An Efficient Hash Index Structure for Solid State Disks

Hongchan Roh[1], and Sanghyun Park[1]
[1] Department of Computer Science, Yonsei University, Seoul, South Korea

Abstract - *An SSD is a data storage device which has excellent features such as fast access time, shock-resistance, and power-efficiency. However, the SSD is vulnerable to small-size random writes. Therefore, a good index structure is needed in order to completely utilize these superior features and address the drawback. This paper proposes an efficient hash index structure using a write buffer and log-based flushing policy, based on linear hashing and deferred splitting. A write buffer accumulates index records, and logs them to index buckets when flushing them. Through a simple performance analysis, the proposed index confirmed that it significantly reduces the number of small-size random writes.*

Keywords: SSD, hash index, linear hashing, flash memory

1 Introduction

For the last several decades, hard disks have performed well as storage devices, increasing their capacity sharply. However, hard disks' access time has not been enhanced much for the last 20 years while the processing and access time of other primary computer components such as the CPU and main memory have made significant advancements. Hard disk's slow access time has become a primary bottleneck in recent computer systems.

An SSD is a data storage device that doesn't use magnetic disks but memory chips such as SDRAM and flash memory. It provides much faster access time than hard disks. SSDs were originally designed for military and limited industry purposes to develop an enhanced storage device which has more robust features and faster access time than hard disks. They consist of CPU, SDRAM and a backup battery. However, they had not been popularly used until a different type of SSD, which substituted flash memory for SDRAM, emerged.

It is not clear who first proposed a flash based SSD, but we could find several patents regarding this. The oldest patent that contains the idea using flash memory chips among the found patents is the patent proposed by Clay et al. in 1995 [1]. Later, several patents regarding more detailed algorithms for the flash based SSD were proposed [4-7]. The recent SSD controller configuration is well illustrated in the patent proposed by Ryu [11]. These SSDs inherited flash memory's robust features such as shock-resistance, power-efficiency, and fast read access. Flash memory is organized by many blocks each of which consists of a fixed number of pages. A block is the smallest unit of erase operation meanwhile a page is the smallest unit of write operation. Writing a page is much slower than reading a page. Therefore, SSDs improved flash memory's long write latency and limited capacity [3]. However, SSDs' write unit was designed to be considerably large to improve the write latency. Consequently, small-size random writes degrade their write performance.

Flash based SSDs will substitute hard disks in the near future. Flash memory's integration level has been significantly increasing with a factor of approximately 2 per year for the last decade. SSDs can be produced with price competitiveness and the same capacity as hard disks in the near future. Since SSDs have far superior features to hard disks, they can be used as storage devices for not only personal computers, but also enterprise database systems.

To take advantage of SSDs' excellent features and hasten their use, a good index structure that reflects unique features of SSDs and improves their drawbacks is needed. Hash indexes are popular indexes which have very small access times to execute an equality query on a hard disk. Several hash index structures has been proposed and a considerable amount of related research has been conducted. To address the problems of static hashing, dynamic hashing [8] and extendible hashing [2] were proposed in 1978. Later, linear hashing advanced these by eliminating directories. Scholl [12] suggested a deferred splitting technique using buddy buckets as overflow buckets. The performance analysis between these was researched by Veklerov [13] and Rathi [10]. Recently Lin et al. proposed a hash index for flash sensor devices, which didn't focus on flash memory issues but on sensor device issues [14].

The goal of this paper is to propose an efficient hash index structure for SSDs which addresses SSDs' drawbacks and enhances their performance more. For this purpose, we designed a buffer that is closely coupled with this hash index and accumulates index update requests. In addition, log-based deletion policy, which delays real deletion of index records until a proper time, is applied to the index.

Section 2 explains linear hashing and a recent SSD controller in more detail. In Section 3, we define the problem, and Section 4 introduces proposed index structure and related algorithms to address the problem. Section 5 provides a simple performance analysis and the conclusion is presented in Section 6.

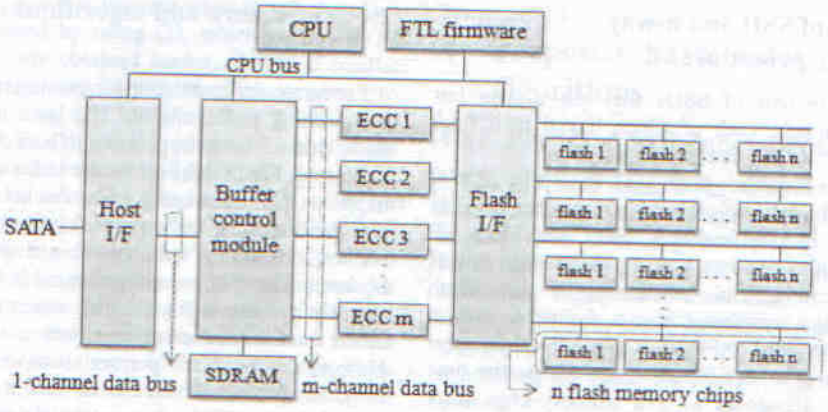


Figure 1. SSD controller overview

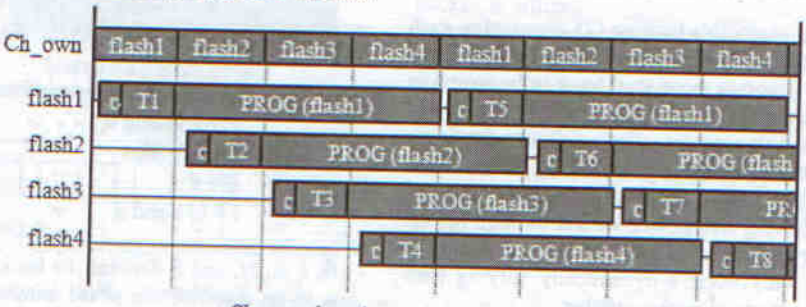
2 Related Work

2.1 SSD controller's configuration

Figure 1 shows a general configuration of recent SSD controllers which consist of several modules such as a CPU, FTL firmware, SDRAM, a Host I/F (host interface), a Flash I/F (flash memory interface), a Buffer control module, ECC modules, flash memory chips, and data and CPU buses.

When write requests are given to an SSD, the data to be written are first delivered to Host I/F, and next the Host I/F transfers the data through 1-channel bus, which has very fast data transfer rate, and then the Buffer control module temporarily stores them in the SDRAM buffer and divides and flushes them into Flash I/F through the m-channel bus, and finally the Flash I/F writes the data from each channel to n flash memory chips.

The CPU translates exterior logical addresses to internal physical flash memory addresses by using FTL (Flash Translation Layer) [3], which is an address-mapping program loaded from the FTL firmware to the SDRAM when the SSD controller is initialized, and give commands to the Host I/F and the Flash I/F to read and write data. The Host I/F transfers data between the 1-channel data bus and the host computer's storage interfaces, like SATA (Serial Advanced Technology Attachment) and PATA (parallel ATA). When triggered by CPU's commands, the Host I/F transfers data from one interface to the other, transforming the data into the data fitted to the other interface. The Buffer control module temporarily stores the transferred data in the SDRAM buffer. It not only efficiently divides them in order to transfer them into the m-channel data bus, but also restores the distributed data to their original form. The Flash I/F transfers data between the m-channel data bus and flash memory chips. When triggered by CPU's write-commands, the Flash I/F writes the data transferred from each channel of the m-channel data bus to the n flash memory chips linked to the channel by means of the n-way interleaving procedure (which will be explained later). When CPU's read commands are given, the Flash I/F reads data from physical addresses of m flash memory chips and transfers the data to the m-channel data bus. The ECC modules not only create ECC data when data are transferred from the Buffer control module to the Flash I/F but also check errors and request the Buffer control module to correct the error by using the ECC data when data are transferred from the Flash I/F to the Buffer control module [11].



Ch_own: channel owner, c: command period, Tr: data transfer period,
PROG (flashi): time to program a page of ith flash memory chip

Figure 2. 4-way interleaving procedure

2.2 Write unit of SSD and n-way interleaving procedure

The smallest write unit of SSDs that has a data bus consisting of m channels, each of which linked to n flash memory chips, is m * n flash memory pages.

To perform a write procedure to an SSD, the Buffer control module efficiently divides original data into m pieces of data and transfers them into m channels. Next, the Flash I/F simultaneously transfers the data from m channels to m sets of n flash memory chips. While transferring the data, Flash I/F interleaves the data transferred from a channel to n flash memory chips as illustrated in Figure 2. Since the time spent transferring the data of a page is much shorter than the time spent programming a page in a flash memory chip, after transferring the amount of data a page can hold to one flash memory chip, the flash I/F transfers the remaining data to the other flash memory chips during the delay time spent programming a page of the flash memory chip. In Figure 2, during the page programming time of flash memory chip 1, the Flash I/F transfers the data to flash memory chip 2, 3, and 4 and gives program-commands to the flash memory chips. After transferring data to flash memory chip 4, the Flash I/F repeats to transfer data to from flash memory chip 1 to flash memory chip 4 in turn.

By doing so, an SSD can simultaneously transfers data to m channels and writes them continuously to n flash memory chips without waiting until programming a flash memory ends. This makes an SSD handle write requests much faster than a flash memory since an SSD can write more than m * 4 pages while a flash memory write two pages as illustrated in Figure 2.

However, small-size write requests to an SSD severely degrades the SSD' write performance. For example, if user requests for the SSD to write only m pages, then the SSD will write m pages through m channels without interleaving. Even if a user requests for the SSD to write a single page, then the SSD will write m pages since m pages are closely coupled with the same data bus even though it may cause several empty pages.

2.3 Overall concept of linear hashing

In dynamic [8] and extendible hashing [2], directories, each of which points to a bucket address, are essential and cause additional hard disk accesses since they have to be stored in hard disks without enough main memory.

To eliminate these disk accesses, after a collision the linear hashing doesn't split the index bucket where the collision occurred, but the next index bucket in a predetermined sequential order, resolving overflowed bucket through bucket chaining. The linear hashing technique can directly access to an index bucket by only using a dynamically varying hash function according to the total bucket number.

2.4 Structure and algorithms of linear hashing index

For an example of the linear hashing index, Figure 3 shows the splitting process of it. The hash index of Figure 3(a) initially has 3 buckets, pointing to bucket 0 as the next bucket to be split. Figure 3(b) shows the index status after a collision happens. While bucket 0 splits, bucket 3 is created and the data in bucket 0 are evenly distributed into bucket 0 and bucket 3. After the split, the pointer sequentially moved to bucket 1. The split process presented in Figure 3(c) is same as the split process in Figure 3(b), which is moving the pointer to the next bucket, creating a bucket, and dividing the data. However, moving the pointer should be different in Figure 3(d). The pointer can be moved until it reaches bucket 2, the end of the original buckets which are not created by the split process in this phase, and then should be return to bucket 0. After this, a new phase starts, and whenever a collision occurs, this split process continues until the pointer reaches bucket 5.

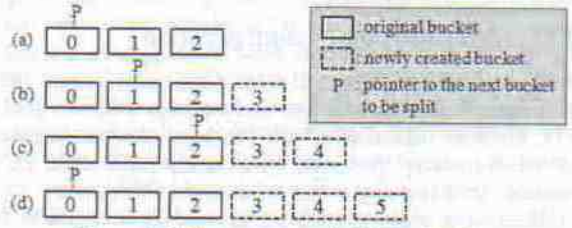


Figure 3. Splitting process in Linear Hashing

In general, the hash function of linear hashing can be represented as follows:

$$
\begin{aligned}
&i = 0,\ P = 0 \\
&E_i = 2^{i+1}N \\
&H_0(K) = K \bmod N \\
&H_i(K) = K \bmod 2^i N
\end{aligned}
\qquad (1)
$$

$$
\begin{aligned}
&\text{if } P = 0 \text{ then } B = H_i(K) \\
&\text{else} \\
&\quad B = H_{i-1}(K) \\
&\quad \text{if } B < P \text{ then } B = H_i(K) \text{ end if} \\
&\text{End if}
\end{aligned}
\qquad (2)
$$

$$
\begin{aligned}
&\text{If a collision happens then} \\
&\quad P = P + 1 \text{ end if} \\
&\text{If } P = E_i \text{ then} \\
&\quad P = 0 \\
&\quad i = i + 1 \text{ end if}
\end{aligned}
\qquad (3)
$$

B, i, K, N, and P denotes the bucket address for the index data to be inserted, the phase number, the key of the index record, the initial total bucket number, and the pointer to the next bucket to be split, respectively.

The example of Figure 3 is the case when N = 3. A bucket address can be obtained by using (2), which returns $H_i(K)$ except when the already obtained bucket address is smaller than the bucket address pointed by P. A collision moves P to the next bucket by using (3). Resetting P to 0, the phase number is increased by (3) when P reaches the end of the original buckets that is determined by E_i in (1).

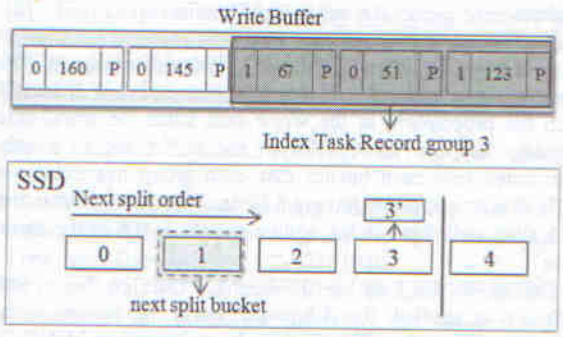
This linear expansion of bucket space makes it possible for a hash index to access a bucket without directories. However, this feature also raised a low storage utilization problem because of frequent splitting. In his paper, Litwin proposed a deferred splitting technique which delays splitting a bucket and makes overflow buckets linked to the collided one until a load factor reaches a certain threshold defined by users. They proposed two types of load factor functions such as T/B and T/B' where T, B, and B' denotes the total number of the inserted index records, the number of records that can be held in all the buckets including overflow buckets, and the number of records that can be held in the buckets not including overflow buckets, respectively [9].

3 Problem Definition

SSDs are very vulnerable to small-size random writes. Since the SSD controller writes as much space as the write unit regardless of how small the space size of the write requests is, the time spent performing write requests to the smaller space than the write unit is same as the time spent writing the total write unit space. Consequently, these small-size writes cause numerous erases of the blocks corresponding to the write units.

A hash index requires a number of small-size random writes since write requests are distributed to numerous buckets by a hash function and even one write request can cause writing several buckets such as overflow buckets, the collided bucket, and the newly created bucket by splitting.

In this respect, designing a new hash index structure that can reduce the number of small-size random writes is essential in order to enhance SSDs' write performance and lengthen SSDs' life time.



$i = 0, H_0(K) = K \bmod 4, H_1(K) = K \bmod 8$
If $H_0(K) < 1$ then $B = H_1(K)$ else $B = H_0(K)$

Figure 4. An overview of proposed index structure

4 Proposed hash index structure and algorithms

4.1 Proposed hash index structure overview

The proposed hash index structure is based on linear hashing since linear hashing not only removes additional accesses to the directory, but also saves main memory space, which can be utilized as a write buffer, by eliminating directories. The index structure exploits the deferred splitting technique because linear hashing originally brings excessive splitting overhead. Among the proposed deferred splitting techniques, we chose the Litwin [9]'s technique that delays splitting by using a load factor function and a certain threshold. In addition, the load factor function not including overflow buckets was chosen.

Moreover, we extended the hash index structure as follows: First, the extended index structure is coupled with a write buffer containing index records in order to reduce the number of writes to SSD. Second, the index structure utilizes a log-based flushing policy when logical blocks on SSDs are updated.

Figure 4 illustrates the overall view of our index structure. The write buffer resides on main memory and the buckets are stored on the SSD. The number of the initial buckets is 4. Therefore, when the pointer to the next split bucket reaches bucket 3, the pointer returns to bucket 0. Bucket 3 has 1 overflow bucket chained to it.

4.2 Buffer Strategy

In an linear hashing index not using write buffer, in the worst case scenario, inserting an index record can cause 4 bucket rewrites such as updating the bucket which collision happened, writing a newly created overflow bucket, updating the split bucket, and writing a newly created bucket for splitting on an SSD (except when splitting cannot completely resolves overflow buckets linked to the split bucket). However it is possible to reduce this write count up to 4/W bucket writes even in the worst case if a write buffer, which gathers index records and flushes W records at once into a bucket, is utilized.

For this purpose, we designed a data structure for an Index Task Record (ITR) which contains an index record and the task type. As shown Figure 4, each entry of the write buffer consists of the following 3 types of information: 1 bit for identification of a request type such as 0 for an insert request and 1 for a delete request, as many bytes as the key size for an index record, 4 bytes for an index record pointer to a data block.

4.3 Write Buffer Management Algorithm

The maximum buffer size that can be allocated in their current system is specified by users.

First, if there are index update requests from users, then the index makes the ITRs by using user request type and the index records. Otherwise, the index waits until an index update request is given.

Second, the index classifies those ITRs by using the hash function and inserts them into the corresponding ITR group which belongs to the bucket designated by the hash function.

Finally, if the buffer is fulfilled by newly inserted entries, locate the largest group among the ITR groups and flush them to the bucket on the SSD. Otherwise, the index repeats the above 3 steps

Through this algorithm, the index flushes ITR group 3 whose size is 3 to bucket 3 in Figure 4.

4.4 Write Buffer flush Algorithm

First, the index locates a bucket designated by the hash function.

Next, the index locates the last bucket, the current bucket if no overflow bucket exists or the end of the bucket chain if overflow buckets exist.

Then, write the ITR group simultaneously to the empty space in the chosen bucket.

Finally, if the bucket overflows by flushing the ITR group, and if the bucket is not the next split bucket, the index appends another overflow bucket to the end of the chain and writes the remaining index records in the ITR group to the bucket. If the bucket overflows, and if the bucket is the next split bucket, and if the load factor reaches to the threshold, then perform the split algorithm which will be explained later.

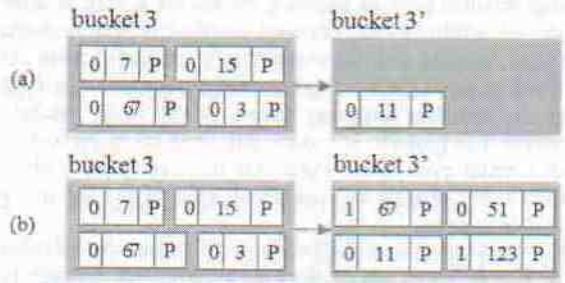


Figure 5. The status of bucket 3 and bucket 3' before and after flushing ITR group 3 in Figure 4

4.5 Log-based deletion technique

During write buffer flushing process, delete index task records can cause needless updates of buckets. To be more exact, if m overflow buckets linked to the designated bucket exists and the group records being flushed include more than m delete index task records, then rewriting m+1 buckets can happen in the worst case.

Merely logging them into a bucket can reduce this overhead very simply, not immediately executing the deletion process. We defined this as the log-based deletion technique.

Figure 5(a) illustrates in more detail bucket 3 and its overflow bucket of Figure 4. After flushing ITR group 3, 2 delete ITRs and 1 insert ITR is logged in bucket 3' causing only 1 bucket write as shown in Figure 5(b). Without the log-based deletion, 2 buckets have to be written by the flushing procedure, since bucket 3 that contains the insert ITR having the key of 67, have to be updated because of the delete ITR having the key of 67 in the ITR group.

Delete index task records are processed in exactly same way as insert index task records during flushing the write buffer, merely appending the index task record group next to the last record in the tail bucket of the bucket chain. This also makes the flush algorithm simpler than individually deleting index records on different buckets.

However, this technique can make the overflow bucket chain become too long, increasing the number of accessed buckets per search request. Therefore, real delete operations corresponding to the logged ITRs should be periodically executed to reduce the bucket chain length. This happens through the following splitting algorithm when splitting occurs.

4.6 Splitting algorithm

First, the index locates and reads the bucket directed by the split pointer.

Next, the index reads all the overflow buckets linked to the located one and performs real deletion of index records indicated by delete ITRs in the read buckets.

Finally, the index evenly divides the decreased ITRs into two buckets such as the located old bucket and newly created one.

5 Simple Performance Analysis

We assumed that the main memory space that is afforded to save the number of ITRs more than (4*NB+1) can be guaranteed, where NB represents the number of current buckets.

Even in the worst case, proposed index assures the number of bucket writes less than 0.8 per index update request. The write buffer chooses the ITR group whose size is at least 5 with the proposition in the worst case since the worst case scenario happens when ITRs in the buffer are so evenly distributed into each bucket that each group has exactly 4 ITRs except one group having 5 ITRs. Therefore, the number of bucket writes per index update request is 0.8 in the worst case.

The aspect the Log based deletion affects on this is that unless it is applied, the deletion ITRs in the chosen group increases the number of buckets to be written up to MIN(ND, OB+1), where ND and OB denotes the number of the delete ITRs and the number of the overflow buckets linked to the designated bucket for the flush operation, respectively.

6 Conclusion

SSDs will eventually substitute hard disks due to their more robust features and much faster access time than hard disks in the near future. However, SSDs are vulnerable to small-size random writes since the writes degrade the write performance and shorten the SSD life time. In order to utilize SSDs' superior features and hasten their use, a good index structure that reflects their own features and reduces the small-size random writes is needed.

We proposed a linear hashing index extension which assures the number of the bucket writes less than 0.8 per index update request including write buffer that gathers index records and then flushes the largest index record group and log-based deletion technique that delays deleting index records by when spitting occurs.

Since proposed index has 0.8 bucket writes per index update request compared with original linear hashing's 4 bucket writes per index update request in the worst case, this index can contribute to reducing write latency of SSDs and prolong the SSD life time.

7 References

[1] D. W. Clay and S. A. Anderson, "Flash Solid State drive that emulates a disk drive and stores variable length and fixed length data blocks," 1995.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," ACM Transactions on Database Systems, Vol. 14, No. 3, pp. 315-344, 1979.

[3] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," ACM Computing Surveys, Vol. 37, No. 2, pp. 138-163, 2005.

[4] Intel Corporation, "Method and Apparatus for Error Management in A Solid State Disk Drive," 2001.

[5] Intel Corporation, "Method and System for Performing Clean-up of A Solid State Disk during Host Command Execution," 1997.

[6] Intel Corporation, "Method for Allocating Memory in A Solid State Memory Disk," 1996.

[7] Intel Corporation, "Method of Controlling Clean-up of A Solid State Memory Disk Storing Floating Sector Data," 1995.

[8] P. Larson, "Dynamic Hashing," BIT 18, pp. 184-201, 1978.

[9] W. Litwin, "Linear Hashing : A New Tool for File and Table Addressing," Proceedings of the 6th Conference on Very Large Database, pp. 212-223, 1980.

[10] A. Rathi, H. Lu, and G. E. Hedrick, "Performance comparison of extendible hashing and linear hashing techniques," Proceedings of the ACM SIGSMALL/PC Symposium on Small Systems, 1990.

[11] D. R. Ryu, "Solid State Disk Controller Apparatus," 2006.

[12] M. Scholl, "New File Organizations based on Dynmic hashing," ACM Transactions on Database Systems, Vol. 6, No. 1, pp. 194-211, 1981.

[13] E. Veklerov, "Analysis of Dynamic Hashing with Deferred splitting," ACM Transactions on Database Systems, Vol. 1, No. 1, pp. 90-96, 1985.

[14] S. Lin and V. Kalogeraki, "Efficient Indexing Data Structures for Flash-Based Sensor Devices," ACM Transactions on Storage, Vol. 2, No. 4, pp. 468-503, 2006.