

RS-Wrapper: Random Write Optimization for Solid State Drive

Da Zhou
Information School
Renmin University of China
Beijing 100872, China
cadizhou@gmail.com

Xiaofeng Meng
Information School
Renmin University of China
Beijing 100872, China
xfmeng@ruc.edu.cn

ABSTRACT

Solid State Drive (SSD), emerging as new data storage media with high random read speed, has been widely used in laptops, desktops, and data servers to replace hard disk during the past few years. However, poor random write performance becomes the bottle neck in practice. In this paper, we propose to insert unmodified data into random write sequence in order to convert random writes into sequential writes, and thus data sequence can be flushed at the speed of sequential write. Further, we propose a clustering strategy to improve the performance by reducing quantity of unmodified data to read. After exploring the intrinsic parallelism of SSD, we also propose to flush write sequences with the help of the simultaneous program between planes and parallel program between devices for the first time. Comprehensive experiments show that our method outperform the existing random-write solution up to one order of magnitude improvement.

Categories and Subject Descriptors: H.2.2 Database Management: Physical Design-Access methods

General Terms: Algorithm, Design, Performance

Keywords: Flash Memory, Database, Random Write, Parallelism

1. INTRODUCTION

SSD, emerging as a new electronic storage device, is widely adopted in laptops and personal computers during the past few years. This mainly benefits from the high read performance of SSD, especially random read performance. As we know, SSD does not have mechanical part like the magnetic head of Hard Disk Drive (HDD), therefore there is no latency for random read of SSD. As a result, random read has similar speed with sequential read. This characteristic improves the read performance of system fundamentally. Besides this, SSD has other attractive characteristics, such as low power consumption, high shock resistance and lightweight form. All of these advantages make SSD as outstanding data storage instead of HDD.

However, the random write performance of SSD, especially small random writes, is very poor as shown in table 1. Read and sequential write is faster than random write in two orders. The costly erase

Table 1: IO Performance Values of Mtron SSD[5]

Sequential Read	Sequential Write	Random Read	Random Write
11,100	16,600	11,200	120

operation of flash memory lays down the main reason for the slow performance of random writes. Erase is peculiar to flash memory. In a word, erase operation has two important characteristics: erase before rewrite and high cost. We must erase the whole block if we want to rewrite pages of it. On the other hand, the cost of erase is 1.5ms, while that of write is only 0.22ms according to Micron electronics datasheet[3]. Besides the cost of erase itself, large quantity of data need to be transferred before erase operation is executed. Although erase operation has high cost and leads to low write performance, every small updates will still trigger erase operation in the worst situation. FTL embedded in SSD can reduce the number of erase by implementing out-of-place update with the help of Physical-to-Logical mapping[4], but the efficiency is very low according to random write performance as shown in table 1. As a result, low performance of random write becomes the bottleneck of wider applications of SSD.

In this paper we propose a novel and efficient method for avoiding random write. Based on the key observation that SSD has high sequential write performance, our method avoids random writes by converting random write sequence into sequential write sequence in order to take full advantage of high performance of sequential write. We novelly insert unmodified data into the random write sequence, which locate between the lower and upper limit of the random write sequence. Finally we flush the constructed sequential write sequence into SSD instead of original random write sequence. Compared with flushing random write sequence, cost of writing the converted sequential write sequence is much lower. Therefore the write performance is enhanced obviously. Density and granularity of write sequence are two key factors of the efficiency of our method in this paper. We also propose to reduce the cost of getting addresses and reading unmodified data further by cluster when density is low than MD(Minimum Density).

We also propose to improve write performance by intrinsic parallelism of SSD further. SSD contains several flash memory devices and one chip is made up of a number of planes. According to this, we explore the simultaneous program between planes and parallel program between devices for the first time. With help of intrinsic parallelism of SSD, we partition sequential write sequence and flush partitions in parallel. Results of comparison experiments show the write performance are enhanced obviously, especially for sequence with low density.

The rest of this paper is organized as follows. Section 2 explains

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'09, November 2–6, 2009, Hong Kong, China.

Copyright 2009 ACM 978-1-60558-512-3/09/11 ...\$10.00.

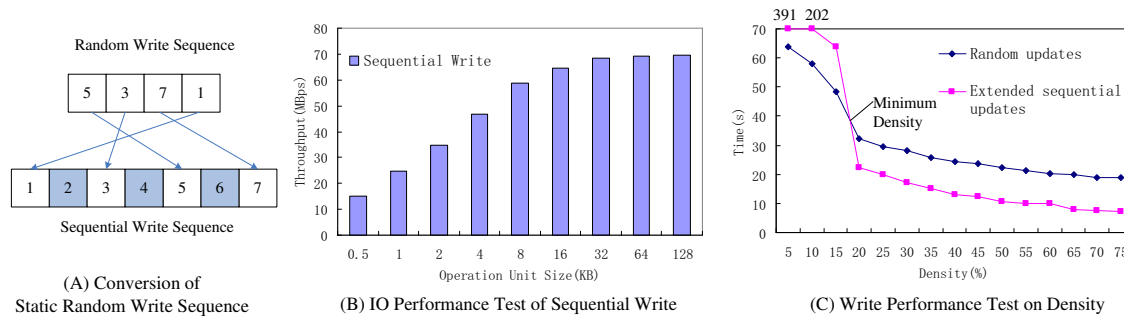


Figure 1: Conversion of static random write sequence and performance experiments

the basic idea. Section 3 expresses our method in detail with data stream. We also enhance performance further in section 4. Experiments are shown in section 5. Finally, section 6 concludes.

2. STATIC RANDOM WRITE SEQUENCE

In this section, we first describe the basic idea of our method by static random write sequence, and then discuss how density and granularity impact the efficiency of our method. Finally, we optimize our method according to density.

2.1 Converting Random Write Sequence

Static random write sequence is defined as a random write sequence without changes during the course of conversion. We insert unmodified data into the random write sequence, which locate between the lower and upper limit of the random write sequence. So we flush the constructed sequential write sequence into SSD instead of original random write sequence. Therefore the cost of write will be reduced due to high sequential write performance. For example, the random write sequence has four data items as shown in Fig. 1(A). The addresses of these items are 5, 2, 7 and 1. It may take four random writes if we flush this write sequence into SSD directly. According to table 1, the cost will be 33 ms. However, after we extend the random sequence with unmodified data items, item 2, 4 and 6, we can flush the sequential write sequence as shown in Fig. 1(A) sequentially. As a result, we only cost seven sequential writes. The cost of flushing is only 0.4 ms ideally. Although the approach is straightforward in concept, when one actually attempts to implement such a facility, one is faced with myriad options and difficult decisions every step of the way. One of important problems is how to decide the granularity and density of random write sequence. We will explore the impact as follows.

2.2 Granularity and Density

Granularity is defined as the write unit when we flush the converted sequential write sequence into SSD. As shown in Fig. 1(B), the throughput becomes higher when we increase the write unit size. The results shows that the write performance of SSD is not taken full use of when the write unit size is less than 32KB. Therefore, 32KB is the optimum granularity for our SSD to perform sequential write. According to this experiment, we define this special write unit as *OSWG* (Optimum Sequential Write Granularity). Different SSDs have different *OSWGs*.

Density is defined as the length ratio of random write sequence to the converted sequential write sequence. Density is direct ratio to the performance. The less the density is, the more unmodified data items are inserted, and the larger the cost of reading and flushing these data is. According to different density, we get run-

time of flushing random write sequence and the converted sequential write sequence. According to the experiment results as shown in Fig. 1(C), the larger density is, the higher the efficiency of our method is. We define the density of intersection point as *MD*. Our method outperforms random write about 100%~150% when density is larger than *MD*. Different SSDs have different *MDs*. The *MD* of the Mtron SSD in our lab is 18%.

2.3 Optimization

According to Fig. 1(C), our method is worse than random write when density is less than *MD*. For example, when the density is 10%, the runtime of our method is as high as 201.7 seconds. In our method, the runtime of getting addresses of each data item, reading unmodified data items and flushing converted sequential write sequence are 6.4, 50.7 and 144.6 seconds, respectively. In this case, we need to read too much unmodified data, and then flush them into SSD. In a word, the large quantity of unmodified data lead to low performance of our method.

A sub-sequence is defined as a *cluster* if the density of it is larger than *MD*. After sorting the random write sequence according to addresses, we generate clusters as follows. Firstly we decide the first data item as a cluster. Secondly, this cluster and its next data item are treated as a new sub-sequence. If the density of it is larger than *MD*, the next data item are grouped into this cluster. If not, the next data item will be grouped as a new cluster. This course cycles until all data items are grouped into clusters. After getting clusters, we convert these clusters into sequential write sequence instead of the whole write sequence. We re-execute the experiments with the help of clusters when density is less than *MD*. Experiments show our method outperforms random write about 10%. This attributes to obvious decrease of the quantity of unmodified data to read, and then the costs of random read and sequential write decrease.

3. DATA STREAM

Section 2 explains our basic idea by static random write sequence. However real applications usually generate writes continuously as data stream. According to characteristics of dynamic write sequence, we first load and initialize the write sequence, and then generate initial clusters. Lastly the final clusters are evolved from initial clusters and flushed into SSD. The above steps repeat until the end of data stream. We will explain each steps in following.

3.1 Initialization

After loading writes from data stream, we need to calculate density. The address of each data item is used to calculate the density of write sequence. Therefore, we need to get the address of each data item in the first step. Actually, this step takes full ad-

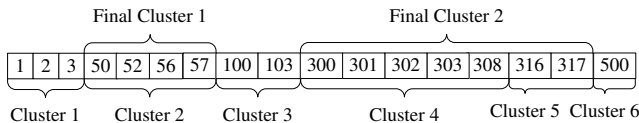


Figure 2: Initialized random write sequence. Data items are grouped into initial clusters, and initial clusters are merged into final clusters.

vantage of characteristics of SSD. During the course of getting an address of a data item, we need to look up it from index, and so several random read operations must be executed. Compared with HDD, SSD gets high performance because of its high random read performance, and then the cost of getting address is low for SSD. After getting addresses of data items, we need to sort the write sequence in cache according to addresses and generate initial clusters as shown in Fig. 2. The numbers denote the addresses of data items.

3.2 Final Cluster

The design objective of final cluster is to avoid small granularity of write. Final cluster is defined as the random write sub-sequence, the length of which is larger than OSWG and the density of which is larger than MD. The basic design principle is the length should be as long as possible in order to take full advantage of high sequential write speed of SSD. Therefore we will get final clusters by merging initial clusters. There are two cases according to the maximum length of initial clusters: The maximum length is no less or less than OSWG. We will discuss them in following.

No Less Than OSWG. The initial clusters are defined as candidate clusters if their lengths are no less than OSWG. In order to lengthen the final cluster, we need to merge adjacent initial clusters into candidate clusters with the grantee of density. The detail step of merger is described as following. For each candidate cluster, we first merge preceding initial clusters into candidate cluster. If the density of merged cluster is less than MD, the course of the merger will be terminated. In the same way, the following initial clusters are merged. For example, as to cluster 4 in Fig. 2, preceding clusters will be merged into cluster 4 firstly. Because the density of cluster 3 \cup cluster 4 is less than MD, the course of merging preceding clusters is terminated. As for following initial clusters, cluster 5 will be merged with cluster 4. At last we get final cluster 2.

Less than OSWG. We need to merge initial clusters into final clusters when lengths of all initial clusters are less than OSWG. [cluster i, cluster j] means the cluster union from cluster i to j. In order to get the maximum length efficiently, we use the method of top to down. For example, all initial clusters are shown in Fig. 3. Firstly, the density of [clusters 1, cluster 6] is calculated. If the density is larger than MD, the course of merger will be terminated. If not, the course will continue. For example, the density of [clusters 1, cluster 5] is calculated. This course will be continued until the cluster union only contain two clusters. As shown in Fig. 3, final clusters will be gotten in the last step. The density of [cluster 1, cluster 2] is larger than MD and length is larger than OSWG, they are merged into final cluster 1. So does the Final cluster 2.

4. OPTIMIZATION WITH INTRINSIC PARALLELISM

In this section we will explore the intrinsic parallelism of SSD, and then optimize our method further.

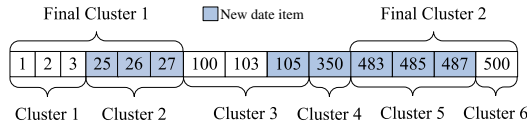


Figure 3: Random write sequence with lengths of all initial clusters are less than OSWG.

4.1 Intrinsic Parallelism

After disassembling the SSD used in my lab, we can see that sixteen flash devices are arrayed on the circuit board, and the part number is MT29F8G08DAA. Therefore we assure that parallelism exists between flash devices because each device can be operated individually. Besides this, parallelism also exists in the interior of one flash device. According to specifications[3], one flash device contains two CE#s(Chip Enable) which has two planes. Both CE#s and planes can be accessed in parallel. As to smaller storage unit, plane is made up from blocks which contain pages.

In order to testify the parallelism, we design five experiments. Each experiment will write 131072 pages of data into SSD. Experiment 1 sequentially writes pages in a single plane. Experiment 2, 3 and 4 alternately and sequentially write pages between two planes, two CE#s and two devices. Finally, experiment 5 sequentially write pages according page NO. We allocate storage area sequentially in logic layer and suppose it is physically sequential. The experiment results show the reasonableness of our assumption.

The runtime of experiment 1, 2, 3, 4 and 5 are 28.2, 25.8, 23.2, 23 and 25.2 seconds, respectively. The write performance of experiment 1 is less than that of experiment 2 about 10%. The reason is experiment 1 only can write data one by one. However, experiment 2 can write data into two planes of the same CE# at the same time by TWO-PLANE PROGRAM. So do experiments 3 and 4. Finally we find that sequential write almost has the same performance with experiments 2. Because even-numbered and odd-numbered blocks belong to different planes, sequential write also utilizes the TWO-PLANE PROGRAM to speed up write as experiment 2. But sequential write does not utilize intrinsic parallelism between CE#s and devices, so the performance is lower than experiment 3 and 4.

4.2 Optimization

According to above experiments, we propose to speed up our method further with intrinsic parallelism between CE#s and devices. Data items in write sequence are programmed alternately into flash devices if they belong to different CE#s or devices. For example, we suppose the final cluster 1 and 2 in Fig. 2 belong to different devices. As to previous method, we will flush final cluster 1 firstly, and then final cluster 2. However, we change the write sequence according to the parallelism of SSD. Final cluster 1 and 2 will be written alternately. The write sequence will be organized as 50, 300, 51, 301.....57, 307, 308, 309.....317. After re-organizing the write sequence, the parallelism between devices is triggered, and then the write performance is further improved.

5. PERFORMANCE EVALUATION

In this section we will firstly introduce the hardware platform and benchmarks in our experiments. In the next, comparison experiment results are shown and analyzed.

5.1 Experiments Setup and Workloads

We implement our experiments on a desktop PC powered by Intel Core 2 Pentium 4 Duo CPU 2.83GHz running Linux fedora 8

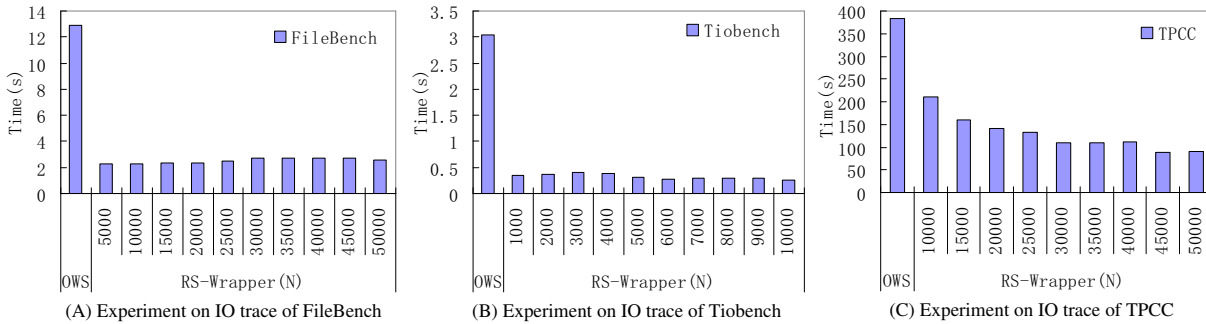


Figure 4: Comparison experiments about performance of RS-Wrapper.

with 2GB main memory. The kernel version is Linux-2.6.23. The SSD used in our experiments is 16GB Mtron SSD(MSD-SATA 3035-016). We use three benchmark tests to testify the IO performance enhancement brought by our RS-Wrapper. We also use blktrace[1] to trace the IO activities at the block level. After running benchmarks, we get the IO activities, *OWS* (Original Write Sequence). In our comparison experiments, we firstly write *OWS* into SSD directly. Secondly, we re-organize *OWS* by our RS-Wrapper and flush the new sequences. For each benchmark, a series of experiments are run by varying the length of write sequence.

5.2 File System Benchmarks

FileBench[2] is a framework of workload for measuring file system performance. In our experiments, the number of files is set 50,000, 50,000 and 25,000 for creatfiles, deletefiles and copyfiles respectively. Besides this, both the file size and IO size are set as 2 kilobytes. We also convert the *OWS* by our method when the length of sequence varies from 5,000 to 50,000. The runtime is plotted in Fig. 4 (A). Our method outperforms *OWS* about 600% as a whole. The main reason is that IO operations of a file are basically sequential. In this case, the density of final cluster is very high in our method. Therefore, our RS-Wrapper gains high performance.

5.3 File IO Benchmarks

IOzone[6] is a file system benchmark tool. In order to simulate the real workload mostly, we select the full automatic mode. The IO operations cover all tested file operations for record sizes of 4KB to 16MB for file sizes of 64KB to 512MB. The comparison experiment results are shown in Fig. 4(B). With the length of write sequence varying from 1,000 to 10,000, our RS-Wrapper is faster than *OWS* about 800% as a whole. As for IOzone, the IO activities are random and converge in a limited range. In common test, the maximum size is 512MB. Therefore, every writes are random, and thus the write cost of *OWS* is very high. However, converted sequential sequence has high density. Therefore our method takes full advantage of, and then performance is enhanced obviously.

5.4 TPCC Benchmark

In order to test the write performance enhancement on databases, we select the typical write intensive benchmark, TPC-C. In this experiment, write sequence comes from the disk IO operations by running TPC-C on PostgreSQL database system version 8.3.5. The operation system is Red Hat Linux 2.6.27. After setting the number of warehouses as 50, page size as 8KB, the number of threads as 200, we run TPC-C 30 minutes. We modify postgresql to record disk IO operations when executing the routines of writing data to disk. In the same way, we flush IO trace by two ways. The experi-

ment results are shown as Fig. 4(C). Our method outperforms *OWS* about 300% when length *N* is no less than 25000. The high performance is obtained because our method converges random writes into high density clusters and flushes them in parallel.

6. CONCLUSIONS

SSD is applied widely due to its high random read speed and sequential access performance. However, poor random write leads to the low performance of write-intensive applications on SSD. We novelly propose to extend random write sequence into sequential write sequence by inserting unmodified data into write sequence. Firstly, we explore the impact of density and length on performance of our method with static random write sequence. Secondly, we optimize our method with cluster which reduces the quantity of data to read and flush during the course of conversion. Thirdly, we improve the performance in further by merging initial clusters into final clusters. Finally, we improve write performance with its intrinsic parallelism. The experiments show our method improves write performance of SSD obviously under all tested workloads.

7. ACKNOWLEDGMENT

We would like to thank Yinan Li and Prof. Qiong Luo for their help in the experiments, Prof. Lei Chen and Jianliang Xu for their constructive comments, and Prof. Jiaheng Lu for his revision. This research was supported by the grants from the Natural Science Foundation of China under grant number 60833005.

8. REFERENCES

- [1] J. Axboe, A. D. Brunelle, and N. Scott. blktrace(8) - linux man page, 2006. <http://linux.die.net/man/8/blktrace>.
- [2] R. McDougall, J. Crase, and S. Debnath. Filebench: File system microbenchmarks, 2006. <http://www.opensolaris.org/os/community/performance/filebench/>.
- [3] Micron. Nand flash memory mt29f4g08aaa, mt29f8g08baa, mt29f8g08daa, mt29f16g08faa, 2007. http://download.micron.com/pdf/datasheets/flash/nand/4gb_nand_m40a.pdf.
- [4] Mtron. Solid gear | mtron ssd technology, 2008. <http://www.solidgear.sg/technology/mtron-ssd-technology.php>.
- [5] Mtron. Solid state drive msd-sata 3035 product specification, 2008. http://mtron.net/Upload_Data/Spec/ASiC/MOBI/SATA/MSD-SATA3035_rev0.4.pdf.
- [6] W. Norcott. Iozone filesystem benchmark, 2006. <http://www.iozone.org/>.