server buffer cache performance. We set the first level buffer cache sizes for the two Oracle 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 is collected from a storage system connecting to an Oracle 8i database client running the standard TPC-C benchmark [42, 27] for about two hours. The Oracle buffer cache replacement algorithm is similar to LRU [5]. 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 Oracle process to the storage system. That is, the trace includes only reads that are missed on the Oracle buffer cache and writes that are flushed back to the storage system periodically or at commit time. The trace ignores all accesses to log disks. In order to better represent the workload on a real database system, we used 128 MBytes for the Oracle buffer cache.

Oracle Miss Trace-16M is collected with the same setup as the previous trace except the database buffer cache (first level buffer cache) size is set to 16 MBytes. For both traces, we fixed the execution time to be 2 hours instead of fixing the total number of transactions. Oracle Miss Trace-128M has performed many more transactions than the second trace. That is why both traces have similar amount of misses.

HP Disk Trace was collected at Hewlett-Packard Laboratories in 1992 [33, 32]. 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 Hewlett-Packard Laboratories to do simulations, compilation, editing and 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 [16]. The system included 237 clients spread over four Ethernets, each of which connected directly to the central server. The trace covers seven days. We preprocessed the trace to include only block and directory read and write accesses.

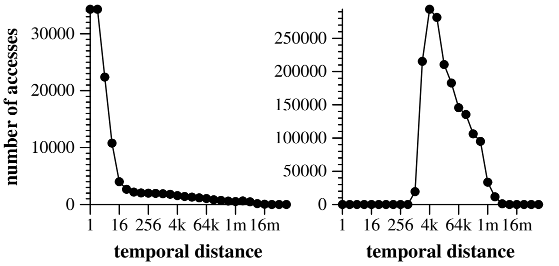
Similarly to [16], we first split the trace into small trace files according to the client host ID. We then ran these traces through a multi-node cache simulator and collected the interleaved misses from different client caches as our server buffer cache trace. The multi-node client cache simulator uses 8 MBytes for each client cache and runs the LRU replacement algorithm.

3 Server Access Pattern

We have studied the access pattern of these four traces with the goal of understanding the behavior of server buffer cache accesses by examining their temporal locality and the distribution of access frequency.

3.1 Temporal Locality

The first part of our study explored the temporal locality of server buffer cache accesses. Past studies have shown that client buffer cache accesses exhibit a high degree of spatial and temporal locality. An accessed block exhibits temporal locality if it is likely to be accessed again in the near future. An accessed block exhibits spatial locality, if blocks near it are likely to be accessed in the near future [11, 38]. The LRU replacement algorithm, typically used in client buffer caches, takes advantage of temporal locality. Thus, blocks with a high degree of temporal locality are likely to remain in a client buffer cache, but accesses to a server buffer cache are misses from a client buffer cache. Do server buffer cache acccesses exhibit temporal locality similar to those of a client buffer cache?



(a) Auspex Client Trace (b) Auspex Server Trace

Figure 2: Comparison of temporal locality of client and server buffer cache accesses using temporal distance histograms. (Note: figures are in different scales)

We used *temporal distance* histograms to observe the temporal locality of the server buffer cache traces. A *reference sequence* (or *reference string*) is a numbered sequence of temporally ordered accesses to a server buffer cache. The *temporal distance* is the distance between two accesses to the same block in the reference sequence. It is similar to the inter-reference gap in [30]. For example, in the reference sequence $ABCDBAX$, the temporal distance from A_1 to A_6 is 5 and the temporal distance from B_2 to B_5 is 3. Formally speaking, if we denote the sequence number of the current access and previous access to a block b as $current(b)$ and $prev(b)$ respectively, then the temporal distance of the current access to block b is $current(b) - prev(b)$. A temporal distance histogram shows how many *correlated accesses* (accesses to the same block) for various temporal distances reside in a given reference sequence.

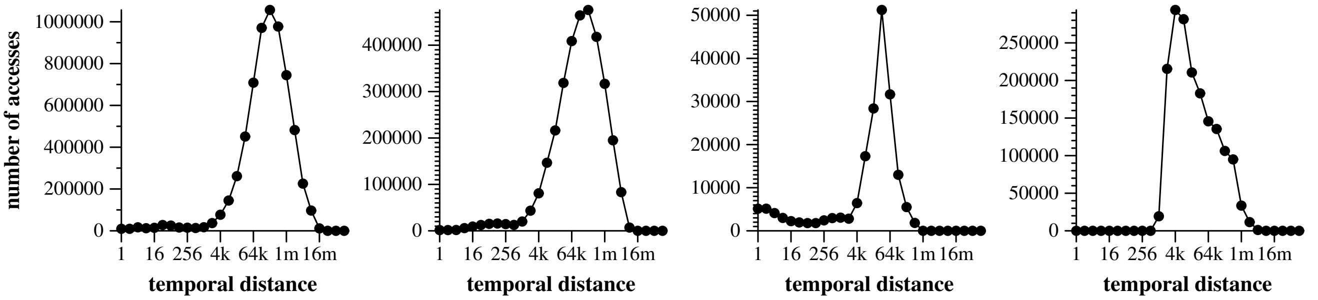
Figure 2 compares the client and server buffer cache's temporal locality using temporal distance histograms with the Auspex traces. The client buffer cache trace is collected at an Auspex client, while the server buffer cache trace is captured at the Auspex File Server. Each Auspex client uses an 8 megabyte buffer cache. The data in the figure shows the histograms by grouping temporal distances by powers of two. The block size is 8 Kbytes. Results are similar with other block sizes. Distances that are not a power of two are rounded up to the nearest power of two. Significantly, for the client buffer cache 74% of the correlated references have a temporal distance less than or equal to 16. This indicates a high degree of temporal locality. Even more significantly, however, 99% of the correlated accesses to the server buffer cache have a temporal distance of 512 or greater, exhibiting much weaker temporal locality.

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

A good replacement algorithm for server buffer 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 and end of this "hill" region *minimal distance* (or $minDist$) and *maximal distance* (or $maxDist$) respectively. We picked $minDist$ and $maxDist$ for each trace by examining the histogram figure for simplicity. Since the temporal distance values in the "hill" are relatively large, a good replacement algorithm should keep most blocks in this region for at least the $minDist$ time. We call this property *minimal lifetime property*. It is clear that when the number of blocks in a server buffer cache is less than the $minDist$ of a given workload, the LRU policy tends to perform poorly, because most blocks do not stay in the server buffer cache long enough for subsequent correlated accesses.

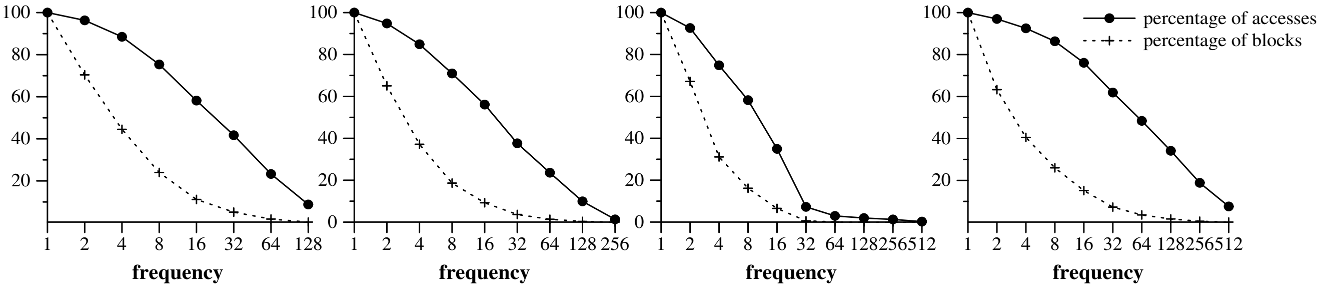
3.2 Access Frequency

Next, we examined the behavior of server buffer cache accesses in terms of frequency. While it is clear that server buffer cache accesses represent misses from client buffer caches, the distribution of access frequencies among blocks remains uncertain. If the distribution is even, then most replacement algorithms will perform similarly to or worse than LRU. If the distribution is uneven, then a good replacement algorithm will keep frequently accessed blocks in a server buffer cache. Past studies [11, 38] 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 server buffer caches?



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

Figure 3: Temporal distance histograms of server buffer 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 (d) Auspex Server Trace

Figure 4: Access and block distribution among different frequencies. A point (f, p_1) on the block percentage curve indicates that p_1% of 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% of total number of accesses are to blocks accessed at least f times.

Our hypothesis is that both hot and cold blocks will be referenced less frequently in server buffer caches, because hot blocks will stay in client 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 server buffer caches should be uneven, though probably not as uneven as those at client buffer caches. A good server buffer cache replacement algorithm should be able to identify warm blocks and keep them in server buffer caches for a longer period of time than others.

In order to understand the frequency distributions of reference sequences seen at server 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. Figure 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% accesses are made to less than 10% blocks, each of which are accessed at least 16 times. This shows that the access frequency distribution among blocks at server buffer caches is uneven. In other words, a subset of blocks are accessed more frequently than others even though the average temporal distance between two correlated accesses in this subset is very large (Figure 3). Thus, if the replacement algorithm can selectively keep those blocks for a long period of time, it will significantly reduce the number of misses, especially when the server buffer cache size is small.

3.3 Properties

To summarize our study results, a good server buffer cache replacement algorithm should have the following three properties:

1. **Minimal lifetime**: warm blocks should stay in a server buffer cache for at least *minDist*

time for a given workload.

2. **Frequency-based priority**: blocks should be prioritized based on their access frequencies.

3. **Temporal frequency**: blocks that were accessed frequently in the past, but have not been accessed for a relatively long time should be replaced.

The first two properties are derived from our study of the traces. The third property is obvious. It addresses the common access pattern where a block is accessed very frequently for some time and then has no accesses for a relatively long time.

Algorithms developed in the past do not possess all three properties. Both LRU and MRU algorithms satisfy the temporal frequency property, but lack the other two. The basic LFU algorithm possesses only the second property. With frequency aging, it can satisfy the third. LRU-2 can satisfy the third property but it only partially satisfies the first and second. FBR and LFRU vary between LRU and LFU depending on the input parameters, but it is almost impossible to find parameters that satisfy all three properties at once. 2Q satisfies the third property, but it can only partially satisfy the second property. When the server buffer cache is small, 2Q lacks the first property, but, for large cache sizes, satisfies it.

4 Multi-Queue Algorithm

We have designed a new replacement algorithm, called Multi-Queue (MQ), that satisfies the three properties above. This algorithm maintains blocks with different access frequencies for different periods of time in the second level buffer cache.

The MQ algorithm uses multiple LRU queues: Q_0, ..., Q_{m-1}, where m is a parameter. Blocks in Q_j have a longer lifetime in the cache than those in Q_i ($i < j$). MQ also uses a history buffer Q_{out}, similarly to the 2Q algorithm [23], 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 8th access to a block that is already in the second level 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 non-empty queue from the second level 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 demotes blocks from higher to lower level queues in order to eventually evict blocks that have been accessed frequently in the past, but have not been accessed for a long time. MQ does this by associating a value called $expireTime$ with each block in the server buffer cache. "Time" here refers to logical time, measured by 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. Figure 5 gives a pseudo-code 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 [23] 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 life time 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

```
/* 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;
      }
   }
}
```

Figure 5: MQ algorithm

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.

MQ satisfies the three properties that a good second level buffer cache replacement algorithm should have. It satisfies the minimal lifetime property because warm blocks are kept in high level LRU queues for at least $expireTime$ time, which is usually greater than the $minDist$ value of a given workload. It satisfies the frequency-based priority property because blocks that are accessed more frequently are put into higher level LRU queues and are, therefore, less likely to be evicted. It also satisfies the temporal frequency property because MQ demotes blocks from higher to lower level queues when its lifetime in its current queue expires. A block that has not been accessed for a long time will be gradually demoted to queue Q_0 and eventually evicted from the second level buffer cache.

5 Simulation and Results

This section reports our trace-driven simulation results of nine replacement algorithms including LRU, MRU, LFU, LRU-2, FBR, LFRU, 2Q, OPT (an optimal off-line algorithm), and MQ. Our goal is to answer three questions:

- How does MQ compare with other algorithms? How does recently proposed single level cache replacement algorithms such as LRU-2, LFRU and 2Q perform for second level buffer caches?
- How can we use the second level buffer cache access behaviors to explain the performance?
- How do one use the simulation information to tune the performance of the MQ algorithm?

The following addresses each question in turn.

5.1 Simulation Experiments

We have implemented the nine replacement algorithms in our buffer cache simulator. The block size for all simulations is 8 KBytes. With experiements, 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 seperate the warmer blocks from the others. The history buffer Q_{out} size is set to be four times of the number of blocks in the 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% of the 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 temporal distance distributions from access history. Due to page limits, we will not discuss it in this paper, but details can be found in [44].

The history buffer size for 2Q is one half of the number of blocks in the cache as suggested by Johnson and Shasha in [23]. For fairness, we have extended the LFRU, LRU-2, FBR and LFU algorithms to use a history buffer to keep track of CRF values, second-to-last reference time and access frequencies for recently evicted blocks respectively (see section 2), using a history buffer of size equal to that in MQ. We have tuned the FBR and LFRU algorithms with several different parameters as suggested by the authors and report the best performance. The off-line optimal algorithm (OPT) was first proposed by Belady [2, 17] and is widely used to derive the lower bound of cache miss ratio for a given reference string. This algorithm replaces the block with the longest future reference distance. Since it relies on the precise knowledge of future references, it cannot be used on-line.

ws Belady's OPT algorithm and WORST algorithm [2, 17]

As with all cache studies, interesting effects can only be observed if the size of the working set exceeds the cache capacity. The three traces provided by other sources (HP Disk Trace, Auspex Server Trace and Web 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 second level buffer cache size to be larger than the first level buffer cache size. However, this property does not always hold in real systems, For example, most of storage systems such as the IBM Enterprise Storage Server have less than 1 Giga Bytes of storage cache (second level buffer cache), while the frontier server, database or file servers, typically have more than 2 Gigabytes of buffer cache (first level buffer cache). Because of this reason, we have also explored a few cases where the second level buffer cache is equal to or smaller than the first level buffer cache.

5.2 Results

Table 2 shows that the MQ algorithm performs better than other on-line algorithms. 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% for a 512 Mbytes server cache, whereas MQ's is 47.5%, a 53% improvement. For the same cache size, MQ has a 10% higher hit ratio than FBR. The main reason for MQ's good performance is that this algorithm can selectively keep warm blocks in caches for a long period of time till subsequent correlated accesses.

LRU does not perform well for the four server cache access traces, though it works quite well for client buffer caches. This is because LRU does not keep blocks in the cache long enough. The LFU algorithm performs worse than LRU. The long temporal distance ($minDist$) at server buffer caches makes frequency values inaccurate. Of the eight on-line algorithms, the MRU algorithm has the worst performance. Although this algorithm can keep old blocks for a long time in server buffer caches, it does not consider frequencies. As a result, some blocks kept in server buffer caches for a long time 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 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 temporal distances.

2Q performs better than other on-line 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 server buffer cache size is small, 2Q performs worse than MQ. For example, with Oracle Miss Trace-128M, 2Q has a 4% lower hit ratio than MQ for a 512 MBytes cache. With Oracle Miss Trace-16M, the gap between MQ and 2Q is 6.7% for a 64 MBytes cache. This is because the lifetime of a block in the 2Q server buffer cache is not long enough to keep the block resident for the next access.

5.3 Performance Analysis

To understand the performance results in more detail, we use temporal 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 replacement algorithm at

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) Oracle Miss Trace-128M

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) Oracle Miss Trace-16M

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) HP Disk Trace

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) Auspex Server Trace

Table 2: Hit ratios in percentage

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
LFU	1077k	769k	1196k	3368k
MRU	285k	1560k	434k	4131k

Table 3: Oracle Miss Trace-128M hits and misses distribution with a 512 MBytes cache (Note: first-time accesses to any blocks are not counted in either category).

server caches primarily depends on how well they can satisfy the life time property. As we have observed from Section 3, accesses to server caches tend to have long temporal distances. If the majority of accesses have temporal 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 temporal distances: $< C$ and $\geq C$ where C is the number of entries in the server buffer cache. Table 3 shows the number of hits and misses in the two categories for a 512 MBytes server 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 (Figure 3(a)), LRU does not perform well.

The 2Q, FBR, LFRU and LRU2 algorithms reduce the number of misses in the right catefory by 15-25% 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 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 on Table 3, MQ has 2,646k misses in the right category, 36% 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% of the total number of misses. Overall, the MQ algorithm performs better than other on-line algorithms.

6 Implementation and Results

We have implemented the MQ and LRU algorithms in a storage server system. The goals of our implementation are:

- to validate the simulation results;
- to study the end performance improvement on a real system.

This section describes the storage system architecture, the MQ and LRU implementation, the experiment setups, and the experimental results of the TPC-C benchmark with the Oracle 8i Enterprise Server.

6.1 Architecture

We have implemented a storage server system using a PC cluster. The storage system manages multiple virtual volumes (virtual disks). A virtual volume can be implemented using a single or multiple physical disk partitions. Similarly to other clustered storage system [35], our storage system runs on a cluster of four PCs. Each PC is 400 MHz Pentium II with 512 KBytes second level cache and 1 GB of main memory. All PCs are connected together using Giganet [22]. Clients communicate with storage server nodes using the Virtual Interface (VI) communication model [29]. The peak communication bandwidth is about 100 MBytes/sec and the one-way latency for a short message is about 10 microseconds [22]. Data transfer from Oracle's buffer to the storage buffer uses direct DMA without memory copying. Each PC runs Windows NT 4.0 operating system. The interrupt time for incoming messages is 20 microseconds. Each PC box holds seven 17 GBytes IBM SCSI disks, one of which is used for storing the operating system. 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.

We have implemented both MQ and LRU as the cache replacement algorithms. The parameters of the MQ algorithm are the same as described in the previous section. It uses eight queues. The history buffer size is four times the number of cache blocks. The $lifeTime$ parameter is set dynamically after the warmup phase and adjusted periodically using the statistic information [44].

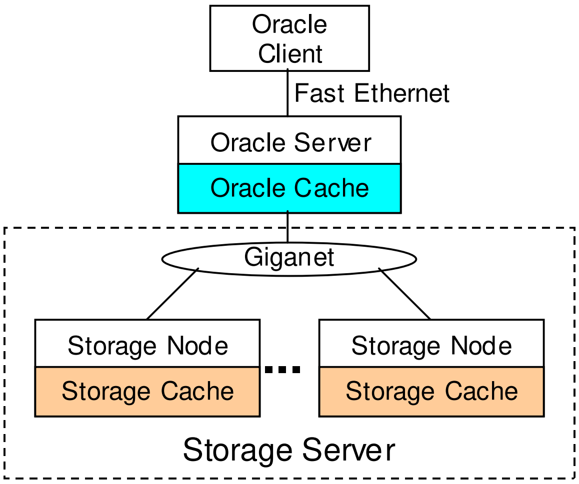


Figure 6: The architecture of a storage server.

We measure the performance using the TPC-C benchmark [27] running on the Oracle 8i Enterprise Server [5]. Figure 6 gives the architecture of our experiments. The hardware and software setups are similar to those used for collecting the Oracle Miss Trace-128M. The Oracle 8i 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 Oracle server are forwarded to the storage system through Giganet. The Oracle buffer cache is configured to be 128 MBytes. Other parameters of the Oracle Server are well tuned to achieve the best TPC-C performance. Each test runs the TPC-C script on an Oracle client machine for 2 hours. The Oracle client also runs on a separate PC which connects to the Oracle server through Fast Ethernet. The TPC-C script is provided by the Oracle Corporation. It simulates 48 clients, each of which generates transactions to the Oracle server. The TPC-C benchmark emulates the typical transaction processing of warehouse inventories. Our database contains 256 warehouses and occupies 100 GBytes disk space excluding logging disks. Logging disk data is not cached in the storage system. The storage system employs a write-through cache policy.

Storage cache size	MQ	LRU
128MB	19.85	8.85
256MB	31.42	17.66
512MB	44.34	31.69

Table 4: Percentage hit ratios of the storage buffer cache. The Oracle buffer cache (first level buffer cache) size is always 128 Mbytes.

6.2 Results

Table 4 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 than 10%, which validates our simulation study. The small difference is mainly caused by two reasons. 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, the average access time decreases. As a result, more I/O requests are forwarded to the storage system.

As shown on Table 4, MQ achieves much higher hit ratios than LRU. For a 512 MBytes storage buffer cache, MQ has a 12.65% higher hit ratio than LRU. Since miss penalty dominates the average access time, we use the relative miss ratio difference to estimate the upper bound of MQ's improvement on the end performance. For a 512 MBytes buffer cache, the relative miss ratio difference between MQ and LRU is 18.5% $(12.65/(100-31.69))$. Therefore, the upper bound for end performance improvement with MQ over LRU is 18.5%.

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 MBytes cache is slightly greater than that of LRU with a 256 MBytes cache. The hit ratio of MQ with a 256 MBytes cache is about the same as LRU with a 512 MBytes cache.

Figure 7 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 8-11%. 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 MBytes buffer cache using the LRU replacement algorithm. Because of the high hit ratios, the MQ algorithm improves the transaction rates over LRU by 8%, 12%, and 10% for 128 MBytes, 256 MBytes and 512 MBytes cache sizes respectively.

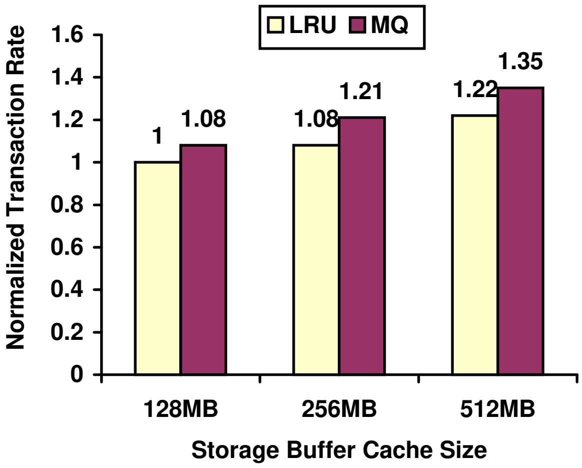


Figure 7: Normalized TPC-C transaction rate with different storage buffer cache sizes (All numbers are normalized to the transaction rate achieved by LRU with an 129 MBytes storage buffer cache).

Similar to the cache hit ratio improvement, using the MQ algorithm is equivalent to using LRU with a double sized cache. With a 128 MBytes buffer cache, MQ increases the transaction rate by 8%, which is exactly same as the improvement achieved by LRU with a 256 MBytes buffer cache. MQ with a 256 MBytes cache achieves a similar transaction rate to LRU with a 512 MBytes cache.

7 Related Work

A large body of literature has examined cache replacement algorithms. Examples of buffer cache replacement algorithms include the LRU [9, 6], GCLOCK [36, 19], First in First Out (FIFO), MRU, LFU, Random, FBR [31], LRU-k [15], 2Q [23], and LFRU [14]. In the spectrum of off-line algorithms, Belady's OPT algorithm and WORST algorithm [2, 17] are widely used to derive a lower and upper bound on the cache miss rate. Other closely related works include Muntz and Honeyman's file server caching study [28] and Eager and Bunt's disk cache study [43]. Most of these works have been described in our Introduction and Methodology sections.

Cache replacement policies have been intensively studied in various contexts in the past, including processor caches [38], paged virtual memory systems [36, 3, 40, 4, 12, 4, 34, 13, 10], and disk caches [37]. Although several studies [1, 20, 24] focus on 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 [21, 25].

Many past studies have used metrics such as LRU stack distance [17], marginal distribution of stack distances [18] or distance string models [39] to analyze the temporal locality of programs. However, the proposed LRU stack distance models were designed specifically for stack replacement algorithms like LRU. Moreover, distance string models do not capture the long-range relationships among references. O'Neil and et. al. recently proposed the inter-reference gap (IRG) model [30] to characterize temporal localities in program behavior. The IRG value for an address in a trace represents the time interval between successive references to the same address. But this model looks at each address separately and does not look at the overall distribution of the IRG values. Thesefore, it cannot well capture global access behavior.

Our study uses the distribution of temporal distances to measure temporal locality. The idea of using multiple queues with feedback has appeared in process scheduling [26, 41]. With this method, the priority of a process increases on an I/O event and decreases when its time slice expires without an I/O event.

8 Conclusions

Our study of second level buffer cache access patterns has uncovered two important insights. First, accesses to server buffer caches have relatively long temporal distances, unlike those to client buffer caches, which are much shorter. Second, access frequencies are distributed unevenly; some blocks are accessed significantly more often than others.

These two insights helped us identify three key properties that a good server buffer cache replacement algorithm should possess: *minimal lifetime, frequency-based priority, and temporal frequency.* Known replacement algorithms such as LRU, MRU, LFU, FBR, LRU-2, LFRU, and 2Q do not satisfy all three properties; however, our new algorithm, Multi-Queue (MQ), does.

Our simulation results show that the MQ algorithm performs better than other on-line algorithms and that it is robust for different workloads and cache sizes. In particular, MQ performs substantially better than the FBR algorithm which was the best algorithm in a previous study [43]. In addition, another interesting result of our study is that the 2Q algorithm, which does not perform as well as other algorithms for single level buffer caches [14], outperforms them for second level buffer caches, with the exception of MQ.

We have implemented the Multi-Queue and LRU algorithms on a storage server using an Oracle 8i Enterprise Server as the client. The 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 8-12% over LRU. For LRU to achieve a similar level of performance, the cache size needs to be doubled.

Our study has two limitations. First, we implemented only MQ and LRU replacement algorithms on our storage system. It would be interesting to compare these with other algorithms. Second, this paper assumes that the only information an L2 buffer cache algorithm has is the misses from the L1 buffer cache. It is conceivable that the L1 buffer cache might pass hints to the L2 cache in addition to the misses themselves. We have not explored this possibility.

References

[1] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *15th ISCA*.

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

[3] R. W. Carr and J. L. Hennessy. WSClock - A Simple and Effective Algorithm for virtual Memory Management. In *SOSP*, 1981.

[4] H. T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, 1985.

[5] Oracle Co. *Oracle 8i Concepts*.

[6] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. 1973.

[7] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide. 1999.

[8] IBM Corporation. White Paper: ESS-The Performance Leader. 1999.

[9] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5), May 1968.

[10] P. J. Denning. Virtual Memory. *ACM Computing Surveys*, 28(1), March 1996.

[11] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3), March 1972.

[12] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4), December 1984.

[13] C. Lee et.al. HiPEC: High Performance External Virtual Memory Caching. In *1st OSDI*, 1994.

[14] D. Lee et.al. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *SIGMETRICS-99*, pages 134–143, 1999.

[15] E.J. O'Neil et.al. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD-93*, 1993.

[16] M. D. Dahlin et.al. A Quantitative Analysis Scalability for Network File Systems. In *SIGMETRICS-94*, 1994.

[17] R. L. Mattson et.al. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.

[18] V. Almeida et.al. Characterizing reference locality in the WWW. In *PDIS-96*, 1996.

[19] V. F. Nicola et.al. Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. In *SIGMETRICS-92*, 1992.

[20] W.-H. Wang et.al. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *16th ISCA*, 1989.

[21] J. Gecsei and J. A. Lukes. A Model for the Evaluation of Storage Hierarchies. *IBM Systems Journal*, 13(2):163–178, 1974.

[22] Giganet Inc. Giganet.

[23] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB-94*, 1994.

[24] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, 1990.

[25] C. Lam and S. E. Madnick. Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data. *ACM Transactions on Database Systems*, 4(3):345–367, September 1979.

[26] B. W. Lampson. A Scheduling Philosophy for Multiprocessing Systems. *Communications of the ACM*, 11(5), 1968.

[27] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. *SIGMOD Record*, 22(2), June 1993.

[28] 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 Technical Conference*, 1992.

[29] VI Architecture Organization. Virtual Interface Architecture Specification version 1.0. 1997.

[30] V. Phalke and B. Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *SIGMETRICS-95*, 1995.

[31] J. Robinson and M. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *SIGMETRICS-90*, 1990.

[32] 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, CA, USA, April 5 1993.

[33] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *USENIX-93*, 1993.

[34] G. M. Sacco and M. Schkolnick. Buffer Management in Relational Database Systems. *ACM Transactions on Database Systems*, 11(4), December 1986.

[35] R. A. Shillner and E. W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Princeton University CS Department, 1996.

[36] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3), September 1978.

[37] A. J. Smith. Disk cache - miss ratio analysis and design considerations. *TOCS*, 3, 1985.

[38] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[39] J. R. Spirn. Distance string models for program behavior. *Computer*, 9(11), November 1976.

[40] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), July 1981.

[41] A. S. Tanenbaum. *Modern Operating Systems.* 1992.

[42] Transaction Processing Performance Council. *TPC Benchmark C.* May 1991.

[43] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *ICDCS*, May 1993.

[44] Yuanyuan Zhou. Memory Management for Networked Servers (Thesis). Technical report, Princeton University, Computer Science Department, November 2000.