InnoDB DoubleWrite Buffer as Read Cache using SSDs[*]

Woon-Hak Kang [1], Gi-Tae Yun [1], Sang-Phil Lim [1], Dong-In Shin [1], Yang-Hun Park [1], Sang-Won Lee [1], and Bongki Moon [2]

[1]Sungkyunkwan University, Suwon, Korea
[2]University of Arizona, Tucson, AZ, U.S.A

1 Introduction

As the technology of flash memory solid state drives (SSD hereafter for short) continues to advance, they are increasingly adopted in a wide spectrum of storage systems. Nevertheless, it is still true that the price per unit capacity of flash memory SSDs is higher than that of disk drives, and the market trend is likely to continue for the foreseeable future. Therefore, for applications dealing with large scale data, it may be economically more sensible to use flash memory SSDs to supplement disk drives rather than to replace them. Along this vein, a few studies have been recently proposed to use SSD as a cache between the RAM buffer and harddisk [2], and they are very promising in terms of performance and price.

Meanwhile, a recent empirical study showed that, due to low latency of SSDs, significant performance improvement can be achieved in OLTP databases just by replacing magnetic harddisk with SSD for a certain type of special storage tablespaces, where sequential writes and random reads are prevalent, which are favorable to SSD [4]. Another interesting storage space not covered in the study is the *doublewrite buffer* of InnoDB, a transactional storage engine for the popular MySQL open source DBMS. As explained later, it acts as a backup copy of recently evicted dirty pages for guaranteeing the atomic writes. Also in the doublewrite buffer (*dw-buffer* for short), the write pattern is sequential (the default write unit is 1MB extent) and thus the write pattern itself would not adversely affect the system performance when SSD is used as the storage device of *dw-buffer* .

In this paper, based on these observations, we propose a simple but elegant extension of *dw-buffer* : by deploying SSD as the storage device for *dw-buffer* and enlarging its size (e.g. 5% of total database capacity), we can utilize it as read cache. Now, *dw-buffer* is used to serve the dual purposes, namely, (1) atomic write support and (2) read cache for random read. The evaluation result shows that, only by deploying a commodity SSD as *dw-buffer* , our scheme can improve the performance by more than 50% compared to the harddisk-only InnoDB with 8 raided enterprise-class HDDs.

[*]This work was supported in part by the NRF of Korea grant funded by the Korea government(MEST) (No. 2011-0027613) and (No. 2011-0026492), also supported in part by the the MKE, Korea, under IT/SW Creative research program supervised by the NIPA(NIPA-2011-C1820-1102-0058)

2 InnoDB Doublewrite Buffer and SSDs

For recovery, database systems should guarantee that all the page writes be atomic. However, modern harddisks do not ensure the atomic write of data page of multiple sectors, and thus the *partial page write* problem can be encountered when a multisector page is being written to disk but all the sector writes are yet completed. Therefore, modern database systems have developed various approaches to achieve atomic write for data pages on top of harddisks [5].

One popular scheme to cope with partial page writes is the *doublewrite buffer* of InnoDB [1]. *dw-buffer* is a special reserved area, with two extents each of 1MB size, which is keeping recently written pages to ensure that every page write is atomic and durable against crash or power failure. When InnoDB flushes dirty pages from the RAM buffer to the disk, it writes them synchronously first to *dw-buffer* , then asynchronously to the main data area to which they really belong. InnoDB maintains two extents for the purpose of double buffering: while one extent is being written, the in-memory buffer of the other extent is buffering the dirty pages flushed by the buffer manager. If there is a partial page write to *dw-buffer* itself, the original page will still remain intact it its real location on disk. Thus, during recovery, InnoDB will use the original page instead of the inconsistent copy in *dw-buffer* . However, if the page is successfully written to *dw-buffer* but the write to its real location fails, InnoDB will use the copy in *dw-buffer* during recovery. In InnoDB, each page has a checksum at its end to check whether is is partially written.

From the above description, we can make two observations about the relationship between *dw-buffer* and SSDs. First, because InnoDB writes several pages at once to *dw-buffer* sequentially and only then calls fsync() to sync them to disk, its write pattern is sequential-only, which is favorable to SSD. Second, since each dirty page is written to *dw-buffer* always prior to its main storage, the page copy residing in *dw-buffer* is most recent one and thus can be safely used to serve the read request for the page. Therefore, with moderate size of SSD as *dw-buffer* (e.g. 5% of the total database size), it can be used as read cache, especially considering the random read performance of modern SSDs.

3 Proposed Scheme

Based on observations made in Section 2, we propose to use SSD as the storage device for *dw-buffer* and more im-

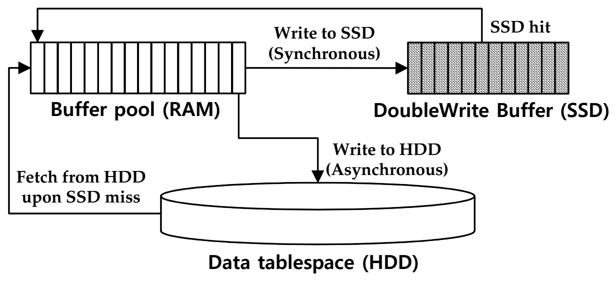

Figure 1: System Overview

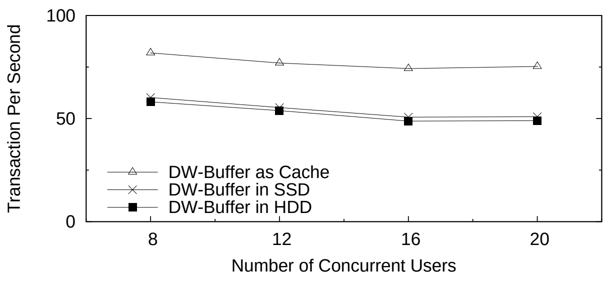

Figure 2: Transaction throughput by user

portantly to enlarge the size of *dw-buffer* enough to cache the recently evicted pages so that the hit ratio from the cache is considerable.

Figure 1 shows the system architecture of our proposed scheme. As shown in Figure 1, *dw-buffer* is separated from the data tablespace in harddisk and is stored in a dedicated SSD. *dw-buffer* is moderately sized (e.g. 5% of the total database capacity) so that it can cache quite a large number of most recently evicted dirty pages, thus being able to provide high hit ratio for the read request from InnoDB buffer manager. In order to keep the mapping information between each page in *dw-buffer* and the main page in harddisk and, when page_id is given, quickly search the corresponding copy from *dw-buffer* , a mapping table and a hash structure, although not depicted in Figure 1, are maintained in main memory. To ensure the durability of the mapping information against crash, the mapping table is periodically check-pointed.

Now, let us explain how read and write operations work in our scheme. For dirty page writes, like the original InnoDB, each page is sequentially written first to *dw-buffer* in SSD, then to its main location in harddisk. But, for page reads, unlike the original InnoDB, *dw-buffer* in SSD is first searched using the hash table. If a copy of the page being read is found there (i.e., SSD hit), the copy is returned. Otherwise, the disk copy is fetched from the main tablespace. As will be shown later, due to the high temporal locality to re-read the pages recently written in OLTP database [3], about 40% of read requests could be served by *dw-buffer* of moderated size (e.g. 10% of total database size) in SSD. From this, we can confirm that *dw-buffer* in SSD is very effective as read cache, taking into account the fast read speed of SSDs.

4 Performance Evaluation

We have implemented this scheme in the MySQL 5.5.1. The benchmark tests were carried out on a Linux system with 2.8GHz Intel i7-860 processor and 1GB DRAM. This platform was equipped with a Samsung S470 Series 128GB MLC SSD and an RAID-0 disk array with eight enterprise-class harddisks (Seagate ST3146356SS). The database size was set to approximately 30 GB (at the scale of 470), and the RAM buffer pool was deliberately limited to 50MB in order to amplify I/O effects. The benchmark database and workload were created by the Benchmark Factory tool.

Both SSD and harddisks were bound as a raw device to minimize interference from data caching in file system.

Figure 2 graphs the number of transactions exectued per second (TPS) of three configurations by varying the number of concurrent users from eight to twenty. In the first configuration ('DW-Buffer in HDD' in Figure 2), *dw-buffer* is stored in harddisk together with main data tablespace. In the second one ('DW-Buffer in SSD' in Figure 2), *dw-buffer* is separated from main tablespace in harddisk and stored in SSD, but its size is not enlarged and thus it consists of two extents each of 1MB. Finally, the third configuration ('DW-Buffer as Cache' in Figure 2) represents our scheme with *dw-buffer* of 1GB in SSD. By comparing the first two configurations, we can observe that no performance improvement is gained only by separating *dw-buffer* from harddisk and storing it in SSD. And, this is not surprising because the sequential write bandwidth of RAIDed eight harddisks (i.e. more than 1GB/sec) surpasses that of the SSD (i.e. 200MB/sec) used in our experiment. Meanwhile, our scheme can improve the performance by more than 50% over the first two configurations regardless the number of concurrent users. And this is due to the high hit ratio from *dw-buffer* used to serve as a read cache for the recently evicted dirty pages and the fast random read speed of SSD.

References

[1] BARON SCHWARTZ, P. Z., TKACHENKO, V., ZAWODNY, J. D., LENTZ, A., AND BALLING, D. J. *High Performance MySQL(2nd ed.).* O'Reilly, 2008.

[2] DO, J., ZHANG, D., PATEL, J. M., DEWITT, D. J., NAUGHTON, J. F., AND HALVERSON, A. Turbocharging DBMS Buffer Pool using SSDs. In *Proceedings of ACM SIGMOD* (2011).

[3] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the TPC benchmarks-an analysis at the logical level. *ACM Transactions on Database Systems 26*, 1 (2001).

[4] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of ACM SIGMOD* (2008).

[5] MOHAN, C. Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging. In *Proceedings of ICDE* (1995).