CLIC: CLient-Informed Caching for Storage Servers

Abstract

Traditional caching policies are known to perform poorly for storage server caches. One promising approach to solving this problem is to use hints from the storage clients to manage the storage server cache. Previous hinting approaches are ad hoc, in that a predefined reaction to specific types of hints is hard-coded into the caching policy. With ad hoc approaches, it is difficult to ensure that the best hints are being used, and it is difficult to accommodate multiple types of hints and multiple client applications. In this paper, we propose CLient-Informed Caching (CLIC), a generic hint-based policy for managing storage server caches. CLIC automatically interprets hints generated by storage clients and translates them into a server caching policy. It does this without explicit knowledge of the application-specific hint semantics. We demonstrate using trace-based simulation of database workloads that CLIC outperforms hint-oblivious and state-of-the-art hint-aware caching policies, and that it scales well and effectively limits the cost of cache management.

1 Introduction

Multi-tier block caches arise in many situations. For example, running a database management system (DBMS) on top of a storage server results in at least two caches, one in the DBMS and one in the storage system. The challenges of making effective use of caches below the first tier are well known [14, 17, 20]. Poor temporal locality in the request streams experienced by the second-tier caches reduces the effectiveness of recency-based replacement polices [20], and failure to maintain *exclusivity* among the contents of the caches in each tier leads to wasted cache space [17].

Many techniques have been proposed for improving the performance of second-tier caches. Section 7 provides a brief survey. One promising class of techniques relies on *hinting*: the application that manages the first-tier cache generates hints and attaches them to the I/O requests that it directs to the second tier. The cache at the second tier then attempts to exploit these hints to improve its performance. For example, an *importance hint* [5] indicates the priority of a particular page to the buffer cache manager in the first-tier application. Given such hints, the second-tier cache can infer that pages that have high priority in the first tier are likely to be retained there, and can thus give them low priority in the second tier. As another example, a *write hint* [10] indicates whether the first tier is writing a page to ensure recoverability of the page, or to facilitate replacement of the page in the first-tier cache. The second tier may infer that replacement writes are better caching candidates than recovery writes, since they indicate pages that are eviction candidates in the first tier.

Hinting is valuable because it is a way of making application-specific information available to the second (or lower) tier, which needs a good basis on which to make its caching decisions. However, previous work has taken an *ad hoc* approach to hinting. The general approach is to identify a specific type of hint that can be generated from an application (e.g., a DBMS) in the first tier. A replacement policy that knows how to take advantage of this particular type of hint is then designed for the second tier cache. For example, the TQ algorithm [10] is designed specifically to exploit write hints. The desired response to each possible hint is hard-coded into such an algorithm.

Ad hoc algorithms can significantly improve the performance of the second-tier cache when the necessary type of hint is available. However ad hoc algorithms also have some significant drawbacks. First, because the response to hints is hard-coded into an algorithm at the second tier, any change to the hints requires changes to the cache management policy at the second-tier server. Second, even if change is possible at the server, it is difficult to generalize ad hoc algorithms to account for new situations. For example, suppose that applications can generate *both* write hints and importance hints. Clearly, a low-priority (to the first tier) replacement write is probably a good caching candidate for the second tier, but what about a low-priority recovery write? A related problem arises when multiple first-tier applications are served by a single cache in the second tier. If different applications generate hints, how is the second tier cache to compare them? Is a write hint from one application more or less significant than an importance hint from another?

In this paper, we propose CLient-Informed Caching (CLIC), a *generic technique* for exploiting application hints to manage a second-tier cache, such as a storage server cache. Unlike ad hoc techniques, CLIC does not hard-code responses to any particular type of hint. Instead, it is an adaptive approach that attempts to learn to exploit any type of hint that is supplied to it. Applications in the first tier are free to supply any hints that they believe may be of value to the second tier. CLIC analyzes the available hints and determines which can be exploited to improve second-tier cache performance. Conversely, it learns to ignore hints that do not help. Unlike ad hoc approaches, CLIC decouples the task of generating hints (done by applications in the first tier) from the task of interpreting and exploiting them. CLIC naturally accommodates applications that generate more than one type of hint, as well as scenarios in which multiple applications share a second-tier cache.

The contributions of this paper are as follows. First, we define an on-line cost/benefit analysis of I/O request hints that can be used to determine which hints provide potentially valuable information to the second-tier cache. Second, we define an adaptive, priority-based cache replacement policy for the second-tier cache. This policy exploits the results of the hint analysis to improve the hit ratio of the second-tier cache. Third, we use trace-based simulation to provide a performance analysis of CLIC. Our results show that CLIC outperforms ad hoc hinting techniques and that its adaptivity can be achieved with low overhead, even when the number of hints is large.

2 Generic Framework for Hints

We assume a system in which multiple storage server client applications generate requests to a storage server, as shown in Figure 1. We are particularly interested in client applications that cache data, since it is such applications that give rise to multi-tier caching.

The storage server's workload is a sequence of block I/O requests from the various clients. When a client sends an I/O request (read or write) to the server, it may attach hints to the request. Specifically, each storage client may define one or more *hint types* and, for each such hint type, a *hint value domain*. When the client issues an I/O request, it attaches a *hint set* to the request. Each hint set consists of one hint value from the domain

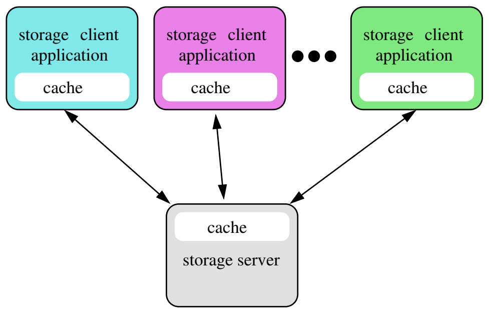
storage client application
cache
storage client application
cache
storage client application
cache
cache
storage server

Figure 1: System Architecture

of each of the hint types defined by that client. For example, we used IBM DB2 Universal Database[1] as a storage client application, and we instrumented DB2 so that it would generate five types of hints, as described in Figure 2. Thus, each I/O request issued by DB2 will have an attached hint set consisting of 5 hint values: a pool ID, an object ID, an object type ID, a request type, and a DB2 buffer priority.

CLIC does *not* require these specific hint types. We chose these particular types of hints because they could be generated easily from DB2, and because we believed that they might prove useful to the underlying storage system. Each application can generate its own types of hints. CLIC itself only assumes that the hint value domains are *categorical*. It neither assumes nor exploits any ordering on the values in a hint value domain. Each storage client application may have its own hint types. In fact, even if two storage clients are instances of the same application (e.g., two instances of DB2) and use the same hint types, CLIC treats each client's hint types as distinct from the hint types of all other clients.

3 Hint Analysis

Every I/O request, read or write, represents a caching opportunity for the storage server. The storage server must decide whether to take advantage of each such opportunity by caching the requested page. Our approach is to base these caching decisions on the hint sets supplied by the client applications with each I/O request. CLIC associates each possible hint set H with a numeric priority, $\texttt{Pr}(H)$. When an I/O request (read or write) for page p with attached hint set H arrives at the server, the server uses $\texttt{Pr}(H)$ to decide whether to cache p. Cache management at the server will be described in more detail in Section 4, but the essential idea is simple: the server caches p if there is some page p' in the cache that was requested with a hint set H' for which $\texttt{Pr}(H') < \texttt{Pr}(H)$.

We expect that some hint sets may signal pages that are likely to be re-used quickly, and thus are good caching

[1]DB2 Universal Database is a registered trademark of IBM.

Hint Type	Value Domain Cardinality	Description
pool ID	2	Identifies which DB2 buffer pool generated the I/O request.
object ID	21	Identifies a group of related database objects, such as a table and its associated indices.
object type ID	6	Distinguishes among the objects with a given object ID. Together, a pool ID, object ID and object type ID uniquely identify a database object, such as a table or index.
request type	5	For read requests, distinguishes regular reads from prefetch reads. For writes, provides write hints ([10]), which distinguish between recovery writes, replacement writes, and synchronous writes. Synchronous writes are replacement writes that are not performed by an asynchronous page cleaning thread.
buffer priority	4	Identifies the priority of the page in its DB2 buffer cache.

Figure 2: Types of Hints in the DB2 I/O Request Traces

candidates. Other hint sets may signal the opposite. Intuitively, we want the priority of each hint set to reflect these signals. But how should priorities be chosen for each hint set? One possibility is to assign these priorities, in advance, based on knowledge of the client application that generates the hint sets. Most existing hint-based caching techniques use this approach. For example, the TQ algorithm [10], which exploits write hints, understands that replacement writes likely indicate evictions in the client application's cache, and so it gives them high priority.

CLIC takes a different approach to this problem. Instead of predefining hint priorities based on knowledge of the storage client applications, CLIC assigns a priority to each hint set by *monitoring and analyzing I/O requests that arrive with that hint set*. Next, we describe how CLIC performs its analysis. To simplify the presentation, we will ignore, for now, the cost (in time and space) of performing the analysis. We will address the issue of *efficient* analysis in Section 5.

We will assume that each request that arrives at the server is tagged (by the server) with a sequence number. Suppose that the server gets a request (p, H), meaning a request (read or a write) for a page p with an attached hint set H, and suppose that this request is assigned sequence number s_1. CLIC is interested in whether and when page p will be requested again after s_1. There are three possibilities to consider:

write re-reference: The first possibility is that the *next* request for p in the request stream is a write request occurring with sequence number s_2 ($s_2 > s_1$). In this case, there would have been no benefit whatsoever to caching p at time s_1. A cached copy of p would not help the server handle the subsequent write request any more efficiently. A cached copy of p may be of benefit for requests for p that occur after s_2, but in that case the server would be better off caching p at s_2 rather than at s_1. Thus, the server's caching opportunity at s_1 is best ignored.

read re-reference: The second possibility is that the *next* request for p in the request stream is read request at time s_2. If the server caches p at time s_1 and keeps p in the cache until s_2, it will benefit by being able to serve the read request at s_2 from its cache. For the server to obtain this benefit, it must allow p to occupy one page "slot" in its cache during the interval $s_2 - s_1$.

no re-reference: The third possibility is that p is never requested again after s_1. In this case, there is clearly no benefit to caching p at s_1.

Of course, the server cannot determine which of these three possibilities will occur for any particular request, as that would require advance knowledge of the future request stream. Instead, we propose that the server base its caching decision for the request (p, H) on an analysis of previous requests with hint set H. Specifically, CLIC tracks three statistics for each hint set H:

$N(H)$**:** the total number of requests with hint set H.

$N_r(H)$**:** the total number requests with hint set H that result in a read re-reference (rather than a write re-reference or no re-reference).

$D(H)$**:** for those requests (p, H) that result in read re-references, the average number of requests that occur between the request and the read re-reference.

Using these three statistics, CLIC performs a simple benefit/cost analysis for each hint set H, and assigns higher priorities to hint sets with higher benefit/cost ratios. Suppose that the server receives a request (p, H) and that it elects to cache p. If a read re-reference subsequently occurs while p is cached, the server will have obtained a benefit from caching p. We arbitrarily assign a value of 1 to this benefit (the value we use does not affect the relative priorities of pages). Among all previous requests with hint set H, a fraction

$$f_{hit}(H) = N_r(H)/N(H) \tag{1}$$

eventually resulted in read re-references, and would have provided a benefit if cached. We call $f_{hit}(H)$ the *read hit rate* of hint set H. Since the value of a read re-reference is 1, $f_{hit}(H)$ can be interpreted as the expected benefit of caching and holding pages that are requested with hint set H. Conversely, $D(H)$ can be interpreted as the expected *cost* of caching such pages, as it measures how long such pages must occupy space in the cache before the benefit is obtained. We define the *caching priority* of hint set H as:

$$\mathtt{Pr}(H) = \frac{f_{hit}(H)}{D(H)} \tag{2}$$

which is the ratio of the expected benefit to the expected cost.

Figure 3 illustrates the results of this analysis for a trace of I/O requests made by DB2 during a run of the TPC-C benchmark. Our DB2 traces will be described in more detail in Section 6. Each point in Figure 3 represents a distinct hint set that is present in the trace, and describes the hint set's caching priority and frequency of occurrence. All hint sets with non-zero caching priority are shown. Clearly, some hint sets have much higher priorities, and thus much higher benefit/cost ratios, than others. For illustrative purposes, we have indicated partial interpretations of two of the hint sets in the figure. For example, the most frequently occurring hint set represents replacement writes to the STOCK table in the TPC-C database instance that was being managed by the DB2 client. We emphasize that CLIC does not need to understand that this hint represents the STOCK table, nor does it need to understand the difference between a replacement write and a recovery write. Its interpretation of hints is based entirely on the hint statistics that it tracks, and it can automatically determine that the shown STOCK table hint set is a better caching opportunity than the shown ORDERLINE table hint set.

To implement its hint set analysis, CLIC maintains two tables, called the *hint table* and the *page table*. The hint table contains one entry for each distinct hint set H that has been observed by the storage server. The hint table entry for H records the current values of the statistics $N(H)$, $N_r(H)$ and $D(H)$. The page table contains one entry for each distinct page that has appeared in the I/O request stream. The entry for page p contains two pieces of information: $\mathtt{seq}(p)$, which is the sequence number most recent request for p, and $\mathtt{H}(p)$, which is the hint set attached to the most recent request for p.

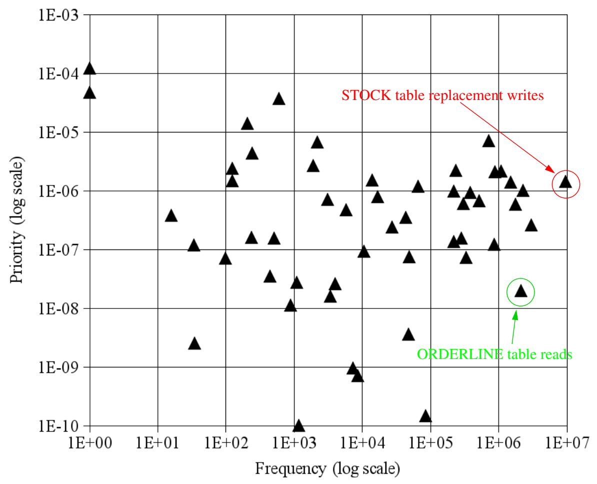

Figure 3: Hint Set Priorities for the 60_400 Trace
Each point represents a distinct hint set. All hint sets are shown.

CLIC updates the page table and hint table each time an I/O request is received. Suppose that the server receives a request (p, H) with sequence number s. If the request is a read, CLIC first checks whether the request is a read re-reference of p. If there is an entry $(\mathtt{seq}(p), \mathtt{H}(p))$ for p in the page table, then the current request is a read re-reference and CLIC increments $N_r(\mathtt{H}(p))$ and updates $D(\mathtt{H}(p))$ in the hint table based on the re-reference distance for this request, $s - \mathtt{seq}(p)$. After checking for a read re-reference, the server updates the entry for p in the page table, setting $\mathtt{seq}(p) = s$ and $\mathtt{H}(p) = H$, and increments $N(H)$ in the hint table.

4 Cache Management

In the previous section, we described how CLIC assigns a caching priority to each hint set H. In this section, we describe how the server uses these priorities to manage the contents of its cache.

Figure 4 describes CLIC's priority-based replacement policy. This policy evicts a lowest priority page from the cache if the newly requested page has higher priority. The priority of a page is determined by the priority $\mathtt{Pr}(H)$ of the hint set H with which that page was last requested. Note that if a page that is cached after being requested with hint set H is subsequently requested with hint set H', its priority changes from $\mathtt{Pr}(H)$ to $\mathtt{Pr}(H')$. The most recent request for each cached page, which is tracked in the page table, always determines its caching priority.

The policy described in Figure 4 can be implemented to run in constant expected time. To do this, CLIC maintains a heap-based priority queue of the hint sets. For each hint set H in the heap, all pages with $\mathtt{H}(p) = H$ are recorded in a doubly-linked list that is sorted by $\mathtt{seq}(p)$.

```
1    if p is not cached then
2      if the cache is not full then
3        cache p
4      else
5        let m be the minimum priority
6            of all pages in the cache
7        let v be the page with the minimum
8            sequence number seq(v) among
9            all pages with priority m
10       if Pr(H)>m then
11         evict v from the cache
12         cache p
13       else do not cache p
```

Figure 4: Hint-Based Server Cache Replacement Policy
This pseudo-code describes how the server handles a request for page p, with associated hint set H and request sequence number s.

This allows the victim page to be identified (Figure 4, lines 5-9) in constant time. CLIC also maintains a hash table of all cached pages so that it can tell which pages are cached (line 1). Finally, CLIC implements the hint table as a hash table so that it can look up $\mathtt{Pr}(H)$ (line 10) in constant expected time.

As it monitors each I/O request, CLIC updates its hint set statistics as described in Section 3. Since the priority $\mathtt{Pr}(H)$ of a each hint set H depends on these statistics, hint set priorities are effectively changing continuously as I/O requests are monitored. Rather than continuously adjusting the hint set priority queue to account for these changes, CLIC applies the changes lazily. Our implementation periodically recalculates $\mathtt{Pr}(H)$ for all hint sets H and then rebuilds the hint set priority queue based on the newly calculated priorities. Hint set priorities then remain unchanged until the next recalculation. Our implementation currently recalculates priorities after every one million I/O requests.

5 Efficient Hint Analysis

In Section 3, we ignored the cost of tracking the statistics required by CLIC. We return to this topic now. There are two potential threats to the efficiency of the analysis described in Section 3. The first threat comes from the page table, which is used to track information about the most recent reference for each page. In the worst case, the size of this table will be proportional to the total number of pages stored at the server. Thus, the space requirements of the page table could substantially reduce the amount of space available for caching, and in fact could overwhelm the cache completely.

We take a simple approach to handle this problem. Instead of tracking the most recent request for *every* page, we track the most recent request for each page that is currently in the cache, and for a fixed number (N_{outq}) of additional pages. Information ($\mathtt{seq}(p)$ and $\mathtt{H}(p)$) about the additional pages is stored in a data structure called the *outqueue*. N_{outq} is a CLIC parameter that can be used to bound the amount of space required for tracking re-reference distances.

When a page p is evicted from the cache, an entry for p is inserted into the outqueue. An entry is also placed in the outqueue for any requested page that CLIC elects not to cache (line 13 in Figure 4). If the outqueue is full when a new entry is to be inserted, the least-recently inserted entry is evicted from the outqueue to make room for the new entry.

Whenever an I/O request (p, H) occurs, CLIC checks both the cache and the outqueue to see if there is information about a previous request for p. If so, then the analysis proceeds as described in Section 3. If not, CLIC updates $N(H)$ as usual, but is unable to determine whether the current request is a read re-reference of page p. As a result, some error may be introduced into CLIC's read hit rate and read re-reference distance estimates. The N_{outq} parameter controls a tradeoff between space consumption and such estimation error.

Although some error is inevitable when the page table is replaced by an outqueue, this approach to tracking page re-references has several advantages. First, since CLIC tracks the most recent reference to all pages that are in the cache, we expect to have accurate re-reference distance estimates for hint sets that are believed to have the highest priorities, since pages requested with those hint sets will be cached. If the priority of such hint sets drops, CLIC should be able to detect this. Second, by evicting the oldest entries from the outqueue when eviction is necessary, CLIC will tend to miss read re-references that have long re-reference distances. Conversely, read re-references that happen quickly are likely to be detected. These are exactly the type of re-references that lead to high caching priority. Thus, the estimation procedure is biased in favor of read re-references that are likely to lead to high caching priority.

The second threat to the efficiency of CLIC is the number of possible hint sets for which CLIC must track statistics. Although the amount of statistical information tracked per hint set is small, the number of distinct hit sets from each client might be as large as the product of the cardinalities of that client's hint value domains. In our DB2 traces, the number of possible hint sets is 5040, and the number that we actually observed in each trace is an order of magnitude smaller than that. For another application, however, these numbers could be much larger.

We propose two techniques for restricting the number of hint sets that CLIC must consider, one based on hint set frequency and one based on generalization. We describe these techniques in Sections 5.1 and 5.2.

5.1 Frequently-Occurring Hint Sets

All of the hint types in our DB2 test traces exhibit frequency skew. That is, some values in the hint domain occur much more frequently than others. As a result, some hint sets occur much more frequently than others. One way to cope with large numbers of hint sets is to exploit this skew by tracking statistics for the hint sets that occur most frequently in the request stream and ignoring those that do not. Ignoring infrequent hint sets may lead to errors. In particular, we may miss a hint set that would have had high caching priority. However, since any such missed hint set would occur infrequently, the impact of the error on the server's caching performance is likely to be small.

The problem with this approach is that we must determine, on the fly, which hint sets occur frequently, without actually maintaining a counter for every hint set. Fortunately, this *frequent item problem* arises in a variety of settings, and numerous methods have been proposed to solve it. We have chosen one of these methods: the so-called *Space-Saving* algorithm [13], which has recently been shown to outperform other frequent item algorithms [6]. Given a parameter k, this algorithm tracks the frequency of k different hint sets, among which it attempts to include as many of the actual k most frequent hint sets as possible. It is an on-line algorithm which scans the sequence of hint sets attached to the requests arriving at the server. Although k different hint sets are tracked at once, the specific hint sets that are being tracked my vary over time, depending on the request sequence.

After each request has been processed, the algorithm can report the k hint sets that it is currently tracking, as well as an estimate of the frequency (total number of occurrences) of each hint set and an error indicator which bounds the error in the frequency estimate. By analyzing the frequency estimates and error indicators, it is possible to determine which of the k currently-tracked hint sets are guaranteed to be among the actual top-k most frequent hint sets and which are not. However, for our purposes this is not necessary.

We adapted the Space-Saving algorithm slightly so that it tracks the additional information we require for our analysis. Specifically:

$N(H)$**:** For each hint set H that is tracked by the Space-Saving algorithm, we use the frequency estimate produced by the algorithm as $N(H)$.

$N_r(H)$**:** We modified the Space-Saving algorithm to include an additional counter for each hint set H that is currently being tracked. This counter is initialized to zero when the algorithm starts tracking H, and it is incremented for each read re-reference involving H that occurs while H is being tracked. We use the value of this counter as $N_r(H)$.

$D(H)$**:** We track the expected re-reference distance for all read re-references involving H that occur while H is being tracked, i.e., those read re-references that are included in $N_r(H)$.

For all hint sets H that are not currently tracked by the algorithm, we take $N_r(H)$ to be zero, and hence $\texttt{Pr}(H)$ to be zero as well.

Since the Space-Saving algorithm's frequency estimates may be overestimates, $N(H)$ will in general be an overestimate of the true frequency of hint set H. On the other hand, since $N_r(H)$ is only incremented while H is being tracked, $N_r(H)$ will in general underestimate the true frequency of read re-references involving H. As a result, we expect in general to underestimate $f_{hit}(H)$ when we use the Space-Saving algorithm. However, the higher the true frequency of H, the more time H will spend being tracked and the more accurate this estimate will be.

The Space-Saving algorithm requires two counters for each tracked hint-set, and we added several additional counters for the sake of our analysis. Overall, the space required is proportional to k. Thus, this parameter can be used to limit the amount of space required to track hint set statistics. With each new request, the data structure used by the Space-Saving algorithm can be updated in constant time [13], and the statistics for the tracked hint sets can be reported, if necessary, in time proportional to k.

5.2 Generalizing Hint Sets

An alternative to focusing on frequently-occurring hint sets is to group similar hint sets together, and then track re-reference statistics for the *groups*, rather than the individual hint sets. In our setting, each hint set consists of a set of values, one value from the domain of each hint type. We have considered a restricted form of grouping these hint sets based on the hint types.

To group hint sets, CLIC recursively partitions the space of possible hint sets one hint type at a time. For example, for the DB2 traces, CLIC might first partition the hint sets according to the value of the "object type ID" hint in each hint set. This would result in six partitions, one for each possible object type ID. CLIC would recursively consider each of these six partitions for further sub-partitioning using one of the remaining hint types. The result of this process can be represented as a *decision tree* in which the final partitions correspond to the leaf nodes of the tree. Figure 5 illustrates a small decision tree based on the hint types from the DB2 traces.

Once constructed, the decision tree can be used to assign caching priorities to requested pages. Each node,

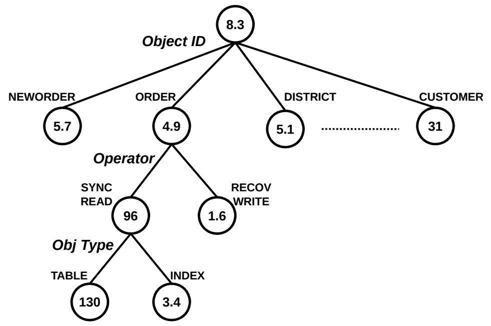

Figure 5: Sample Decision Tree for the 540_40 Trace. Node priorities are shown (all numbers ×1E-5).

or partition, n in the tree is assigned a caching priority, $\texttt{Pr}(n)$. When an I/O request (p, H) occurs, CLIC determines the node n into which the decision tree classifies the hint set H, and assigns priority $\texttt{Pr}(n)$ to page p.

For a given set of hint types, there are many possible decision trees that can be constructed, depending on the order in which the hint types are used for partitioning, and depending on how deeply the sub-partitioning continues along each branch of the tree. Ideally, CLIC should construct its decision tree on-line and continuously adapt it as it observes new I/O requests. To approximate such behavior, our current implementation of CLIC instead simply rebuilds its decision tree periodically. Specifically, it divides the I/O request stream into intervals of fixed length L. During the ith interval, it monitors the statistics described in Section 3 for a limited number, k, of distinct hint sets. At the end of the interval, it uses these k hint sets as training data for a decision tree construction algorithm. The resulting tree is then used to assign priorities for all requests that arrive during interval $i + 1$.

To construct a decision tree after each interval, CLIC begins with a root node representing a single partition containing all of the hint sets in the training data. The priority of the node is taken to be the frequency-weighted average of the priorities of the training hint sets that map to that node:

$$\texttt{Pr}(n) = \frac{\sum_{H \in n} N(H)\texttt{Pr}(H)}{\sum_{H \in n} N(H)} \tag{3}$$

It then considers partitioning the node according to each of the hint types. Partitioning a node using a given hint type creates one child node for each possible value of that hint type. Suppose that CLIC is considering splitting a node n to produce child nodes $c_1, c_2, \ldots, c_m$. Define the *priority variance* V introduced by this split to be

$$V = \frac{\sum_i N(c_i)(\texttt{Pr}(c_i) - \texttt{Pr}(n))^2}{\texttt{Pr}(n)} \tag{4}$$

where $N(c_i)$ is the sum of the request counts $N(H)$ of the training hint sets that map to node c_i. Splits with high priority variance are desirable, as they separate training hint sets with substantially different priorities into different nodes. Among the possible hint types for which CLIC could split n, it chooses the hint type with maximum priority variance.

CLIC continues to partition nodes until either of two stopping conditions is reached. First, CLIC will not split a node n if $N(n) < N_{min}$. Second, CLIC will not split a node if the priority variance introduced by the best possible split is less than a *gain threshold* G. Both N_{min} and G are parameters of the decision tree construction algorithm that can be used to influence the size of the final tree, controlling a tradeoff between space and accuracy of priority assignments.

6 Experimental Evaluation

Objectives: We used trace-driven simulation to evaluate our proposed mechanisms. The goal of our experimental evaluation is to answer the following questions:

1. Can CLIC identify good caching opportunities for storage server caches, and thereby improve the cache hit ratio over other caching policies? (Section 6.1)
2. How well do the mechanisms used by CLIC to reduce the cost of tracking statistics affect performance? (Sections 6.2 and 6.3)
3. Does CLIC scale as the number of hint types grows? (Section 6.4)
4. Can CLIC improve performance for multiple storage clients by prioritizing the caching opportunities of the different clients based on their observed reference behavior? (Section 6.5)

Simulator: To answer these questions, we implemented a simulator of the storage server cache. In addition to CLIC, the simulator implements the following caching policies for purpose of comparison:

OPT: This is an implementation of the well-known optimal off-line MIN algorithm [3]. It replaces the cached page that will not be read for the longest time. This algorithm requires knowing the future so it cannot be used for cache replacement in practical systems, but its hit ratio is optimal so it serves as an upper bound on the performance of any caching algorithm.

LRU: This algorithm replaces the least-recently used page in the cache. Since temporal locality is often poor in second-tier caches, we expect CLIC to perform significantly better than LRU.

TQ: This is a hint-aware algorithm that was proposed for use in second-tier caches [10]. Unlike the algorithms proposed here, it works only with one specific type of hint that can be associated with write requests from database systems. We expect our proposed algorithms, which can automatically exploit any type of hint, to do at least as well as TQ when the write hints needed by TQ are present in the request stream.

The TQ algorithm has previously been compared to a number of other second-tier caching policies that are not considered here. These include MQ [20], a hint-oblivious policy, and write-hint-aware variations of both MQ and LRU. TQ was shown to be generally superior to those alternatives when the necessary write hints are present [10], so we use it as our representative of the state of the art in hint-aware second-tier caching policies.

The simulator accepts a stream of I/O requests with associated hint sets, as would be generated by one or more storage clients. It simulates the caching behavior of one of the four supported cache replacement policies (CLIC, OPT, LRU, and TQ) and computes the *read hit ratio* for the storage server cache. The read hit ratio is the number of read hits divided by the number of read requests.

Workloads: In this paper, we use DB2 Universal Database (version 8.2) as our storage system client. DB2 is a widely-used commercial relational database system to which we had access to source code. We instrumented DB2 so that it would generate I/O hints and dump them into an I/O trace. The types of hints generated by the instrumented DB2 are described in Figure 2.

To generate our traces, we ran a TPC-C workload on DB2. TPC-C is a well-known on-line transaction processing (OLTP) benchmark, and we ran it at scale factor 25. At this scale factor, the TPC-C database initially occupied 606,317 4KB blocks, or about 2.3 GB, in the storage system. The TPC-C workload inserts new items into the database, so the database grows during the TPC-C run.

We varied two different DB2 configuration parameters, and collected traces under the resulting configurations. The first parameter is the size of DB2's internal buffer cache (the buffer pool), which we set at 10%, 50%, and 90% of the initial database size (60,000, 300,000, and 540,000 pages, respectively). We expect this parameter to be significant because it affects temporal locality in the request stream seen by the underlying storage server. The larger DB2's buffer cache, the less temporal locality we expect to be available at the storage server. The second DB2 parameter we varied is `softmax`, which controls the urgency with which DB2 forces dirty pages from its buffer cache to disk for recoverability reasons. Smaller values of `softmax` result in more (and more frequent) write requests in the request stream. Figure 6 summarizes the I/O request traces we used for our evaluation.

6.1 Base Results

In our first experiment, we compare the cache hit ratio of CLIC to the other replacement policies that we consider (LRU, TQ, and OPT). We tested both CLIC and an off-line version of CLIC. The off-line version makes two passes over the input trace. During the first pass it gathers statistics for each hint set and computes hint set caching priorities. During the second pass, it uses those priorities to manage the cache. In this experiment, CLIC (both on-line and off-line) was given *unlimited space* for tracking page references and for recording hint statistics, and this space was not subtracted from the server cache size. Thus, the CLIC algorithm uses more space than its competitors.

Figure 7 shows the results of this experiment for the three traces with `softmax=400`. The traces with `softmax=40` resulted in behavior similar to that shown in the figure. There are several observations to be made from Figure 7. First, all of the algorithms, including LRU, have similar performance for the 60_400 trace. The 60_400 trace comes from the DB2 configuration with the smallest buffer cache, and there is a significant amount of temporal locality in the trace that is not "absorbed" by that cache. This temporal locality can be exploited by the storage server cache. As a result, even LRU performs reasonably well. Both of the hint-based algorithms (TQ and CLIC) also do well.

The performance of LRU is significantly worse on the other two traces, as there is very little temporal locality. The hint-based algorithms perform much better. CLIC, which learns how to exploit the available hints, does better than TQ, which implements a hard-coded response to one particular hint type. CLIC's performance approaches that of OPT on the 300_400 trace, but the gap is greater on the 540_400 trace. The 540_400 trace comes from the DB2 configuration with the largest buffer cache, so it has the least temporal locality of all traces and therefore presents the most difficult cache replacement problem. We also note that there is very little difference between the off-line and on-line versions of CLIC. This indicates that on-line collection of hint set statistics is effective.

This experiment shows that the hint analysis performed by CLIC results in better second-tier caching decisions than LRU and the hint-aware TQ. The downside is that CLIC as used in this experiment does not bound the amount of memory used for the metadata that is required for tracking cache statistics. In the next two sections, we show that the techniques we propose for efficient tracking of statistics can bound the amount of memory used to a small value without a significant penalty in performance.

Trace Name	DB2 Buffer Size (blocks)	softmax	Requests	Distinct Hint Sets	Distinct Pages
60_40	60K	40	38101851	169	919215
60_400	60K	400	37699091	164	930688
300_40	300K	40	32102429	128	1130925
300_400	300K	400	31869377	154	1320882
540_40	540K	40	49279589	105	1684878
540_400	540K	400	21863719	140	1807431

Figure 6: I/O Request Traces

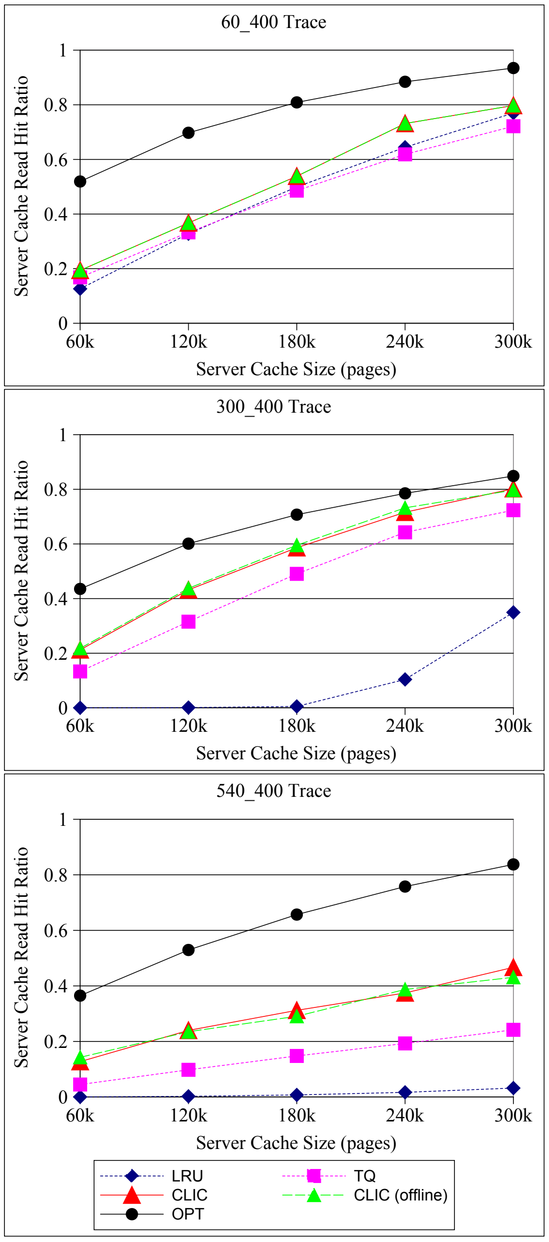

Figure 7: Read Hit Ratio of Caching Policies

6.2 Limiting the Outqueue Size

In this experiment, we study the effect of limiting the size of the outqueue that CLIC uses to track the sequence number of each page's most recent reference (Section 5). In the previous section, we showed the performance of CLIC with unlimited outqueue size, and in this section we show that we can limit the size of the outqueue without a large performance penalty.

We ran two sets of experiments, one in which the number of outqueue entries was limited to one per page in the storage server's cache, and one in which the number of entries was limited to 5 per storage server cache page. If the cache holds C pages, this means that CLIC tracks the most recent reference for $2C$ pages in the former case, and $6C$ pages in the in the latter case (since it tracks this information for all cached pages, plus those in the outqueue). For each tracked page, CLIC records a sequence number and a hint set. If each of these is represented as a 4-byte integer, this represents a space overhead of 0.2% at our 4KB page size. An outqueue limit of 5 entries per server cache page represents a space overhead of roughly 1%, which is still quite small.

Figure 8 shows the results of this experiment for all six traces and a server cache size of 180K pages. In all cases, an outqueue limit of 5 entries per server cache page (the middle bar in each group in Figure 8) results in a server cache read hit ratio very close to what was obtained with an unlimited outqueue. In most cases, an outqueue with only one entry per server cache page also does well (with some exceptions for the 540_40 trace). Similar experiments with server cache sizes of 60K pages and 300K pages resulted in identical conclusions. Thus, we can see that limiting the outqueue size saves space without degrading the performance of CLIC. For all of our remaining experiments, we use an outqueue limit of 5 entries per server cache page.

6.3 Tracking Only Frequent Hint Sets

In this experiment, we study the effect of tracking only the most frequently occurring hint sets (top-k hint set filtering). We vary the number of hint sets tracked by CLIC, k, and measure the server cache hit ratio. Figure 9

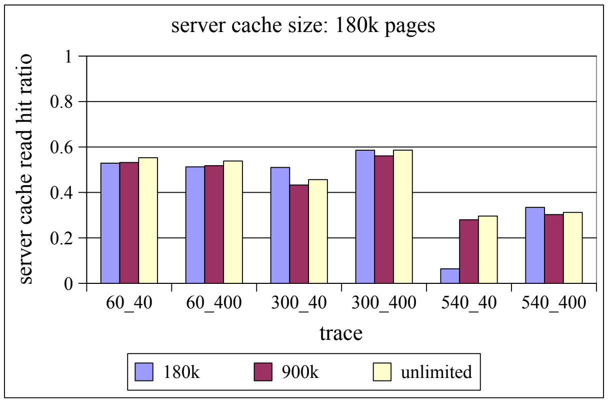

Figure 8: Effect of Outqueue Size on Read Hit Ratio
Each bar represents a different outqueue size.

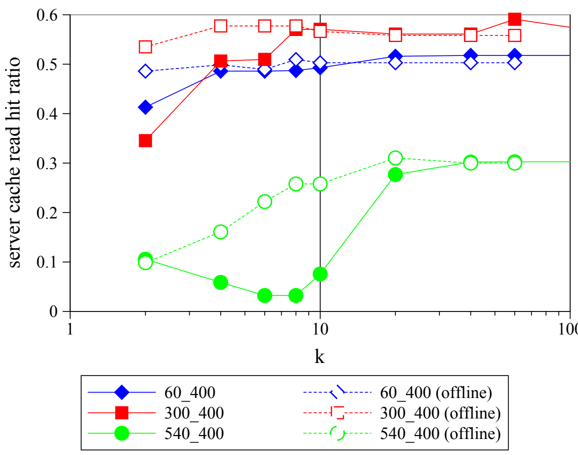

Figure 9: Effect of Top-K Hint Set Filtering on Read Hit Ratio

shows the results of this experiment for the three traces with `softmax=400`, and a server cache size of 180K pages. The rightmost point for each trace is the result obtained when tracking all of the hint sets in the trace. For every trace, the figure shows results for the Space-Saving algorithm used by CLIC and an off-line top-k algorithm. The off-line top-k algorithm makes two passes over the input trace. In the first pass it computes the frequency of every hint set and determines the k most frequent hint sets. In the second pass, it tracks cache statistics for the top-k hint sets determined in the first pass. Thus, this off-line algorithm enables CLIC to track the *exact* top-k hint sets, while the Space-Saving algorithm enables it to track an *approximate* top-k group of hint sets.

The figure shows that we can significantly reduce the number of hint sets tracked with almost no sacrifice in performance. For all traces, tracking 20 hint sets (or more) results in performance that is almost identical to tracking all hint sets. The figure also shows that we do not lose accuracy because we are tracking an approximate top-k and not an exact top-k. The off-line and Space-Saving results are almost identical for $k > 10$.

6.4 Increasing the Number of Hints

In addition to the top-k approach studied in the previous section, we also propose using decision trees to limit the space required for tracking hint statistics. The decision tree approach works by reducing the number of hint types that we track, so we present an evaluation of this approach in the context of studying the scalability of CLIC as the number of hints in the hint sets (i.e., the different hint types) increases.

To increase the number of hints we injected additional synthetic hints into our DB2 traces. Each DB2 trace record contains 5 real hint types. In this experiment, we add T additional hint types and hence T additional hints to each hint set. Each injected hint is chosen randomly from a domain of D possible hint values. A particular value from the domain is selected using a Zipfian distribution with skew parameter $z = 1$. When $T > 1$, each injected hint value is chosen independently of the other injected hints for the same record. This injection procedure potentially increases the number of distinct hint sets in a trace by a factor D^T. For our experiments, we chose $D = 10$, and we varied T.

Since the injected hints are chosen at random, we do not expect them to provide any information that is useful for server cache management. We are interested in whether the methods used by CLIC to limit the number of hint sets will be effective in filtering out the injected hints, focusing instead on those original hint types that provide predictive value.

Figure 10 shows the read hit ratios in a server cache of size 180K when adding extra hints to the three traces with `softmax=400`. The figure shows the hit ratio with no extra hints ($T = 0$) and with $T = 1$ to 3 extra hints in every hint set. Results for both decision tree and top-k hint set filtering are shown. For the top-k hint set filtering, we use the Space-Saving algorithm with $k = 100$. For the decision tree, we use a rebuild interval of $L = 10^6$ requests, a monitoring limit of $k = 100$ hint sets, $N_{min} = 5000$, and a gain threshold of $G = 10^{-4}$. These values of G and N_{min} gave us good results for all of our traces, and we leave the problem of optimally tuning the decision tree parameters for future work. All of the decision trees constructed by CLIC during these experiments had fewer than 150 nodes.

The figure shows that decision trees and top-k filtering are equally effective at making caching decisions for $T = 0$. As T increases and more "noise dimensions" are added to the hints, decision trees perform better because they are able to ignore entire dimensions, although top-k

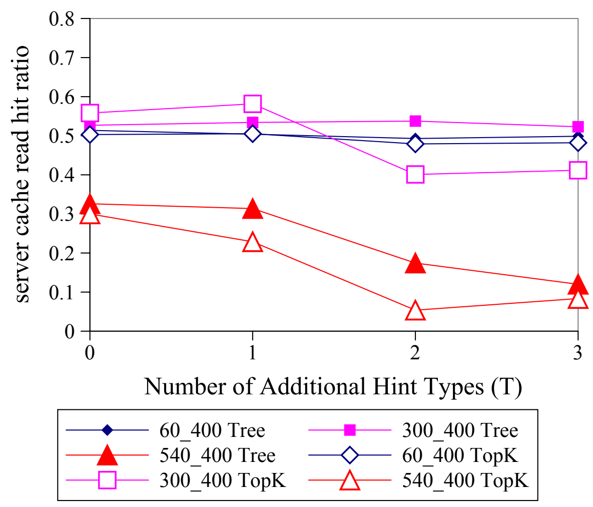

Figure 10: Effect of Number of Additional Hint Types on Read Hit Ratios

remains competitive. For the 60_400 and 300_400 traces (but not the more difficult 540_400 trace), decision trees are able to filter out the noise in the trace and experience almost no drop in performance. To put this in perspective, we note that the number of different hint sets increases from about 1000 to about 50000 in each trace as T increases from 1 to 3. Thus, from this experiment we conclude that both frequency-based hint set filtering and decision trees enable CLIC to effectively deal with large numbers of hint types without requiring a large amount of space.

6.5 Multiple Storage Clients

One desirable feature of CLIC is that it should be capable of accommodating hints from multiple storage clients. The clients independently send their different hints to the storage server without any coordination among themselves, and CLIC should be able to effectively prioritize the hints to get the best overall cache hit ratio.

To test this, we simulated a scenario in which multiple instances of DB2 share a storage server. Each DB2 instance manages its own separate database, and represents a separate storage client. All of the databases are housed in the storage server, and the storage server's cache must be shared among the pages of the different databases. To create this scenario, we create a multi-client trace for our simulator by interleaving requests from several DB2 traces, each of which represents the requests from a single client. We interleave the requests in a round robin manner, one from each trace. We truncate all traces to the length of the shortest trace being interleaved to eliminate bias towards longer traces. We treat the hint types in each trace as distinct, so the total number of distinct hint sets in the combined trace is the sum of the number of distinct hint sets in each individual trace.

Figure 11 shows results for the trace generated by interleaving the three traces with `softmax=400`. The server cache size is 180K pages, and CLIC uses top-k filtering with $k = 100$. We also performed this experiment with decision trees and the results were almost identical to top-k filtering. The figure shows the read hit ratio for the requests from each individual trace that is part of the interleaved trace. The figure also shows the overall hit ratio for the entire interleaved trace. For comparison, the figure shows the hit ratios for the full-length (untruncated) traces when they use independent caches of size 60K pages each (i.e., the storage server cache is partitioned equally among the clients). The figure shows a dramatic improvement in hit ratio for the 60_400 trace and also an improvement in the overall hit ratio as compared to equally partitioning the server cache among the traces. CLIC is able to identify that the 60_400 trace has the best caching opportunities (since it has the most temporal locality), and to focus on caching pages from this trace. This illustrates that CLIC is able to accommodate hints from multiple storage clients and prioritize them so as to maximize the overall hit ratio.

Note that it is possible to consider other objectives when managing the shared server cache. For example, we may want to ensure fairness among clients or to achieve certain quality of service levels for some clients. This may be accomplished by statically or dynamically partitioning the cache space among the clients. In CLIC, the objective is simply to maximize the overall cache hit ratio without considering quality of service targets or fairness among clients. This objectives results in the best utilization of the available cache space. Our experiment illustrates that CLIC is able to achieve this objective, although the benefits of the server cache may go disproportionately to some clients at the expense of others.

7 Related Work

Many algorithms have been proposed to improve on LRU, including MQ [20], ARC [12], CAR [2], and 2Q [9]. These algorithms use a combination of recency of use and frequency of use to make replacement decisions. They can be used to manage a cache at any level of a cache hierarchy, though some, like MQ, were explicitly developed for use in second-tier caches, for which there is little temporal locality in the workload.

There are also caching policies that have been proposed explicitly for second (or lower) tier caches in a cache hierarchy. Chen et al [5] have classified these as either *hierarchy-aware* or *aggressively collaborative*. Hierarchy-aware methods specifically exploit the knowledge that they are running in the second tier, but they are transparent to the first tier. Some such approaches, like

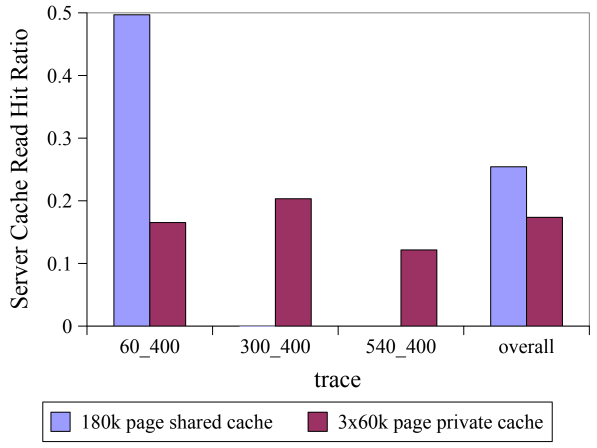

Figure 11: Read Hit Ratio with Three Clients
Read hit ratio is near zero for the 300_400 and 540_400 traces in the 180K page shared cache, so bars are not visible.

X-RAY [1], work by examining the contents of requests submitted by a client application in the first tier. By assuming a particular type of client and exploiting knowledge of its behavior, X-RAY can extract client-specific semantic information from I/O requests. This information can then be used to guide caching decisions at the server. X-RAY has been proposed for file system clients [1] and DBMS clients [15].

Aggressively collaborative approaches require changes to the first tier. Examples include PROMOTE [7] and DEMOTE [17], both of which seek to maintain exclusivity among caches. Hint-based techniques, including CLIC, are aggressively collaborative. At the extreme, it is possible to give control of the second tier cache to the first tier, as is done in ULC [8]. Among the aggressively collaborative techniques, hint-based approaches are arguably the least intrusive and least costly. Hints are small and can be piggybacked onto I/O requests, and the policies used to manage the first tier cache need not be changed.

Several hint-based techniques have been proposed, including importance hints [5] and write hints [10], which have already been described. Karma [19] relies on application hints to group pages into "ranges", and to associate an expected access pattern with each range. As was described in Section 1, these techniques are ad hoc and difficult to generalize. Unlike CLIC, they are designed to exploit specific types of hints.

A previous study [5] suggested that aggressively collaborative approaches provided little benefit beyond that of hierarchy-aware approaches and thus, that the loss of transparency implied by collaborative approaches was not worthwhile. However, that study only considered one ad hoc hint-based technique. Li et al [10] found that the hint-based TQ algorithm could provide substantial performance improvements in comparison to hint-oblivious approaches (LRU and MQ) as well as simple hint-aware extensions of those approaches.

There has also been work on the problem of sharing a cache among multiple competing client applications [4, 11, 16, 18]. Often, the goal of these techniques is to achieve specific quality-of-service objectives for the client applications, and the method used is to somehow partition the shared cache. This work is largely orthogonal to CLIC, in the sense that CLIC can be used, like any other replacement algorithm, to manage the cache contents in each partition. CLIC can also used to directly control a shared cache, as in Section 6.5, but it does not include any mechanism for enforcing quality-of-service requirements or fairness requirements among the competing clients.

The problem of identifying frequently-occurring items in a data stream occurs in many situations. Metwally et al [13] classify solutions to the frequent-item problem as counter-based techniques or sketch-based techniques. The former maintain counters for certain individual items, while the latter collect information about aggregations of items. For CLIC, we have chosen to use the Space-Saving algorithm [13] as it is both effective and simple to implement. A recent study [6] found the Space-Saving algorithm to be one of the best overall performers among frequent-item algorithms.

8 Conclusion

We have presented CLIC, a technique for managing a storage server cache based on hints from storage client applications. CLIC provides a general, adaptive mechanism for incorporating application-provided hints into cache management. We used trace-driven simulation to evaluate CLIC, and found that it was very effective at learning to exploit hints. In our tests, CLIC learned to perform as well as or better than TQ, an ad hoc hint based technique. CLIC also performed substantially better than LRU, which is hint-oblivious. Our results also show that CLIC, unlike TQ and other ad hoc techniques, can accommodate hints from multiple client applications.

A potential drawback of CLIC is the overhead of learning which hints are valuable. We considered two techniques for limiting this overhead, one based on identifying frequently-occurring hints and the other based on aggregating hints with similar properties, using decision trees. We found both approaches to be effective. The frequency-based approach is very simple, while the decision tree approach is more complex and has several parameters that must be tuned. We intend to explore simpler and more easily controllable decision tree techniques for CLIC as part of our future work.

References

[1] Bairavasundaram, L. N., Sivathanu, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 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).

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

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

[4] Brown, K. P., Carey, M. J., and Livny, M. Goal-oriented buffer management revisited. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (June 1996), pp. 353–364.

[5] Chen, Z., Zhang, Y., Zhou, Y., Scott, H., and Schiefer, B. 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)* (2005), pp. 145–156.

[6] Cormode, G., and Hadjieleftheriou, M. Finding frequent items in data streams. In *Proc. Int'l Conference on Very Large Data Bases (VLDB'08)* (Aug. 2008).

[7] Gill, B. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proc. USENIX Conference on File and Storage Technologies (FAST'08)* (2008), pp. 49–65.

[8] Jiang, S., and Zhang, X. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS'04)* (2004), pp. 168–177.

[9] Johnson, T., and Shasha, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)* (1994), pp. 439–450.

[10] Li, X., Aboulnaga, A., Salem, K., Sachedina, A., and Gao, S. Second-tier cache management using write hints. In *USENIX Conference on File and Storage Technologies (FAST'05)* (Dec. 2005), pp. 115–128.

[11] Martin, P., Li, H.-Y., Zheng, M., Romanufa, K., and Powley, W. Dynamic reconfiguration algorithm: Dynamically tuning multiple buffer pools. In *11th International Conference on Database and Expert Systems Applications (DEXA)* (2000), pp. 92–101.

[12] Megiddo, N., and Modha, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. USENIX Conference on File and Storage Technology (FAST'03)* (2003).

[13] Metwally, A., Agrawal, D., and Abbadi, A. E. Efficient computation of frequent and top-k elements in data streams. In *Proc. International Conference on Database Theory (ICDT)* (Jan. 2005).

[14] Muntz, D., and Honeyman, P. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Conference* (Jan. 1992), pp. 305–313.

[15] Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Database-aware semantically-smart storage. In *Proc. of the USENIX Symposium on File and Storage Technologies (FAST'05)* (2005), pp. 239–252.

[16] Soundararajan, G., Chen, J., Sharaf, M., and Amza, C. Dynamic partitioning of the cache hierarchy in shared data centers. In *Proc. Int'l Conference on Very Large Data Bases (VLDB'08)* (Aug. 2008).

[17] Wong, T. M., and Wilkes, J. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference (USENIX 2002)* (June 2002), pp. 161–175.

[18] Yadgar, G., Factor, M., Li, K., and Schuster, A. Mc2: Multiple clients on a multilevel cache. In *Proc. Int'l Conference on Distributed Computing Systems (ICDCS'08)* (June 2008).

[19] Yadgar, G., Factor, M., and Schuster, A. Karma: Know-it-all replacement for a multilevel cache. In *Proc. USENIX Conference on File and Storage Technologies (FAST'07)* (Feb. 2007).

[20] Zhou, Y., Chen, Z., and Li, K. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems 15*, 7 (July 2004).