An Efficient Design and Implementation of Multi-level Cache for Database Systems

Jiangtao Wang, Zhiliang Guo, and Xiaofeng Meng(✉)

School of Information, Renmin University of China, Beijing, China
{jiangtaow,zhiliangguo,xfmeng}@ruc.edu.cn

Abstract. Flash-based solid state device(SSD) is making deep inroads into enterprise database applications due to its faster data access. The capacity and performance characteristics of SSD make it well-suited for use as a second-level buffer cache. In this paper, we propose a SSD-based multilevel buffer scheme, called flash-aware second-level cache(FASC), where SSD serves as an extension of the DRAM buffer. Our goal is to reduce the number of disk accesses by caching the pages evicted from DRAM in the SSD, thereby enhancing the performance of database systems. For this purpose, a cost-aware main memory replacement policy is proposed, which can efficiently reduce the cost of page evictions. To take full advantage of the SSD, a block-based data management policy is designed to save the memory overheads, as well as reducing the write amplification of flash memory. To identify the hot pages for providing great performance benefits, a memory-efficient replacement policy is proposed for the SSD. Moreover, we also present a light-weight recovery policy, which is used to recover the data cached in the SSD in case of system crash. We implement a prototype based on PostgreSQL and evaluate the performance of FASC. The experimental results show that FASC achieves significant performance improvements.

Keywords: Solid state driver · Flash · Cost · Cache

1 Introduction

Large-scale data intensive applications have gained tremendous growth in recent years. Since these applications always contain massive random and high-concurrent accesses over large datasets, traditional disk-based database systems(DBMSs) cannot support the system gracefully. Compared with magnetic hard disks, flash memory has a myriad of advantages: higher random read performance, lighter weight, better shock resistance, lower power consumption, etc. Today, flash-based solid

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61379050,91224008); the National 863 High-tech Program (No. 2013AA013204); Specialized Research Fund for the Doctoral Program of Higher Education(No. 20130004130001), and the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010).

M. Renz et al. (Eds.): DASFAA 2015, Part I, LNCS 9049, pp. 160–174, 2015.
DOI: 10.1007/978-3-319-18120-2_10

state device(SSD) is making deep inroads into enterprise applications with its increasing capacity and decreasing price. The unique features of flash memory make it the best storage media for DBMSs. However, the comparatively small capacity and high price hinder flash memory from full replacement of hard disks. Flash memory will be used along with hard disks in enterprise-scale data management system for a long time[1,2]. Therefore, the hybrid storage may be an economical way to deal with the rapid data growth.

In this paper, we propose a flash-aware second-level cache scheme(FASC). The FASC takes flash-based SSD as a non-volatile caching layer for hard disk. When a pages is evicted from main memory, the FASC adopts a data admission policy to selectively cache the evicted page in the SSD for subsequent requests. In this way, the cost of disk access can be reduced due to the faster read/write performance of flash memory. Unlike the traditional single-level buffer management system, the multilevel cache design results in a complicated page flow across the different storage devices. Meanwhile, flash memory has some inherent physical characteristics[3,4], such as erase-before-write limitation, poor random write and limited lifetime. The data scheduling policies designed and optimized for the single-level buffer cache must be carefully reconsidered when switching to a SSD-based multilevel buffer cache. Otherwise, the performance improvement may be suboptimal. How to take full advantages of the flash memory is the key issue and challenge in the design of SSD-based multilevel buffer system.

The main contributions of the paper are summarized as follows:

1. We present a SSD-based multilevel cache strategy, called FASC, which aims to improve I/O efficiency of DBMSs by reducing the penalty of disk accesses. FASC is a hybrid storage system which uses flash-based SSD as an extension of main memory. To reduce the I/O cost for each page eviction, we discuss the different states of the memory pages, and propose a cost-aware buffer replacement policy for the DRAM buffer.
2. We design a memory-efficient data management policy for the SSD. The data cached in the SSD is organized into blocks. Our block-based policy can not only reduce the memory overheads for maintaining the metadata, but also the number of scattered random write operation on flash memory. We also propose a novel hot page identification method to find the pages that are accessed more frequently. In addition, we propose a lightweight metadata recovery scheme to ensure transaction durability.
3. We implement a prototype system based on PostgreSQL, and evaluate its performance using TPC-C and TPC-H benchmarks. The experimental results show that FASC not only provides significant performance improvements, but also outperforms other cache policies presented in recent studies.

The rest of the paper is organized as follows: Section 2 describes the related works about SSD-based hybrid storage system. We present the system overview and some critical components related to FASC in Section 3, while Section 4 details our cost-aware memory replacement policy. In Section 5, we elaborate the data management policy for the SSD buffer. The data recovery scheme designed

for FASC is described in Section 6. We give the results of our experiment evaluation in Section 7. Finally, Section 8 concludes this paper.

2 Related Work

Flash memory based SSD has drawn more and more attention in recent years. Many research works have been done to solve different problems. Some works focus on how to adjust the traditional methods in DBMSs to take full advantage of the unique characteristics of SSDs[5–7]. Recently, some works have attempted to use SSD store frequently accessed data to enhance transactional DBMS performance[8–10]. We cannot cover all the excellent works here, so we just give a brief review for the works related to FASC in this section.

TAC[8] is a SSD-based bufferpool extension scheme to accelerate reads and writes from hard disks by caching frequently accessed pages in SSDs. This work adopts a temperature-based data admission policy to cache the pages evicted by the bufferpool layer. Whenever a page request arrives, TAC computes the regional temperatures of the extent(32 contiguous pages) where the requested page resides. If the extent is identified as a hot region, the page will be written to both SSD and main memory synchronously. TAC adopts a *write-through* caching policy, the dirty page evicted by the DRAM will be directly written to disk. Therefore, it does not reduce the total number of write operation to the disk.

The benefits of using SSD in an extended cache manner have been discussed in work[9]. In that work, the authors propose a lazy cleaning(LC) method to handle the pages evicted by the DRAM buffer. Unlike the TAC, LC adopts a *write-back* caching policy to handle the dirty pages evicted from the main memory. All the dirty pages are only written to the SSD first, and these dirty pages are not flushed to the hard disk until the SSD is full. The SSD data replacement policy adopted by the LC is based on LRU-2 algorithm, which triggers many expensive random writes when replacing a page in SSD. In addition, considering that each mapping record occupies 88 bytes memory space, LC suffers huge memory overheads, which may potentially result in a low hit ratio.

Kang et al. [10] propose a flash-based second-level cache policy, called FaCE, to improve the throughput of DBMS. FaCE adopts a multi-version policy to manage the pages cached in SSD. Whenever a page is evicted from the DRAM buffer, it is written to SSD sequentially in an append-only fashion. Stale copies of the page, if exist, will be invalidated. FaCE manages the SSD buffer in a first-in-first-out(FIFO) fashion. In this way, the expensive random writes on SSD are avoided. However, the FIFO policy is hard to capture the pages that contribute more accesses, which degrades the performance of extended buffer cache system.

Previous works mainly focus on how to flush pages from the DRAM to SSD. However, for a SSD-based multilevel cache, the asymmetric speed of read/write operations on SSD is an important factor to speed up I/O access, and we must reduce the cost of page migration among SSD and DRAM. On the other hand, considering the large capacity of SSD, it is necessary to use memory-efficient data structures and algorithms to manage the page cached in the SSD. In this paper,

we analyze the different page states from the data flow perspective, and discuss the critical issues related to the performance of bufferpool extension system. Based on our observations, we propose a cost-aware data management policy to reduce the overhead of I/O accesses. We also design a lightweight data recovery policy to minimize the negative impact of maintaining the mapping table.

3 System Architecture

The FASC design aims to exploit the high-speed random read of flash memory to enhance the performance of DBMSs. To support large-scale data processing, we take hard disk as the main storage medium. Figure 1 gives an overview of FASC, which employs a three-part storage architecture integrating main memory, SSD, and hard disk. We take SSD as the extension of DRAM due to its fast access speed. When the DRAM buffer is full, a page evicted from the DRAM buffer is admitted to be cached in the SSD buffer only when it is not present in the SSD. If the evicted page is dirty, the evicted page is cached in the SSD until it is written back to the disk. The *write-back* policy can reduce write traffic to disk if the dirty pages cached the SSD are frequently updated. Upon a buffer fault, FASC first checks the SSD to see if the requested page exists. If it does, FASC just reads it from the SSD. Otherwise, the request is forwarded to the disk.

3.1 Architectural Components

The main components of FASC are as the follows:

1)SSD buffer manager: The SSD buffer manager is in charge of data transfer between the SSD and main memory. Specifically, when the DRAM buffer receives a page request, if the requested page does not reside in main memory, FASC will invoke the SSD buffer manager to search the requested page in the SSD buffer. If the DRAM buffer has no available space, the SSD buffer manager is invoked to accommodate the victim pages evicted from main memory. With the increase of incoming evicted pages, the buffer manager needs to reclaim rarely accessed pages periodically.

2) Victim pages buffer: This is an in-memory data structure. It is responsible for accumulating the pages evicted by the DRAM buffer. If the evicted page is clean, the page is admitted into the victim pages buffer only when it is not resident in the SSD. The dirty pages are admitted to the SSD buffer. Note that if an old version of the evicted dirty page exists in the SSD, the old copy of data is marked as invalid. The invalid pages will be recycled by the cleaner thread. When the victim pages buffer is full, FASC flushes the victim pages to the SSD.

3) Temporary buffer: It is a temporary data structure used for migrating the dirty pages to the hard disk. The temporary buffer is sized to 8 times the page size. In the actual implementation, we use double buffers. Once one buffer is full, all the write threads will not be blocked but directed to the second buffer.

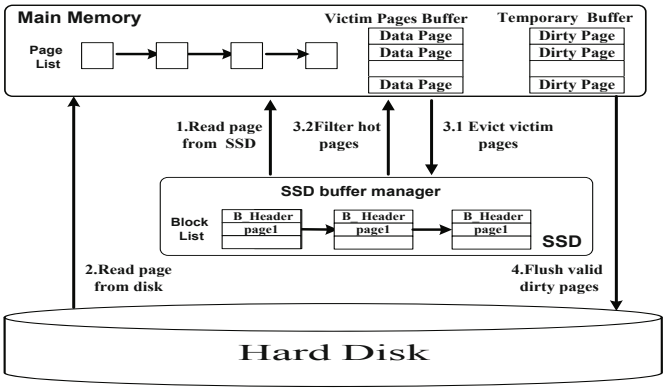

Fig. 1. FASC System Overview

4 Cost-Aware Buffer Management Policy

Buffer is one of the most fundamental component in modern storage systems. The goal of existing buffer replacement policies is to save the retrieval of data from disk by minimizing the buffer miss ratio. Unfortunately, this performance metric is ineffective on a SSD-based multilevel buffer scheme. This is because flash memory provides relatively slow performance for write, and the cost of evicting a dirty page is higher than that of evicting a clean page. This case indicates that maintaining a high hit ratio does not necessarily bring a higher I/O performance. We must revisit the replacement policies to make them fit well in SSD-based multilevel cache system. In fact, the memory pages present different states due to the complicated page flow, and the cost difference on evicting the page with various states is significant. Considering the inconsistency between cache hit ratio and I/O performance, we design a cost-aware replacement policy which is aware of the read/write characteristics of flash memory. Our cost model includes two parts: accessing a SSD page and writing a dirty page to the disk. Except for the latency for reading a page from the SSD, the cost of accessing a SSD page also includes the latency for writing a page to the SSD(if the page is not currently cached in the SSD). Considering that some dirty pages need to be migrated to disk, the latency for data migration is introduced in our cost model.

Based on the analysis of the state of the page, we classifies the memory pages into two types: *SSD-present* page and *SSD-absent* page. Here, a page which has the same copy in the SSD is called a *SSD-present* page, while a *SSD-absent* page represents the page that is not currently cached in the SSD. Considering the asymmetric cost of page migration, we further divide the *SSD-present* pages into two groups: the R_{rc} which keeps the clean pages and the R_{rd} which keeps the dirty pages. Because the R_{rd} page needs to be migrated to disk, its replacement cost is always expensive than that of the R_{rc} page. We also classify the *SSD-absent* pages into two types, say M_{mc} and M_{md}. M_{mc} keeps the clean pages

while M_{md} keeps the dirty pages. Note that evicting a M_{md} page which has an invalid version in the SSD may incur an expensive data transfer overhead, since it may be updated or dirtied multiple times within the buffering period. To reduce write traffic to flash memory, it should have the priority to reside in main memory when executing data eviction. In the following discussion, the cost of reading a page from the SSD is C_r, while the cost of writing a page to the SSD is C_w. D_w represents the cost of fetching a page from the hard disk.

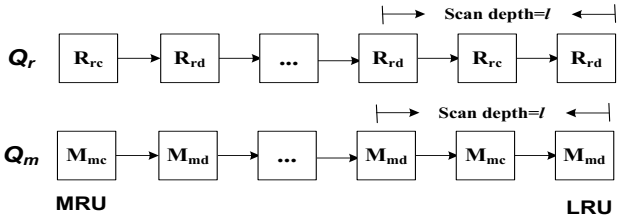

Fig. 2. Cost-based Memory Management

Based on the above analysis, we split the DRAM buffer into two LRU queues: Q_r and Q_m. As shown in Figure 2, Q_r is responsible to maintain the *SSD-present* pages, while Q_m is used to manage the *SSD-absent* pages. The size ratio of the two queues(i.e.,Q_r and Q_m) is dynamically adjusted using the Formula 1:

$$\beta = \frac{C(Q_r) + \theta * M(Q)}{C(Q_m) + \theta * M(Q)} = \frac{C_r + \theta * (C_r + D_w)}{C_r + C_w + \theta * (C_r + D_w)} \tag{1}$$

In Formula 1, the cost of fetching a *SSD-present* page, denoted by C(Q_r), only needs to spend a SSD read; while for a *SSD-absent* page, we must write the page to the SSD first. So, the cost of fetching a *SSD-absent* page from the SSD is $C_r + C_w$(denoted by C(Q_m)). M(Q) represents the cost of migrating a page to the hard disk(i.e., C_r+D_w). To ensure that the size ratio of the two queues can dynamically adapt to different workloads, we add a tuning parameter θ in Formula 1(0< θ < 1). If the workload is update-intensive, there are a lot of dirty pages in main memory, which need to be merged to the disk. In this case, the FASC will reduce the value of θ to keep more memory for the Q_m list.

When the main memory becomes full, if the size ratio of Q_r and Q_m is greater than β, the FASC will select a page from Q_r as the victim page. The cost of searching the victim page in Q_r or Q_m is bounded by a parameter l called the scan depth. Specifically, if Q_r serves as the victim queue, FASC selects the R_{rc} page closest to the end of Q_r region as the victim page. If there is no R_{rc} page within a scan depth, the R_{rd} page closest to the end of Q_r will serve as the victim. If the victim queue is Q_m, the procedure of replacement is similar to the generation of victim page in Q_r. If we do not find a M_{mc} page within a scan depth, the M_{md} page residing in the end of list is selected as victim page.

5 SSD Data Management

Using flash memory as an extended cache is a cost-effective way to improve the performance of system. However, flash memory is not perfect due to its inherent characteristics, such as poor random write. To obtain substantial performance benefits, we need to carefully design the SSD data management policy to reduce the small random write traffic to flash memory.

5.1 Block-Based Data Admission Policy

The small random write pattern is not an ideal one for flash memory due to its erase-before-write limitation. Hence it must be reduced or avoided. The previous study has shown that flash memory can handle random writes with larger I/O request more efficiently[4]. So, we select a larger page size for the SSD buffer. Specifically, when the in-memory victim page buffer is full, all the pages residing in this buffer are gathered into a large data block and then written to the SSD buffer. To reduce the overhead of read operation, we provide an asymmetric I/O pattern. In FASC, each block consists of multiple data pages, and we employ a block header to summarize the pages distribution in the block. The read unit size is a data page, and we can get the requested page through checking its block header. The block-based data admission policy has advantages over a page level policy. First, the capacity of SSD is always much larger than that of the DRAM buffer. In order to keep track of the pages cached on SSD, the memory consumption for the in-memory data structures is very large. Using a large block size can save a lot of main memory by reducing the size of DRAM directory, which in turn increases the hit ratio. Second, using a large block can crease sequentiality in write traffic, which reduces write amplification and improves the performance of SSDs[11]. Third, our block-based policy can reduce the number of erase/write cycles on SSD, which indirectly extends the lifetime of the SSD.

5.2 Memory-Efficient Replacement Policy

As the incoming data blocks consume the space of SSD, the cold pages need to be migrated from the SSD to the disk. In order to find the rarely-accessed pages, the data replacement policy always has to maintain some in-memory data structures to record the access information for the SSD pages(e.g., recency or frequency). Considering the large capacity of SSD, if we directly apply the existing caching policies to the SSD buffer, the overhead of in-memory index is expensive. Therefore, we must provide a memory-efficient cache replacement policy for the SSD buffer.

Hot data identification: The FASC aims to use flash memory as buffer caches to speed-up I/O accesses. If we can identify frequently accessed pages and buffer them in the SSD buffer, the number of disk accesses can be greatly reduced. So, hot data identification plays a crucial role in improving the performance of SSD-based multilevel cache system. Most of existing hot data identification policies typically employ a locality-based algorithm. However, because

all the accesses to the extended buffer are misses from the DRAM buffer, the second-level caches exhibit poor temporal locality. The previous work has shown that pages referenced more than twice contribute more accesses for the second-level buffer cache[12]. Replacement algorithm needs to give higher priorities for the pages that are accessed more frequently.

According to above analysis, we propose a memory-efficient hot data identification algorithm for the SSD buffer. Our hot identification method not only offers a tradeoff between temporal locality and access frequency, but also realizes a low memory consumption used for maintaining the access information. The thorough consideration of our policy makes it suitable for implementing a large SSD-based extended cache. Our hot page identification policy can be divided into two phases: victim block identification and hot page filtration.

(a) Finding victim block: In order to identify and retain the hot pages, the FASC dynamically maintains a hotness-based replacement priority for each block. When the SSD buffer is low in available space, the block with the lowest priority is selected as a victim block. For the block b, its replacement weight *replace_pri(b)* is computed according to Formula 2:

$$replace\_pri(b) = \frac{block\_fre(b) * \theta(t)}{invalidpage\_c(b)} * dirtypage\_c(b) \tag{2}$$

In this formula, the function $block\_fre(b)$ denotes the access frequency of a block b. The aging function, denoted by $\theta(t)$, is used to decrease the replacement priority of a block that has not been accessed for a long time. The FASC dynamically adjusts the value of $\theta(t)$ by scanning the access information maintained in two time windows. We will further elaborate the two time windows in Section(b). The function $invalidpage\_c(b)$ represents the number of invalid pages in block b. If there exists many invalid pages in a block due to frequent updates, the FASC will assign a low priority for it according to Formula 2. $dirtypage\_c(b)$ denotes the number of dirty page in block b. The benefits of keeping dirty page in the SSD tend to be higher than that of keeping clean page.

(b) Filtering hot page: The disadvantage of block-based replacement policy is that some recently accessed pages of the victim block may be evicted from the SSD. We must pick recently accessed pages from the victim block. For this purpose, we need to keep track of the recency of data. In order to reduce the overhead of maintaining access information, we propose a window-based hot page identification policy for the SSD, where a window is defined as a predefined number of consecutive page requests. In this paper, the size of a time window is set to 4096. Unlike previous caching strategies, we only maintain the pages that are accessed within two time windows. For ease of presentation, both time windows are denoted as *Wpre* and *Wcur*, respectively. We maintain two bitmaps to mark the recency of the accessed pages during the two time windows. Each bit in this bitmap indicates whether the page has been recently accessed. A bitmap, called *Bcur*, is used to mark the recently accessed pages within the time window *Wcur*, and all the pages that are accessed during the time window *Wpre* are maintained in another bitmap(called *Bpre*). Periodically, we shift two

bitmaps: discard *Bpre* and replace it with the current bitmap (i.e., *Bcur*), reset *Bcur* to capture the recency within the next window. All the recently accessed pages within two time windows will be found by scanning the two bitmaps. To facilitate fast lookups, we also employ a bloom filter to keep track of the blocks which are accessed within a time window. In this way, the search for a recently accessed page can be avoided when lookups are done on a non-existing block.

Algorithm 1. Evict_SSDPage()

Require: The SSD buffer $SSDBuffer$
Ensure: Evict some pages from $SSDBuffer$

```
1: if the current time window is finished then
2:     for each data block B ∈ SSDBuffer do
3:         if B is not found in BFcur or BFpre then
4:             calculate the value of tuning parameter f;
5:             decay the frequency of B;
6: update the weight heap according to formula 2;
7: find the victim block with the lowest priority victimB;
8: for each page bp ∈ victimB do
9:     if bp can be found in two recency bitmaps then
10:        write bp to the victim pages buffer ;
11: write the cold dirty pages of victimB back to disk;
12: return;
```

Replacement Algorithm: We present the details of replacement policy for the SSD buffer in Algorithm 1. When a read request arrives, if the requested page hits the SSD buffer. FASC updates the recency of the page. But if the SSD buffer receives a write request, our replacement policy checks if the available space on SSD exceeds a predefined threshold. If so, FASC will evict a block out of the SSD to accommodate the incoming pages. To find a proper victim block, FASC performs an aging mechanism for the elder data blocks first, and updates the replacement priority according to Formula 2. The block with the lowest replacement priority serves as the victim block. Next, the FASC checks every page in the victim block and picks out recently accessed pages. The hot pages are retained in the victim page buffer to serve the subsequent requests.

6 Data Recovery for System Failure

In FASC, the dirty pages evicted by the DRAM buffer will be temporarily cached in the SSD. As a result, data consistency and recovery become a concern. Therefore, we also introduce a recovery policy that ensures the system can reuse the pages cached in the SSD during crash recovery. In fact, because the process of transaction recovery always incurs an intensive random read/write operations, the crash recovery can operate with a warm cache if the pages cached in the SSD are recovered. For this purpose, we always need to maintain an accurate SSD

mapping table to keep track of the pages cached in the SSD. Considering the non-volatility of flash memory, the SSD buffer mapping table can be stored in the SSD. To guarantee the validity of the SSD mapping table, a simple approach is to flush, at all times, the updates that belong to the SSD mapping table to the SSD. Such scheme ensures that the SSD buffer manager can quickly recover the data cached in the SSD. However, the drawback of this approach is that it may generate a large amount of additional I/O traffic. In fact, the SSD mapping table may be updated frequently with the increase of incoming pages, and the SSD buffer has to spend an effort to flushing these updates. This may hinder the overall performance of transaction processing.

Based on the above analysis, we must design an efficient method to manage the SSD mapping table. In FASC, each mapping table entry includes block_ID, validation flag that indicates the page whether is valid or not, dirty flag, frequency information, some pointers such as next block pointer, and etc. The FASC organizes the mapping table into a series of fixed-size chunks, and each chunk consists of multiple flash pages. To reduce the overhead of hardening the mapping table, we assign an exclusive flash page for each chunk as the log page. Whenever the most recent update to some mapping table chunk is finished, the SSD buffer manager writes a log record describing the change to the mapping table to its log page. When a log page runs out of free storage space, it is merged with the associated mapping table chunk. Overwriting a mapping table record will most likely incur a random write. By combining the modifications of the mapping table chunk into a single batch update, we avoid a number of small random writes on flash memory, and reduce the negative effect on system throughput.

7 Performance Evaluation

In this section, we implement the FASC policy in PostgreSQL open source DBMS, and evaluate its performance using TPC-C and TPC-H benchmarks. Our experiments are run on a 2.00 GHz Intel Xeon(R) E5-26200 processor with 16GB of DRAM. Operating system (Ubuntu Linux with kernel version 3.2.0) is installed on a Seagate 15K RPM 146 GB hard disk. We use a 128GB Sumsung 840 Pro Series SSD as the second-level buffer cache.

7.1 Prototype Design and Implementation

We implement a prototype based on PostgreSQL, and perform a serial of experiments. A *SSD-buffer-manager* is added to the original buffer manager for PostgreSQL. It is an independent module which provides a rich interface layer to match the buffer manager. We modify the processing of buffer allocation and page eviction. The functions added to the prototype can ensure that all the evicted pages are written to SSD instead of hard disk. We also modify the *BufferAlloc* function, and add a *page flag logic* to indicate the state of a page. An *Adaptive_VictimBuffer* module is introduced in our prototype, which is responsible for searching a victim

page in Q_r or Q_m. We also introduce a recovery component which ensures the durability of transactions in case of system crash. When the data cached in SSD exceeds a certain threshold, a cleaner module is in charge of reclaiming invalid pages. The block size of our policy is set to 1MB.

7.2 TPC-C Evaluation

In the experiments, we implement a throughput test according to the TPC-C specifications, and measure the number of new orders that can be fully processed per minute (tpmC). We run the benchmark with 500 warehouses. The size of database files is approximately 60GB. To achieve a steady-state throughput, each design runs for 1 hour. The PostgreSQL buffer pool size, including 512MB share memory, is set to 1GB. We vary the size of the SSD from 8GB and 16GB.

1) Transaction Throughput: To demonstrate the effectiveness of our cache policy, we compare FASC with two state-of-the-art SSD caching strategies—FaCE[10] and lazy cleaning(LC)[9]. In order to compare the performance, our experiments include the cases where the PostgreSQL is stored on an entire hard disk (denoted by disk-only) or an entire SSD (denoted by SSD-only). Figure 3 shows the transaction throughput achieved by different buffer extension designs. As the figure shows, except for the SSD-only case, our method outperforms all other designs. For example, with a 16GB SSD cache, we observed up to 3.78X improvement in throughput compared to the noSSD design(i.e.,Disk-only). In the following, we analyze the behavior of each caching design. The LC method uses LRU-2 replacement algorithm to manage the SSD pages, and the eviction operation always incurs many costly random write on flash memory due to overwriting a page. This results in a degradation in throughput. Under the FaCE design, although the FIFO-based data management policy ensures that all pages are written to SSD sequentially. However, the replacement policy is hard to capture the frequently accessed pages that contribute more accesses on the SSD. In addition, the multi-version method adopted by FaCE incurs a lot of invalid pages that are scattered over the entire SSD buffer. The cleaner module needs to reclaim the invalid data constantly, which in turn generates a lot of expensive write operations, and seriously aggravates the performance of SSD-based I/O processing. For a 16GB SSD buffer cache, the SSD-only case outperforms our caching design, by roughly 20%, although the database engine is running entirely on the SSD. Therefore, using SSD as the extended cache is a cost-effective and attractive solution to improve the performance of transaction throughput.

To gain a thorough comparative analysis of different SSD designs, we describe the average throughput with a 6 minutes interval, and present the cumulative distribution in terms of tpmC. As shown in Figure 4, our policy gains its peak faster than other policies. After 30 minutes, the performance of FASC slightly decreases, and an explanation for this behavior is that the accumulative pages evicted from main memory soon use up the available SSD space. Thus, the SSD has to consume a significant I/O throughput to handle the page eviction. We can see that the throughput provided by the LC design is very close to that of the FaCE design in the initial 20 minutes, while the performance gap between

LC and FaCE becomes more obvious after 36 minutes. This is because the LC design uses a LRU-2 replacement policy manage the pages cached in the SSD. Hence, evicting a page may incur an expensive random write.

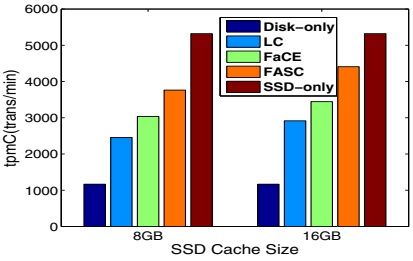

Fig. 3. Transcription Throughput

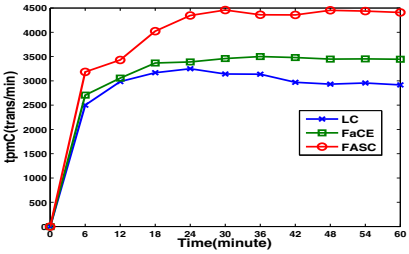

Fig. 4. Throughput Curves over Time

Table 1. SSD Write Ratio and Disk Access Reduction.

(Measured in %)	FASC	FaCE	LC
Ratio of SSD Writes	33.63	41.32	38.41
Disk Access Reduction	62.12	51.39	55.47

2) Memory-efficient SSD Data Management: To demonstrate that our SSD replacement policy is a memory-efficient method, we reduce the size of main memory assigned for PostgreSQL, and repeat the transaction throughput experiment using the same TPC-C instance described above. In this evaluation, 512MB of DRAM is dedicated to the DBMS buffer, and the share memory pool size is limited to 256MB. As we can see from Figure 5, the FASC shows the best performance with respect to transaction throughput, and a speedup of 3.8X is achieved in terms of the tpmC figures when the SSD buffer size is set to 16GB. In contrast, the performance efficiency of the LC design is the worst. We suspect that the main reason lies in the fact that implementing the LC design consumes too much memory, which in turn results in a low DRAM hit ratio. From the results of experimentation, we can see that the metadata overhead is closely related to the throughput of database. Our replacement policy designed for the SSD buffer offers an efficient hot pages identification with low memory overhead, and improves transaction processing performance dramatically.

3)I/O Traffic Reduction: Our cost-aware replacement policy aims to reduce I/O accesses to and from the hard disk, as well as reducing the write traffic to flash memory. In this section, we compare FASC with existing SSD-based caching methods with respect to I/O traffic reduction. As shown in Table 1, the ratio of the SSD write requests to the total SSD I/O requests is lower than those of other policies. An important reason is that our policy considers the asymmetric read/write cost of flash memory when evicting the victim pages, and tends to

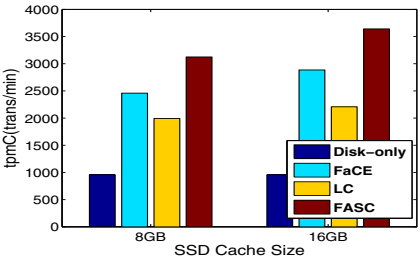

Fig. 5. Throughput with 512MB memory

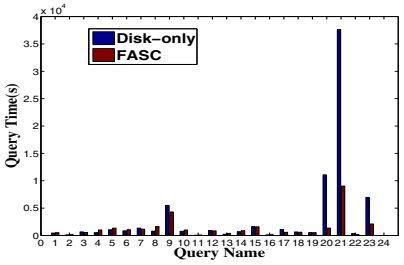

Fig. 6. Concurrency stream(SF=30)

Table 2. Time Taken to Recovery the System(SSD Cache Size=16GB)

Restart Design	recovery time(second)	sustained throughput(tpmC)
FASC	189	4409
FASC_T	203	4122
FaCE	192	3445
Disk-only	847	1166

minimize the data transmission cost between main memory and SSD. In addition, our block-based SSD management policy reduces the memory consumption for maintaining mapping table, which improves the cache hit rate. The low write traffic to SSD contributes to the performance improvement of the FASC. We can see that FASC is approximately 6 to 10 percent higher than FaCE and LC in disk I/O reduction. The LC slightly outperforms the FaCE with respect to I/O traffic reduction. However, the FaCE achieves a higher transaction throughput than the LC method. We suspect that an important reason is that the FaCE adopts an append-only fashion to manage the SSD, which reduces the number of random writes on SSD. The I/O behavior dominated by random write degrades the performance of the LC method. These experimental results explain well why the FASC obtains higher throughput compared to other policies, and the I/O cost reduction is the major factor for performance improvement.

4) Performance of Data Recovery: To evaluate the recovery performance of FASC, we implement FASC with another recovery method(denoted by FASC_T). In FASC_T, an accurate SSD mapping table is stored in the SSD at all times. That is, any update on the mapping table is immediately written in the SSD. Because the LC design employs an in-memory mapping table, it cannot provide a warm SSD cache in case of system crash, and we omit the recovery times evaluation for this design. We compare the three caching designs concerning time taken to restart from a crash. As shown in Table 2, the FASC provides 4.5x speedup over Disk-only(the noSSD cache restarting design). This is because the SSD extension buffer provides a warm cache for DBMS, which accelerates the process of transaction recovery. The recovery times of FASC is similar to other SSD caching methods. However, we can see in Table 2 that

the FASC obtains a higher sustained throughput than FASC_T. An important reason for this is that the FASC implements a light-weight method to harden the mapping table, and our policy reduces the SSD write traffic triggered by continuous updates for the SSD mapping table.

7.3 TPC-H Evaluation

In this section, we evaluate our SSD caching design by conducting experiments with TPC-H benchmark. In our evaluation, the scale factor is set to 30, which occupies a total of 70 GB of disk space. The query stream consists of 22 TPC-H queries and two update queries. As for the platform, we select 8GB SSD and 1GB main memory. We first run a power test by the TPC-H specifications. Then, we run a throughput test by running four query streams concurrently.

Table 3. Power and Throughput test

Scale Factor=30	Disk-only	FASC	FaCE	LC	SSD-only
Power test	336	442	409	374	589
Throughput test	94	316	232	209	402
QphH@30SF	177	374	308	280	487

1) Query performance evaluation: We run the TPC-H evaluation entirely on different designs. Table 3 shows the detailed results. In terms of the throughput test, FASC provides a speedup of 3.36X relative to the Disk-only. The throughput test always needs to handle multiple concurrent queries simultaneously, which creates a lot of random disk accesses. These random disk accesses degrade the system performance. Under the SSD caching scheme, part of the page requests that misses in DRAM can be serviced from the SSD, and thus considerable random disk accesses can be saved. The performance improvement is similar on three caching designs. This is because TPC-H is a read-intensive workload, while each design makes the best use of the excellent read performance of flash memory. The FASC is slightly better than FaCE and LC. An important reason is that our block level data management policy can effectively reduce the metadata overhead. In addition, our policy tends to keep the dirty page in the SSD, which reduces the transmission cost between SSD and disk.

2) Concurrency stream query : In this section, we look into the execution time of each query during the throughput test. As we can see from Figure 6, some queries achieve significant performance improvements on query latency. For example, for Q20 and Q21, the speedup is 8.1 and 4.2 respectively, We analyze the two queries closely, and find that both queries generate intensive random data accesses. Using the SSD buffer can significantly amortize the high cost of disk seeks, and shorten the executing time. It is worth noting that some queries are better than those of Disk-only(but no clear winner is found). This is because that the dominant I/O pattern of these queries is sequential read. In this case, considering the additional transmission costs between SSD and disk, there is no significant performance gain over the SSD caching scheme.

8 Conclusions

In this paper, we propose FASC, a SSD-based multilevel cache scheme, to improve the performance of DBMSs. Following this design principles, we propose a cost-aware replacement algorithm for main memory. To reduce the I/O traffic to the SSD, we implement a block-based data management policy for SSD. We develop a prototype system based on PostgreSQL. The experiment results show that FASC enjoys substantial performance improvements.

References

1. Do, J., Zhang, D.H., Patel, J.M., DeWitt, D.J.: Fast peak-to-peak behavior with SSD buffer pool. In: 30th International Conference on Data Engineering, pp. 1129–1140. IEEE Press, Brisbane (2013)
2. Koltsidas, I., Viglas, S.D.: Designing a Flash-Aware Two-Level cache. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 153–169. Springer, Heidelberg (2011)
3. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M.S., Panigrahy, R.: Design tradeoffs for SSD performance. In: 2008 USENIX Annual Technical Conference, pp. 57–70. USENIX Association, Boston (2008)
4. Bouganim, L., Jnsson, B.T., Bonnet, P.: uFLIP: Understanding flash IO patterns. In: Online Proceedings of the 4th Biennial Conference on Innovative Data Systems Research, pp. 1–12, Asilomar (2009)
5. Chen, S.M.: FlashLogging: exploiting flash devices for synchronous logging performance. In: ACM SIGMOD International Conference on Management of Data, pp. 73–86. ACM Press, Rhode Island (2009)
6. Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query processing techniques for solid state drives. In: ACM SIGMOD International Conference on Management of Data, pp. 59–72. ACM Press, Rhode Island (2009)
7. Lee, S.W., Moon, B., Park, C., Kim, J.M., Kim, S.W.: A case for flash memory ssd in enterprise database applications. In: ACM SIGMOD International Conference on Management of Data, pp. 1075–1086. ACM Press, Vancouver (2008)
8. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: SSD Bufferpool Extensions for Database Systems. Proceedings of the VLDB Endowment **3**(2), 1435–1446 (2010)
9. Do, J., Zhang, D.H., Patel, J.M., DeWitt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: ACM SIGMOD International Conference on Management of Data, pp. 1113–1124. ACM Press, Athens (2011)
10. Kan, W.H., Lee, S.W., Moon, B.: Flash-based Extended Cache for Higher Throughput and Faster Recovery. Proceedings of the VLDB Endowment **5**(1), 1615–1626 (2012)
11. Hu, X.Y., Eleftheriou, E., Haas, R., Iliadis, I., Pletka, R.: Write amplification analysis in flash-based solid state drives. In: Israeli Experimental Systems Conference 2009, pp. 82–90. ACM Press, Haifa (2009)
12. Zhou, Y.Y., Chen, Z.F., Li, K.: Second-Level Buffer Cache Management. IEEE Transactions on Parallel and Distributed Systems **15**(6), 505–519 (2004)