Integrating SSD Caching into Database Systems

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

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

Abstract

Flash-based solid state storage devices (SSDs) are now becoming commonplace in server environments. In this paper, we consider the use of SSDs as a persistent second-tier cache for database systems. We argue that it is desirable to change the behavior of the database system's buffer cache when a second-tier SSD cache is used, so that the buffer cache is aware of which pages are in the SSD cache. We propose such an SSD-aware buffer cache manager, called GD2L. An interesting side effect of SSD-aware buffer cache management is that the rate with which a page will be evicted or written from the buffer cache will change when that page is moved into or out of the second-tier SSD cache. We also propose a technique, called CAC, for managing the contents of the second-tier cache. CAC is aware that moving pages into or out of the SSD cache will change their physical read and write rates. It anticipates these changes when making decisions about which pages to cache at the second tier.

1 Introduction

Flash-based solid state storage devices (SSDs) are now becoming commonplace in server environments. When SSDs are present, there are several ways in which they could be exploited by database management systems (DBMS). On servers that have *hybrid storage systems*, consisting of both SSDs and disk drives (HDDs), one option for the DBMS is to partition the database between the SSDs and the HDDs, so that each chunk of data is stored persistently either on the SSD or on the HDD, but not both [1, 9, 13]. Alternatively, SSDs can be used as an intermediate cache between the DBMS buffer cache and the HDDs [2, 4, 5, 8, 10, 12]. In this case, the entire database resides on the HDDs, and portion of the database is also cached on the SSDs.

In this paper, we consider the latter scenario, in which the SSD is used as a persistent, intermediate cache. We also assume that both the SSD cache and the HDDs are visible to the database management system, so that it can take responsibility for managing the contents of the SSD cache. This is illustrated in Figure 1. When writing data to storage, the DBMS chooses which type of device to write it to.

The primary focus of previous work has been on how a DBMS should decide which data to cache in the SSD to maximize the system's overall I/O performance [1, 2, 5, 12, 13]. Because SSDs provide much better I/O performance than HDDs, this question is clearly important. However, if the goal is to maximize I/O performance then this work considers only part of the problem, since the DBMS manages two tiers of caching, not one. Most previous work assumes that the policies used to manage the database system's in-memory buffer cache are given and fixed. The goal is to design a caching strategy for the SSD without changing the way the DBMS buffer cache is managed.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

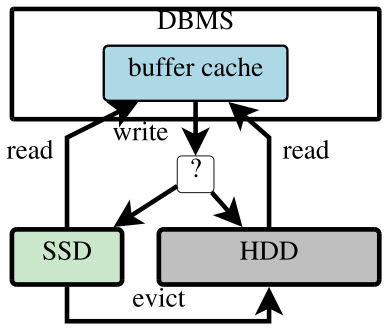


Figure 1: System Architecture

In this paper, we take the broader view, which brings both the database system's in-memory buffer cache and the SSD within scope. We consider two related problems. The first is determining which data should be retained in the DBMS buffer cache. The answer to this question is affected by the presence of an intermediate SSD cache because a database page that is evicted from the in-memory buffer cache can be brought back into the cache much more quickly if it is present in the SSD cache than if it is not. Thus, we propose a *cost-aware* DBMS buffer management technique, called *GD2L*, which takes this distinction into account.

The second problem is deciding which database pages should be retained in the SSD cache. Like some previous work, our approach bases these decisions on page read and write frequencies, attempting to fill the SSD with pages that will provide the greatest boost to I/O performance. However, the key observation is that, if the DBMS buffer cache is cost-aware, then those page read and write frequencies *depend on whether or not the page is present in the SSD*. Specifically, if a page is placed in the SSD cache, its read and write frequencies are likely to increase, since the buffer cache will view the page as a good eviction candidate. Conversely, moving a page out of the SSD cache will probably cause its read and write frequencies to drop. Thus, we propose an *anticipatory* technique, called CAC, for managing the contents of the SSDs. When deciding whether to place a page in the SSD cache, CAC predicts how such a move will affect the page's I/O frequencies and places the page into the SSD cache only if it is determined to be a good candidate under this predicted workload.

In this paper, we try to provide some intuition as to why it is important to consider both cache tiers in order to maximize I/O performance, and we present an overview of GD2L and CAC. More information, including a more detailed performance evaluation, can be found in a longer paper that appeared at VLDB'13 [11].

2 System Overview

As illustrated in Figure 1, we assume that 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 cache. Any given page may have a copy in the SSD, in the buffer cache, or both.

When the DBMS needs to read a page, the buffer cache is consulted first. If the page is cached in the buffer cache, the DBMS reads the cached copy. If the page is not in the buffer cache but it is in the SSD, it is read into the buffer cache from the SSD. If the page is in neither the buffer cache nor the SSD, it is read from the HDD.

If the buffer cache 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 4. 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

	in SSD	not in SSD
in buffer cache	$w_i W_S$	$w_i W_D$
not in buffer cache	$r_i R_S + w_i W_S$	$r_i R_D + w_i W_D$

Figure 2: Total I/O Cost for the ith Page, Depending on Cache Placement

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 cache 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.

3 Page Placement Example

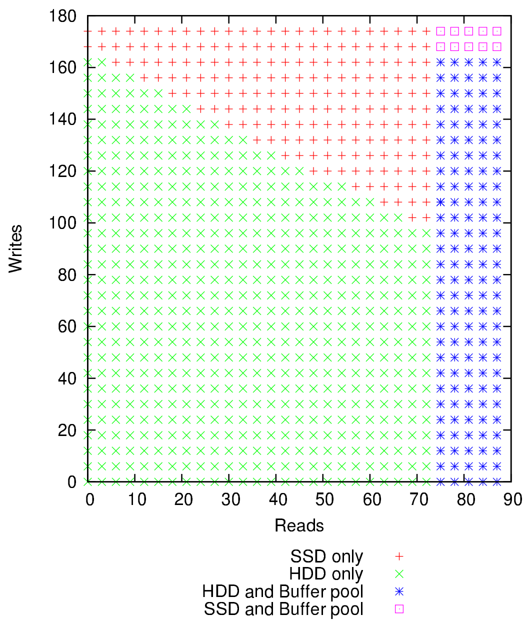
Before presenting the GD2L and CAC algorithms, we first try to develop some intuition for our two-tiered cache problem. To do this, we consider a simple static placement problem in a two-tier cache setting. Clearly, placement decisions will not be static in practice, in either cache. However, by first considering a simple static example, we hope to get some understanding of which pages belong in each cache.

Suppose we have a database consisting of N pages of data, of which at most C_M can fit in the DBMS in-memory buffer cache and C_S can fit in the SSD cache ($N > C_S > C_M$). We assume that a sequence of page read and write requests arrives from the DBMS, and that the ith page is read r_i times and written w_i times in the sequence.

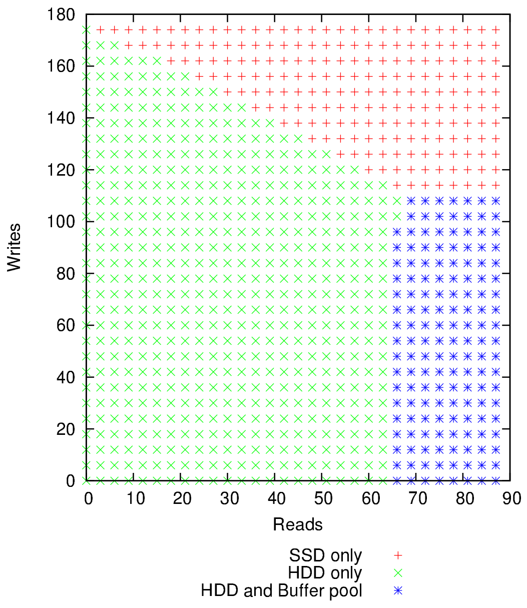
Our goal is to determine which pages to place in each cache so that the total I/O cost of the request sequence is minimized. Placement is static. We'll assume R_D and W_D represent the costs of reading and writing a single page to the HDD, and R_S and W_S are the costs of reading and writing a page to the SSD. With these parameters, we can determine the total cost of the requests for each page, depending on its placement, as shown in Figure 2. The total I/O cost for the entire request sequence is simply the sum of the costs of the pages. Note that if a page is in the SSD, page writes are directed only to the SSD, not to the HDD, since the SSD cache is persistent. In this way, the SSD cache can improve the performance of writes as well as reads.

To illustrate the tradeoffs that arise in solving the two-tiered static placement problem, we consider a single problem instance in which the request sequence is chosen so that a 2D scatter plot of the pages according to their read and write counts will fill the space, as illustrated in Figure 3. This is not intended to be a realistic request sequence. Rather, it is merely intended to illustrate the read/write characteristics of the pages that get placed into each cache. We also choose $N = 900$, $C_S = 200$, and $C_M = 150$, and for I/O costs we choose $R_D = 12$, $R_S = 0.16$, $W_D = 12.5$ and $W_S = 0.4$. (These particular costs were adopted from Graefe [6].)

Figure 3(a) shows a *non-optimal* solution to this problem instance that was produced by a simple two-step process. In the figure, each point represents a page, and the points are coded to indicate where that page was placed in the solution, giving a visual overview of page placement. The first step in the two-step solution process was to determine an optimal placement of pages into memory, under the assumption that no pages are in the SSD, i.e., all pages are stored persistently on the HDD only. The second step in the process is to determine an optimal placement of pages into the SSD, given the memory placement chosen in the first step. This two-step approach is analogous to existing approaches to SSD cache management in which the management of the in-memory cache



(a) Page Placement from Two-Step Technique



(b) Optimal Page Placement

Figure 3: Two Solutions to an Instance of the Two-Tier Static Placement Problem

is taken as fixed and unchangeable.

Figure 3(b) shows an optimal solution to the same problem instance. The key differences between this solution and the two-step solution occur in the top right of the figures. In the optimal solution, pages with high read and write counts are placed in the SSD and not in the in-memory buffer cache. Instead, the buffer cache is used for pages with high read counts and lower write counts. In contrast, the two-step solution places pages with high read and write counts in the buffer cache. Most are not in the SSD, but the most frequently written pages are in both the buffer cache and the SSD.

Of course, this is only a single problem instance, and the placement boundaries will shift as the parameters change. However, this does provide some intuition about how to place pages in the two caches to minimize I/O costs. The optimal solution ensures that frequently written pages are in the SSD, to keep write costs low. It also avoids *cache inclusion* [15], i.e., it avoids keeping the same page in both the buffer cache and the SSD.

4 Buffer Cache Management

Most replacement policies for DBMS buffer caches, such as 2Q [7] or variants of LRU, are cost-oblivious, i.e., they do not consider the cost of reloading a page when making eviction decisions. In this section, we describe our proposed *cost-aware* buffer cache management technique, which we refer to as GD2L. GD2L is based on the GreedyDual algorithm [16], a cost-aware technique that was originally proposed for file caching.

The original GreedyDual algorithm takes into account the access costs of cached objects when making replacement decisions. It associates a non-negative cost $H(p)$ with each cached object p. When p is brought into the cache or referenced in the cache, $H(p)$ is set to the cost of retrieving p. To make room for a new object, the object with the lowest H in the cache, H_{min}, is evicted and the H values of all remaining objects are reduced by H_{min}. By reducing the objects' H values and resetting them upon access, GreedyDual ages objects that have not been accessed for a long time. The algorithm thus integrates temporal locality and cost concerns in a seamless fashion. GreedyDual is usually implemented using a priority queue of cached objects, prioritized based on their H value. With a priority queue, handling a hit and an eviction each require $O(\log k)$ time. Cao et al. [3] have proposed a technique to avoid the cost of subtracting H_{min} from the H values of all cached objects when a page is evicted. That technique involves maintaining a global inflation value L that increases on each eviction, and increasing the initial H value of each newly-cached page by L.

In our setting, the cached objects are database pages, and there are only two possible initial values for $H(p)$: one corresponding to the cost of retrieving p from the SSD (R_S) and the other to the cost of retrieving p from the HDD (R_D). The GD2L algorithm is designed for this special case.

We implemented GD2L in MySQL's InnoDB storage engine. GD2L uses two queues to manage pages in the buffer cache: one queue (Q_S) is for pages that are located on the SSD, the other (Q_D) is for pages that are not on the SSD. Each queue is managed using the scan-resistant variant of LRU that is used by InnoDB. When eviction is necessary, GD2L evicts the page with the lowest H value, which will be located either at the LRU end of Q_S or at the LRU end of Q_D. In part by leveraging the technique proposed by Cao et al, GD2L achieves $O(1)$ time for handling both hits and evictions.

In a DBMS, when pages are modified in the buffer cache (dirty pages), they need to be copied back to the underlying storage device. InnoDB, like many other database systems, uses dedicated by *page cleaner* threads to clean dirty pages asynchronously. Since the page cleaners attempt to clean pages that are likely eviction candidates, we changed the page cleaners to reflect the eviction policies of the new cost-aware replacement policy. Our modified page cleaners check pages from the tails of both Q_S and Q_D. 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.

The original GreedyDual algorithm also assumed that a page's retrieval cost does not change while it is cached, but this is not true in our setting. 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 (e.g, when it is cleaned), then GD2L takes that page out of Q_D and places it into Q_S. It is also possible that a buffered page that is in the SSD will be evicted from the SSD (while remaining in the buffer cache). 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.

5 SSD Management

Pages are considered for SSD admission when they are cleaned or evicted from the DBMS buffer cache. Pages are always admitted to the SSD if there is free space available on the device. If there is no free space on the SSD when a page is cleaned or evicted from the DBMS buffer cache, 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 cache manager, like the GD2L algorithm presented in Section 4.

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.

We refer to a read and write requests issued to the SSD or the HDD as *physical* read and write requests. Suppose that a page p is not currently in the SSD, and has experienced $r(p)$ physical read requests and $w(p)$ physical write requests over some measurement interval prior to the admission decision. If this 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

$$B(p) = r(p)(R_D - R_S) + w(p)(W_D - W_S) \tag{3}$$

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
α	Miss rate expansion factor

Figure 4: Summary of Notation

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

Unfortunately, when the DBMS buffer manager is cost-aware, like GD2L, the physical read and write counts experienced by p in the past may be particularly poor predictors of its future 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. This is because GD2L will be more likely to evict p from the database buffer cache once p has been moved into the SSD. Since p will spend less time in the buffer cache, attempts by the database system to read p will be more likely to result in physical reads, causing p's physical read rate to increase. Since the DBMS page cleaners try to keep likely eviction candidates clean, p's physical write rate will increase as well. Put another way, the *buffer pool miss rate* of p will increase if p is moved into the SSD. Conversely, if a page p that is located on the SSD is evicted from the SSD, then we expect its buffer pool miss rate, and hence its physical I/O rates, to drop. Thus, we do not expect Equation 3 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

$$B(p) = (\widehat{r_D}(p)R_D - \widehat{r_S}(p)R_S) + (\widehat{w_D}(p)W_D - \widehat{w_S}(p)W_S) \qquad (4)$$

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 $\widehat{w_S}(p)$. The notation used in these formulas is summarized in Table 4.

To estimate $\widehat{r_S}(p)$, CAC uses *two* measured read counts: $r_S(p)$ and $r_D(p)$. In general, p may spend some time on the SSD and some time not on the SSD. $r_S(p)$ is the count of the number of physical reads experienced by p while p is on the SSD. $r_D(p)$ is the number of physical reads experienced by p while it is not on the SSD. The total number of physical reads experienced by p during the measurement interval is $r_S(p) + r_D(p)$. To estimate what p's total physical read count would be if it had been on the SSD full time during the measurement interval ($\widehat{r_S}$), CAC uses

$$\widehat{r_S}(p) = r_S(p) + \alpha r_D(p) \qquad (5)$$

In this expression, the number of physical reads experienced by p while it was not on the SSD ($r_D(p)$) 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*. 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, although our implementation of CAC uses a more fine-gained estimate. CAC estimates the values of $\widehat{r_D}(p)$, $\widehat{w_D}(p)$, and $\widehat{w_S}(p)$ in a similar fashion:

$$\widehat{r_D}(p) = r_D(p) + \frac{r_S(p)}{\alpha} \quad (6) \qquad \widehat{w_S}(p) = w_S(p) + \alpha w_D(p) \quad (7) \qquad \widehat{w_D}(p) = w_D(p) + \frac{w_S(p)}{\alpha} \quad (8)$$

6 Performance Evaluation

We performed a variety of experiments to evaluate the performance of CAC and GD2L. All of our experiments were performed using TPC-C workloads [14]. The longer paper [11] includes a more thorough performance evaluation, including tests with different systems configurations and comparisons of GD2L and CAC to other proposed techniques for managing SSDs in database systems. Here, we present two of the experiments from that evaluation.

Our first experiment compares the TPC-C performance obtained by our MySQL test system when it uses different combinations of algorithms for managing its buffer cache and the SSD. Here, we consider three alternatives:

LRU+CC: This uses InnoDB's original, unmodified cost-oblivious algorithm (a scan-resistant variant of LRU) to manage the database buffer cache, and uses CC to manage the SSD. CC is identical to our proposed CAC technique, except that it is non-anticipatory, i.e., it does not attempt to predict the changes in page read and write rates that will occur as the page is moved into and out of the SSD. Specifically, CC uses Equation 3, rather than Equation 4, to estimate the benefit of each page.

GD2L+CC: This uses GD2L to manage the InnoDB buffer cache, and CC to manage the SSD.

GD2L+CAC: This uses GD2L to manage the buffer cache and CAC to manage the SSD.

We used a TPC-C scale factor of 300 warehouses, corresponding to an initial database sizes of approximately 30GB, fixed the SSD size at 10GB, and tested database buffer cache sizes of of 10%, 20%, and 40% of the SSD size (1GB, 2GB, and 4GB, respectively). Thus, in this setting, the database is substantially larger than the SSD.

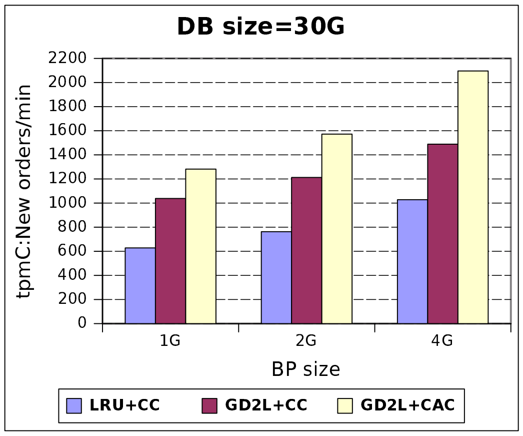
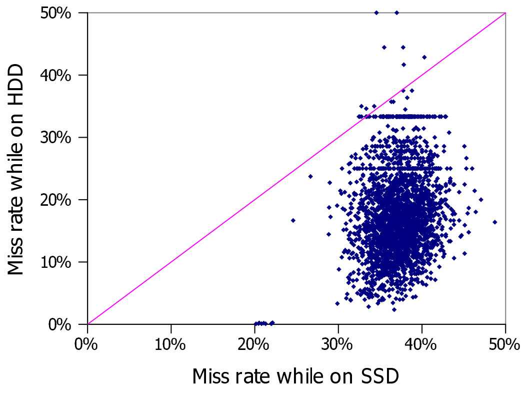
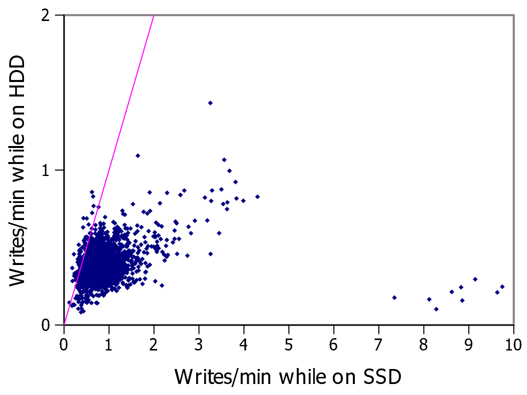


Figure 5: TPC-C Throughput Under Different Algorithm Combinations for Various DBMS Buffer Cache Sizes.

Figure 5 shows the TPC-C throughput of each of these algorithm combinations for each InnoDB buffer cache size. By comparing LRU+CC with GD2L+CC, we can see the benefit that is obtained by switching from InnoDB's original cost-oblivious buffer manager to GD2L. In our setting, this resulted in TPC-C throughput increases of about 40%-75%, depending on the size of the buffer cache. By comparing GD2L+CC with GD2L+CAC, we see the additional benefit that is obtained by switching from a non-anticipatory cost-based SSD manager (CC) to an anticipatory one (CAC). Figure 5 shows that GD2L+CAC provides additional performance gains above and beyond those achieved by GD2L+CC. Thus, it is important to use *both* cost-aware buffer management and compatible SSD manager, like CAC, to obtain the full benefit of the SSD cache. Together, GD2L and CAC provide a TPC-C performance improvement of about a factor of two relative to the LRU+CC baseline in these tests.



(a) Miss rate



(b) Write rate

Figure 6: Miss rate/write rate while on the HDD (only) vs. miss rate/write rate while on the SSD. Each point represents one page

Our second experiment is intended to show how cost-aware buffer managers, like GD2L, cause the read and write rates of database pages to change as those pages are moved into or out of the SSD. For this experiment, we drove our MySQL test system, with GD2L used for buffer management, using a TPC-C workload, and generated a log of all page I/O as well as movements of pages into and out of the SSD. By analyzing this log, we can calculate read and write rates for individual pages. We identified approximately 2000 pages in this trace that spent significant amounts of time both in the SSD cache and not in the SSD cache. For each such page, we calculated the read miss rate (in the DBMS buffer cache) of the page while it was in the SSD cache, and the read miss rate while it was not in the SSD cache. In addition, we calculated the pages physical write rate while in the SSD cache, and its physical write rate while not in the SSD cache.

Figure 6(a) shows scatter plots of the read miss rates of pages while in the SSD cache and while not in the SSD cache. Each point represents a single page. From these graphs, we can see that most pages have higher read miss rates when they are located in the SSD. This is the effect of making the DBMS buffer manager cost-aware: it is more likely to evict pages that are located in the SSD. Figure 6(b) is similar to Figure 6(a), but it shows page write rates rather than miss rates. Again, most pages have higher write rates while in the SSD cache, because GD2L's page cleaners try to clean pages that are likely to be evicted.

7 Conclusion

In this paper we present two new algorithms, GD2L and CAC, for managing the buffer cache and the SSD in a database management system. Both algorithms are cost-based and the goal is to minimize the overall I/O cost of the workload. We implemented the two algorithms in the InnoDB storage engine and evaluated them using a TPC-C workload. Our results show that we can achieve the best I/O performance using both cost-aware management of the DBMS buffer cache (GD2L) and a compatible SSD manager, like CAC. The SSD manager needs to anticipate I/O workload changes that will result from its SSD placement decisions.

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 18–18, 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] G. Graefe. The five-minute rule 20 years later: and how flash memory changes the rules. *Queue*, 6:40–52, July 2008.

[7] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

[8] 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.

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

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

[11] X. Liu and K. Salem. Hybrid storage management for database systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.

[12] 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.

[13] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 939–950, New York, NY, USA, 2010. ACM.

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

[15] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

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