Optimize write performance for DBMS on Solid State Drive

Yu Li

Abstract

Solid State Drive (SSD) is believed to be the replacement for magnetic disk in computer systems. SSD is a complex storage device composed of flash chips, controller hardware, and proprietary software that together provide a block device interface via a standard interconnect. Comparing to its fast sequential read/write and random read operations, the random write on SSD is generally believed to be poor. DBMS applications such as online transaction processing (OLTP) will suffer from it because of issuing random write stream to the storage. It is desirable to improve random write performance for these DBMS applications on SSD. Being different from previous solution in literature, we propose a solution which does not rely on modifying the firmware or hardware of SSD, but tries to decomposite the write streams of good write patterns from the archived write requests in a temporary focused area called StableBuffer. It is implemented as a software model called StableBuffer manager extending DBMS buffer manager. We discuss the motivation, design and implementation of StableBuffer manager, and report preliminary evaluation results.

1 Introduction

Solid State Drive (SSD) is believed to be the replacement for magnetic disk in computer systems. SSD is a complex storage device composed of flash chips (*i.e.*, NAND), controller hardware, and proprietary software (*i.e.*, firmware) that together provide a block device interface via a standard interconnect (*e.g.*, USB, IDE, SATA) [1]. Though it is made by assembling flash chips, the properties of SSD can not be easily derived from properties of flash chips. Because SSD read and write data in fixed sized blocks through a block interface, and integrates layers of software that manage block mapping(*i.e.,*, Flash Translation Layer, FTL), wear-leveling and error correction. In terms of IO performance, SSD does inherent characteristics of other flash storage media such as USB Flash memory and CompactFlash card. In general, sequential read/write and random read are believed to be fast on SSD, while random write on the other hand is believed to be slow.

The performance of random write of SSD is vital to DBMS applications such as online transaction processing (OLTP) . As often observed in OLTP applications, the access pattern to database is random and scattered with small granularity insert/delete/update requests, which cause the poor random write to be the bottleneck of whole system. Therefore, improve the overall write performance on SSD, especially when the write pattern trends to be random, is highly desired. In literature, Lee and Moon [10] studied this problem and proposed an in-page logging solution. The basic idea of in-page logging is that only write update logs inside the flash chip, and merge them with old data page when enough logs are collected. The in-page logging solution actually transfers random write stream to log appending stream which is naturally sequential. Even though lately merging logs into old data page costs extra resources, the transferred sequential logging appending stream actually save much more than random write stream. To implement the in-page logging mechanism, supporting from FTL, which is usually inside the firmware of SSD, is needed. However modifying the functions inside the firmware of SSD is not always possible to current commercial SSD products in market, which limits the in-page solution in practice.

Our focus in this paper is also to improve the write performance for DBMS applications. We try not to modify the firmware of SSD, and only to utilize the characteristics that SSD provides to developer. In the rest of this section, we describe the basic idea of our research by first introducing interesting characteristics of SSD as the motivation.

1.1 Good Write Patterns on SSD

Systematic performance studies [2, 6] show that the performance of IO on SSD is complex. It does not only sensitive to operation type (*i.e.*, read/write), but also sensitive to parameters such as access type (*i.e.*, sequential/random), granularity (*i.e.*, page size) and locality (*i.e.*, the distribution of addresses of accessed data). Recently a research work proposing a micro IO benchmark called uFlip for SSD by L. Bouganim et al [2] shows that, even different kinds of random write patterns perform different on SSD. They actually identified some good write patterns on SSD which could be considered as random. We will review them one by in following paragraphs.

1. *Random write limited to a focused area is fast.* A focused area is a small logical area of access addresses. For example, the space inside a pre-allocated small file smaller than 8MB can be viewed as a focused area. The size of a focus area is usually not bigger than the size of RAM in the controller of SSD. We name this kind of write pattern *focused random write*. According to result reported in uFlip paper, the response time of focused random write is only 1x$\sim$2x of the response time of sequential write, which is far better than the response time (17x$\sim$30x) of general random write to large area.

2. *Partitioned sequential write is fast.* As an example, write sequence "`1`[1]`,50,2,51,3,52,...`" is an instance of partitioned sequential write pattern. It is similar to random write in global, but actually is mixture of two sequential write streams (*i.e.*, "`1,2,3,...`" and "`50,51,52,...`"). There are two sequential write streams, corresponding to two partitions. The response time of partitioned sequential write can be just 1x$\sim$2x of response time of sequential write, as long as there is not too many partitions (8$\sim$16).

3. *Ordered sequential write is fast.* As an example, the write sequence "`1,3,5,7,9,11,...`" is an instance of ordered sequential write pattern. The distance between addresses is $+2$. The distance could also be negative number, such as -2, such as the example "`111,109,107,105,103,101...`". The response time of ordered sequential write can be 1x$\sim$4x of response time of sequential write.

1.2 StableBuffer Idea

It is natural to thinkd about utilizing good write patterns (*i.e.*, sequential write, partitioned sequential write, ordered sequential write and clustered write patterns) to improve the write performance of SSD for DBMS applications. Suppose that we have several transactions concurrently committing pages to be written on SSD in an OLTP application. By only observing the combined write stream, it is generally random. So if we write pages one by one in their submission orders, the performance could be very poor. But actually some transactions may originally issue pages in good write pattern, or several write streams from different transactions may be combined to form instances of good write patterns. For instance, clustering pages by destination addresses may form focused random write stream. Then if we have the chance to delay each write request a little time, and collect enough write requests, we may either decomposite instances of good write patterns, or rearrange the write requests to follow good write patterns. Therefore we can only issue write requests to SSD by following good write patterns and the performance is improved.

However as the write requests are issued by online transactions, we usually have to flush them immediately to disk. Therefore we think about to write the pages temporarily first to a well managed place. In particular, we write them first to *StableBuffer*. StableBuffer is a temporary pre-allocated storage on SSD, and we limit it to be a focused area. In implementation, it can be implemented as small pre-allocated temporary file. Now with StableBuffer, the detailed process of writing a page will be

1. Page will be temporarily written to StableBuffer when it is issued. Since StableBuffer is a focused area, the temporarily write follows the focused random write pattern. And we keep some in memory data structures to record the temporarily write.

2. After we collect enough pages (usually when the StableBuffer is full), we decomposite some instance of good write pattern from StableBuffer. Pages of this instance will be read from StableBuffer and write again to their real destinations. The involved read request is a random read. The involved write request follows good write pattern.

We can roughly estimate the cost of write one page through StableBuffer. We denote the cost of sequentially writing one page as C. To flush one page to its destination through StableBuffer, there are two writes following good write patterns on SSD. According to the uFlip result, the cost is at most $4 \times C \times 2 = 8C$. There is another random read. But random read is even faster than sequential write on SSD. We at most add C cost to the total most, which is $C + 8C + C = 9C$. Now consider that we directly flush the page to SSD. Since it could follow a random write pattern, the cost could be as big as $13C \sim 30C$ according to the result of uFlip paper. We can easily notice that, although it writes same page twice, the StableBuffer idea is still able to improve the overall performance.

It is remarkable that the StableBuffer idea does not require modification on firmware and hardware of SSD. We can implement it as a software model inside the buffer manager of DBMS, which is the focus of our current research. In the rest of this paper, we discuss the design of StableBuffer manager in section 2 and 3. After that, we present some preliminary evaluation result in section 4 and list the related work in section 5. Finally, we conclude the paper in section 6.

[1]We use simple number to denote the logical address of a page.

2 System Overview

Fig. 1 gives an overview of the StableBuffer manager. It is an extension of traditional DBMS buffer manager. New components such as *StableBuffer*, *StableBuffer Translation Table* and *Write Stream Decompositors* are added.

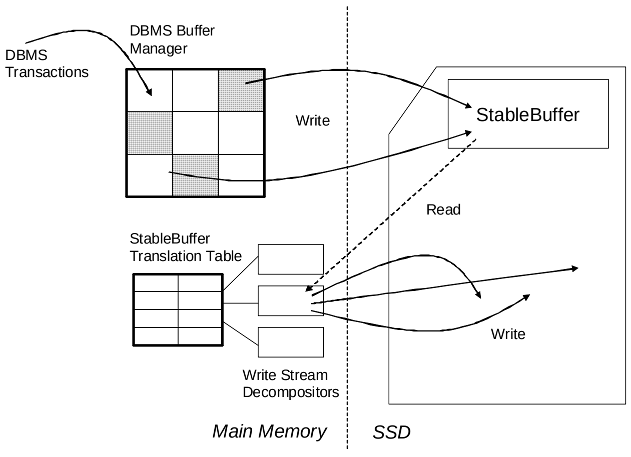

Figure 1: The System Overview of StableBuffer Manager

1. *StableBuffer* is an pre-allocated focused area on SSD. It is usually implemented as small pre-allocated temporary file. The size of the StableBuffer should not exceeds the size of the RAM in the controller of SSD. For example, in a 16GB MTron MSD-SATA-3525 SSD, the size should be less than 8MB. The StableBuffer will be accessed in the granularity of pages, which has the same size as pages in buffer manager of DBMS.

2. *StableBuffer Translation Table* is a data structure maintaining the mapping between the offset of page inside the StableBuffer and the destination address. For example, if a page is issued to write to address `12345678AB` but now temporarily written to the `32`th slot of StableBuffer, a mapping entry "`<12345678AB, 32>`" is inserted into StableBuffer Translation Table. In order to facility fast lookup, insert and delete, StableBuffer Translation Table can be implemented as a hash table.

3. *Write Stream Decompositors* are programs finding instances of good write patterns. They run in concurrent threads, scan the entries of StableBuffer Translation Table and try to decomposite instances of good write patterns. When StableBuffer runs out of slots, some instance of good write patterns will be selected to be read its pages from StableBuffer and then written to their destinations. Thus some of the slots of StableBuffer will be freed for new coming pages.

Now we elaborate how a page is written through StableBuffer manager. Consider that a dirty page is issued by a transaction (as you can see in up left corner of Fig. 1). First it goes through a traditional DBMS buffer manager and may result as a random write to SSD. When we get the write request, we try to write the page to StableBuffer (shown in up center of Fig. 1). If we find a free slot in StableBuffer, the page is written and a new mapping entry will be inserted into StableBuffer Translation Table. Otherwise, we have to free some slots of StableBuffer first. The Write Stream Decompositors will be examined one by one to select an instance of good write patterns (the details will be discussed in section 3). With an instance, we read its pages from StableBuffer, write them to their destinations according to the mapping entries in StableBuffer Translation Table. After that we delete these entries from StableBuffer Translation Table. Now we have free slots of StableBuffer and can write the issued page to it. In either case, we only write pages to StableBuffer, which is a focused area, or write pages in good write patterns. Though we pay extra IO and computation cost in managing StableBuffer, it is still could be better than directly write pages in the order they are issued.

Notice that when write a page into StableBuffer, besides inserting mapping entry to StableBuffer Translation Table, its destination address and a timestamp should also be embedded into the metadata of the page. This is to serve the recovery of StableBuffer Translation Table after the system crush. For a page at offset O whose destination address is D, by comparing its timestamp T to the latest update time T_0 of page at destination D, we know whether the page is swapped out from StableBuffer. In detail, if $T > T_0$, the page is not swapped out yet. Therefore we can insert a mapping entry "$<D, O>$" to StableBuffer Translation Table. Otherwise (*i.e.*, $T \leq T_0$), since the page is already swapped out, we can mark the Oth slot of StableBuffer as free.

Because pages could be temporary stored in StableBuffer, we have to query StableBuffer when answer the request of retrieving page by transaction through DBMS buffer manager. In detail, when get a request of retrieving some page at D, first we need to check whether there is an entry "$<D, O>$" in StableBuffer Translation Table. If there is, we read page at Oth slot in StableBuffer and return it. Otherwise we issue a read request to SSD for the page at D. By implementing StableBuffer Translation Table as a hash table on D, this query to StableBuffer is efficient and will not harm overall performance.

3 Write Stream Decompositors

Fig. 2 illustrates how write stream decompositors work in StableBuffer manager. Write Stream Decompositors are programs which scan the entries of StableBuffer Transla-

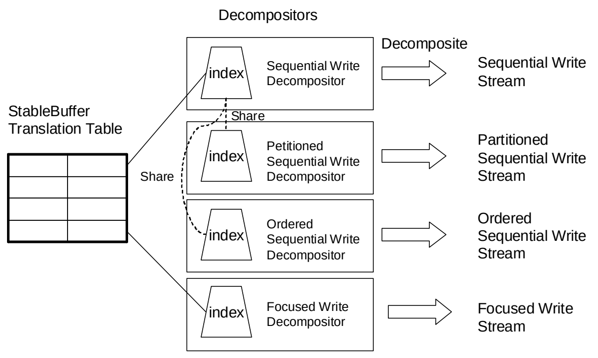
Decompositors
Decomposite
StableBuffer Translation Table
Share
Share
index
Sequential Write Decompositor
Petitioned Sequential Write Decompositor
Ordered Sequential Write Decompositor
Focused Write Decompositor
Sequential Write Stream
Partitioned Sequential Write Stream
Ordered Sequential Write Stream
Focused Write Stream

Figure 2: The Framework of Decompositors

tion Table for instances of good write patterns. They are designed to work concurrently, and may identify different instances of different good write patterns sharing same set of entries. They may require local data structures to track on relevant information of entries in StableBuffer Translation Table. For example, a sequential write decompositor may build a search tree index on destination addresses of mapping entries to facility the searching process. These data structures can be viewed as an index on entries of StableBuffer Translation Table. Some decompositor can share other's index to save the system recourses. E.g., partitioned sequential write can reuse the index of sequential write decompositor to search sequential write instances. As every index should be updated whenever insert/delete entry to/from StableBuffer Translation Table, sharing index between decompositors can save both computation resource and memory.

With multiple instances of good write patterns, selecting which instance to swap out from StableBuffer becomes a non-trivial problem. Intuitively, we trend to select the instance of write pattern which performs the best on SSD. For example, if there are instances of sequential write and focused random write pattern, we should select the instance of sequential write, because the performance of sequential write is usually better than focused random write. On the other hand, how many slots the instance can free from StableBuffer is also important. The reasons are: First the write sequence of local short instances of good write patterns could be random in global. For example, consider the write sequence "`1,2,56,57,6,7,42,43,3,4,...`". It is random in global, but actually could be generated by every time selecting instance of sequential write with two pages, *i.e.*, "`1,2`", "`56,57`", "`6,7`", "`42,43`", "`3,4`" and so on. Second selecting short instances may cause frequent demands on freeing slots for new write requests because each time few slots can be freed. Since updating StableBuffer Translation Table and indices does consume system resources, these frequent demands could cause a degeneration on overall performance. Therefore, in our solution, we select the instances $\{I_i\}$ of different write patterns $\{P_i\}$ based on formula 1,

$$min\{\frac{T_i}{L_i}\} \tag{1}$$

where T_i is average time needed to write single page of the write pattern P_i and L_i is the size of I_i. In particular, we will select the instance with fastest write speed and biggest size.

In our system, we propose four write stream decompositors, corresponding to four good write patterns described in section 1. Following we will discuss the function, design and implementation of them one by one.

Sequential Write Decompositor

This decompositor maintains a search tree index on the destination addresses of mapping entries. The decompositor scans the index in the ascending order of destination addresses to find continuous subsequences on destination address. Each continuous subsequence is corresponding to an instance of sequential write. Finally the largest continuous subsequence will be return as the decomposition result.

Partitioned Write Decompositor

Partitioned write stream is a group of instances of sequential write with same length, so it can share the search tree index of Sequential Write Decompositor. When to decomposite, the Partitioned Write Decompositor first finds continuous subsequences similar to Sequential Write Decompositor. After that, it groups continuous subsequences according to their lengths. Any group with two or more continuous subsequences is then an instance of partitioned write. The largest group will be return as the decomposition result.

Ordered Write Decompositor

Ordered write differs from sequential write in that there is fix distance between destination addresses. But it can also reuse the search tree index of Sequential Write Decompositor. When scan the entries, we try to find not the continuous subsequence, but subsequence with even distance between destination addresses. Finally the largest subsequence will be returned as the decomposition result.

Focused Write Decompositor

Focused Write Decompositor searches for clusters of entries whose destination addresses are focused in small area. It maintains a hash index of entries of StableBuffer Translation Table. In detial, given an entry "$< D, O >$", it will be hashed into bucket $\lfloor D/M \rfloor$, where M is the upper bound of size of focused area of the SSD. It is clear that either a page cluster is inside a bucket, or at most expands to two sibling buckets. When to decomposite, we can efficiently find the biggest cluster by scanning the buckets in ascending order. The biggest cluster will be returned as the decomposition result.

4 Preliminary Performance Evaluation

We have implemented a prototype StableBuffer manager on top of a Windows desktop PC equipped a 16GB MTron MSD-SATA-3525 SSD. In the prototype, StableBuffer is implemented as a pre-allocated temporary file with the size not exceeding 8MB, and it is accessed with a page oriented interface. The prototype can read write stream from write trace file. All four kinds of decompositors described in section 3 are implemented.

In our preliminary performance evaluation, we use the write trace from Oracle 11g DBMS running TPC-C benchmark. The benchmark simulates an enterprise OLTP retailing application, which processes transactions keeping insert/delete/update records from a 8GB database. Therefore the write addresses in write trace expand from 0 to 8GB. In total there are 488623 write requests in our write trace for testing. The page size is set to be 4KB, and the StableBuffer is configured to be 8MB, corresponding to 2048 pages.

We compare the performance of StableBuffer with the Direct method, which processes write requests one by one in their appearing order in the trace file. Fig. 3 shows the evaluation result. We can see that there is a 1.5x performance gain with StableBuffer against Direct.

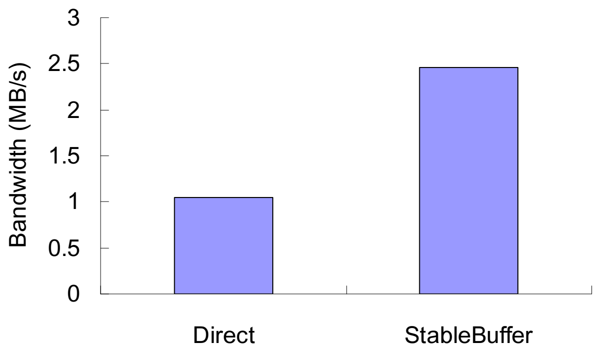

Figure 3: Evaluation result on Oracle 11g TPC-C Trace

5 Related Work

Database management on flash-based storage media has attracted increasing research attention in recent years. Early work focused on assembling flash chips to simulate traditional hard disk [8, 4, 9] and guaranteeing long life span of data [3, 5, 9]. Recent research work starts to tune DBMSs to meet the characteristics of flash disks for a better performance. In view of the asymmetric read/write speed and the erase-before-write limitation, Wu et al. [14] proposed log-based indexing scheme for flash memory. Observing that the log-based indexing scheme is not suitable for read-intensive workload on some flash devices, Nath and Kansal [12] developed an adaptive indexing method that adapts to the workload and storage device. Lee and Moon [10] presented a novel storage design called in-page logging (IPL) for DBMS in order to overcome the possible write issues. S. Chen [7] investigated exploiting flash devices to improve the synchronous logging performance of DBMS.

More recently there are more and more research works focus on improving DBMS performance on Solid State Drive. Agrawal et al. [1] first published a paper on revealing the internal architecture showing that the SSD is indeed very different devices comparing to simple flash-based disks, such as USB flash memory, Compact Flash card and Secure Disk card. uFLIP by Bouganim et al. [2] proposes a microbench mark to help researchers systematically understanding flash IO pattern on SSD. It also provides several interesting hints on the best practices when write pages on SSD in the conclusion part. In parallel, Feng et al. [6] did a similar work to Bouganim. In their work they also point out that fragmentation can cause unignorable performance degeneration. [11] investigated how the performance of standard DBMS algorithm is affected when the conventional magnetic disk is replaced by SSD. [13] presented fast scanning and joining method for relational DBMS on SSD.

6 Conclusion

We focus our study on overcoming poor random write performance of SSD for DBMS applications. Different from previous research work, we propose the StableBuffer solution which does not rely on modifying the firmware or hardware of SSD. It is motivated by systematic study of IO performance of various write patterns on SSD. The basic idea is to decomposite the write streams of good write patterns after archiving the write requests into a temporary focused area called StableBuffer. It is implemented as a software model inside DBMS buffer manager. Preliminary evaluation result shows that there is 1.5x performance gain with StableBuffer manager on TPC-C benchmark trace.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08*, pages 57–70. USENIX Association, 2008.

[2] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[3] L. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC'07*, pages 1126–1130, 2007.

[4] L. Chang, T. Kuo, and S. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[5] Y. Chang, J. Hsieh, and T. Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC'07*, pages 212–217, 2007.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*, pages 181–192. ACM, 2009.

[7] S. Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD '09*, pages 73–86. ACM, 2009.

[8] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[9] H. Kim and S. Lee. A new flash memory management for flash storage system. In *COMPSAC'99*, page 284, 1999.

[10] S. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07*, pages 55–66, 2007.

[11] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD*, pages 1075–1086, 2008.

[12] S. Nath and A. Kansal. Flashdb: Dynamic self-tuning database for nand flash. Technical Report MSR-TR-2006-168, Microsoft Research, 2006.

[13] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD '09*, pages 59–72. ACM, 2009.

[14] C. Wu, T. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.