# Second-Tier Cache Management Using Write Hints

Xuhui Li
*University of Waterloo*

Ashraf Aboulnaga
*University of Waterloo*

Kenneth Salem
*University of Waterloo*

Aamer Sachedina
*IBM Toronto Lab*

Shaobo Gao
*University of Waterloo*

## Abstract

Storage servers, as well as storage clients, typically have large memories in which they cache data blocks. This creates a two-tier cache hierarchy in which the presence of a first-tier cache (at the storage client) makes it more difficult to manage the second-tier cache (at the storage server). Many techniques have been proposed for improving the management of second-tier caches, but none of these techniques use the information that is provided by *writes* of data blocks from the first tier to help manage the second-tier cache. In this paper, we illustrate how the information contained in writes from the first tier can be used to improve the performance of the second-tier cache. In particular, we argue that there are very different reasons why storage clients write data blocks to storage servers (e.g., cleaning dirty blocks vs. limiting the time to recover from failure). These different types of writes can provide strong indications about the current state and future access patterns of a first-tier cache, which can help in managing the second-tier cache. We propose that storage clients inform the storage servers about the types of writes that they perform by passing *write hints*. These write hints can then be used by the server to manage the second-tier cache. We focus on the common and important case in which the storage client is a database system running a transactional (OLTP) workload. We describe, for this case, the different types of write hints that can be passed to the storage server, and we present several cache management policies that rely on these write hints. We demonstrate using trace driven simulations that these simple and inexpensive write hints can significantly improve the performance of the second-tier cache.

## 1 Introduction

Current storage servers have large memories which they use to cache data blocks that they serve to their clients. The storage clients, in turn, typically cache these data blocks in their own memories. This creates a *two-tier cache hierarchy* in which both the storage server and the storage client cache the same data with the goal of improving performance.[1]

Managing the *second-tier* (storage server) cache is more difficult than managing the *first-tier* (storage client) cache for several reasons. One reason is that the first-tier cache captures the accesses to the hot blocks in the workload. This reduces the temporal locality in the accesses to the second-tier cache, which makes recency-based replacement policies (e.g., LRU or Clock) less effective for the second tier.

Another reason why managing second-tier caches is difficult is that the second-tier cache may include blocks that are already present in the first-tier cache. Accesses to these blocks would hit in the first tier, so caching them in the second tier is a poor use of available cache space. Hence, second-tier cache management has the additional requirement of trying to maintain *exclusiveness* between the blocks in the first and second tiers [20].

Managing second-tier caches is also difficult because the cache manager needs to make placement and replacement decisions without full knowledge of the access pattern or cache management policy at the first tier. For example, a request to the second-tier for a block indicates a first-tier miss on that block, but does not provide information on how many first-tier hits to the block preceded this miss.

The difficulty of managing second-tier caches has been recognized in the literature, and various techniques for second-tier cache management have been proposed. Examples of these techniques include:

- Using cache replacement policies that rely on frequency as well as recency to manage second-tier caches [22].
- Passing hints from the storage client to the storage server about which requested blocks are likely to be retained and which are likely to be evicted [8, 5].

- Using knowledge of the algorithms and access patterns of the storage client to prefetch blocks into the second-tier cache [18, 2].

- Placing blocks into the second-tier cache not when they are *referenced* but when they are *evicted* by the first-tier cache [20, 6, 21].

- Evicting blocks requested by the first tier quickly from the second-tier cache [8, 20, 2].

- Using a single cache manager to manage both the client and the server caches [11].

Some of these techniques place extra responsibilities on the storage client for managing the storage server cache, and therefore require modifying the storage client [8, 20, 11, 5]. Other techniques do not require any modifications to the storage client, but spend CPU and I/O bandwidth trying to infer the contents of the storage client cache and predict its access patterns [1, 6, 2]. A common characteristic of all these techniques is that they do not have any special treatment for *writes of data blocks* from the storage client to the storage server.

In this paper, we focus on using *write requests* from the storage client to improve the performance of the storage server cache. Storage clients write data blocks to the storage server for different reasons. For example, one reason is writing a dirty (i.e., modified) block while evicting it to make room in the cache for another block. Another, very different, reason is periodically writing frequently modified blocks to guarantee reliability. The different types of writes provide strong indications about the state of the first-tier cache and the future access patterns of the storage client, and could therefore be used to improve cache management at the storage server.

We propose associating with every write request a *write hint* indicating its type (i.e., why the storage client is writing this block). We also present different methods for using these write hints to improve second-tier cache replacement, either by adding hint-awareness to existing replacement policies (e.g., MQ [21] and LRU) or by developing new hint-based replacement policies.

Our approach requires modifying the storage client to provide write hints. However, the necessary changes are simple and cheap. In particular, we are not requiring the storage client to make decisions about the management of the second-tier cache. We are only requiring the storage client to choose from a small number of explanations of why it is writing each block it writes, and to pass this information to the storage server as a write hint.

The write hints that we consider in this paper are fairly general, and could potentially be provided by a variety of storage clients. However, to explore the feasibility and efficacy of the proposed write hints, we focus on one common and important scenario: a database management system (DBMS) running an on-line transaction processing (OLTP) workload as the storage client (Figure 1). For this scenario, we demonstrate using trace driven simulations that write hints can improve the performance of the recently proposed MQ cache replacement policy by almost 30%, and that TQ, a new hint-based replacement policy that we propose, can perform twice as well as MQ.

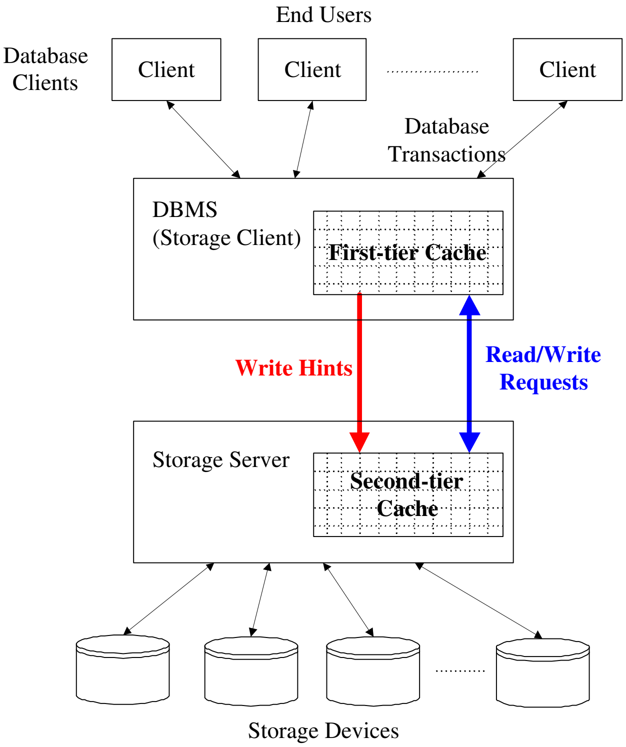


Figure 1: DBMS as the Storage Client

Our approach, while not transparent to the storage client, has the following key advantages:

- It is simple and cheap to implement at the storage server. There is no need to simulate or track the contents of the first-tier cache.

- It is purely opportunistic, and does not place additional load on the storage devices and network. When the storage server receives a write request, the request (a) contains a copy of the data to be written, and (b) must be flushed to the storage device at some point in time. Thus, if the second-tier cache manager decides, based on the write hints, to cache the block contained in a write request, it does not need to fetch this block from the storage device. On the other hand, if the second-tier cache manager decides not to cache the block contained in a write request, it has to flush this block to the storage device, but this flushing operation must be performed in any case, whether or not hints are used.

- As mentioned earlier, the first-tier cache typically captures most of the temporal locality in the workload. Thus, many reads will be served from the first-tier cache. Writes, on the other hand, must go to the second tier. Thus, the second-tier cache will see a higher fraction of writes in its workload than if it were the only cache in the system. This provides many opportunities for generating and using write hints.
- Using write hints is complementary to previous approaches for managing second-tier caches. We could exploit other kinds of hints, demotion information, or inferences about the state of the first-tier cache in addition to using the write hints.
- If the workload has few writes (e.g., a decision-support workload), the behavior of the proposed hint-aware replacement policies will degenerate to that of the underlying hint-oblivious policies. In that case, we expect neither benefit nor harm from using write hints.

Our contributions in this paper can be summarized as follows. We propose different types of write hints that can be generated by storage clients, and we propose second-tier cache replacement policies that exploit these hints. We evaluate the performance of these policies using traces collected from a real commercial DBMS running the industry standard TPC-C benchmark, and we compare them to the hint-oblivious alternatives. We also study an *optimal* replacement technique to provide an upper bound on how well we can do at the second tier.

The rest of this paper is organized as follows. In Section 2, we give some background about the architecture of a modern DBMS and its characteristics as a storage client. In Section 3, we present our proposal for using write hints, and in Section 4, we present three cache replacement policies that use these hints. Section 5 presents an evaluation of the proposed policies. Section 6 provides an overview of related work. We present our conclusions in Section 7.

## 2 Background

The I/O workload experienced by a storage server depends on the properties of its clients. Since we are considering a scenario in which the storage client is a DBMS, we first present, in this section, the relevant aspects of the process architecture and buffer management of a modern commercial DBMS. The specifics of this presentation are taken from DB2 Universal Database [9]. However, similar features are found in other major commercial and open-source database management systems.

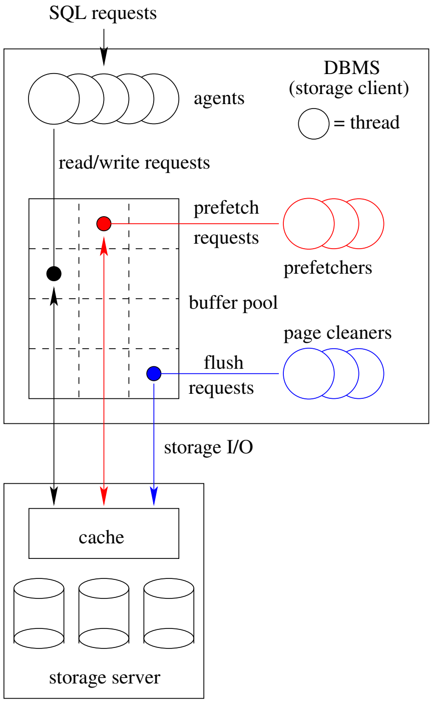


Figure 2: DBMS Architecture

Figure 2 provides a simplified illustration of the multi-threaded (or multi-process, depending on the platform) execution architecture of the DBMS. The DBMS is capable of processing several application SQL requests concurrently. One or more threads, known as agents, are used to execute each SQL statement. As the agents run, they read and update the database structures, such as tables and indexes, through a block-oriented buffer pool. The DBMS may actually maintain several, independently managed buffer pools (not illustrated in Figure 2). Together, these pools constitute the storage client cache.

Each buffer pool is managed using a clock-based algorithm, so recency of reference is important in replacement decisions. However, the replacement policy also considers a number of other factors, including the type of data in the block, whether the block is clean or dirty, and the expected access pattern of the last agent to have used the block. Blocks are loaded into the buffer pool on demand from agents. Depending on the type of query being executed, prefetching may also be employed as a means of removing demand paging delays from the critical paths of the agents. Agents send read-ahead requests to a prefetching queue, which is serviced by a pool of prefetching threads. Prefetching threads retrieve blocks from the underlying storage system and load them into the buffer pool, replacing blocks as necessary.

As agents run, they may modify the contents of blocks that are cached in the buffer pools. Modified (dirty) data blocks are generally not written immediately to the underlying storage system. Instead, one or more threads known as *page cleaners* are used to implement asynchronous (with respect to the agents) copy-back of dirty blocks from the buffer pool. In the event that the buffer replacement policy calls for the eviction of an updated block that has not been cleaned by a page cleaner, the agent (or prefetcher) that is responsible for the replacement flushes (writes) the dirty block back to the underlying storage system before completing the replacement. Note that flushing a dirty block does not by itself remove that block from the buffer pool. It simply ensures that the underlying storage device holds an up-to-date copy of the block.

The page cleaners must choose which dirty blocks to copy back to the storage system. There are two issues which affect this choice. First, the page cleaners try to ensure that blocks that are likely to be replaced by the agents will be clean at the time of the replacement. This removes the burden and latency associated with flushing dirty blocks from the execution path of the agents. To accomplish this, the page cleaners try to flush dirty blocks that would otherwise be good candidates for replacement.

The second issue considered by the page cleaners is failure recovery time. The DBMS uses write-ahead logging to ensure that committed database updates will survive DBMS failures. When the DBMS is recovering from a failure, the log is replayed to recreate any updates that were lost because they had not been flushed to the underlying storage system prior to the failure. The amount of log data that must be read and replayed to recover the proper database state depends on the age of the *oldest* changes that are in the buffer pool at the time of the failure. By copying relatively old updates from the buffer pools to the storage system, the page cleaners try to ensure that a configurable recovery time threshold will not be exceeded.

Several aspects of these mechanisms are worth noting. First, block writes to the underlying storage system usually do not correspond to evictions from the DBMS buffer pools. Writes correspond closely to evictions only when they are performed synchronously, by the agents. However, in a well-tuned system, the page cleaners try to ensure that such synchronous block writes are rare. Thus, if management of the storage server cache depends on knowledge of evictions from the client cache, that knowledge must be obtained by some other means, e.g., through the introduction of an explicit DEMOTE operation [20]. Second, the replacement algorithm used to manage the DBMS buffer pool is complex and uses application-specific information. This poses a challenge to storage server cache managers that rely on simulation of the storage client as a means of predicting which blocks are in the client's cache [2].

## 3 Write Hints

As was noted in Section 1, we propose to use write requests to improve the performance of the storage server cache. Each write request generated by the storage client includes a copy of the block being written, so write requests provide low-overhead opportunities to place blocks into the storage server's cache. Furthermore, the fact that the storage client has written block $b$ to the storage server may also provide some clues as to the state of the storage client's cache. The storage server can exploit these hints to improve the exclusiveness of its cache with respect to the client's.

What can the storage server infer about the storage client from the occurrence of a write? One key to answering this question is the fact that there are several distinct reasons why the storage client issues write requests, as described in Section 2. The first reason is block replacement: if the client wants to replace block $b$ and $b$ has been updated, then the client must write $b$ back to the storage server before replacing it. We call such write requests *replacement writes*. The second reason for writing is to limit data loss in the event of a failure at the storage client. Thus, the storage client may write a block to the storage server even through that block is not a likely replacement candidate, in order to ensure the recoverability of changes that have been made to that block. We call such write requests *recoverability writes*.

A second key issue is the relationship between the time of the client's write of block $b$ and the time of $b$'s eviction from the client's cache. In some cases, the client writes a dirty block $b$ to the storage server because it is about to evict $b$ from its cache. In the DBMS architecture described in Section 2, such writes may be generated by the agent threads when they need to replace a dirty block in the buffer pool. We call these *eviction-synchronous writes*, or simply *synchronous writes*. In other cases, such as when pages are flushed by the page cleaners, the eviction of the block is not imminent, and in fact may not occur at all. We call these *eviction-asynchronous writes*, or simply *asynchronous writes*. Note that the distinction between synchronous and asynchronous writes and the distinction between replacement and recoverability writes are essentially orthogonal.

Assuming that the storage server could somehow make these distinctions, what kinds of hints could it take from write requests? We present several cases here.

- **synchronous writes:** A synchronous write of block $b$ indicates that $b$ is about to be evicted from the storage client's cache. If the storage server chooses to place $b$ into its cache, it can be confident that $b$ is not also in the storage client's cache.

- **asynchronous replacement writes:** An asynchronous replacement write of block $b$ indicates two things. First, $b$ is present in the storage client's cache. Second, the storage client is preparing $b$ for eventual eviction, although eviction may not be imminent. Thus, in this case, it is not obvious what the storage server should infer from the occurrence of the write. However, we observe that if the storage client is well-designed, an asynchronous replacement write does suggest that $b$ is quite likely to be evicted from the storage client cache in the near future. This is a weaker hint than that provided by a synchronous write. However, given that a well-designed client will seek to avoid synchronous writes, asynchronous replacement write hints may ultimately be more useful because they are more frequent.

- **asynchronous recoverability writes:** An asynchronous recoverability write of block $b$ indicates that $b$ is present in the storage client's cache and that it may have been present there for some time, since recoverability writes should target old unwritten updates. Unlike an asynchronous replacement write, a recoverability write of block $b$ does not indicate that $b$'s eviction from the storage client cache is imminent, so $b$ is a poor candidate for placement in the storage server cache.

To exploit these hints, it is necessary for the storage server to distinguish between these different types of writes. One possibility is for the server to attempt to infer the type of write based on the information carried in the write request: the source of the block, the destination of the block in the storage server, or the contents of the block. Another alternative is for the storage client to determine the type of each write and then label each write with its type for the benefit of the storage server. This is the approach that we have taken. Specifically, we propose that the storage client associate a *write hint* with each write request that it generates. A write hint is simply a tag with one of three possible values: SYNCH, REPLACE, or RECOV. These tags correspond to the three cases described earlier.

The necessity of tagging means that the use of write hints is not entirely transparent to the storage client. Thus, under the classification proposed by Chen et al [5], write hints would be considered to be an "aggressively collaborative" technique, although they would be among the least aggressive techniques in that category. On the positive side, only a couple of bits per request are required for tagging, a negligible overhead. More importantly, we believe that it should be relatively easy and natural to identify write types from within the storage client. As noted in Section 5, we easily instrumented DB2 Universal Database to label each write with one of the three possible write types described above. Moreover, the types of write requests that we consider are not specific to DB2. Other major commercial database management systems, including Oracle [17] and Microsoft SQL Server [14], distinguish recoverability writes from replacement writes and try to do the writes asynchronously, resorting to synchronous writes only when necessary. Non-DBMS storage clients, such as file systems, also face similar issues. Finally, it is worth noting that the storage client does not need to understand how the storage server's cache operates in order to attach hints to its writes. Write hints provide information that may be useful to the storage server, but they do not specify how it should manage its cache.

## 4 Managing the Storage Server Cache

In this section, we discuss using the write hints introduced in Section 3 to improve the performance of second-tier cache replacement policies. We present techniques for extending two important cache replacement policies (LRU and MQ) so that they take advantage of write hints. We also present a new cache replacement algorithm that relies primarily on the information provided by write hints. But first, we address the question of how write hints can be used to achieve the goals of second-tier cache management.

### 4.1 Using Hints for Cache Management

Our goals in managing the second-tier cache are twofold. We want to maintain *exclusiveness* between the first- and second-tier caches, which means that the second tier should not cache blocks that are already cached in the first tier. At the same time, we want the second tier to cache blocks that will eventually be useful for the first tier. These are blocks whose re-reference distance (defined as the number of requests in the I/O stream between successive references to the block) is beyond the locality that could be captured in the first tier, and so will eventually miss in the first tier.

When choosing blocks to cache in the second tier, we should bear in mind that hits in the second tier are only useful for *read* requests from the first tier, but not write requests. Thus, the second-tier cache management policy should try to cache blocks that will cause read misses in the first tier.

We should also bear in mind that the second tier does not have to cache every block that is accessed by the first tier. The storage server could choose not to cache a block that is accessed, but rather to send the block from the storage device directly to the storage client (on a read miss), or from the client directly to the device (on a write).[2] This is different from other caching scenarios (e.g., virtual memory) in which the cache manager must cache every block that is accessed. Thus, storage server cache management has an extra degree of flexibility when compared to other kinds of cache management: when a new block arrives and the cache is full, the cache manager can evict a block to make room for the new block, or it can choose *not to cache the new block*.

With these points in mind, we consider the information provided by SYNCH, REPLACE, and RECOV write requests and also by read requests (which we label READ). SYNCH and REPLACE writes of a block $b$ indicate that the block will be evicted from the first tier, so they provide hints that $b$ should be cached in the second tier, with SYNCH providing a stronger hint than REPLACE. Caching $b$ in the second tier will not violate exclusiveness, and future read accesses to $b$, which most likely will miss in the first tier, will hit in the second tier.

Conversely, a READ request for block $b$ indicates that $b$ will have just been loaded into the first-tier cache. We cannot determine from the READ request how long $b$ will be retained in the first-tier cache. If recency-of-use plays a role in the storage client's cache management decisions, then we can expect that $b$ will be a very poor candidate for caching at the storage server, as it is likely to remain in the client's cache for some time. On the other hand, the client's cache manager may take factors besides recency-of-use into account in deciding to evict $b$ quickly. For example, if $b$ is being read as part of a large sequential table scan performed by a database system then $b$ may be quickly evicted from the client, and potentially re-referenced.

RECOV writes provides little information to the storage server cache. On the one hand, the written block is known to be in the storage client cache, which makes it a poor candidate for caching at the server. On the other hand, a RECOV write of $b$ indicates that $b$ has probably been in the storage client cache for a long time. Thus, the RECOV write does not provide as strong a negative hint as a READ.

Next, we illustrate how two important cache replacement policies (LRU and MQ) can be extended to take advantage of hints, and we present a new algorithm that relies primarily on request types (i.e., hints) to manage the cache.

## 4.2 LRU+Hints

We extend the least recently used (LRU) cache replacement policy by using hints to manage the LRU list and to decide whether or not to cache accessed blocks. We consider a simple extension: we cache blocks that occur in SYNCH or REPLACE write requests, since such blocks are likely to be evicted from the storage client cache. Blocks that occur in RECOV write requests or READ requests are not added to the cache.

Specifically, in the case of a SYNCH or REPLACE write for block $b$, we add $b$ to the cache if it is not there and we move it to the most-recently-used (MRU) end of the LRU list. If a replacement is necessary, the LRU block is replaced. In the case of a RECOV or READ request for block $b$, we make no changes to the contents of the cache or to the recency of the blocks, except during cold start, when the cache is not full. During cold start, RECOV and READ blocks are cached and placed at the LRU end of the LRU list. Of course, in the case of a READ request, the server checks whether the requested block is in its cache, and it serves the requested block from the cache in case of a hit. This hint-aware policy is summarized in Algorithm 1.

## 4.3 MQ+Hints

The Multi-Queue (MQ) [21] algorithm is a recently proposed cache replacement algorithm designed specifically for second-tier cache management. It has been shown to perform better than prior cache replacement algorithms, including other recently proposed ones such as ARC [13] and LIRS [10]. The algorithm uses multiple LRU queues, with each queue representing a range of reference frequencies. Blocks are promoted to higher frequency queues as they get referenced more frequently, and when we need to evict a block, we evict from the lower frequency queues first. Thus, MQ chooses the block for eviction based on a combination of recency and frequency.

To implement its eviction policy, MQ tracks the recency and frequency of references to the blocks that are currently cached. MQ also uses an auxiliary data structure called the *out queue* to maintain statistics about some blocks that have been evicted from the cache, Each entry in the out queue records only the block statistics, not the block itself, so the entries are relatively small. The out queue has a maximum size, which is a configurable parameter of the MQ policy, and it is managed as an LRU list.

We extend the MQ algorithm with hints in the same way in which we extended LRU. If a request is a SYNCH or REPLACE, we treat it exactly as it would be treated under the original MQ algorithm. If the request is a READ,

**Algorithm 1** LRU+Hints

```
LRUWITHHINTS($b$ : block access)
if $b$ is already in the cache /* cache hit */
    then if $type(b)$ = SYNCH or $type(b)$ = REPLACE
            then move $b$ to the MRU end of the LRU list;
elseif $type(b)$ = SYNCH or $type(b)$ = REPLACE /* cache miss */
    then insert $b$ at the MRU end of the LRU list, evicting the LRU block to make room if needed;
elseif cache is not full /* cache miss and not SYNCH or REPLACE */
    then insert $b$ at the LRU end of the LRU list;
```

we check the queues for a hit as usual. However, the queues are not updated at all unless the cache is not full, in which case the block is added as it would be under the original algorithm. RECOV requests are ignored completely unless the cache is not full, in which case the block is added as in the original algorithm.

## 4.4 The TQ Algorithm

In this section, we present a new cache replacement algorithm that relies primarily on request types, as indicated by write hints, to make replacement decisions. We call this algorithm the *type queue (TQ) algorithm*. Among our hint-aware algorithms, TQ places the most emphasis on using request types (or hints) for replacement. We show in Section 5 that the TQ algorithm outperforms other candidate algorithms. TQ is summarized in Figure 3 and Algorithm 2.

As described earlier, blocks that occur in SYNCH and REPLACE write requests are good candidates for caching at the storage server, since there is a good chance that they will soon be evicted from the storage client. Blocks that are requested in READ requests are not likely to be requested soon, although we can not be certain of this. The TQ policy accounts for this by caching READ requests at the server, but at lower priority than SYNCH and REPLACE requests. Thus, if a block is read, we will retain it in the storage server cache if possible, but not at the expense of SYNCH or REPLACE blocks. RECOV writes provide neither a strong positive hint to cache the block (since the block is known to be at the client) nor a strong negative hint that the block should be removed from the server's cache. To reflect this, the TQ policy effectively ignores RECOV writes.

The TQ algorithm works by maintaining two queues for replacement. A high priority queue holds cached blocks for which the most recent non-RECOV request was a SYNCH or REPLACE write. A low priority queue holds cached blocks for which the most recent non-RECOV request was a READ. When a SYNCH or REPLACE request for block $b$ occurs, $b$ is added to the high priority queue if $b$ is not cached, or moved to the high priority queue if it is in the low priority queue. If $b$ is in the high priority queue and a READ request for $b$ occurs, then it is moved to the low priority queue. Thus, the sizes of these two queues are not fixed, and will vary over time depending on the request pattern. Replacements, when they are necessary, are always made from the low priority queue unless that queue is empty. If the low priority queue is empty, then replacements are made from the high priority queue.

RECOV writes are ignored, which means that they do not affect the contents of the cache or the order of the blocks in the two queues. The only exception to this is during cold start, when the cache is not full. During cold start, blocks that occur in RECOV write requests are added to the low priority queue if they are not already in the cache.

The low priority queue is managed using an LRU policy.[3] The high priority queue, which is where we expect most read hits to occur, is managed using a replacement policy that we call *latest predicted read*, or LPR. When block $b$ is placed into the high priority queue (because of a REPLACE or SYNCH write to $b$), the TQ algorithm makes a prediction, $nextReadPosition(b)$, of the time at which the next READ request for $b$ will occur. When block replacement in the high priority queue is necessary, the algorithm replaces the block $b$ with the latest (largest) $nextReadPosition(b)$.

This policy is similar in principle to the optimal off-line policy. However, unlike the off-line policy, LPR must rely on an imperfect prediction of $nextReadPosition(b)$. To allow it to make these predictions, the TQ algorithm maintains an estimate of the expected *write-to-read distance* of each block, which is the distance (number of cache requests) between a REPLACE or SYNCH write to the block and the first subsequent READ request for the block. When block $b$ is added to the high priority queue, $nextReadPosition(b)$ is set to the current cache request count plus the expected write-to-read distance for $b$.

The TQ policy uses a running average of all the past

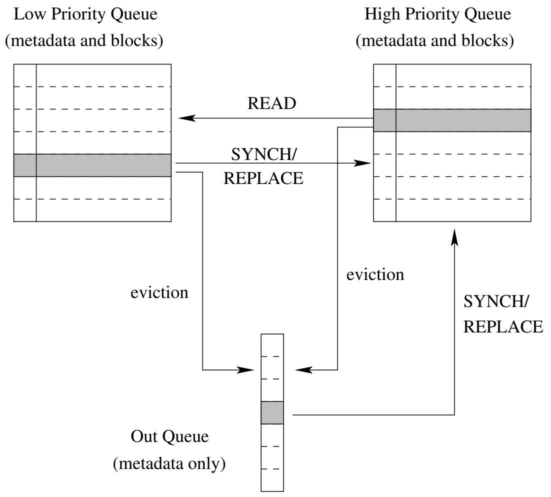


Figure 3: Structures used by TQ. Arrows show possible movements between queues in response to cache requests.

write-to-read distances of a block as its estimate of the expected write-to-read distance of this block. The policy maintains this running average of write-to-read distances for each block in the cache. In addition, like the MQ policy, TQ maintains an auxiliary data structure in which it tracks write-to-read distances and other reference statistics for a limited number of blocks that have previously been in the cache but have been evicted. For consistency with the terminology used by MQ, we call this data structure the TQ *out queue*. The maximum number of entries in the out queue is a parameter to the TQ algorithm. When an eviction from the out queue is necessary, the entry with the largest write-to-read distance is evicted.

When a block is added to the cache, TQ checks the out queue for an entry containing reference statistics about this block. If the block is found in the out queue, its write-to-read distance is obtained from the out queue, and the entry for the block is then removed from the out queue. If the block is not found in the out queue, its expected write-to-read distance in the cache is assumed to be infinite.

To maintain the running average of write-to-read distances for the blocks in the cache, TQ tracks the cache request count of the last REPLACE or SYNCH write request to each block in the high priority queue of the cache. This is done whether the request is bringing a new block into the cache, or whether it is a hit on a block already in the cache. When a READ request is a hit on a block in the high priority queue of the cache, the distance between this read and the most recent REPLACE or SYNCH request to this block is computed. The running average of write-to-read distances for this block is updated to include this new write-to-read distance.

When a block is evicted from the cache, an entry recording the expected write-to-read distance and the position of the most recent REPLACE or SYNCH write of this block is added to the out queue, and the out queue entry with the highest write-to-read distance is evicted to make room if necessary.

# 5 Evaluation

We used trace-driven simulations to evaluate the performance of the cache management techniques described in Section 4. The goal of our evaluation is to determine whether the use of write hints can improve the performance of the storage server cache. We also studied the performance of an optimal cache management technique to determine how much room remains for improvement.

## 5.1 Methodology

For the purposes of our evaluation, we used DB2 Universal Database (version 8.2) as the storage system client. We instrumented DB2 so that it would record traces of its I/O requests. We also modified DB2 so that it would record an appropriate write hint with each I/O request that it generates. These hints are recorded in the I/O trace records.

To collect our traces, we drove the instrumented DB2 with a TPC-C [19] OLTP workload, using a scale factor of 25. The initial size of the database, including all tables and indexes, is 606,317 4KB blocks, or approximately 2.3 Gbytes. The database grows slowly during the simulation run. The I/O request stream generated by DB2 depends on the settings of a variety of parameters. Table 1 shows the settings for the most significant parameters. We studied DB2 buffer pools ranging from 10% of the (initial) size of the database to 90% of the database size. The `softmax` and `chngpgs_thresh` parameters are important because they control the mix of write types in the request stream. The `chngpgs_thresh` gives the percentage of buffer pool pages that must be dirty to cause the page cleaners to begin generating replacement writes to clean them. The `softmax` parameter defines an upper bound on the amount of log data that will have to be read after a failure to recover the database. Larger values of `softmax` allow longer recovery times and result in fewer recoverability writes by the page cleaners. By fixing `chngpgs_thresh` at 50% (near DB2's default value) and varying `softmax`, we are able to control the mix of replacement and recoverability writes generated by the page cleaners.

Table 2 summarizes the traces that we collected and used for our evaluation. The 300_400 trace is our baseline trace, collected using our default DB2 parameter set-

**Algorithm 2** The TQ Algorithm

```
TQACCESS($b$ : block access)
/* for the sake of simplicity, this assumes that the cache and the out queue are already full */
if $type(b) = $ READ
   then if $b$ is in $Q_{high}$ /* $b$ is in high priority queue */
            then move $b$ to $Q_{low}$; /* move $b$ to low priority queue */
         /* if this READ follows a SYNCH or REPLACE, update write-to-read distance */
         if $b$ is in cache or $Q_{out}$ and $lastWritePosition(b) > 0$
            then update $avgWriteReadDist(b)$ using $(currentPosition - lastWritePosition(b))$;
                 $lastWritePosition(b) = 0$;
elseif $type(b) = $ SYNCH or $type(b) = $ REPLACE
   then if $b$ is in $Q_{low}$
            then $nextReadPosition(b) = currentPosition + avgWriteReadDist(b)$;
                 move $b$ to $Q_{high}$; /* move $b$ to high priority queue */
                 $lastWritePosition(b) = currentPosition$; /* remember when this write happened */
         elseif $b$ in in $Q_{out}$ and $lastWritePosition = 0$
            then $nextReadPosition(b) = currentPosition + avgWriteReadDist(b)$;
                 move $victim$ from cache to $Q_{out}$;
                 /* victim is LRU in $Q_{low}$, or latest $nextReadPosition$ in $Q_{high}$ if $Q_{low}$ is empty */
                 move $b$ to $Q_{high}$; /* put $b$ into high priority queue */
                 $lastWritePosition(b) = currentPosition$; /* remember when this write happened */
         elseif $b$ is not in $Q_{high}$ and $b$ is not in $Q_{out}$
            then remove $Q_{out}$ entry with largest $avgWriteReadDist$;
                 move $victim$ from cache to $Q_{out}$;
                 /* victim is LRU in $Q_{low}$, or latest $nextReadPosition$ in $Q_{high}$ if $Q_{low}$ is empty */
                 put $b$ into $Q_{high}$; /* put $b$ into high priority queue */
                 $lastWritePosition(b) = currentPosition$; /* remember when this write happened */
                 $nextReadPosition(b) = \infty$;
```

| Parameter | Our Default Value | Other Values | Description |
|---|---|---|---|
| `bufferpool size` | 300000 4KB blocks | 60000, 540000 blocks | size of the DBMS buffer pool |
| `softmax` | 400 | 50, 4000 | recovery effort threshold |
| `chngpgs_thresh` | 50% | - | buffer pool dirtiness threshold |
| `maxagents` | 1000 | - | maximum number of agent threads |
| `num_iocleaners` | 50 | - | number of page cleaner threads |

Table 1: DB2 Parameter Settings

| Trace Name | Buffer Pool Size in blocks | `softmax` | Number of Requests | Synch. Writes | Asynchronous Replacement Writes | Asynchronous Recoverability Writes | Reads |
|---|---|---|---|---|---|---|---|
| 300_400 | 300K (1.1 GB) | 400 | 13269706 | 0.00% | 62.57% | 3.60% | 33.83% |
| 60_400 | 60K (234 MB) | 400 | 15792519 | 0.08% | 48.89% | 0.18% | 50.85% |
| 540_400 | 540K (2.1 GB) | 400 | 12238848 | 0.00% | 35.78% | 49.89% | 14.33% |
| 300_4000 | 300K (1.1 GB) | 4000 | 13226138 | 0.01% | 65.37% | 0.11% | 34.51% |
| 300_50 | 300K (1.1 GB) | 50 | 15175377 | 0.00% | 0.03% | 74.33% | 25.64% |

Table 2: I/O Request Traces

tings. The remaining traces were collected using alternative buffer pool sizes and `softmax` settings. Not surprisingly, increasing the size of the DB2 buffer pool decreases the percentage of read requests in the trace (because more read requests hit in the buffer pool). Large buffer pools also tend to increase the frequency of recoverability writes, since updated pages tend to remain in the buffer pool longer. As discussed above, smaller values of `softmax` increase the prevalence of recoverability writes. The 300_50 trace represents a fairly extreme scenario with a very low `softmax` setting. This causes DB2 to issue a recoverability write soon after a page has been updated, so that recovery will be extremely fast. Although these settings are unlikely to be used in practice, we have included this trace for the sake of completeness.

We used these traces to drive simulations of a storage server buffer cache running the various algorithms described in Section 4. In addition, we implemented a variation of the off-line MIN algorithm [4], which we call OPT, as a means of establishing an upper bound on the hit ratio that we can expect in the storage server's buffer. Suppose that a storage server cache with capacity $C$ has just received a request for block $b$. The OPT algorithm works as follows:

- If the cache is not full, put $b$ into the cache.
- If the cache is full and it includes $b$, leave the cache contents unchanged.
- If the cache is full and it does not include $b$, then from among the $C$ blocks currently in the cache plus $b$, eliminate the block that will not be *read* for the longest time. Keep the $C$ remaining blocks in the cache.

Note that this algorithm may choose *not* to buffer $b$ at all if it is advantageous to leave the contents of the cache unchanged.

For the MQ, MQ+Hints, and TQ algorithm, we set maximum number of entries in the out queue to be equal to the number of blocks that fit into the server's buffer cache. Thus, for each of these algorithms, the server tracks statistics for the pages that are currently buffered, plus an equal number of previously buffered pages. We subtracted the space required for the out queue from the available buffer space for each of these algorithms so that our comparisons with LRU and LRU+Hints, which do not require an out queue, would be on an equal-space basis.

On each simulation run, we first allow the storage server's cache to fill. Once the cache is warm, we then measure the *read hit ratio* for the storage server cache. This is the percentage of read requests that are found in the cache.

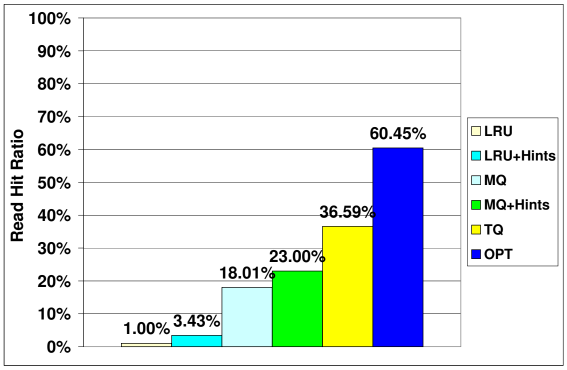


Figure 4: Read Hit Ratios in Storage Server Cache. Baseline (300_400) trace. Storage client cache size is 300K blocks (1.1 Gbytes), storage server cache size is 120K blocks (469 Mbytes).

## 5.2 Results: Baseline Case

Figure 4 shows the read hit ratios of the storage server cache under each of the techniques described in Section 4 for the baseline 300_400 trace with the storage server cache size set to 120K blocks (469 Mbytes). These results show that the LRU policy has very poor performance, which is consistent with other previous evaluations of LRU in second-tier caches [15, 21]. The LRU+Hints algorithm, which takes advantage of write hints, results in a hit ratio more than three times that of LRU, but it is still very low in absolute terms. The MQ algorithm, which considers frequency as well as recency, performs significantly better than LRU, and MQ+Hints improves on hint-oblivious MQ. The write hint based TQ algorithm provides the best performance, with a hit ratio nearly double that of MQ. TQ achieves more than half of the hit ratio of the off-line OPT algorithm.

## 5.3 Results: Sensitivity Analysis

We evaluated the sensitivity of the baseline results in Figure 4 to changes in three significant parameters: the size of the storage server cache, the size of the storage client cache (i.e., the DBMS buffer pool), and the value of the `softmax` parameter, which controls the mix of write types among the I/O requests.

Figure 5 shows the read hit ratio of the storage server cache as its size varies from 60K blocks (234 Mbytes) to 300K blocks (1.1 Gbytes), which is the size of the first-tier cache. Several observations can be made about these data. First, the relative advantage of the TQ algorithm is consistent until the server's cache reaches the largest size (300K blocks, 1.1 Gbytes) that we considered, at which point the advantage of TQ begins to diminish. For this large cache size, the improvement obtained by adding hints to MQ also becomes negligible.

However, for large cache sizes the performance of the simple LRU+Hints algorithm is much better than that of plain LRU, and comparable to that of TQ and the MQ policies. As the storage server cache gets smaller, the performance of LRU+Hints (and plain LRU) drops off quickly.

Figure 6 illustrates the impact of changing the storage client (DBMS) cache size, with the storage server cache size fixed at 120K blocks (469 Mbytes). These results show that management of the storage server cache becomes more difficult as the storage client cache becomes larger. Large storage client caches absorb most of the locality available in the request stream, leaving little for the storage server cache to exploit. Larger storage client caches also make it more difficult to maintain exclusiveness between the client and server caches. For very large client caches, the TQ algorithm performs more than five times better than the best hint-oblivious algorithm. However, *all* of the algorithms, including TQ, have poor performance in absolute terms, with read hit ratios far below that of the off-line OPT algorithm. When the storage client buffer is very small (60K blocks), all of the algorithms provide similar performance. In this case, the small storage client cache leaves temporal locality for the storage server cache to exploit, so that the difference between LRU and the remaining algorithms is not as great as it is when the client's cache is large.

Finally, Figure 7 shows the server cache read hit ratios as the `softmax` parameter increases from 50 to 4000. When `softmax` is very large (4000), the DBMS is effectively being told that long recovery times are acceptable. Under those conditions (trace 300_4000), the DBMS generates almost no recoverability writes; this is the primary difference between the baseline 300_400 trace and the 300_4000 trace. This has little impact on the performance of any of the algorithms.

At a `softmax` setting of 50, all of the hint-based algorithms, have similar performance, which is better than that of MQ and much better than LRU. When `softmax` is 50, almost three quarters of the I/O requests are recoverability writes, and there are no replacement writes. As was noted earlier, this represents an extreme scenario in which changes are flushed to the storage server almost immediately. As a result, this `softmax` setting generally gives poor overall system performance because of the substantial I/O write bandwidth that it requires, and is unlikely to be used in practice.

## 6 Related Work

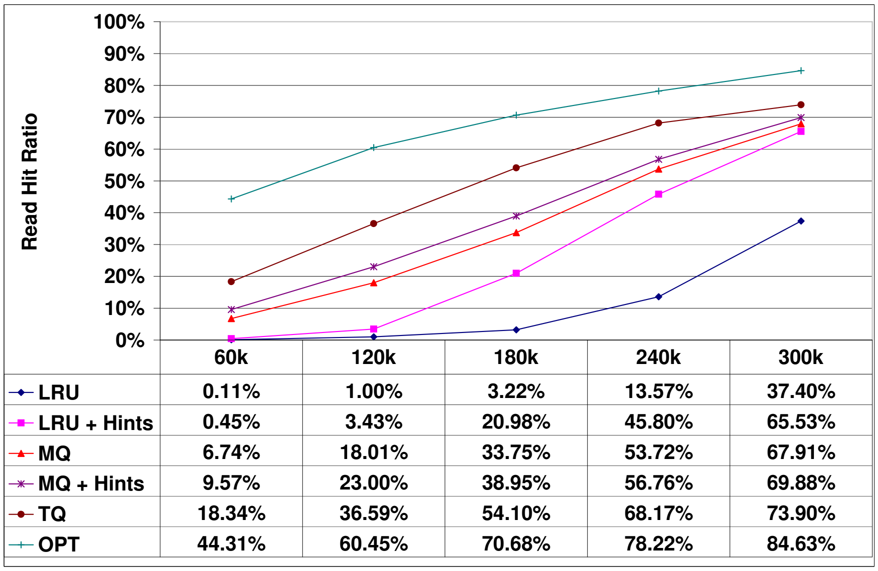
Classical, general-purpose replacement algorithms, such as LRU and LFU, rely on the recency and frequency of requests to each block to determine which blocks to replace. More recent general-purpose algorithms, such as 2Q [12], LRU-k [16], ARC [13], and CAR [3] improve on these classical algorithms, usually by balancing recency and frequency when making replacement decisions. Special purpose algorithms have been developed for use in database management systems [7] and other kinds of applications that cache data.

While any of the general-purpose algorithms can be used at any level of a cache hierarchy, researchers have recognized that cache management at the lower tiers of a hierarchy poses particular challenges, as was noted in Section 1. Zhou et al observed that access patterns at second tier caches are quite different from those at the first tier [21]. Muntz and Honeyman found that the second-tier cache in a distributed file system had low hit ratios because of this problem [15]. A second problem, pointed out by Wong and Wilkes, is that lower tier caches may contain many of the same blocks as upper tier caches [20]. This lack of exclusiveness wastes space and hurts the overall performance of the hierarchy.

Several general approaches to the problem of managing caches at the lower tiers in a hierarchy have been proposed. Since there is little temporal locality available in requests to second-tier caches, one strategy is to use a general-purpose replacement policy that is able to consider request frequency in addition to recency. Zhou, Philbin, and Li propose the multi-queue (MQ) algorithm (Section 4.3) to address this problem [22].

Although the MQ algorithm has been shown to be a better choice than LRU for managing a second-tier cache, the algorithm itself is not sensitive to the fact that it is operating in a hierarchy. Much of the work on caching in hierarchies focuses instead on techniques that are explicitly aware that they are operating in a hierarchy. One very simple technique of this type is to quickly remove from a lower-tier cache any block that is requested by an upper-tier, so that the block will not be cached redundantly [8, 5]. Other techniques involve tracking or simulating, at the second tier, certain aspects of the operation of the first-tier cache. One example of this is eviction-based caching, proposed by Chen, Zhou, and Li [6]. Under this technique, the second-tier cache tracks the target memory location of every block read by the first tier. This identifies where in the first tier cache each cached block has been placed. When the second-tier observes a new block being placed in the same location as a previously-requested block, in infers that the previously-requested block has been evicted from the first-tier cache and should be fetched into the second-tier cache. This places an extra load on the storage system, because it speculatively prefetches blocks.

The X-RAY mechanism takes a similar approach [2]. However, X-RAY assumes that the first tier is a file system, and it takes "gray-box" approach [1] to inferring the contents of the file system's cache. X-RAY can distin-



| | 60k | 120k | 180k | 240k | 300k |
|---|---|---|---|---|---|
| ◆ LRU | 0.11% | 1.00% | 3.22% | 13.57% | 37.40% |
| ■ LRU + Hints | 0.45% | 3.43% | 20.98% | 45.80% | 65.53% |
| ▲ MQ | 6.74% | 18.01% | 33.75% | 53.72% | 67.91% |
| ✳ MQ + Hints | 9.57% | 23.00% | 38.95% | 56.76% | 69.88% |
| ● TQ | 18.34% | 36.59% | 54.10% | 68.17% | 73.90% |
| + OPT | 44.31% | 60.45% | 70.68% | 78.22% | 84.63% |

Figure 5: Read Hit Ratios in the Storage Server Cache. Baseline (300_400) trace. Storage client cache size is 300K blocks, storage server cache size varies from 60K blocks to 300K blocks.

guish file meta-data (i-nodes) from file data. It inspects the meta-data when it is flushed to the tier-two cache, and it uses the resulting information (e.g., access and update timestamps) to predict which blocks are likely to be in the file system's cache. Sivathanu et al proposed a related technique called semantically-smart disks [18]. Like X-RAY, this assumes that the first tier is a file system. A probe process running against the file system allows the disk system at the second tier to discover, e.g., which blocks hold file system meta-data. It then uses this information to improve caching performance in the disk system.

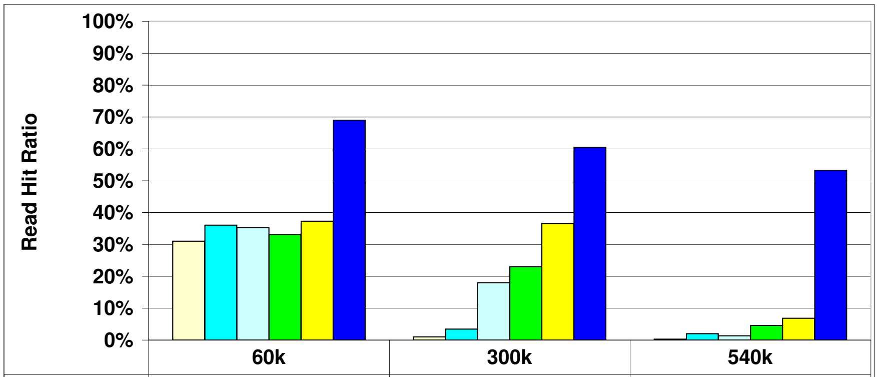
All of the techniques discussed above share the property that they are transparent to the first-tier cache, i.e., they can be deployed without modifying the code that manages the first tier. Chen et al called these techniques "hierarchically aware" [5]. Other techniques, called "aggressively collaborative" by the same authors, require some modification to the first-tier. Wong and Wilkes defined a DEMOTE operation that is issued by the first tier cache to send evicted blocks to the second tier [20]. This operation can be used to achieve the same effect as eviction-based caching, except that with DEMOTE it is not necessary for the second-tier to infer the occurrence of first-tier evictions. Another possibility is for the first tier to pass hints to the second tier. For example, Chen et al describe Semantics-Directed Caching, in which the first-tier cache provides hints to the second tier about the importance (to the first tier) of blocks that it requests [5]. Franklin et al propose a technique for collaboratively managing the caches at a database client and a database server, in which the client passes a hint to the server before it evicts a block, and the server can then ask the client to send it the block on eviction if the client has the only cached copy of this block [8].

The write hints proposed in this paper belong to the general class of "aggressively collaborative" techniques. However, they are complementary to previously proposed techniques of this class. For example, we could still exploit demotion information [20] or other kinds of hints [8, 5] while using write hints.

Another approach for managing two or more tiers of caches in a hierarchy is to use a single, unified controller. The Unified and Level-aware Caching (ULC) protocol controls a cache hierarchy from the first tier by issuing RETRIEVE and DEMOTE commands to caches at the lower tiers to cause them to move blocks up and down the cache hierarchy [11]. Zhou, Chen, and Li describe a similar approach, which they call "global" L2 buffer cache management, for a two-level hierarchy [5].

## 7 Conclusion

In this paper we observe that write hints can provide useful information that can be exploited by a storage server to improve the efficiency of its cache. We propose hint-aware versions of two existing hint-oblivious replacement policies, as well as TQ, a new hint-based policy. Trace-driven simulations show that the hint-aware policies perform better than the corresponding hint-oblivious policies. Furthermore, the new policy, TQ, had the best performance under almost all of the conditions that we studied.



| | 60k | 300k | 540k |
|---|---|---|---|
| □ LRU | 31.04% | 1.00% | 0.27% |
| 🟦 LRU + Hints | 36.04% | 3.43% | 2.01% |
| □ MQ | 35.26% | 18.01% | 1.32% |
| 🟩 MQ + Hints | 33.14% | 23.00% | 4.61% |
| 🟨 TQ | 37.28% | 36.59% | 6.84% |
| 🟦 OPT | 68.97% | 60.45% | 53.28% |

Figure 6: Read Hit Ratios in the Storage Server Cache. Traces 60_400, 300_400, and 540_400. Storage client cache size varies from 60K blocks (234 Mbytes) to 540K blocks (2.1 Gbytes). Storage server cache size is 120K blocks (469 Mbytes).

Our work focused on a configuration in which a DBMS, running an OLTP workload, acts as the storage client. In this common scenario, write hints are quite valuable to the storage server. The write hints themselves, however, are general, and reflect issues that must be faced by any type of storage client that caches data. Thus, we are optimistic that the benefits of write hints will extend to other types of storage clients that experience write-intensive workloads.

Possibilities for future work include investigating the use of write hints for other types of workloads or storage clients. They also include adding an aging mechanism to the TQ policy, and investigating avenues for the real world adoption of write hints, possibly through enhancements to the the SCSI interface.

## 8 Acknowledgments

We would like to thank Matt Huras and Calisto Zuzarte, from the IBM Toronto Lab, for their comments and assistance. This work was supported by IBM through the Center for Advanced Studies (CAS), and by Communications and Information Technology Ontario (CITO).

## References

[1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, October 2001.

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

[3] S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *Proc. of the 3nd USENIX Symposium on File and Storage Technologies (FAST'04)*, March 2004.

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

[5] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005.

[6] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based cache placement for storage caches. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 269–282, June 2003.

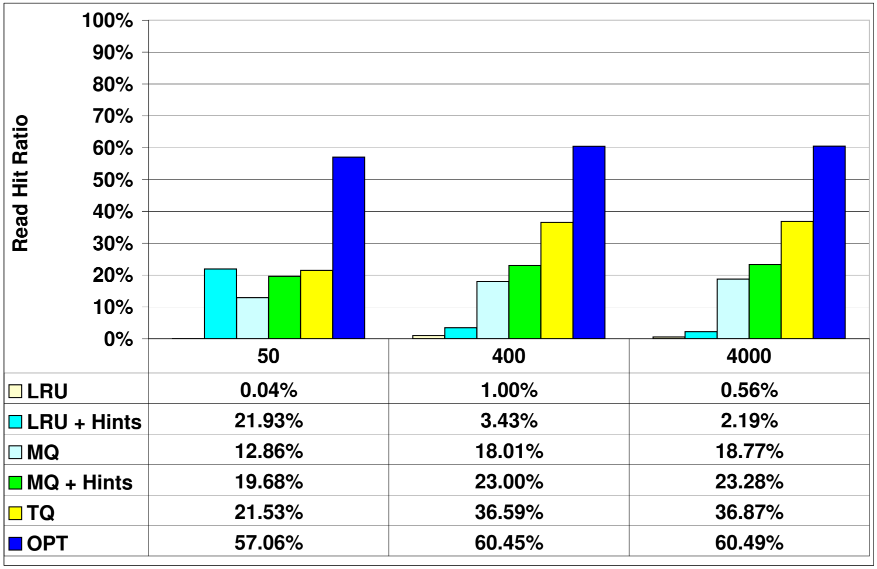
[7] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB)*, pages 127–141, August 1985.

[8] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In *Proc. International Conference on Very Large Data Bases (VLDB'92)*, pages 596–609, 1992.

[9] International Business Machines Corporation. *IBM DB2 Universal Database Administration Guide: Performance*, version 8.2 edition.

[10] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, pages 31–42, June 2002.

[11] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 168–177, March 2004.



| | 50 | 400 | 4000 |
|---|---|---|---|
| □ LRU | 0.04% | 1.00% | 0.56% |
| ■ LRU + Hints | 21.93% | 3.43% | 2.19% |
| □ MQ | 12.86% | 18.01% | 18.77% |
| 🟩 MQ + Hints | 19.68% | 23.00% | 23.28% |
| 🟨 TQ | 21.53% | 36.59% | 36.87% |
| 🟦 OPT | 57.06% | 60.45% | 60.49% |

Figure 7: Read Hit Ratios in the Storage Server Cache as `softmax` is varied. Traces 300_50, 300_400, and 300_4000. Storage client cache size is 300K blocks (1.1 Gbytes), storage server cache size is 120K blocks (469 Mbytes).

[12] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.

[13] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, March 2003.

[14] Microsoft Corporation. *Microsoft SQL Server 2000 Books Online.*

[15] 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 Conference*, pages 305–313, January 1992.

[16] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 297–306, 1993.

[17] Oracle Corporation. *Oracle Database Concepts*, version 10g release 2 (10.2) edition, June 2005.

[18] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003.

[19] Transaction Processing Performance Council. *TPC Benchmark C*, revision 5.4 edition, 2005.

[20] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference (USENIX 2002)*, pages 161–175, June 2002.

[21] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.

[22] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 91–104, June 2001.

## Notes

[1] We focus on two-tier cache hierarchies for clarity of presentation, but our discussion and proposed techniques extend to cache hierarchies with more than two tiers.

[2] Some short-term buffering may be required to accommodate transfer speed mismatches and request bursts. We have ignored this for the sake of simplicity.

[3] We expect that hits in the low priority queue will be uncommon, and that the behavior of the TQ policy will not be very sensitive to the replacement policy in the low priority queue.