hUBI: An Optimized Hybrid Mapping Scheme for NAND Flash-Based SSDs

Guangjun Xie
Baidu Inc., Beijing, China
Nankai-Baidu Joint Lab, Nankai University, Tianjin, China
Email: xieguangjun1980@163.com

Guangzhi Xu, Gang Wang, Xiaoguang Liu, Rui Cao, Yan Gao
Nankai-Baidu Joint Lab, Nankai University, Tianjin, China
Email: xgz.nku@mail.nankai.edu.cn, wgzwp@163.com, liuxg74@yahoo.com.cn

***Abstract*—NAND flash-based SSDs have become attractive alternatives to hard disk drivers due to their high random read performances and low power consumptions. However, the poor random write performances highly limit their popularization in commercial applications.**

In this paper, we propose a novel mapping scheme called hybrid mapping unsorted block images (hUBI). hUBI aims for (1) optimized random write performances, (2) low write latencies, and (3) low space consumptions. It optimizes traditional hybrid mapping schemes to achieve (1). In hUBI, the merge operation involves only a single block. So, (2) is guaranteed. To avoid introducing a high space cost maintaining the metadata, hUBI puts its metadata into out-of-band (OOB) areas of SSD pages, which obtains (3). Our experimental results show that hUBI provides a considerable random write performance and holds low write latencies at the same time.

I. Introduction

Over the last decade the CPU speed has increased dramatically while the access time of traditional disks has only improved slowly [1]. The bottleneck seems more apparent when parallel workloads introduced by rapid developing IT applications increase. As the traditional hard disks using spinning disks and movable read/write heads can not meet applications' requirements, people seek for new storage medium that can solve this problem. At the end of last century, solid-state disks (SSD) were introduced into the storage industry, and nowadays become increasingly alternatives for traditional hard disks [2].

SSDs have the properties of low power consumptions and high access speeds. Now, most SSDs are based on NAND flash chips. Along with the attractive properties of low costs and high densities, the NAND flash chip has a problem that blocks should be erased before being written and erase operations are quite slow. In consideration of wear leveling and read/write latencies, dynamic mappings are used in most SSD drivers.

Mapping schemes can be divided into three dimensions: block-mapping, page-mapping and hybrid mapping. Block-Mapping schemes [3] maintain mappings between logical blocks and physical blocks. Each page update will cause a block erase and several page copy operations. Block-Mapping schemes imply simple implementations and low space costs for their metadata. However, due to their high write latencies, block-mapping schemes have poor random write performances. To improve the write performance, page-mapping schemes [4] maintain fine-grained mappings between logical pages and physical pages. In page-mapping schemes, page writes are performed on free pages. Compared to block-mapping schemes, page-mapping schemes need fewer erase operations and thus get higher random write performances. However, page-mapping schemes consume a lot of spaces to store their metadata. Hybrid mapping schemes [5] [6] [7] combine the advantages of block-mapping schemes and page-mapping schemes. They group blocks into data area and log area. Pages in the log area are addressed like page-mapping schemes. Page writes are performed on free pages in the log area. When the log area has no free pages, a merge operation [10] is triggered to erase log area blocks and stores valid pages in data area blocks. By this means, hybrid mapping schemes get considerable random write performances and hold space-efficiency at the same time. However, in traditional hybrid mapping schemes, complex merge operations may cause high write latencies, which could be catastrophic in time-sensitive applications.

In this paper, we present a novel hybrid mapping scheme called hybrid mapping unsorted block images (hUBI). hUBI contributes in several aspects:

- hUBI uses a optimized hybrid mapping algorithm giving high random write performances.
- The merge operation in hUBI involves only one block, which leads to a guaranteed low write latency.
- hUBI maintains its metadata in the out-of-band (OOB) areas, which reduces the space consumption and the number of I/O operations.

Experimental results show that hUBI provides an considerable random write performance and a low timeout rate under mixed read and write workload.

The following parts of this paper are organized as follow: Section 2 introduces other related works. Section 3 describes the design of hUBI in details. Section 4 shows our analysis and experiments of hUBI. And finally Section 5 gives the conclusion.

II. Related Work

Unsorted Block Images (UBI) [11] is a mapping scheme used in the Linux kernel. It has a layered architecture and most other schemes adopt its design. UBI is a block-mapping scheme. As discussed above, each page update in UBI brings a block erase and several page copy operations. Block erase and page copy are both time consuming operations. So page updates in UBI have a high time cost.

There have been researches on hybrid mapping schemes. Kim [5] presented a log block scheme in which each log block (i.e., a block in log area) is dedicated to one data block (i.e., a block in data area). This may cause low space utilization of log blocks. e.g., there is an update to one data block. Although the log area still has lots of free rooms, if the log block it owns is fully occupied, a merge operation will still be necessary. To fully utilize the log area space, Lee [7] proposed a mapping scheme called fully associative sector translation (FAST). In FAST, a page in log block can be mapped to any data blocks. FAST merges fewer times than [5], but it brings complex merge logic. A merge in [5] only involves two blocks. While at the worst case, a merge in FAST can involve as many as $n+1$ blocks (n is the number of pages in a log block). Park [8] introduced a compromising solution that maps K log blocks to N data blocks. He ran a lot of workloads trying to find proper values for K and N. Liu [9] tried to apply hybrid associations between log blocks and data blocks. For one hot data block (i.e., the block that accessed frequently), a exclusive log block is assigned, while other cold data blocks use fully associations described in FAST. However, the merge problem remains.

Partition based UBI (pUBI) is an improved hybrid mapping scheme. In pUBI, blocks are grouped into partitions. Each partition is a contiguous region of the address space. Blocks in a partition are divided into data blocks, log blocks and m-blocks. Updates to a partition will be performed on its corresponding log blocks. In one partition, there is only one active m-block storing its metadata. A merge operation is called when there are no free pages in the log area. Compared with UBI, pUBI provides a better random write performance. However, the merge operation in pUBI may involve several data and log blocks, therefore it can bring a number of block erase and page copy operations. In some circumstance, such a time consuming merge operation may cause a data update timeout.

hUBI adopts the layered architecture of UBI but designs a new hybrid mapping mechanism. To solve the timeout problem in pUBI, hUBI focuses on optimizing merge operations. Experimental results show that hUBI solves this problem effectively and provides a comparable random write performance with pUBI.

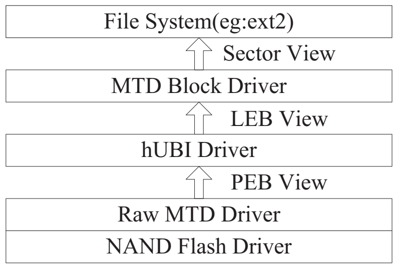

Figure 1. Overview Architecture

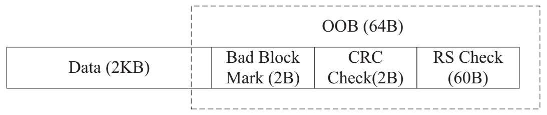

Figure 2. Huawei SSD Page Layout

III. Design of hUBI

The major objective of hUBI is lowering the write latencies and space consumptions of hybrid mapping schemes. Our work is based on the NAND flash-based SSDs produced by Huawei Inc. and can be applied to other SSDs. Overview architecture of hUBI is shown in Figure1.

The raw memory technology device (MTD) driver provides a physical erasable block (PEB) view without wear leveling. The hUBI driver implements a hybrid mapping algorithm and provides a logical erasable block (LEB) view. Finally, to use SSDs as normal block devices, the MTD block driver emulates a sector view for upper applications.

A. Metadata Layout

1) Disk Layout: hUBI divides pages in each block into log area and data area. The page-mapping info for log area pages is stored on disk to record the relationships between logical pages and physical pages. Since the page-mapping info changes along with page updates, by storing the page-mapping along with the page data, only one I/O operation is needed when writing both the page data and the page-mapping info. Figure2 shows the page layout of Huawei MLC SSDs. The out-of-band (OOB) area is used for error control and the first two bytes of OOB are used as a badblock flag. Only 2 bits of the flag are actually used, therefore we can use the second byte to hold the page-mapping info, as shown in Figure3.

Two main advantages are gained by the above design: (1) The number of I/O operations is reduced, as the page data and the page-mapping can be written to disks in one I/O operation. (2) By using the OOB areas to store the metadata, hUBI does not consume extra storage spaces.

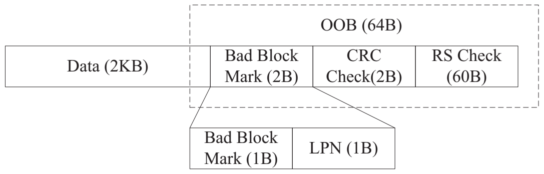

Figure 3. hUBI Page Layout

2) Memory Layout: There are two alternatives to maintain the in-memory page-mapping table (PMT). One of them stores the physical page number (PPN) for each page in the log area. For a read/write access, if the required page is in the log area, the PPN can be gotten by searching the PMT. Otherwise, if the required page is in the data area, as data area pages are stored in the order of logical addresses, the PPN is equal to the logical page number (LPN). By using this solution, fewer memory spaces are consumed to maintain the metadata. However, a read operation may cause as many as n comparisons to get the corresponding PPN (n is the total number of log pages in one block). The other method maintains the page-mapping info for both log and data area pages, which is similar to the page-mapping scheme. By this way, read operations avoid PMT searching. But, more spaces will be occupied and things get even worse when the size of storage grows larger.

To balance between memory consumptions and performances, we come up with a compromising method. hUBI optimizes the first design by adding a bitmap. One bit in the bitmap corresponds to a unique logical page and indicates whether it is in the log area or not. When accessing a page, the corresponding bit in the bitmap is first tested. Since bit test is a fast operation, we pay little performance penalty to save a lot of memory space.

An overview of metadata layout for hUBI is shown in Figure4. hUBI extends the eraseblock mapping table (EMT) used by UBI to support hybrid mapping. Each entry of EMT stores the metadata for a logical block. It contains four parts: the block-mapping info, a pointer, an index and a bitmap. The pointer points to the PMT. The index records the next free log area page in PMT. The bitmap indicates whether a page is in the log area. e.g., in Figure4, LEB 1 is mapped to PEB 10. In PEB 10, an update operation is performed after a merge operation. The LPN of the written page is 2.

B. Read and Write Algorithms

In this section, we describe the read and write algorithms of hUBI in details.

The key logic of the read algorithm is the address translation. In hUBI, a logical address can be divided into three parts: a LEB (logical erasable block) number, a LPN (logical page number) in the LEB and an offset in the page. Usually, read requests are aligned by pages, thus we can assume the offset in the page to be zero and use a tuple <LEB, LPN> to represent a logical address. In the following parts, we use this form of logical address to demonstrate the read and write algorithms.

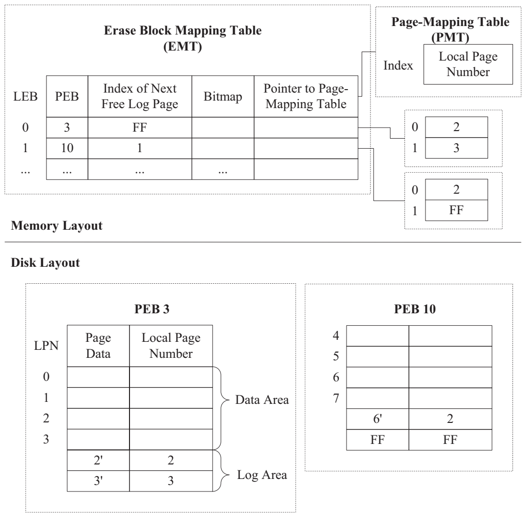

Figure 4. hUBI Metadata Layout

Algorithm 1 shows the read algorithm of hUBI. To read a page, the algorithm first locates the page-mapping table for a given LEB. Then the corresponding bit in the bitmap is tested to check whether the required page is in the log area. For a read request to log area pages, the algorithm will search the page-mapping table sequentially to find the corresponding PPN. Otherwise, the required page is in the data area, the PPN is equal to the LPN.

Algorithm 1: hUBI Read Algorithm

```
Input: $leb$, $lpn$
begin
    $emt\_entry$ = get_entry_from_emt($leb$);
    $peb$ = $emt\_entry.peb$;
    $pmt$ = $emt\_entry.pmt$;
    $bitmap$ = $emt\_entry.bitmap$;
    if $is\_in\_log\_area(lpn, bitmap) == true$ then
    $ppn$ = get_ppn_from_pmt($lpn$, $pmt$);
    else  $ppn = lpn$;
    read_page($peb$, $ppn$);
end
```

Algorithm 2 shows the write algorithm of hUBI. The algorithm first writes the data and the page-mapping info to a free log area page. Then a new entry is added to the PMT and the corresponding bit in the bitmap is also set. When there are no free pages in the log area, a merge operation is called to clean the log area.

Algoritm 3 describes the merge logic. When there are no free log pages in the log area of current PEB P (LEB L), a new PEB P' is allocated from the free PEB list. Then valid

Algorithm 2: hUBI Write Algorithm

```
Input: $leb$, $lpn$, $data$
begin
    $emt\_entry$ = get_entry_from_emt($leb$);
    $peb$ = $emt\_entry.peb$;
    $pmt$ = $emt\_entry.pmt$;
    $bitmap$ = $emt\_entry.bitmap$;
    $ppn$ = get_next_free_log_page($pmt$);
    set_oob_mapping($oob$, $lpn$);
    write_physical_page_with_oob($peb$, $ppn$, $oob$,
    $data$);
    set_bitmap($lpn$, $bitmap$);
    set_pmt($lpn$, $pmt$);
    if no free log pages then
        merge($leb$);
    end
end
```

pages are copied from P to P'. After that, P will be put on the erase list waiting to be erased by a garbage collector. Finally, the EMT will be updated to map L to P'. e.g., in Figure4, according to algorithm 2, the second data update to PEB 3 will trigger a merge operation, then a new PEB K will be allocated and valid pages will be copied to PEB K. Then entry 0 in EMT will be updated to map LEB 0 to PEB K.

Algorithm 3: hUBI Merge Algorithm

```
Input: $leb$
begin
    $emt\_entry$ = get_emt_from_emt($leb$);
    $peb$ = $emt\_entry.peb$;
    $peb'$ = alloc_free_peb();
    $bitmap$ = $emt\_entry.bitmap$;
    $pmt$ = $emt\_entry.pmt$;
    foreach $ppn$ in $pmt$ do
        copy_page($peb'$, $peb$, $ppn$);
    end
    foreach $ppn$ in Data area do
        if is_in_log_area(ppn, bitmap) $== false$ then
            copy_page($peb'$, $peb$, $ppn$);
        end
    end
    $emt\_entry.peb$ = $peb'$;
    add_to_erase_queue($peb$);
end
```

IV. Analysis and Experiments

A. Analysis

By using bitmap, read algorithms of hUBI bring tiny extra time cost. So the analysis below is focused on the write algorithm.

Two performance metrics are concerned in evaluating the write performance of SSDs.

Mean Time To Page Write(MTTPW) shows the average time cost to write back a page. From this metric, the average latency of write operations can be derived.

Mean Time To Merge(MTTM) shows the time cost of one merge operation. Since write requests are blocked when merging, the worst case of write latency can be derived from this metric.

Assume the time cost of reading a page as t_r, the time cost to write a page as t_w, the time cost to erase a block as t_e and the time cost to deliver a page from upper application to SSD storage as t_x. For a single block, d stands for the number of pages in the data area and l stands for the number of pages in the log area. Since in hUBI, every l continuous write requests will trigger a merge operation, and a merge operation needs a block copy and an block erase operations, we have:

$$MTTM_h = d(t_r + t_w + 2t_x) + t_e \tag{1}$$

$$MTTPW_h = \frac{MTTM_h}{l} + t_w + t_x \tag{2}$$

We compare hUBI with two UBI-based mapping schemes, UBI and pUBI.

UBI is a typical block-mapping scheme. A page update in UBI will cause a merge operation, which then needs a block read, a block write and a block erase operations. Assume there are k pages in a block, then the $MTTPW$ and $MTTM$ of UBI are:

$$MTTM_u = k(t_r + t_w + 2t_x) + t_e \tag{3}$$

$$MTTPW_u = MTTM_u + t_w + t_x \tag{4}$$

pUBI is a hybrid mapping version of UBI. In pUBI, blocks are grouped into partitions. Blocks in a partition are divided into data blocks, log blocks and m-blocks. Write requests of a partition go to its log blocks and data blocks store the merged data. Each partition stores its mapping info in a m-block. To analyze the performance of pUsBI, we introduce two metrics.

Locality ratio α: Suppose that there are D data blocks, L log blocks in a partition and k pages in a block. As pUBI uses FAST [7] on its log blocks, every $L \times k$ write requests cause a merge operation. We define α as the average ratio of different LEBs covered by $L \times k$ continuous write requests to all data blocks in a partition.

Sequential ratio β: When merging, if a log block covers exactly one LEB, pUBI will turn this log block into a data block. We define β as the average ratio of log blocks that can be turned into data blocks to all data blocks in a partition.

We know that every $L \times k$ write requests cause a merge operation. For these requests, there are $D \times \alpha$ different LEBs covered, and $D \times \beta$ LEBs can be turned into data blocks directly. Therefore, in a merge operation, $D \times k \times (\alpha - \beta)$ pages are copied and $D \times (\alpha - \beta) + L$ blocks are erased. In addition, as every partition in pUBI stores its mapping info in a m-block page and mappings are changed during merging, each merge operation will cause a page write to a m-block, which then implies a $1/k$ erase operation. From the above analysis, we can get the $MTTW$ and $MTTPM$ of pUBI:

$$MTTM_p = D(kt_r + kt_w + 2kt_x + t_e)(\alpha - \beta) + Lt_e + t_w + \frac{t_e}{k} \quad (5)$$

$$MTTPW_p = \frac{MTTM_p}{kL} + t_w + t_x \quad (6)$$

To optimize the sequential write performance of mapping schemes, MTD block driver implements a write-back cache. The size of the cache is equal to the size of a LEB. For a write request, if the required LEB is contained in the buffer, the write operation is performed directly on the page cache. Otherwise, the cached LEB is discarded, dirty pages are flushed to the disk and the required LEB is fetched. To fully utilize this cache, hUBI optimizes its write algorithm in two ways: (1) When the number of flushed pages is below the size of a log area, each write request is performed using the write logic described in Algorithm 2. (2) When the number of flushed pages is not less than the size of a log area, the required block will be merged and valid pages will be directly copied to the data area of a newly allocated block. From the above analysis, under sequential write workloads, the $MTTPMs$ of the three mapping schemes can be changed into:

$$MTTPW_h = \frac{d(t_w + 2t_x + t_r) + t_e}{d} \quad (7)$$

$$MTTPW_p = \frac{Lt_e + t_w + \frac{t_e}{k}}{kL} + t_w + t_x \quad (8)$$

$$MTTPW_u = \frac{k(t_w + 2t_x + t_r) + t_e}{k} \quad (9)$$

The performance metrics for ssds used in our experiment are shown in Table I

Table I
HUAWEI MLC SSD PERFORMANCE METRICS

Operation		Time(us)
256KB Block Erase	t_e	1500
2KB Page Read	t_r	50
2KB Page Write	t_w	800
2KB Data Transfer	t_x	50

From Table I and eqs. (1) - (9). we compute the theoretical write performances for UBI, pUBI and hUBI (Table II). Assume each block contains 128 pages (k=128). For pUBI, each partition has 16 data blocks (D=16) and 4 log blocks (L=4). For hUBI, each block contains 100 pages for data area (d=100) and 26 pages for log area (l=26), two pages are remained for the block metadata.

Table II
THEORETICAL WRITE PERFORMANCE

	UBI	pUBI	hUBI
SW(Sequential Write) RW(Random Write)			
SW MTTPW	0.961ms	0.863ms($\alpha = \beta$)	0.965ms
RW MTTPW	123.1ms	4.71ms($\alpha = 1, \beta = 0$)	4.561ms
MTTM	123.1ms	6.65ms-1976.32ms	96.5ms

As we can see from TableII, under sequential workloads, all three schemes have similar performances. And as hUBI merges more times than pUBI, pUBI gets a higher sequential write performance. While under random workload, the complex merge operation of pUBI effects its performance and hUBI achieves a little better. Moreover, in the worst case, pUBI will take nearly 2s to perform a merge operation, which is unacceptable for some time sensitive applications. In a word, hUBI provides a considerable write performance and holds a steady write latency.

B. Experimental results

A overview of our experiment environment is given in Table III.

Table III
EXPERIMENT ENVIRONMENT

OS	64-bit Redhat Linux AS 5 with kernel 2.6.18
Host	
CPU	Intel Xenon 5250 × 2
Memory	DDR2 FBD 16GB
HDD	SAS 146GB × 8 RAID 5
SSD	
Manufacturer	Huawei Inc.
Chip Type	2GB NAND MLC
Block Size	256K
page Size	2K
Tools	
Iometer	Ver 2006_07_27 (x86_64)
MySQL	Ver 14.7 for Redhat-Linux-GNU (i686)

First, we use iometer [24] to evaluate the read performance of hUBI(see Figure5 and Figure6). We can see that hUBI performs much better than traditional HDDs. And as read operations in the three schemes do not trigger time

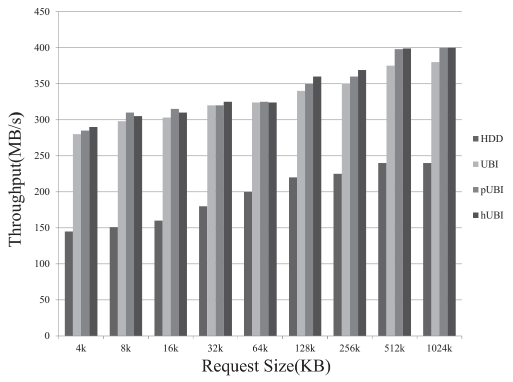

Figure 5. Sequential Read Performance

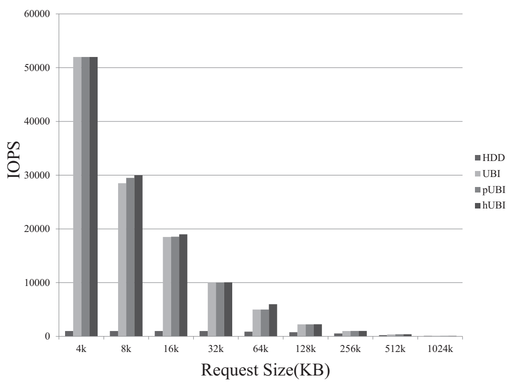

Figure 6. Random Read Performance

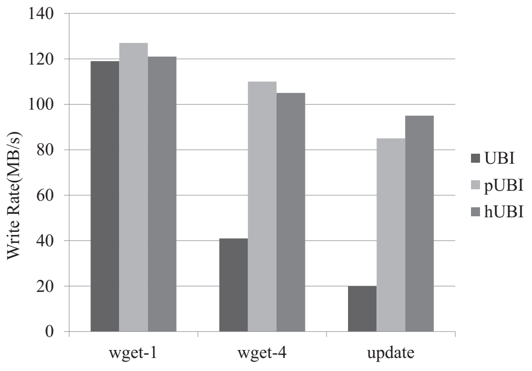

Figure 7. Write Benchmark

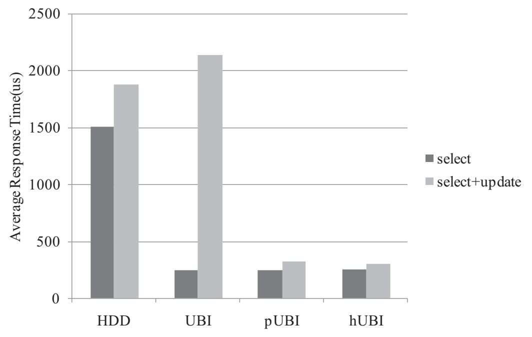

Figure 8. Mixed Read and Write Benchmark

consuming write/erase operations. Their read performances do not show much difference.

Then, we simulated three common scenarios to evaluate the write performance of hUBI: (1) a single threaded wget program [25] downloads files from a library of web indexes, (2) a modified multi-threaded wget program downloads files from a library of web indexes and (3) a program performs random updates to a MySQL database. (1) is a sequential write scenario, while (2) and (3) provide random write workload. For (1) and (2), the total size of the library is 80G, there are 76 files with 1G size and a lot of small files. For (3), there are 5 million records in the database. The average length of records is 497 bytes. Results are shown in Figure7.

From Figure7, we can see that under sequential workloads, the three mapping schemes have similar performances. This is because a write cache is used in the MTD block driver and the results are within our expectation. While in random write scenarios, hUBI and pUBI perform much better, because hUBI and pUBI merge much less times than UBI.

Then, we use a mixed read and write workload to evaluate the average write latency of hUBI. We perform concurrent select and update queries to the MYSQL database used in the above benchmark. Figure8 shows the average response time of select queries. In the experiment, the number of select queries ten times the number of update quires. As we can see, the performance of UBI falls sharply when update quires involve while UBI and pUBI remain a steady response time.

To evaluate the worst case of write latencies for hUBI, we observe the distributions of the response time in the above experiment. Figure9 and TableIV show the detailed statistics for timeout requests. As we can see, for one hundred thousand select queries, pUBI has about four hundred queries that take over 20ms. So, the timeout rate of pUBI is about 0.4%. However, in hUBI, the timeout rate is only 0.001%. This is because the merge operations in hUBI involves only one block. While in pUBI, a merge operation can involve several block erase and a number of page copy operations.

V. Conclusion

Nowadays, NAND-based SSDs are widely used both in the commercial area and the personal computing. However, the poor random write performance of SSDs has proven to be a serious problem. One way to solve such a problem is

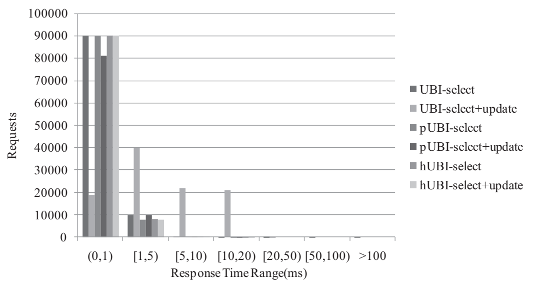

Figure 9. Response Time Distribution

Table IV
STATISTICS OF TIMEOUT REQUESTS

	[20,50)	[50,100)	>100
UBI-select	3	0	0
UBI-select+update	1089	921	389
pUBI-select	0	0	0
pUBI-select+update	123	15	250
hUBI-select	0	0	0
hUbI-select+update	1	0	0

optimizing the mapping algorithms of SSDs. Many works have explored this area. In this paper, we introduce a novel hybrid-mapping scheme for NAND-based SSDs called hUBI. hUBI contributes in three aspects:

- hUBI uses a optimized hybrid mapping algorithm achieving high random write performance.
- The merge operation in hUBI is performed on a single block, which provides a guaranteed low write latency.
- hUBI stores the mapping info in OOB areas of SSD pages which holds space-efficiency.

In a word, for NAND-based SSDs, hUBI improves their random write performances and provides guaranteed low write latencies. However, as a hybrid mapping scheme, the log area of hUBI causes a waste of storage spaces. Our future work will focus on reducing the space cost of mapping schemes and providing a better write performance at the same time.

ACKNOWLEDGMENTS

This work was supported in part by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028, 61070014, 61170301), Science and Technology Development Plan of Tianjin (08JCYBJC13000), and Key Projects in the Tianjin Science and Technology Pillar Program.

REFERENCES

[1] Mendel Rosenblum, John K. Ousterhout. *The Design and Implementation of a Log-Structured File System* In Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1991.

[2] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, JA. A. *Tauber: Storage Alternatives for Mobile Computers* In Proceedings of the 1st Symposium on Operation Systems Design and Implementation, 1994.

[3] A. Ban. *Flash File System* United States Patent, No. 5,404,485, April 1995.

[4] A. Gupta, Y. Kim, and B. Urgaonkar. *DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings* In Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS 09), March 2009.

[5] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho. *A Space-Efficient Flash Translation Layer for CompactFlash Systems* IEEE Transactions on Consumer Electronics, Vol. 48, No. 2, 2002.

[6] Tae-Sun Chung, Dong-Joo Park, Sang-Won Park, Dong-Ho Lee, Sang-Won Lee, Ha-Joo Song. *System Software for Flash Memory: A Survey* In Proceedings of the 2006 IFIP International Conference on Embedded And Ubiquitous Computing, August 2006.

[7] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song *A Log Buffer based Flash Translation Layer Using Fully Associative Sector Translation* IEEE Transactions on Embedded Computing Systems, 6(3):18, 2007. ISSN 1539-9087.

[8] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. *A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications* ACM Transactions on Embedded Computing Systems, 7(4), 2008.

[9] Z. Liu, L. Yue, P. Wei, P. Jin, and X. Xiang. *An Adaptive Block-Set based Management for Large-Scale Flash Memory* In Proceedings of the 2009 ACM Symposium on Applied Computing, pages 1621-1625, 2009.

[10] L. P. Chang and T. W. Kuo. *A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems* The 8th International Conference on Real-Time Computing Systems and Applications, 2002.

[11] Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. *UBI - Unsorted Block Images* IBM Tech Report,2006-06-09.

[12] Y. H. Bae. *Design of a high performance flash memory-based solid state disk* Journal of Korean Institute of Information Scientists and Engineers,25 (6), 2007.

[13] Yuan-Hao Chang , Jen-Wei Hsieh , Tei-Wei Kuo. *Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design* Proceedings of the 44th annual conference on Design automation, June 04-08, 2007, San Diego, California.

[14] Hyun-jin Cho, Dongkun Shin, Young Ik Eom.*KAST: K-associative sector translation for NAND flash memory in real-time systems* DATE 2009: 507-512.

[15] Hyun Jin Choi , Seung-Ho Lim , Kyu Ho Park. *JFTL: A flash translation layer based on a journal remapping for flash memory* ACM Transactions on Storage (TOS), v.4n.4, p.1-22, January 2009.

[16] Jen-Wei Hsieh , Li-Pin Chang , Tei-Wei Kuo. *Efficient on-line identification of hot data for flash-memory management* Proceedings of the 2005 ACM symposium on Applied computing, March 13-17, 2005, Santa Fe, New Mexico.

[17] Li-Pin Chang , Tei-Wei Kuo. *Efficient management for large-scale flash-memory storage systems with resource conservation* ACM Transactions on Storage (TOS), v.1 n.4, p.381-418, November 2005.

[18] Intel Corporation. *Understanding the Flash Translation Layer(FTL) Specification* Tech Report December 1998.

[19] J. Kang, H. Jo, J. Kim, and J. Lee. *A Superblock-based Flash Translation Layer for NAND Flash Memory* In Proceedings of the International Conference on Embedded Software (EMSOFT), pages 161ĺC170, October 2006. ISBN 1-59593-542-8.

[20] J.U. Kang, J. S. Kim, C. Park, H. Park, and J. Lee. *A multi-channel architecture for high-performance nand flash-based storage system* Journal of Systems Architecture, 53(9):644ĺC658, 2007.

[21] A. Kawaguchi, S. Nishioka, and H. Motoda. *A flash-memory based File System* Proceedings of the USENIX Technical Conference, 1995.

[22] Tei-Wei Kuo, Jen-Wei Hsieh, Li-Pin Chang, Yuan-Hao Chang. *Configurability of performance and overheads in flash management* Proceedings of the 2006 conference on Asia South Pacific design automation, January 24-27, 2006, Yokohama, Japan.

[23] Gye-Jeong Kim, Seung-Cheon Baek, Hyun-Sook Lee, Han-Deok Lee, Moon Jeung Joe. *LGeDBMS: a small DBMS for embedded system with flash memory* Proceedings of the 32nd international conference on Very large data bases, September 12-15, 2006, Seoul, Korea.

[24] *Iometer Project* http://www.iometer.org/.

[25] *GNU Wget Project* http://www.gnu.org/s/wget/.