Hybrid Storage Management for Database Systems

Xin Liu
University of Waterloo, Canada
x39liu@uwaterloo.ca

Kenneth Salem
University of Waterloo, Canada
ksalem@uwaterloo.ca

ABSTRACT

The use of flash-based solid state drives (SSDs) in storage systems is growing. Adding SSDs to a storage system not only raises the question of how to manage the SSDs, but also raises the question of whether current buffer pool algorithms will still work effectively. We are interested in the use of hybrid storage systems, consisting of SSDs and hard disk drives (HDDs), for database management. We present cost-aware replacement algorithms, which are aware of the difference in performance between SSDs and HDDs, for both the DBMS buffer pool and the SSDs. In hybrid storage systems, the physical access pattern to the SSDs depends on the management of the DBMS buffer pool. We studied the impact of buffer pool caching policies on SSD access patterns. Based on these studies, we designed a cost-adjusted caching policy to effectively manage the SSD. We implemented these algorithms in MySQL's InnoDB storage engine and used the TPC-C workload to demonstrate that these cost-aware algorithms outperform previous algorithms.

1. INTRODUCTION

Flash memory has been used for many years in portable consumer devices (e.g, cameras, phones) where low power consumption and lack of moving parts are particularly desirable features. Flash-based solid state storage devices (SSDs) are now also becoming commonplace in server environments. SSDs are more expensive per bit than traditional hard disks (HDD), but they are much cheaper in terms of cost per I/O operation. Thus, servers in data centers may be configured with both types of persistent storage. HDDs are cost effective for bulky, infrequently accessed data, while SSDs are well-suited to data that are relatively hot [8]. In this paper, we are concerned with the use of such hybrid (SSD and HDD) storage systems for database management. We consider hybrid storage systems in which the two types of devices are visible to the database management system (DBMS), so that it can use the information at its disposal to decide how to make use of the two types of devices. This is illustrated in Figure 1. When writing data to storage, the DBMS chooses which type of device to write it to.

Proceedings of the VLDB Endowment, Vol. 6, No. 8

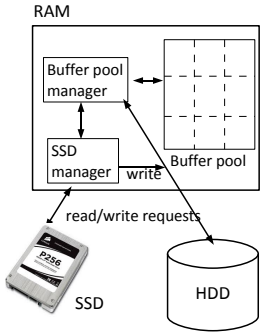

Figure 1: System Architecture

Previous work has considered how a DBMS should place data in a hybrid storage system [11, 1, 2, 16, 5]. We provide a summary of such work in Section 7. In this paper, we take a broader view of the problem than is used by most of this work. Our view includes the DBMS buffer pool as well as the two types of storage devices. We consider two related problems. The first is determining which data should be retained in the DBMS buffer pool. The answer to this question is affected by the presence of hybrid storage because blocks evicted from the buffer cache to an SSD are much faster to retrieve later than blocks evicted to the HDD. Thus, we consider *cost-aware buffer management*, which can take this distinction into account. Second, assuming that the SSD is not large enough to hold the entire database, we have the problem of deciding which data should be placed on the SSD. This should depend on the physical access pattern for the data, which depends, in turn, on both the DBMS workload and the management of the DBMS buffer pool.

Because we consider both buffer pool management and management of the hybrid storage system, we have more scope for optimization than previous work in this area, at the expense of additional invasiveness in the design and implementation of the DBMS. In addition, we must account for the fact that the two problems we consider are mutually dependent. Replacement decisions in the buffer pool depend on the locations (SSD or HDD) of the pages being replaced, since location affects both eviction cost and reloading cost. Conversely, SSD page placement decisions depend on how

the page is used, e.g., how frequently it is read or written, which depends in turn on the buffer manager. For example, under the GD2L replacement policy we propose here, moving a page from the HDD to the SSD may result in a significant increase in the physical read and write rates for that page, since GD2L tends to evict SSD pages quickly from the buffer pool.

Our work addresses these dependencies using an *anticipatory* approach to SSD management. When deciding whether to move a page into the SSD, our proposed admission and replacement policy (called CAC) predicts how such a move will affect the physical I/O load experienced by that page. The page is moved into the SSD only if it is determined to be a good candidate under this predicted workload. The DBMS buffer manager then makes cost-aware replacement decisions based on the current placements of buffered pages.

In this paper we present the following contributions:

- We present GD2L, a cost-aware algorithm for buffer pool management in database systems with hybrid storage systems. GD2L takes the usual concerns of DBMS buffer management (exploiting locality, scan resistance) into account, but also considers the fact that different devices in a hybrid storage system perform differently. GD2L is based on the GreedyDual algorithm [19], but we have restricted GreedyDual to hybrid systems that include only two types of devices. In addition, we have refined GreedyDual for operation in a DBMS environment.

- We present CAC, an anticipatory cost-based technique for managing the SSD. Unlike previous techniques, CAC is intended to work together with a cost-aware buffer manager like GD2L. It expects that moving a page into or out of the SSD will change the access pattern for that page, and it anticipates these changes when making SSD placement decisions.

- We present an empirical evaluation of GD2L and CAC. We have implemented both techniques in MySQL's InnoDB storage manager. We compare the performance of GD2L with that of InnoDB's native buffer manager, which is oblivious to the location of pages in a hybrid storage system. We compare CAC to several alternatives, including a non-anticipatory cost-based technique, LRU-2, and MV-FIFO. Our evaluation uses transactional workloads (TPC-C).

The remainder of this paper is organized as follows. Section 2 gives an overview of the system architecture that we assume. Section 3 presents the GD2L technique for database buffer pool management, and Section 4 shows empirical results that illustrate the effect of GD2L on the physical access patterns of database pages. Section 5 presents the CAC algorithm for managing the contents of the SSD device(s). The results of our evaluation GD2L and CAC are presented in Section 6, and Section 7 summarizes related work.

2. SYSTEM OVERVIEW

Figure 1 illustrates the system architecture we have assumed for this work. The DBMS sees two types of storage devices, SSDs and HDDs. All database pages are stored on the HDD, where they are laid out according to the DBMS's secondary storage layout policies. In addition, copies of some pages are located in the SSD and copies of some pages are located in the DBMS buffer pool. Any given page may have copies in the SSD, in the buffer pool, or both.

When the DBMS needs to read a page, the buffer pool is consulted first. If the page is cached in the buffer pool, the DBMS reads the cached copy. If the page is not in the buffer pool but it is in the SSD, it is read into the buffer pool from the SSD. The SSD manager is responsible for tracking which pages are currently located in the SSD. If the page is in neither the buffer pool nor the SSD, it is read from the HDD.

If the buffer pool is full when a new page is read in, the buffer manager must evict a page according to its page replacement policy, which we present in Section 3. When the buffer manager evicts a page, the evicted page is considered for admission to the SSD if it is not already located there. SSD admission decisions are made by the SSD manager according to its *SSD admission policy*. If admitted, the evicted page is written to the SSD. If the SSD is full, the SSD manager must also choose a page to be evicted from the SSD to make room for the newly admitted page. SSD eviction decisions are made according to an *SSD replacement policy*. (The SSD admission and replacement policies are presented in Section 5.) If a page evicted from the SSD is more recent than the version of that page on the HDD, then the SSD manager must copy the page from the SSD to the HDD before evicting it, otherwise the most recent persistent version of the page will be lost. The SSD manager does this by reading the evicted page from the SSD into a staging buffer in memory, and then writing it to the HDD.

We assume that the DBMS buffer manager implements asynchronous page cleaning, which is widely used to hide write latencies from DBMS applications. When the buffer manager elects to clean a dirty page, that page is written to the SSD if the page is already located there. If the dirty page is not already located on the SSD, it is considered for admission to the SSD according to the SSD admission policy, in exactly the same way that a buffer pool eviction is considered. The dirty page will be flushed to the SSD if it is admitted there, otherwise it will be flushed to the HDD.

The buffer and SSD management techniques that we have described have two key properties. First, admission of pages into the SSD occurs only when pages are evicted or cleaned from the DBMS buffer pool. Pages are *not* admitted into the SSD when they are loaded into the buffer pool from the HDD, as might be expected in a multi-tier cache. The reason for this is to minimize *cache inclusion* [18], i.e., the duplication of pages in the buffer cache and the SSD. Second, each flush of a dirty page from the DBMS buffer pool goes either to the SSD or to the HDD, but not to both (at least not immediately). One advantage of this approach, compared to a write-through design, is that the SSD can potentially improve DBMS write performance, to the extent that writes are directed to the SSD. A disadvantage of this approach is that the latest version of an unbuffered page might, in general, be found on either device. However, because the DBMS always writes a dirty buffer pool page to the SSD if that page is already on the SSD, it can be sure that the *SSD version (if any) of a page is always at least as recent as the HDD version.* Thus, to ensure that it can obtain the most recently written version of any page, it is sufficient for the DBMS to know which pages have copies on the SSD, and to read a page from the SSD if there is a copy of the

Table 1: Storage Device Parameters

Symbol	Description
R_D	The read service time of HDD
W_D	The write service time of HDD
R_S	The read service time of SSD
W_S	The write service time of SSD

page there. To support this, the SSD manager maintains an in-memory hash map that records which pages are on the SSD. To ensure that it can determine the contents of the SSD even after a failure, the SSD manager uses a checkpointing technique (described in Section 5.4) to efficiently persist its map so that it can be recovered quickly.

3. BUFFER POOL MANAGEMENT

In this section, we describe our replacement algorithm for the buffer pool: a two-level version of the GreedyDual algorithm [19] which we have adapted for use in database systems. We refer to it as GD2L.

Most existing cost-aware algorithms, e.g., the balance algorithm [15] and GreedyDual [19], were proposed for file caching. They take into account the size of cached objects and their access costs when making replacement decisions, and target different situations: uniform object size with arbitrary retrieval costs, arbitrary object size with uniform retrieval costs, or arbitrary object size with arbitrary retrieval costs. The GreedyDual algorithm addresses the case of objects with uniform size but different retrieval costs. Young [19] shows that GreedyDual has the same (optimal) competitive ratio as LRU and FIFO [15].

GreedyDual is actually a range of algorithms which generalize well-known caching algorithms, such as LRU and FIFO. Initially, we present the GreedyDual generalization of LRU and our restriction to two retrieval costs. In Section 3.1, we describe how a similar approach can be applied to the LRU variant used by InnoDB, and we also discuss how it can be extended to handle writes.

GreedyDual associates a non-negative cost H with each cached page p. When a page is brought into the cache or referenced in the cache, H is set to the cost of retrieving the page into the cache. To make room for a new page, the page with the lowest H in the cache, H_{min}, is evicted and the H values of all remaining pages are reduced by H_{min}. By reducing the H values and resetting them upon access, GreedyDual ages pages that have not been accessed for a long time. The algorithm thus integrates locality and cost concerns in a seamless fashion.

GreedyDual is usually implemented using a priority queue of cached pages, prioritized based on their H value. With a priority queue, handling a hit and an eviction each require $O(\log k)$ time. Another computational cost of GreedyDual is the cost of reducing the H values of the remaining pages when evicting a page. To reduce the value H for all pages in the cache, GreedyDual requires k subtractions. Cao et al. [3] have proposed a technique to avoid the subtraction cost. Their idea is to keep an "inflation" value L and to offset all future settings of H by L.

Parameters representing the read and write costs to the SSD and the HDD are summarized in Table 1. In our case, there are only two possible initial values for H: one corresponding to the cost of retrieving an SSD page (R_S) and the other to the cost of retrieving an HDD page (R_D). The GD2L algorithm is designed for this special case. GD2L uses two queues to maintain pages in the buffer pool: one queue (Q_S) is for pages placed on the SSD, the other (Q_D) is for pages not on the SSD. Both queues are managed using LRU. With the technique proposed by Cao et al. [3], GD2L achieves $O(1)$ time for handling both hits and evictions.

Figure 2 describes the GD2L algorithm. When GD2L evicts the page with the smallest H from the buffer pool, L (the inflation value) is set to the H value. If the newly requested page is on the SSD, it is inserted to the MRU end of Q_S and its H value is set to $L + R_S$; otherwise it is inserted to the MRU end of Q_D and its H value is set to $L + R_D$. Because the L value increases gradually as pages with higher H are evicted, pages in Q_D and Q_S are sorted by H value. The one having the smallest H value is in the LRU end. By comparing the H values of the LRU page of Q_D and the LRU page of Q_S, GD2L easily identifies the victim page that has the smallest H value in the buffer pool. The algorithm evicts the page with the lowest H value if the newly requested page is not in the buffer pool. In Figure 2, page q represents the page with the lowest H value.

```
1     if p is not cached
2        compare LRU page of Q_S with LRU page of Q_D
3        evict the page q that has the smaller H
4        set L = H(q)
5        bring p into the cache
6     if p is on the SSD
7        H(p) = L + R_S
8        put p to the MRU of Q_S
9     else if p is on HDD
10       H(p) = L + R_D
11       put p to the MRU of Q_D
```

Figure 2: GD2L Algorithm For Reading Page p.

3.1 Implementation of GD2L in MySQL

We implemented GD2L for the buffer pool management in InnoDB, the default storage engine of the MySQL database system. InnoDB uses a variant of the least recently used (LRU) algorithm. When room is needed to add a new page to the pool, InnoDB evicts the LRU page from the buffer pool. Pages that are fetched on demand are placed at the MRU end of InnoDB's list of buffered pages. Prefetched pages are placed near the midpoint of the LRU list (3/8 of the way from the LRU end), moving to the MRU position only if they are subsequently read. Since prefetching is used during table scans, this provides a means of scan resistance.

To implement GD2L, we split InnoDB's LRU list into two LRU lists: Q_D and Q_S. As shown in Figure 3, the cached HDD pages (represented by H) are stored in Q_D and the cached SSD pages (represented by S) in Q_S. Newly loaded pages are placed either at the MRU end or the midpoint of the appropriate list, depending on whether they were prefetched or loaded on demand. When a new prefetched page is inserted at the midpoint of Q_D or Q_S, its H value is set to the H value of the current midpoint page.

When pages are modified in the buffer pool, they need to be copied back to the underlying storage device. In InnoDB, dirty pages are generally not written to the underlying storage device immediately after they are modified in the buffer pool. Instead, *page cleaner* threads are responsible for asynchronously writing back dirty pages. The page cleaners can

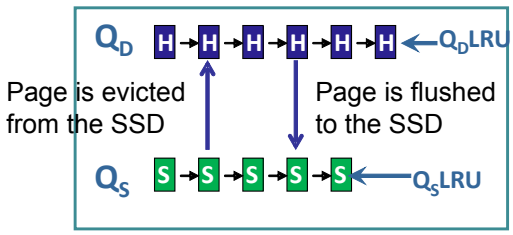

Figure 3: Buffer pool managed by GD2L

issue two types of writes: *replacement writes* and *recoverability writes*. Replacement writes are issued when dirty pages are identified as eviction candidates. To remove the latency associated with synchronous writes, the page cleaners try to ensure that pages that are likely to be replaced are clean at the time of the replacement. In contrast, recoverability writes are those that are used to limit failure recovery time. The InnoDB uses write ahead logging to ensure that committed database updates survive failures. The failure recovery time depends on the age of the oldest changes in the buffer pool. The page cleaners issue recoverability writes of the least recently modified pages to ensure that a configurable recovery time threshold will not be exceeded.

In InnoDB, when the free space of the buffer pool is below a threshold, page cleaners start to check a range of pages from the tail of the LRU list. If there are dirty pages in the range, the page cleaners flush them to the storage devices. These are replacement writes. We changed the page cleaners to reflect the new cost-aware replacement policy. Since pages with lower H values are likely to be replaced sooner, the page cleaners consider H values when choosing which pages to flush. As GD2L maintains two LRU lists in the buffer pool (Q_D and Q_S), the pages cleaners check pages from tails of both lists. If there are dirty pages in both lists, the page cleaners compare their H values and choose dirty pages with lower H values to write back to the storage devices. We did not change the way the page cleaners issue recoverability writes, since those writes depend on page update time and not on page access cost.

The original GreedyDual algorithm assumed that a page's retrieval cost does not change. However, in our system a page's retrieval cost changes when it is moved into or out of the SSD. If a buffered page is moved into the SSD, then GD2L must take that page out of Q_D and place it into Q_S. This situation can occur when a dirty, buffered page that is not on the SSD is flushed, and the SSD manager elects to place the page into the SSD. If the page flush is a replacement write, it means that the page being flushed is a likely eviction candidate. In that case, GD2L removes the page from Q_D and inserts it at the LRU end of Q_S. If the page flush is a recoverability write, then the flushed page should not be inserted to the LRU end of Q_S because it is not an eviction candidate. As Q_S is sorted by page H value, we could find the page's position in Q_S by looking through pages in Q_S and comparing the H values. Since recoverability writes are much less common than replacement writes, pages are rarely moving into the SSD by recoverability writes. Hence, we chose a simple approach for GD2L. Instead of finding the accurate position for the page, GD2L simply inserts the page at the midpoint of Q_S and assigns it the same H value as the previous Q_S midpoint page.

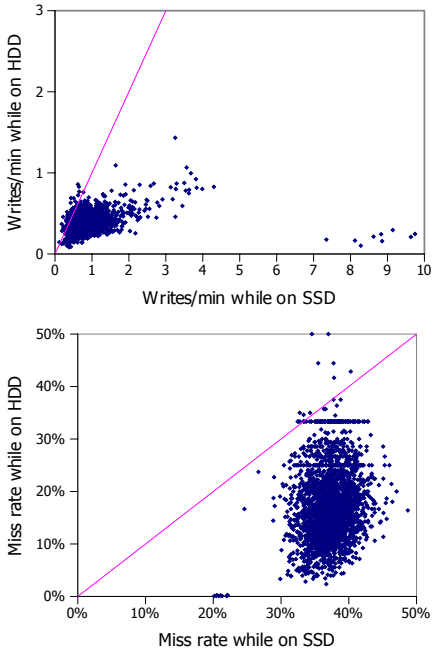

Figure 4: Miss rate/write rate while on the HDD (only) vs. miss rate/write rate while on the SSD. Each point represents one page

It is also possible that a buffered page that is in the SSD will be evicted from the SSD (while remaining in the buffer pool). This may occur to make room in the SSD for some other page. In this case, GD2L removes the page from Q_S and inserts it to Q_D. Since this situation is also uncommon, GD2L simply inserts the page at the midpoint of Q_D, as it does for recoverability writes.

4. THE IMPACT OF COST-AWARE CACHING

Cost-aware caching algorithms like GD2L take into account page location when making replacement decisions. As a result, a page's physical read rate and write rate might be different after its location changes. In this section, we address the following questions: if the buffer pool uses the GD2L caching algorithm, how do pages' physical read rates change when they are placed in the SSD? GD2L also changes the mechanism for asynchronous cleaning of dirty pages. How does this impact the pages' write rates?

To study the impact of the GD2L algorithm on the page access pattern, we drove the modified InnoDB with a TPC-C workload, using a scale factor of 10. The initial size of the database was approximately 1GB. For managing the SSD, we used the policy that will be described in Section 5.

In our experiments, we set the buffer pool size to 200MB, the SSD size to 400MB, and the running duration to sixty minutes. During the run, we monitored the amount of time each page spent on the SSD, and its read and write rates while on the SSD and while not on the SSD. We identified pages that spent at least twenty minutes on the SSD and also spent at least 20 minutes not on the SSD (about 2500 pages). We observed the buffer pool miss rates and physical

write rates for these pages. A logical read request on a page is realized as a physical read when the page is missed in the buffer pool. The page miss rate in the buffer pool is defined as the percentage of logical reads realized as physical reads.

Figure 4 shows the pages' buffer pool miss rates while the pages are on the HDD (only) vs. their miss rate while the pages are on the SSD, and the pages' write rates while the pages are on the HDD (only) vs. their write rates while they are on the SSD. From the two graphs we see that most page miss rates and write rates are larger while the page is cached on SSD. This is as expected. Once pages are placed on SSD, they are more likely to be evicted from the buffer pool because they have lower retrieval costs. As SSD pages in the buffer pool are better eviction candidates, the page cleaner needs to flush dirty ones to the storage before they are evicted. As a result, page read and write rates go up while they are cached in SSD.

5. SSD MANAGEMENT

Section 2 provided a high-level overview of the management of the SSD device. In this section, we present more details about SSD management, including the page admission and replacement policies used for the SSD and the checkpoint-based mechanism for recovering SSD meta-data after a failure.

Pages are considered for SSD admission when they are cleaned or evicted from the DBMS buffer pool. Pages are always admitted to the SSD if there is free space available on the device. New free space is created on the SSD device as a result of *invalidations* of SSD pages. Consider a clean page p in the DBMS buffer pool. If p is updated and hence made dirty in the buffer pool, the SSD manager invalidates the copy of p on the SSD if such a copy exists and it is identical to the copy of p on the HDD. Invalidation frees the space that was occupied by p on the SSD. If the SSD version of p is newer than the HDD version, it cannot be invalidated without first copying the SSD version back to the HDD. Rather than pay this price, the SSD manager simply avoids invalidating p in this case.

If there is no free space on the SSD when a page is cleaned or evicted from the DBMS buffer pool, the SSD manager must decide whether to place the page on the SSD and which SSD page to evict to make room for the newcomer. The SSD manager makes these decisions by estimating the benefit, in terms of reduction in overall read and write cost, of placing a page on the SSD. It attempts to keep the SSD filled with the pages that it estimates will provide the highest benefit. Our specific approach is called Cost-Adjusted Caching (CAC). CAC is specifically designed to work together with a cost-aware DBMS buffer pool manager, like the GD2L algorithm presented in Section 3.

5.1 CAC: Cost-Adjusted Caching

To decide whether to admit a page p to the SSD, CAC estimates the benefit B, in terms of reduced access cost, that will be obtained if p is placed on the SSD. The essential idea is that CAC admits p to the SSD if there is some page p' already on the SSD cache for which $B(p') < B(p)$. To make room for p, it evicts the SSD page with the lowest estimated benefit.

Suppose that a p has experienced $r(p)$ physical read requests and $w(p)$ physical write requests over some measurement interval prior to the admission decision. If the physical I/O load on p in the past were a good predictor of the I/O load p would experience in the future, a reasonable way to estimate the benefit of admitting p to the SSD would be

$$B(p) = r(p)(R_D - R_S) + w(p)(W_D - W_S) \quad (1)$$

where R_D,R_S,W_D, and W_S represent the costs of read and write operations on the HDD and the SSD (Table 1).

Unfortunately, when the DBMS buffer manager is cost-aware, like GD2L, the read and write counts experienced by p in the past may be particularly poor predictors of its future physical I/O workload. This is because admitting p to the SSD, or evicting it from the SSD if it is already there, will change p's physical I/O workload. In particular, if p is admitted to the SSD then we expect that its post-admission physical read and write rates will be much higher than its pre-admission rates, as was illustrated by the experiments in Section 4. Conversely, if p is evicted from the SSD, we expect its physical I/O rates to drop. Thus, we do not expect Equation 1 to provide a good benefit estimate when the DBMS uses cost-aware buffer management.

To estimate the benefit of placing page p on the SSD, we would like to know what its physical read and write workload would be if it were on the SSD. Suppose that $\widehat{r_S}(p)$ and $\widehat{w_S}(p)$ are the physical read and write counts that p would experience if it were placed on the SSD, and $\widehat{r_D}(p)$ and $\widehat{w_D}(p)$ are the physical read and write counts p would experience if it were not. Using these hypothetical physical read and write counts, we can write our desired estimate of the benefit of placing p on the SSD as follows

$$B(p) = (\widehat{r_D}(p)R_D - \widehat{r_S}(p)R_S) + (\widehat{w_D}(p)W_D - \widehat{w_S}(p)W_S) \quad (2)$$

Thus, the problem of estimating benefit reduces to the problem of estimating values for $\widehat{r_D}(p)$, $\widehat{r_S}(p)$, $\widehat{w_D}(p)$, and $\widehat{w_S}(p)$.

To estimate $\widehat{r_S}(p)$, CAC uses *two* measured read counts: $r_S(p)$ and $r_D(p)$. (In the following, we will drop the explicit page reference from our notation as long as the page is clear from context.) In general, p may spend some time on the SSD and some time not on the SSD. r_S is the count of the number of physical reads experienced by p while p is on the SSD. r_D is the number of physical reads experienced by p while it is not on the SSD. To estimate what p's physical read count would be if it were on the SSD full time ($\widehat{r_S}$), CAC uses

$$\widehat{r_S} = r_S + \alpha r_D \quad (3)$$

In this expression, the number of physical reads experienced by p while it was not on the SSD (r_D) is multiplied by a scaling factor α to account for the fact that it would have experienced more physical reads during that period if it had been on the SSD. We refer to the scaling factor α as the *miss rate expansion factor*, and we will discuss it further in Section 5.2. CAC estimates the values of $\widehat{r_D}$, $\widehat{w_D}$, and $\widehat{w_S}$ in a similar fashion:

$$\widehat{r_D} = r_D + \frac{r_S}{\alpha} \quad (4)$$

$$\widehat{w_S} = w_S + \alpha w_D \quad (5)$$

$$\widehat{w_D} = w_D + \frac{w_S}{\alpha} \quad (6)$$

The notation used in these formulas is summarized in Table 5.

An alternative approach to estimating $\widehat{r_S}$ would use only the observed read count while the page was on the SSD (r_S),

Figure 5: Summary of Notation

Symbol	Description
r_D, w_D	Measured physical read/write count while not on the SSD
r_S, w_S	Measured physical read/write count while on the SSD
$\widehat{r_D}$, $\widehat{w_D}$	Estimated physical read/write count if never on the SSD
$\widehat{r_S}$, $\widehat{w_S}$	Estimated physical read/write count if always on the SSD
m_S	Buffer cache miss rate for pages on the SSD
m_D	Buffer cache miss rate for pages not on the SSD
α	Miss rate expansion factor

scaling it up to account for any time in which the page was not on the SSD. While this might be effective, it will work only if the page has actually spent time on the SSD to that r_S can be observed. We still require a way to estimate r_S for pages that have not been observed on the SSD. In contrast, estimation using Equation 3 will work even if r_S or r_D are zero due to lack of observations.

We track reference counts for all pages in the buffer pool and all pages in the SSD. In addition, we maintain an *outqueue* for to track reference counts for a fixed number (N_{outq}) of additional pages. When a page is evicted from the SSD, an entry for the page is inserted into the outqueue. Entries are also placed in the outqueue for pages that are evicted from the buffer pool but not placed into the SSD. Each entry in the outqueue records only the page statistics. When the outqueue is full, the least-recently inserted entry is evicted to make a room for a new entry.

5.2 The Miss Rate Expansion Factor

The purpose of the miss rate expansion factor (α) is to estimate how much a page physical read and write rates will change if the page is admitted to the SSD. A simple way to estimate α is to compare the overall miss rates of pages on the SSD to that of pages that are not on the SSD. Suppose that m_S represents the overall miss rate of logical read requests for pages that are on the SSD, i.e., the total number of physical reads from the SSD divided by the total number of logical reads of pages on the SSD. Similarly, let m_D represent the overall miss rate of logical read requests for pages that are not located on the SSD. Both m_S and m_D are easily measured. Using m_S and m_D, we can define the miss rate expansion factor as:

$$\alpha = \frac{m_S}{m_D} \tag{7}$$

For example, $\alpha = 3$ means that the miss rate is three times higher pages for on the SSD than for pages that are not on the SSD.

While Equation 7 captures our intuition about increased miss rates for pages on the SSD, we have found that it is too coarse. In Equation 7, α is calculated using the buffer pool miss rates of all database pages, meaning that all pages will be assumed to have the same expansion factor. However, since different tables may have different access patterns and the distribution of page requests is not uniform, this may not be true. As an example, Figure 6 illustrates miss rate

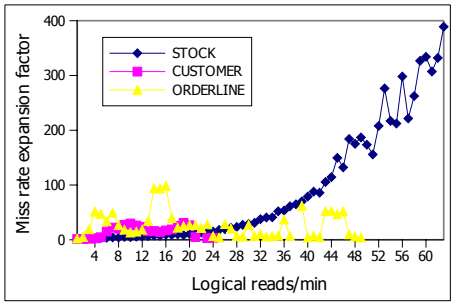

Figure 6: Miss rate expansion factor for pages from three TPC-C tables.

expansion factors of pages grouped by table and by logical read rate. The three lines represent pages holding the TPC-C STOCK, CUSTOMER, and ORDERLINE tables.

Since different pages may have substantially different miss rate expansion factors, we use different expansion factors for different groups of pages. Specifically, we group database pages based on the database object (e.g., table) for which they store data, and on their logical read rate, and we track a different expansion factor for each group. We divide the range of possible logical read rates into subranges of equal size. We define a group as pages that store data for the same database object and whose logical read rates fall in the same subrange. For example, in our experiments, we defined the subrange width as one logical read per minute. If the maximum logical read rate of a table was 1000, this table might have 1000 groups. For each page group g, we define the miss rate expansion factor as in Equation 7:

$$\alpha(g) = \frac{m_S(g)}{m_D(g)} \tag{8}$$

where $m_S(g)$ is the overall miss rate for pages in g while they are in the SSD, and $m_D(g)$ is the overall miss rate for pages in g while they are not in the SSD.

We track logical and physical read counts for each individual page, as well as miss rates for each group. Page read counts are updated with each logical or physical read request to the page. Group miss rates are updated lazily, when certain events occur, using the per-page statistics of the group's pages. Specifically, we update group miss rates when a page is evicted from the buffer pool, when a dirty page is flushed from the buffer pool to the SSD, and when a page is evicted from the SSD. Because pages are grouped in part based on their logical read rates, which can fluctuate, the group to which a page belongs may also change over time. If this occurs, we subtract the page's read counts from those of its old group and add them to the new group.

It is possible that $m_S(g)$ $m_D(g)$ will be undefined for some groups. For example, a group's $m_D(g)$ may be undefined because pages in the group have never been evicted from the buffer pool. We assume that $m_D(g) = 0$ for such groups. Similarly, $m_S(g)$ may be undefined because no pages in the group have been admitted to the SSD. We set $\alpha(g) = 1$ for such groups, which gives a better opportunity for them to be admitted to the SSD, giving us a better chance to collect statistics for them.

A potential efficiency threat is the number of possible groups for which the system must maintain statistics. The

number of possible groups depends on the size of the subrange. If we do not set the subrange size, there is only one group in each table. A smaller subrange size leads to more accurate $\alpha(g)$ at a cost of more space for collecting statistics. In our evaluation (Section 6) we ignore this cost because the space requirement for tracking group statistics was less than 0.01% of the buffer pool size.

5.3 Sequential I/O

Hard disks have substantially better performance for sequential reads than for random reads. To account for this, CAC considers only random reads when estimating the benefit of placing a page in the SSD. In particular, the measured values r_S and r_D used in Equation 3 count only random reads. This requires that the SSD manager classify read requests as sequential or random. Two classification approaches have been proposed in recent work. Canim et al. [2] classify a page request as *sequential* if the page is within 64 pages of the preceding request. Do et al. [5] exploit the existing DBMS *read-ahead* (prefetch) mechanism: a page is marked as sequential if it is read from the disk via the read-ahead mechanism; otherwise, the page is marked as random. Do et al. [5] indicate that leveraging the read-ahead mechanism was much more effective. CAC adopts this approach for identifying sequential reads, using the read-ahead in the InnoDB buffer manager.

5.4 Failure Handling

Since data present in the SSD may be more recent than that in the HDD, the system needs to ensure that it can identify the pages in the SSD after a system failure. TAC [2] does not have such a problem because it writes dirty pages to both the SSD and the HDD. Lazy cleaning, as proposed by Do et al. [5], handles this issues by flushing all dirty pages in the SSD to the HDD when taking a checkpoint. Neither of these approaches exploits the persistence of the SSD. In contrast, CAC assumes that the contents of the SSD will survive a failure, and it will read the latest version of a page from the SSD after a failure if the page was located there. The challenge with this approach is that the SSD manager's in-memory hash map indicating which pages are in the SSD is lost during a failure. Debnath et al. [4] address this problem by checkpointing the hash map and logging all writes to the SSD. During recovery, the hash map can be rebuilt based on the last written hash map and the log.

CAC's approach is also based on checkpointing, but it does not require logging of changes to the hash map. As each page header includes a page identifier, the hash map can be rebuilt without causing any runtime overhead by scanning all pages in the SSD during the failure recovery process. However, this may substantially increase recovery time. For example, based on the read service time of our SSD, to scan a 32G SSD requires about three minutes. Larger SSDs would introduce proportionally larger delays during recovery. To achieve faster recovery, CAC checkpoints the hash map periodically and also identifies a group of k low priority pages as an *eviction zone* on the SSD. Until the next checkpoint, CAC will evict only pages that fall into the eviction zone. After a failure, CAC initializes its hash map using the most recently checkpointed copy, and then checks the k SSD slots where the eviction candidates were located to identify what is actually there, updating the hash map if necessary. The eviction zone size (k) controls a trade-off between operational overhead and the recovery time. CAC will checkpoint its hash map when all of the eviction candidates in the eviction zone have been evicted from the SSD. Thus, smaller values of k result in more frequent hash map checkpoints, but faster recovery.

6. EVALUATION

In this section, we present an experimental evaluation of GD2L and CAC. Our first objective is to provide some insight into the behavior of our proposed algorithm, which is the combination of GD2L (for the buffer pool) and CAC (for the SSD). Specifically, we wish to address two questions. First, how effective is GD2L relative to non-cost-aware buffer management? Second, when GD2L is used to manage the buffer pool, how important is it to use an anticipatory SSD manager, like CAC, that recognizes that page access patterns change when the page is moved between the SSD and the HDD? Our second objective is to compare the performance of our proposed algorithms (GD2L with CAC) to that of other, recently proposed techniques for managing SSDs in database systems.

To answer these questions, we have implemented a variety of algorithms in MySQL's InnoDB storage manager. For the DBMS buffer pool, we have two alternatives: the original buffer pool policies of InnoDB, which we refer to as LRU, and our implementation of GD2L. For SSD management we have implemented CAC as well as three alternatives, which we refer to as CC, MV-FIFO, and LRU2:

CC: CC is cost-based, like CAC, but it is not anticipatory. That is, unlike CAC it does not attempt to predict how a page's I/O pattern will change if that page is moved between the SSD and HDD. It uses Equation 1 to estimate the benefit of placing a page in the SSD, and evicts the page with the lowest benefit from the SSD when necessary. CC's approach for estimating the benefit of placing a page in the SSD is similar to the approach used by TAC [2], although TAC tracks statistics on a region basis, rather than a page basis. However, CC differs from TAC in that it considers pages for admission to the SSD when they are cleaned or evicted from the buffer pool, while TAC admits pages on read. Also, TAC manages the SSD as a write-through cache, while CC, like CAC, is write-back.

LRU2: LRU2 manages the SSD using the LRU2 replacement policy, as recently proposed by Do et al [5] for their *lazy cleaning* (LC) technique. LRU2 is neither cost-based nor anticipatory. Our implementation of LRU2 is similar to LC. Both consider pages for admission when they are cleaned or evicted from the database buffer pool, and both treat the SSD as a write-back cache. Our LRU2 implementation cleans pages in the SSD only when they are evicted, which corresponds to the least aggressive (and best performing) version of LC implemented by Do et al. in SQLServer. The invalidation procedure used for our implementation of LRU2 differs slightly from LC's in that our implementation invalidates an SSD page only if that page is identical to the version of the page on the HDD.

MV-FIFO: MV-FIFO manages the SSD as a FIFO queue of pages. Pages are admitted to the SSD when they are cleaned or evicted from the database buffer pool. If

the page being cleaned or evicted already exists in the SSD and the existing version is older, the existing version is invalidated. MV-FIFO is neither cost-based nor anticipatory. It was proposed for SSD management by Kang et al. as the basis of their FaCE algorithm [10]. The FIFO organization of the SSD ensures that all writes to the SSD are sequential and hence fast - this is the chief advantage FaCE.

Either buffer pool technique can be combined with any of the SSD managers, and we will use the notation X+Y to refer to the combination of buffer pool manager X with the SSD manager Y. For example, LRU+MV-FIFO refers to the original InnoDB buffer manager combined with SSD management using MV-FIFO.

6.1 Methodology

We used the MySQL database management system, version 5.1.45, with the InnoDB storage engine modified to implement our buffer management and SSD management techniques. MySQL ran on a server with six 2.5GHz Intel Xeon cores and 4GB of main memory, running Ubuntu 10.10 Linux with kernel version 2.6.35-22-generic. The server has two 500GB RPM SCSI hard disks. One disk holds all system software, including MySQL, and the test database. The second disk holds the transaction logs. In addition, the server has a 32GB Intel X25-E SATA SSD The database SSD cache is implemented as a single file on the SSD. All files in InnoDB use unbuffered I/O.

All of our experiments were performed using TPC-C workloads [17]. Each of our experiments involved measuring performance under a TPC-C workload for a given system configuration, TPC-C scale factor, and a combined buffer pool and SSD algorithm. Our primary performance metric is TPC-C throughput, measured as the number of TPC-C New-Order transactions that are processed per minute (tpmC). Throughput is measured after the system has warmed up and reached its steady state performance. We also collected a wide variety of secondary metrics, including device utilizations and I/O counts measured at both the database and operating system levels. Experiment durations varied from from four to seven hours, largely because the amount of time required to achieve a steady state varied with the system configuration and the TPC-C scale factor. After each run, we restarted the DBMS to clean up the buffer pool and replaced the database with a clean copy.

Like Do et al. [5], we have focused our experiments on three representative scenarios:

- database much larger than the size of the SSD cache
- database somewhat larger than the SSD cache
- database smaller than the SSD cache

To achieve this, we fixed the SSD size at 10GB and varied the TPC-C scale factor to control the database size. We used TPC-C scale factors of 80, 150, and 300 warehouses, corresponding to initial database sizes of approximately are 8GB, 15GB, and 30GB, respectively. The size of a TPC-C database grows as the workload runs. The number of TPC-C client terminals was set to twice the number of warehouses. For each of these scenarios, we tested database buffer pool sizes of 10%, 20%, and 40% of the SSD size (1GB, 2GB, and 4GB, respectively).

For experiments involving CAC or CC, the maximum number of entries in the outqueue was set to be the same as the number of database pages that fit into the SSD cache. We subtracted the space required for the outqueue from the available buffer space when using CAC and CC, so that all comparisons would be on an equal space basis. Unless otherwise stated, all experiments involving CAC used an eviction zone of 10% of the SSD cache size.

6.2 Cost Parameter Calibration

As introduced in Sections 3 and 5, both GD2L and CAC rely on device read and write cost parameters (listed in Table 1) when making replacement decisions. One characteristic of an SSD is its I/O asymmetry: its reads are faster than its writes because a write operation may involve an erasing delay. We measure R_S and W_S separately.

To measure these access costs, we ran a TPC-C workload using MySQL and use *diskstats*, a Linux tool for recording disk statistics, to collect the total I/O service time. We also used InnoDB to track the total number of read and write requests it made. As *diskstats* does not separate total service time of read requests and that of write requests, we measured the devices' read service time using a read-only workload. The read-only workload was created by converting all TPC-C updates to queries with the same search constraint and deleting all insertions and deletions. Thus, the modified workload has a disk block access pattern similar to that of the unmodified TPC-C workload. First, we stored the entire database on the SSD and ran the read-only workload, for which we found that 99.97% of the physical I/O requests were reads. Dividing the total I/O service time (from *diskstats*) by the total number of read requests (from InnoDB), we calculated $R_S = 0.11$ms. Then, we ran an unmodified TPC-C workload, and measured the total I/O service time, the total number of reads, and the total number of writes. Using the total number of reads and the value of R_S obtained from the read-only experiment, we estimated the total I/O service time of the read requests. Deducting that from the total I/O service time, we had the total I/O service time spending on write requests. Dividing the total I/O service time on writes by the total number of write requests, we calculated $W_S = 0.27$ms. Similarly, we stored the database on the HDD and repeated this process to determine $R_D = 7.2$ms and $W_D = 4.96$ms. For the purpose of our experiments, we normalized these values: $R_S = 1$, $R_D = 70$, $W_S = 3$, and $W_D = 50$.

6.3 Analysis of GD2L and CAC

To understand the performance of GD2L and CAC, we ran experiments using three algorithm combinations: LRU+CC, GD2L+CC, and GD2L+CAC. By comparing LRU+CC and GD2L+CC, we can focus on the impact of switching from a cost-oblivous buffer manager (LRU) to a cost-aware buffer manager (GD2L). By comparing the results of GD2L+CC and GD2L+CAC, we can focus on the effect of switching from a non-anticipatory SSD manager to an anticipatory one. Figure 7 shows the TPC-C throughput of each of these algorithm combinations for each test database size and InnoDB buffer pool size.

6.3.1 GD2L vs. LRU

By comparing LRU+CC with GD2L+CC in Figure 7, we see that GD2L outperforms LRU when the database is much

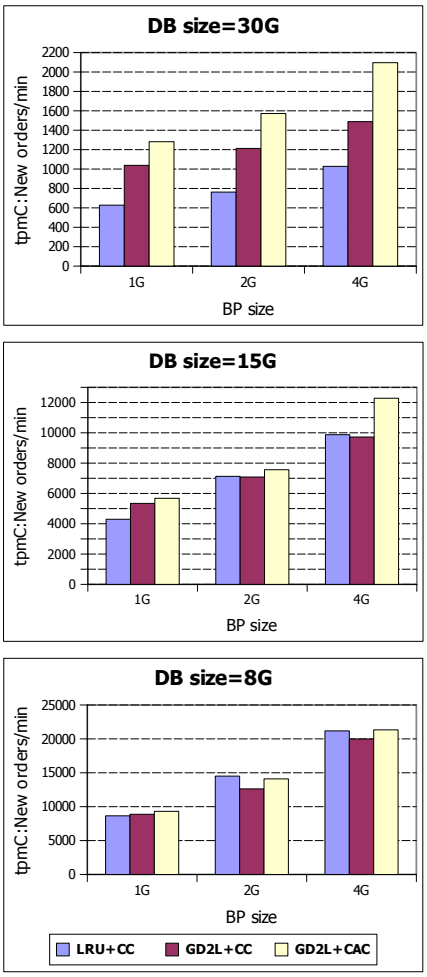

Figure 7: TPC-C Throughput Under LRU+CC, GD2L+CC, and GD2L+CAC For Various Database and DBMS Buffer Pool Sizes.

larger than the SSD. The two algorithms have similar performance for the two smaller database sizes. For the large database, GD2L provides TPC-C throughput improvements of about 40%-75% relative to LRU.

Figures 8 and 9 show the HDD and SDD device utilizations, buffer pool miss rates, and normalized total I/O cost on each device for the experiments with the 30GB and 15GB databases. The normalized I/O cost for a device is the device utilization divided by the New Order transaction throughput. It can be interpreted as the number of milliseconds of device time consumed, on average, per completed New Order transaction. The normalized total I/O cost is the sum of the normalized costs on the HDD and SSD.

For the 30GB experiments, in which GD2L+CC outperforms LRU+CC, Figure 8 shows that GD2L results in a much lower total I/O cost (per transaction) than LRU, despite the fact the GD2L had a higher miss rate in the InnoDB buffer pool. GD2L's higher miss rate is not surprising, since it considers replacement cost in addition to recency of use when making eviction decisions. Although the total number of I/O operations performed by GD2L+CC is higher than that of LRU+CC, GD2L+CC results in less I/O time

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
LRU+CC						
1G	93	88.6	12	11.1	99.7	6.6
2G	93	72.8	8	6.0	78.8	4.4
4G	94	55.1	6	3.3	58.4	2.4
GD2L+CC						
1G	92	53.1	21	12.1	65.2	8.8
2G	90	44.3	20	9.7	54.0	7.4
4G	90	36.3	14	5.8	42.1	4.7
GD2L+CAC						
1G	85	39.6	19	8.8	48.4	7.4
2G	83	31.8	20	7.8	39.6	6.3
4G	82	23.5	20	5.8	29.3	4.8

Figure 8: Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=30GB)
I/O is reported as ms. per New Order transaction.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
LRU+CC						
1G	79	11.1	38	5.4	16.5	4.2
2G	68	5.7	47	4.0	9.7	2.7
4G	73	4.4	43	2.6	7.0	1.3
GD2L+CC						
1G	21	2.5	68	8.0	10.4	6.1
2G	18	1.5	62	5.3	6.8	3.7
4G	14	0.9	61	3.8	4.7	2.3
GD2L+CAC						
1G	30	3.2	73	7.8	11.0	5.7
2G	21	1.6	78	6.2	7.8	4.0
4G	48	2.3	60	2.9	5.3	2.0

Figure 9: Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=15GB)
I/O is reported as ms. per New Order transaction.

per transaction because it does more of its I/O on the SSD and less on the HDD, compared to LRU+CC. This reflects GD2L's preference for evicting SSD pages, since they are cheaper to reload than HDD pages. In the case of the 30GB database, GD2L's shifting of I/O activity from the HDD to the SSD results in significantly higher throughput (relative to LRU+CC) since the HDD is the performance bottleneck in our test environment. This can be seen from the very high HDD utilizations shown in Figure 8.

For the 15GB experiments, Figure 9 shows that GD2L+CC again has lower total I/O cost per transaction than LRU+CC, and shifts I/O activity from the HDD to the SSD. However, the effect is not as pronounced as it was for the larger database. Furthermore, as can be seen from Figure 7, this behavior does not lead to a significant TPC-C throughput advantage relative to LRU+CC, as it does for the 30GB database. This is because the SSD on our test server begins to saturate under the increased load induced by GD2L. (The SSD saturates here, and not in the 30GB case, because most of the database hot spot can fit in the SSD.) In a system with greater SSD bandwidth, we would expect to see a TPC-C throughput improvement similar to what we observed with the 30GB database.

For the experiments with the 8G database, both LRU+CC and GD2L+CC have very similar performance. In those experiments, the entire database can fit into the SSD. As more of the database becomes SSD-resident, the behavior of GD2L degenerates to that of LRU, since one of its two queues (Q_D) will be nearly empty.

6.3.2 CAC vs. CC

Next, we consider the impact of switching from a non-anticipatory cost-based SSD manager (CC) to an anticipatory one (CAC). Figure 7 shows that GD2L+CAC provides additional performance gains above and beyond those achieved by GD2L+CC in the case of the large (30GB) database. Together, GD2L and CAC provide a TPC-C performance improvement of about a factor of two relative to the LRU+CC baseline in our 30GB tests. The performance gain was less significant in the 15GB database tests and non-existent in the 8GB database tests.

Figure 8 shows that GD2L+CAC results in lower total I/O costs on both the SSD and HDD devices, relative to GD2L+CC, in the 30GB experiments. Both policies result in similar buffer pool hit ratios, so the lower I/O cost achieved by GD2L-CAC is attributable to better decisions about which pages to retain on the SSD. To better understand the reasons for the lower total I/O cost achieved by CAC, we analyzed logs of system activity to try to identify specific situations in which GD2L+CC and GD2L+CAC make different placement decisions. One interesting situation we encounter is one in which a very hot page that is in the buffer pool is placed in the SSD. This may occur, for example, when the page is cleaned by the buffer manager and there is free space in the SSD, either during cold start and because of invalidations. When this occurs, I/O activity for the hot page will spike because GD2L will consider the page to be a good eviction candidate. Under the CC policy, such a page will tend to *remain* in the SSD because CC prefers to keep pages with high I/O activity in the SSD. In contrast, CAC is much more likely to to evict such a page from the SSD, since it can (correctly) estimate that moving the page will result in a substantial drop in I/O activity. Thus, we find that GD2L+CAC tends to keep very hot pages in the buffer pool and *out* of the SSD, while with GD2L+CC such pages tend to remain in the SSD and bounce into and out of the buffer pool. Such dynamics illustrate why it is important to use an anticipatory SSD manager (like CAC) if the buffer pool manager is cost-aware.

For the experiments with smaller databases (15GB and 8GB), there is little difference in performance between GD2L+CC and GD2L+CAC. Both policies result in similar per-transaction I/O costs and similar TPC-C throughput. This is not surprising, since in these settings most or all of the hot part of the database can fit into the SSD, i.e., there is no need to be smart about SSD placement decisions. The SSD manager matters most when the database is large relative to the SSD.

6.4 Comparison with LRU2 and MV-FIFO

In this section we compare GD2L+CAC to other recently proposed techniques for managing the SSD, namely lazy cleaning (LC) and FaCE. More precisely, we compare GD2L+CAC against LRU2 and MV-FIFO, which are similar to LC and FaCE but implemented in InnoDB to allow for side-by-side comparison. We combine LRU2 and MV-FIFO with InnoDB's default buffer manager, which results combined algorithms LRU+LRU2 and LRU+MV-FIFO. We also tested GD2L+LRU2.

Figure 10 shows the TPC-C throughput achieved by all four algorithms on our test system for all three of the database sizes that we tested. In summary, we found that GD2L+CAC significantly outperformed the other algorithms in the case of the 30GB database, achieving the greatest advantage over its closest competitor (GD2L+LRU2) for larger buffer pool sizes. For the 15GB database, GD2L+CAC was only marginally faster than LRU+LRU2, and for the smallest database (8GB) they were essentially indistinguishable. LRU+MV-FIFO performed much worse than the other two algorithms in all of the scenarios we test.

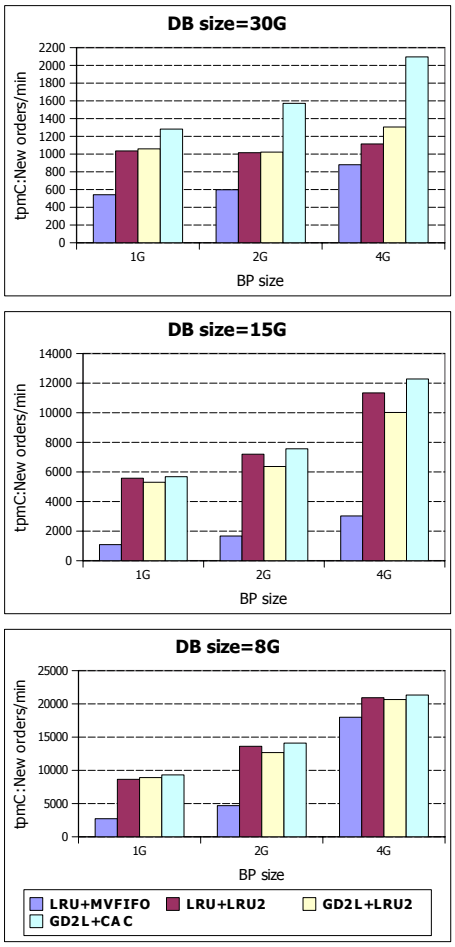

Figure 10: TPC-C Throughput Under LRU+MV-FIFO, LRU+LRU2, GD2L+LRU2, and GD2L+CAC for Various Database and DBMS Buffer Pool Sizes.

LRU+MV-FIFO performs poorly in our environment because the performance bottleneck in our test system is the HDD. The goal of MV-FIFO is to increase the efficiency of the SSD by writing sequentially. Although it succeeds in doing this, the SSD is relatively lightly utilized in our test environment, so MV-FIFO's optimizations do not increase overall TPC-C performance. Interestingly, LRU+MV-FIFO performs poorly even in our tests with the 8GB database, and remains limited by the performance of the HDD. There are two reasons for this. The first is that MV-FIFO makes poorer use of the available space on the SSD than LRU2 and CAC because of versioning. The second is disk writes due to evictions as SSD space is recycled by MV-FIFO.

Figure 11 shows the device utilizations, buffer hit rates and normalized I/O costs for the experiments with the 30GB database. LRU+LRU2 performed worse than GD2L+CAC

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
GD2L+CAC						
1G	85	39.6	19	8.8	48.4	7.4
2G	83	31.8	20	7.8	39.6	6.3
4G	82	23.5	20	5.8	29.3	4.8
LRU+LRU2						
1G	85	49.4	15	8.6	58.0	6.7
2G	87	50.3	12	6.7	57.0	4.4
4G	90	48.5	9	4.7	53.2	2.4
GD2L+LRU2						
1G	73	41.1	43	24.6	65.8	10.7
2G	79	46.5	32	18.9	65.3	8.8
4G	79	34.4	32	13.8	48.2	7.6
LRU+FIFO						
1G	91	101.3	8	9.2	110.5	6.6
2G	92	92.3	6	5.8	98.1	4.4
4G	92	62.9	5	3.7	66.6	2.5

Figure 11: Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=30GB)
I/O is reported as ms. per New Order transaction.

in the 30GB database test because it had higher total I/O cost (per transaction) than GD2L+CAC. Furthermore, the additional cost fell primarily on the HDD, which is the performance bottleneck in our setting. Although it is not shown in Figure 11, GD2L+CAC did fewer reads per transaction on the HDD and more reads per transaction on the SSD than did LRU+LRU2. This may be due partly to CAC's SSD placement decisions and partly to GD2L's preference for evicting SSD pages. In the 30GB tests, the performance of LRU+LRU2 remained relatively flat as the size of the database buffer pool was increased. We observed that SSD read hit ratio of LRU2 dropped as the buffer pool size increased. One potential reason is that the effectiveness of LRU2 (recency-based) is reduced as the temporal locality in the request stream experienced by the SSD is reduced as the buffer pool gets larger. Another reason for this is that LRU+LRU2 generated more write traffic to the HDD because of SSD evictions than did GD2L+CAC.

GD2L+LRU2 had worse performance than LRU+LRU2 and GD2L+CAC in most cases. In Figure 11, GD2L+LRU2 shows higher I/O cost, which is caused by the higher I/O cost on the SSD. Unlike CC and CAC, LRU2 flushes all dirty pages to the SSD. When hot dirty pages are flushed to the SSD, GD2L evicts them from the buffer pool, and then reloads to the buffer pool quickly. As a result, GD2L+LRU2 tends to retain hot pages on the SSD and keeps bouncing them into and out of the buffer pool. With a cost aware buffer policy, hot pages cause more I/O cost when on SSD than on HDD.

When the database size is 15GB, GD2L+CAC's advantage disappears. In this setting, both algorithms had similar per-transaction I/O costs. GD2L+CAC directed slightly more of the I/O traffic to the SSD than did LRU+LRU2, but the difference was small. For the 8GB database there was no significant difference in performance between the two algorithms

6.5 Impact of the Eviction Zone

To evaluation the impact of the eviction zone, we ran experiments with GD2L+CAC using different eviction zone sizes. In these experiments, the database size was 1GB, the buffer pool size is set to 200M and the SSD cache size is set to 400M. We tested k set to 1%, 2%, 5% and 10% of the SSD size, Our results showed that k values in this range had no impact on TPC-C throughput. In InnoDB, the page identifier is eight bytes and the size of each page is 16K. Thus, the hash map for a 400M SSD fits into ten pages. We measure the rate with which the SSD hash map was flushed, and find that even with $k = 1\%$, the highest rate of checkpointing the hash map experienced by any of the three SSD management algorithms (CAC, CC, and LRU2) is less than three per second. Thus, the overhead imposed by checkpointing the hash map is negligible.

7. RELATED WORK

Placing hot data in fast storage (e.g. hard disks) and cold data in slow storage (e.g. tapes) is not a new idea. Hierarchical storage management (HSM) is a data storage technique which automatically moves data between high-cost and low-cost storage media. It uses fast storage as a cache for slow storage. The performance and price of SSDs suggest that "Tape is dead, disk is tape, flash is disk" [9] seems to have come true.

Some research has focused on how to partially replace hard disks with SSDs in database systems. Unlike DRAM, flash memory is non-volatile, i.e. data stored on flash memory will not be lost in case of a loss of power. Koltsidas et al. [11] assume that random reads to SSD are ten times faster than random writes to HDD, while random writes to SSD are ten times slower than random writes to HDD. They design an algorithm to place read-intensive pages on the SSD and write-intensive pages on the HDD. Canim, et al. [1] introduce an object placement advisor for DB2. Using runtime statistics about I/O behavior gathered by the buffer manager, the advisor helps the database administrator make decisions about SSD sizing, and about which database objects, such as tables or indices, should be placed on limited SSD storage. Ozmen et al. [16] present a database layout optimizer which places database objects to balance workload and to avoid interference. It can generate layouts for heterogeneous storage configurations that include SSDs.

Flash memory has also been used as a lower-tier cache in various settings. In Sun ZFS [12], flash memory has been used as an extension of the main memory. The system populates the flash memory cache as entries are evicted from the main memory cache. FlashCache [6], a product of Facebook, sits between the main memory and the disks and is managed using a LRU/FIFO policy. FlashStore [4] uses SSDs as a write-back cache between RAM and the HDD. It organizes data as key-value pairs, and writes the pairs in a log-structure on flash to improve the write performance. Canim, et al. [2] investigate the use of SSD as a second-tier write-through cache. The most-frequently read pages, identified by run-time I/O statistics gathering, are moved into the SSD. Reads are served from the SSD if the page is in the SSD, but writes need to go to the hard disks immediately. Do, et al. [5] propose lazy cleaning, an eviction-based mechanism for managing an SSD as a second-tier write-back cache for database systems. Kang et al. [10] propose FaCE, an alternative write-back design. FaCE is based on the FIFO replacement algorithm. FaCE invalidates stale pages on the SSD and writes new versions to the end of the FIFO queue. Therefore, FaCE always writes pages sequentially to the SSD, and improves the performance by avoiding random writes. hStorage-DB [13] extracts semantic information

from the query optimizer and query planner and passes it with I/O requests to the SSD manager. The semantic information includes hints about the request type, e.g, random or sequential. hStorage-DB associates a priority with each request type, and the SSD manager uses these priorities when making placement and replacement decisions.

Most replacement policies for the buffer cache, such as LRU and ARC, are cost-oblivious. Existing cost-aware algorithms for heterogeneous storage systems, e.g. balance algorithm [15] and GreedyDual, are proposed for file caching. Cao et al. [3] extend GreedyDual to handle cached objects of varying size, with application to web caching. Forney et al. [7] revisit caching policies for heterogeneous storage systems. They suggest partitioning the cache for different classes of storage according to the workload and performance of each class. Lv, et al. [14] design another "cost-aware" replacement algorithm for the buffer pool. However, it is designed for storage systems that only have SSD, not for heterogeneous storage systems. The algorithm is aware that SSD read costs and write costs are different and tries to reduce the number of writes to the SSD. Thus, its "cost-aware" is in a different sense than GD2L.

Our work builds upon previous studies [15, 3, 2, 5]. The GD2L algorithm for managing the buffer pool is a restricted version of GreedyDual [15], which we have adapted for use in database systems. Our CAC algorithm for managing the SSD is related to the previous cost-based algorithm of Canim et al [2]. CAC is aware that the buffer pool is managed by a cost-aware algorithm and adjusts the its cost analysis accordingly when making replacement decisions.

8. CONCLUSION

In this paper we present two new algorithms, GD2L and CAC, for managing the buffer pool and the SSD in a database management system. Both algorithms are cost-based and the goal is to minimize the overall access time cost of the workload. We implemented the two algorithms in InnoDB storage engine and evaluated them using a TPC-C workload. We compared the performance of GD2L and CAC with other existing algorithms. For databases that our large relative to the size of the SSD, our algorithm provided substantial performance improvements over alternative approaches in our tests. Our results also suggest that the performance of GD2L and CAC and other algorithms for managing SSD caches in database systems will depend strongly on the system configuration, and in particular on the balance between available HDD and SSD bandwidth. In our test environment, performance was usually limited by HDD bandwidth. Other algorithms, like FaCE, are better suited to settings in which the SSD is the limiting factor.

9. REFERENCES

[1] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. An object placement advisor for db2 using solid state storage. *Proc. VLDB Endow.*, 2:1318–1329, August 2009.

[2] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. Ssd bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3:1435–1446, September 2010.

[3] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 193–206, 1997.

[4] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010.

[5] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *Proc. SIGMOD Int'l Conf. on Management of Data*, pages 1113–1124, 2011.

[6] Facebook. Facebook: FlashCache, 2012. `http://assets.en.oreilly.com/1/event/45/Flashcache%20Presentation.pdf`.

[7] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proc. the 1st USENIX FAST*, pages 61–74, 2002.

[8] G. Graefe. The five-minute rule 20 years later: and how flash memory changes the rules. *Queue*, 6:40–52, July 2008.

[9] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, July 2008.

[10] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012.

[11] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1:514–525, August 2008.

[12] A. Leventhal. Flash storage memory. *Commun. ACM*, 51:47–51, July 2008.

[13] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[14] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *Proc. ACM SIGMOD Int'l Conf. on Management of data*, pages 13–24, 2011.

[15] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11:208–230, May 1990.

[16] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. In *Proc. ACM SIGMOD Int'l Conf. on Management of data*, pages 939–950, 2010.

[17] The TPC-C Benchmark. http://www.tpc.org/tpcc/.

[18] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc USENIX ATC*, pages 161–175, 2002.

[19] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.