Improving Update-Intensive Workloads on Flash Disks through Exploiting Multi-Chip Parallelism

Bingsheng He, Jeffrey Xu Yu, *Senior Member, IEEE*, and Amelie Chi Zhou

Abstract—Solid state drives (SSDs), or flash disks have been considered as ideal storage for various data-intensive workloads, because of the low random access latency and the intra-disk multi-chip parallelism. However, due to inherent nature of flash memories, update-intensive workloads cause the flash disk fragmented, and trigger costly *internal activities* such as cleaning and wear leveling. We use database transaction processing as a motivating update-intensive workload. Our studies based on a flash disk simulator as well as flash disks show that, these activities result in significant overhead to the I/O response time and system throughput. To resolve the impact of internal activities, we propose *dynamic page replications* to exploit the multi-chip parallelism on the flash disk. Specifically, we replicate the frequently blocked data pages to improve the data availability even when internal activities block the request. To reduce the overhead of replications, we take advantage of the idle periods in the flash chips for the I/O operations by writes to replicas or reads from replicas, and further develop a prediction model for the decisions on those I/O operations to minimize the interference to normal I/O operations. We evaluate our techniques with three public transaction benchmarks in the simulator as well as on the real flash disks. Our results demonstrate the effectiveness of our replication management on improving I/O response time and system throughput.

Index Terms—Multi-chip parallelism, flash disks, solid state drives, update-intensive workloads, transaction processing, replication

1 Introduction

Flash memory has dominated the storage for mobile devices and sensors for its advantages of light weight, power saving and shock resistance. Due to the increasing capacity and dropping price, flash-based solid state drives (SSDs, or flash disks) have emerged as a popular storage for enterprise applications. Recently, efficient data structures and algorithms optimized for flash disks have become a fruitful research field [14], [20], [23], [25], [30], [31], [32], [46], [48]. Those studies have demonstrated that flash disks are ideal storage for various workloads, due to low random access latency and high intra-disk multi-chip parallelism [1]. Main-stream flash disks offer less than 0.1 ms access latency, and each flash disk consists of multiple flash chips (or channels) for serving I/O requests in parallel. However, due to the inherent erase-before-write nature of flash memory, I/O writes, especially the small random writes, make flash disks fragmented. That results in costly internal activities such as cleaning (or garbage collection) and wear leveling [1], [9], [11], [41]. In this paper, we target at addressing the performance issues from fragmentation of running update-intensive workloads on flash disks.

- B. He and A.C. Zhou are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: bshe@ntu.edu.sg, czhou1@e.ntu.edu.sg.
- J.X. Yu is with the Systems Engineering and Engineering Management, Chinese University of Hong Kong, Shatin, N.T., Hong Kong. E-mail: yu@se.cuhk.edu.hk.

Manuscript received 2 Sept. 2013; revised 15 Dec. 2013; accepted 8 Jan. 2014. Date of publication 24 Feb. 2014; date of current version 5 Dec. 2014. Recommended for acceptance by J. Flich. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2014.2308199

Upon serving an I/O write, the flash disk needs to find a clean page for the write request. As a flash disk becomes fragmented, the number of clean pages diminishes. When a clean page is not available, due to the erase-before-write mechanism, a page write requires erasing a large block, called the *erase block*. An erase block usually consists of dozens of flash pages. For an I/O write request of Q pages, the worst case is that these pages are mapped to Q distinct erase blocks, which can result in Q block cleaning operations for the request. In addition to cleaning, wear leveling could move blocks around such that writes are uniformly distributed to blocks [1]. These internal activities are due to the inherent nature of the flash storage media, and they are doomed to occur as the flash disk becomes fragmented.

Previous studies [9], [11], [41] have observed the significant overhead caused by internal activities on flash disks. Chen et al. [9] showed that a fragmented flash disk suffers from ten times slower random writes. Shimpi [41] also showed the severe performance degradation on several commodity SSDs. Chen [11] viewed internal activities as outliers in transaction logging, causing ad-hoc high latency. These studies have demonstrated the significant performance problems of internal activities on specific flash disks. However, there has been little work [11], [47] on resolving the impact of internal activities. Chen [11] proposed scheduling synchronous logging to multiple flash memory drives to improve the logging throughput. Yoo et al. [47] proposed buffer management algorithms considering the number of block erasure operations. While those studies can reduce some occurrences of internal activities, they cannot reduce the overhead when internal activities do happen.

We use database transaction processing as our motivating example for a detailed study on the impact of internal activities. We adopt two complementary approaches, i.e., with a public simulator [27] and on real flash disks. The simulator has been used in previous studies [1], [36]. Since the details on flash translation layer (FTL) [18] and internal activities are mostly private properties of hardware vendors, the simulator allows us to understand how the internal activities affect the performance under different FTL configurations. On both simulated and real flash disks, we have observed that internal activities are an important performance factor when the flash disk is fragmented. The simulations show that fragmentation results in 71 percent throughput degradation, and increases the average response time by 100 percent.

Different from traditional hard disks, flash disks embrace different degrees of parallelism within the disk. For example, flash chips within a flash disk are able to serve I/O requests independently. Even within a flash chip, some degree of parallelism, e.g., interleaving, is offered. Therefore, we propose replication to exploit the multi-chip parallelism within the flash disk. The core idea is to replicate the pages with the most performance gain across the chips. When an I/O request is blocked by any internal activity, it can be served by other available chips. Given a space budget, we develop a novel replication manager to dynamically determine the set of pages to replicate so that the performance gain is maximized. The replication manager considers workload locality and blocking probabilities of different pages. Replications induce extra I/O operations. To reduce their interferences to normal I/O operations, we take advantage of the idle periods in the flash chips for the I/O operations by writes to replicas or reads from replicas. We further develop a prediction model for the decisions on those I/O operations to minimize the interference to normal I/O operations.

We conduct the experiments with simulations and real flash disks. The simulation shows that our replication technique reduces the average response time for transaction processing by 59 percent. The experiments on real flash disks show that replication improves the performance of the three benchmarks with a reduction on the average response time by 22 percent, and reduces the response time of the 1 percent worst transactions by 39 percent.

Organization. The rest of the paper is organized as follows. Section 2 introduces the preliminaries and reviews the related work. Section 3 gives the motivation and an overview of our approach. We present the design and implementation of the replication manager in Section 4, followed by the evaluation in Section 5. Finally, we conclude in Section 6.

2 Preliminary and Related Work

2.1 Flash Disks

Flash disks with different hardware and software designs can vary significantly on the storage capacity and I/O performance [7]. In this paper, we focus on more recent flash disks, each consisting of a set of flash chips that are connected to a controller. Compared with magnetic hard disks, flash disks differ in the following three unique features. First, flash page updates are copy-on-write, and pages must be erased before being overwritten. Second, flash disks have high in-disk parallelism: the chips can serve I/O requests in parallel. While a single chip has relatively low bandwidth, the aggregated bandwidth from parallel chips can be high. Parallelism of different *storage elements* such as channels, packages, dies and planes is available on flash disks [8]. This paper focuses on chip-level parallelism. Due to the chip-level parallelism, the flash disk can serve I/O requests while performing internal activities. Third, a software layer FTL is used to maintain the address mapping, and to perform various internal activities including erasures, cleaning and wear leveling. This paper focuses on reducing the runtime overhead of internal activities on update-intensive workloads.

We briefly introduce two kinds of internal activities including cleaning and wear leveling. More details on internal activities can be found in previous studies [1], [7], [9].

Cleaning. Due to the erase-before-write nature, a page write generates an out-dated page, and may trigger garbage collection when a clean page is not available. Upon garbage collection, it reclaims the outdated pages in the candidate block for reuse. Before cleaning, it reads all the valid pages, and writes them back to the block after erasure.

Wear-leveling. A number of wear-leveling algorithms [1], [4] have been proposed to shuffle around the blocks for making the writes evenly performed.

SATA is a typical interface for connecting SSDs to computing systems. SATA III can offer more support for outstanding I/Os and queued TRIM commands for improved performance. We give more details in Appendix A of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2308199.

2.2 Flash-Aware System Design

There have recently been a lot of research efforts on flash-based data structures and algorithms from a single machine (e.g., [23], [25], [26], [48]) to distributed systems [15] and supercomputers [40]. We briefly describe the relevant related work on flash-aware system design, and more details can be found in a recent tutorial [6].

Lee and Moon [23] proposed an in-page logging strategy to reduce the number of writes. They further performed extensive experiments on a commercial DBMS [24], which show that flash disks can improve the overall performance of transaction processing. Flash-optimized data structures are developed to improve indexing performance [14], [25], [26], [48]. Due to asymmetry of reads and writes on flash disks, different write-optimized policies are developed, e.g., delayed writes [31], [32], [35], partial writes [38], write clustering [34] and repairing [43]. These techniques can reduce the number of occurrences of internal activities. Our approach is complementary to these write-optimized techniques, and focuses on improving the performance with dynamic replications and taking advantage of multi-chip parallelism. Similar to our study, Roh et al. [37] proposed a flash-aware B+-tree structure with optimizations on multi-chip parallelism. They have not considered replications. RAIDs have been adapted to multiple flash disks in order to improve the reliability [2], [17]. We currently investigate our replication management

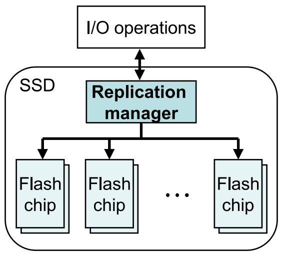


Fig. 1. The replication manager is implemented as a component in FTL handling I/O operations from upper-level applications.

within a single flash disk, and the extensions to multiple flash disks are straightforward by treating different flash disks as another level of parallelism.

There have been other studies leveraging the internal management routines of SSDs. We review those studies in two major categories. The first category is on how to exploit the internal chip parallelism for optimizing the performance of specific applications/operations. Examples are B+tree [37], scan and join [21] and other database operations [10]. Hu et al. [16] studied the impact of different levels of parallelism on the endurance. The second category is on how to make other system components aware of internal activities like erase operations. Yoo et al. [47] developed the page replacement algorithm with the goal of reducing the number of erase operations and improving the wear-leveling degree of flash memory. Lee et al. [22] proposed buffer-aware garbage collection algorithms for reducing unnecessary page migrations. Jeon et al. [19] explored how FTL designs can be adapted to MapReduce workloads. In contrast, this paper develops the replication-based mechanism to dynamically exploit the internal parallelism, with little modification on the application.

3 Overview

We have conducted some benchmark studies on real flash disks and simulations. Our motivation studies have shown that: 1) the overhead of internal activities is significant for update-intensive workloads. 2) the intra-disk multi-chip parallelism enables optimization opportunities for accessing other chips while a chip is blocked by internal activities. More details about those studies can be found in Appendix B of the supplementary file, available online.

The goal of this paper is to improve the performance of update-intensive workloads on the flash disk through reducing the overhead of the internal activities. Ideally, we have two directions for this goal, one is to reduce the frequency of internal activities, and the other is to reduce the overhead when internal activities occur. In this study, we address the latter case with replications. Given a budget of disk space, we aim at maximizing the performance gain with dynamic replication management.

The dynamic replication management is performed by a replication manager, which exploits the inherent intra-disk parallelism. Fig. 1 shows the architectural overview of the replication manager for handling the I/O operations. The replication manager is designed as a system component in

TABLE 1
Notations Throughout the Paper

Parameter	Description
B	Page size (KB)
I	The number of chips in the flash disk
E	Erase block size (in number of pages)
C_E	The cost of erasing a block
C_R, C_W	The cost of reading and writing a page (without cleaning), respectively
K	The maximum number of replicas in a replication group
D	The threshold value for detecting an internal activity (by default, $D = 2C_W$)

FTL. When receiving I/O reads and writes from the upper-level application, it issues the actual I/O reads and writes to flash chips, according to the I/O handling algorithms (Section 4.1). If a request is blocked by an internal activity, the request may be redirected to the other flash chips for reading or writing a replica. As for replication, the replication manager identifies the set of pages to replicate in order to exploit the intra-disk parallelism.

To fully utilize the memory bandwidth, we propose to leverage the idle period to perform the *extra* accesses by the replication manager. This is also to reduce the interference to the normal I/O operations. The extra operations include the writes caused by replications and the reads to the replica. Those extra operations may pose interference to normal I/O operations. Ideally, we should identify the lengthy idle periods that can accommodate the extra operations. However, it is impossible or impractical to predict the length of the next idle period without a priori knowledge. Thus, we periodically estimate the idle period distribution and determine threshold values for individual flash chips according to the distribution. Only if the current idle period exceeds the threshold, we perform an extra operation. The details of deciding extra accesses are presented in Section 4.2.2.

4 Replication Manager

Replication is widely used to increase the data availability in computer systems [3], [42]. We design and implement the replication manager to improve the data availability when internal activities occur. It basically has two functionalities. First, it receives the I/O requests from upper-level applications (such as databases), handles them and returns the data. Second, it determines the set of pages to be replicated and when to perform extra accesses. Given a budget of disk space (i.e., replication space), the replication manager needs to perform replacement when the budget is reached.

In the remainder of this section, we first describe I/O handling algorithms provided by our replication manager. Next, we present the details on replication management, including the policies for selecting pages to replicate and deciding when to perform extra accesses. Table 1 shows the notations used throughout the paper.

4.1 I/O Handling with Replication

We define a *replica group* of a page to be the set of replicas for the page. In our replication design, each page can have

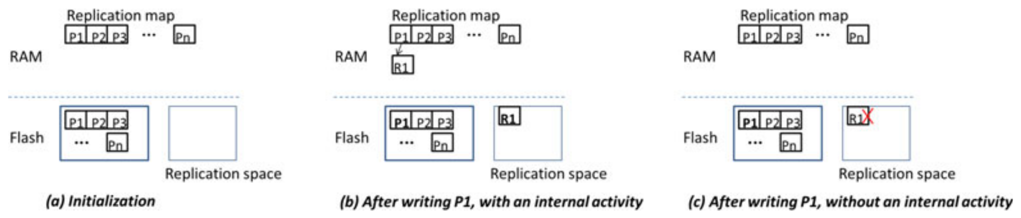

Replication map
P1 P2 P3 ··· Pn
RAM
Flash
Replication space
R1
(a) Initialization
(b) After writing P1, with an internal activity
(c) After writing P1, without an internal activity

Fig. 2. An example of replication manager with *RWrite*. RAM denotes the on-disk buffer.

up to K replicas, where K denotes the maximum number of replicas per page. Ideally, to achieve data availability, we should replicate the page to different storage elements within a flash disk. Thus, K is no larger than the number of parallel storage elements in the flash disk.

The replication manager has two algorithms namely *RRead* and *RWrite* to handle the I/O operations from the upper level applications. Particularly, *RRead* reads a page from a replica group, and *RWrite* writes a page and forms a replica group if necessary. Procedures *RRead* and *RWrite* are shown in Algorithms 1 and 2, respectively. The read and write operations used in those two procedures are implemented with non-blocking I/O operations.

Algorithm 1 *RRead*: handing a page read from a replica group

Description: read a page p from its replica group g.

```
1: while $p = null$ do
2:     let $r$ be the next unissued replica in $g$;
3:     if $r = null$ then
4:         wait;
5:     else
6:         read a page $p$ from $r$;
7:         add reading $r$ into the pending request list, $L$;
8:     if $\exists l \in L$ is done within time $D$ then
9:         let $p$ be $l$'s result;
```

Algorithm 2 *RWrite*: handling a page write

Description: write a page p, and form a new replica group g.

```
1: $iodone = false$;
2: $g = null$;
3: while $iodone \neq true$ do
4:     if the number of replicas in $g$ reaches $K$ then
5:         wait;
6:     else
7:         write the page $p$ to the page $pid$;
8:         add writing $p$ into the pending request list, $L$;
9:         add $pid$ into $g$;
10:    if $\exists l \in L$ is done within time $D$ then
11:        $iodone = true$;
12:    else
13:        pick a random partition and write to a page, and let it be $pid$;
```

These two algorithms are designed with similar ideas, considering both replication and the occurrence of internal activities. Let us illustrate the *RRead* interface in more details. The occurrence of internal activities can be identified through the chip status of FTL [1], [18]. The *RRead* interface submits the I/O request to the replicas one by one. If the requests are blocked by internal activities, we issue a request to another unissued replica in the replica group. When selecting an unissued replica, the one with the lowest blocking probability is preferred. We estimate this blocking probability based on the history access statistics. In particular, we define *blocking probability* to be the ratio of accesses being blocked, i.e., $\frac{\#Blocked}{\#Total}$, where $\#Blocked$ and $\#Total$ are the number of accesses that are blocked by internal activities and the total number of accesses to the page, respectively. If two chips have very similar blocking probabilities (e.g., the difference is smaller than 0.05), we choose the one with lower bandwidth unitization (i.e., with more idle cycles). If such a replica is not available, we need to wait until one of the requests returns. This is the worst case for the response time of *RRead*. To reduce the response time, the result of the earliest finished read is returned, and the results of other unfinished reads are discarded later.

In our implementation of *RRead* and *RWrite*, we need data structures for recording the pages in a replica group as well as whether a page in the replication space is valid or not. In particular, we maintain two map structures in the internal RAM of SSD: replication map and valid map. The replication map maintains the mapping from the physical page ID to its replica group. The valid map is a bitmap indicating whether a page in the replication space is valid or not (zero for an invalid page and one for a valid page). When receiving an I/O request from the upper-level application, the replication manager checks the replication map for the replica group, and uses the valid map to manage the replication space. An example illustrating those two map structures is shown in Fig. 2. Initially, all replica groups have one page only, and all bits in the valid map are zero (Fig. 2a). Next, an *RWrite* operation is performed on P1 and an internal activity occurs. As shown in Fig. 2b, a page R1 is replicated in the replication space, and the valid bit of R1 is set to be one. Next, another *RWrite* operation is performed on P1 and no internal activities occur. There is no page replicated and the valid bit of R1 is set to be zero, indicating R1 can be reclaimed (Fig. 2c).

There are some design issues worth noting for *RWrite*. First, *RWrite* adopts the primary replication protocol [42]. The firstly written page is the primary copy, and others are secondary copies. The second issue is about the data consistency among replicas. When we update a replicated page, we generate its replicas according to *RWrite*, instead of performing in-place updates on other replicas. This is because of the erase-before-write hardware feature of the flash disk. The in-place update is equivalent to rewrite on the flash disk. Therefore, when a page is updated, we mark all its replicas invalid, and create new replicas in *RWrite* if necessary. The replicas are always consistent after *RWrite*. Third, similar to *RRead*, the earliest finished write operation makes *RWrite* return to upper-level applications. This is for the sake of reducing the response time.

We give more implementation details on *RRead* and *RWrite* in Appendix C of the supplementary file, available online.

4.2 Dynamic Replication Management

Having described the algorithms for handling I/O operations, we now present the detailed design of the replication manager. The dynamic replication management has two major functionalities. First, we need to select the page that is most likely to achieve a high performance gain to replicate, given the replication budget. That leads to the design of page selection policies. Second, the extra accesses cause interferences to the normal I/O accesses, and we take advantage of idle periods to perform the extra accesses in order to reduce the interference. Thus, we develop algorithms to decide when to perform the extra accesses.

4.2.1 *Page Selection Policies*

While replication can improve the response time upon the occurrences of internal activities, it has the two drawbacks. First, replicating a page for K times causes $(K-1)$ extra writes. These writes consume disk bandwidth, especially for the poor random write performance on the flash disk. Even worse, they can trigger more internal activities. Second, replication consumes disk space. Ideally, we should replicate the hot pages that are frequently blocked during their accesses, such that their replicas are available when internal activities occur.

Analyzing the replication management, we find that dynamic replication management is analogous to buffer management. Both of these two management problems are to maximize the performance gain given the limited memory/disk space budget. The disk space budget in the replication management corresponds to the buffer size in the buffer management. Upon a page access, a page has been replicated (i.e., a *hit* in the replication management), which corresponds to the page access hit in the buffer management. On the other hand, a page is not replicated (i.e., a *miss* in the replication management), which corresponds to a miss in the buffer management. Thus, the hit rate is an important metric for the replication manager. Based on such a correspondence, a large set of replacement policies in buffer management becomes relevant to the replication management.

We define the *replica group set*, $\mathbb{S}$, to be the set of replication groups in the replication manager. We define the size of $\mathbb{S}$, $|\mathbb{S}|$, to be the total space of the pages in $\mathbb{S}$. For a page access, if the page belongs to any replica group in the replica group set, the access is a hit. Otherwise, it is a miss.

Since least recently used (LRU) is a common and basic algorithm, we start with LRU. The detailed algorithm can be found in Appendix C of the supplementary file, available online. We have developed a more advanced algorithm for replication management.

Due to the hardware design and workload characteristics, the blocking probabilities are different across pages. We observed this phenomenon on both simulated and real flash disks. There are two major causes for the uneven blocking probabilities.

The first cause is on spatial reasons. The blocking probability of a page is affected by other pages within the same erase block. In an SSD with block-based FTL mapping, if a page has a very high update rate, the pages within the same erase block have a relatively high blocking probability. Moreover, in some FTL design, blocks within the same chip are allocated with a high priority [13]. On the other hand, the page/block mapping often uses a hybrid approach combining both static and dynamic page mapping [1], [5], [12]. Static page mapping can lead to non-uniform occurrence of internal activities across chips.

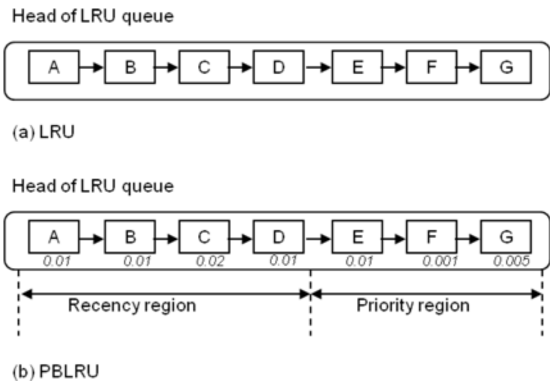


Fig. 3. Dynamic replication management: (a) an example of LRU, (b) an example of two regions in PBLRU.

The second cause is on temporal reasons. Some chips may have more frequent erasures than others, when the free blocks are unevenly distributed among the chips. While wear leveling attempts to resolve this unevenness, it is performed lazily when certain criteria are met [1].

To distinguish the pages with high and low blocking probabilities, we propose priority based least recently used (PBLRU) to manage the replacement of replication. The basic heuristic of PBLRU is to keep the pages with relatively high recency and frequency in the replica group set, and to perform replacement on the pages with low blocking probability. Thus, the pages with relatively low blocking probabilities are more likely to be selected as victims for replacement.

Ideally, we should assign the total order to pages according to recency and blocking probability. However, it is hard to combine those two metrics into a meaningful single metric. Thus, we develop a simple yet effective approach to distinguish the pages with high and low blocking probabilities.

In PBLRU, we divide the LRU queue into two regions, namely *recency region* and *priority region*. Recency region is from the head of the LRU queue (the mostly recently accessed replica group), while priority region is from the tail of the LRU queue. Fig. 3b shows an example of the LRU queue in PBLRU, consisting of seven replication groups. The size of priority region is three replication groups. The example blocking probability is listed below each replica group. We define the blocking probability of the replica group to be the sum of the blocking probability of all the pages in the replica group. The recency region consists of recently accessed replication groups and most of replication hits are generated in this region. The priority region consists of replication groups which are candidates for eviction. PBLRU selects a replica group to evict in the priority region according to the blocking probability of the replica group. The replica group with the lowest blocking probability is chosen as the victim.

The replacement order of PBLRU is different from that of LRU. For example, under LRU, the last element in the LRU queue is always evicted first. Thus, the priority for being a victim is in the order of G, F, and E in Fig. 3a. However, under PBLRU, according to the blocking probability, the evictions are in the order of F, G, and E.

The size of recency region (CR) is a tuning parameter. It is important to adjust CR properly according to the difference among the blocking probability. If the replication groups have almost the same blocking probability, we should keep CR large to avoid evicting the hot replication groups in the priority region. Otherwise, we should set CR to reasonably small, in order to keep the pages with high blocking probability. The experimental tuning evaluation will be presented in our performance study.

PBLRU handles an I/O request according to Algorithm 3. As PBLRU handles the I/O request, replicas are added to the replica groups. I/O reads and writes are handled differently in PBLRU. If the I/O request is a *RWrite*, we produce a new replica group, and replace the one in the replica group set in PBLRU, if available. For a *RRead*, we replicate the page according to its blocking probability. Only when the blocking probability is higher than a threshold (5 percent in our experiments), we replicate the page.

Algorithm 3 *PBLRU*: handling a blocked I/O request in the replication manager

Description: handling a blocked I/O request io on page p.

```
1: if $io$ is a write then
2:     /* After calling RWrite ($p$); */
3:     if $p$ is in the LRU queue then
4:         remove $p$ from the LRU queue;
5:     if $p$ is replicated then
6:         call AddRG to $p$ into the LRU queue;
7:         mark $p$ with L;
8: else
9:     /* After calling RRead ($p$); */
10:    if $p$ is in the LRU queue then
11:        move $p$ to the head;
12:        adjust the mark of $p$;
13:    else
14:        if $P_p$ is higher than the threshold then
15:            Replicate $p$;
16:            call AddRG to add $p$ to the LRU queue;
17:            if the LRU queue is full, remove a replica group from the LRU
               queue.
```

We give an estimation on the number of extra writes. Denote the blocking probability of a page p to be P_p. Extra writes come from the replications caused by *RRead* or *RWrite*. Suppose the numbers of *RRead* and *RWrite* operations on the page p are R_p and W_p, respectively. Thus, the number of extra writes is $E_p = W_p \times P_p + R_p \times P_p \times I(P_p > t)$, where $I(x)$ is an indicator function that equals to 1 if $x \geq 0$ and 0 otherwise and t is the threshold on blocking probability. Thus, the total number of extra writes are the sum of E_p for all pages p. From this analysis, we can see that the blocking probability is an important factor affecting the number of extra writes, which depends on the workload and FTL design. In practice, the blocking probability is expected to be low (mostly smaller than 5 percent in our studies). On the other hand, reads have relatively low chances to be blocked than writes. $I(P_p > t)$ is 1 for only a very small portion of read accesses. This observation is also consistent with the previous studies on real workloads [28]. Therefore, the ratio of extra writes in the total write traffic is small. The extra writes have a relatively small impact on the endurance of SSDs for real workloads.

4.2.2 Decisions on Extra Accesses

Compared with the normal I/O accesses, the replication manager issues two kinds of extra accesses, including writes for creating the replication group and reads for the replica when the target chip performs the internal activity. The extra access can cause interference to the normal I/O accesses, which can offset the performance gain of replication. There are many opportunities to exploit the idle period even for the I/O intensive workloads. Thus, we propose to utilize the idle periods for those extra accesses.

If the idle period is too short, the extra accesses can cause significant penalty, especially for writes. On the other hand, if the idle period is lengthy, there are opportunities of performing multiple extra accesses. However, it is impossible to predict the length of the current idle period without a priori knowledge. Thus, we estimate the idle period length distribution in a slot. Then, we adopt a simple approach to determine whether to perform an extra access: if the current idle period is longer than the *threshold*, we perform an extra access.

Taking advantage of idle periods, we implement writes in *RWrite* with asynchronous writes. We maintain a queue for those asynchronous writes for each chip. Upon a write is issued by *RWrite*, it is buffered in a queue of the corresponding chip. Only when we detect an idle period of the target chip exceeding a threshold value, we remove the write operation at the head of the queue, and perform the corresponding write.

Let us describe our estimation on determining the threshold values. The estimation is performed periodically (the period is called *epoch*). Within an epoch, we use the same threshold values. We estimate the threshold values at the beginning of an epoch according to the histogram of the previous epoch. We use a histogram to represent the distribution of idle lengths. Particularly, the histogram is used to hold the frequency counter of the idle period lengths. Suppose the histogram has $T+1$ buckets denoted as $Hist[i]$, $i = 0, 1, \ldots, T$, where $Hist[i]$ denotes the frequency counter for idle periods with length of i time units (in our experiments, the time unit μ equals to the read latency). $Hist[0]$ simply represents the number of the memory accesses. Given a write threshold of t time units, a memory access delay of $(t + \frac{C_W}{\mu} - i)$ will happen for an idle period of i time units.

To support multiple writes in a lengthy idle period, we use a vector $\vec{t}$ to represent the write threshold values within the same idle period. $\vec{t}[i]$ is the write threshold value for the ith write. Consider the scenario that an idle period starts. We perform the first write if the idle period reaches $\vec{t}[0]$ cycle. After the write finishes, if we are still in the same idle period, we will perform the second write if the idle period lasts for $\vec{t}[1]$ time units more. We repeat this process by M

TABLE 2
Default Parameter Settings in Simulator

Parameters	Default Setting
B	2KB
E	64 pages
I	4
C_R	0.12ms
C_W	0.4ms
C_E	1.5ms
K	4
Buffer pool size (pages)	1024
Reserved disk space, R	10%

times so that $\sum_{i=0}^{M-1}(\vec{t}\,[i]+\frac{C_W}{\mu})\geq L$, where L is the length of the longest idle period in number of time units.

We apply similar ideas to reads and writes. On flash memories, writes have a much higher overhead than reads. Reads and writes have different threshold values. Since writes can be performed in the asynchronous manner and reads cannot, the impact of taking advantage of idle periods is larger on writes than that on reads.

4.3 Discussions

After presenting replication management algorithms, we discuss the following issues for the replication manager.

Overhead. The overhead of PBLRU comes from two parts. The first part is on the storage overhead caused by replications. It is subject to the user-specified budget. The second part is the metadata stored in DRAM. In our experiments, the DRAM usage of the metadata is relatively small.

Limitations. There are several limitations in the design and implementation of the replication manager. First, our replication manager is designed for the SSD with intra-disk parallelism. That means, the proposed techniques are not suitable for low-end flash memories with little or no intra-disk parallelism. Second, extra writes can reduce the endurance, although the impact tends to be small in practice.

5 Evaluation

In this section, we present experimental results on the overhead of internal activities and on the replication management.

5.1 Experimental Setup

Our experiments are organized into two parts: simulations and experiments on real flash disks. The simulation allows us to study different hardware parameters in a fully controlled manner.

In simulations, we adopt the trace-driven flash disk simulator [27], which is used in previous studies [1], [33], [36]. Each trace entry represents the page access information (i.e., the file and the page to access) as well as a time stamp. The simulator is a single-threaded program, which simulates concurrent page accesses through replaying the traces according to the time stamp of the page accesses. We configured the simulator to emulate a 4 GB flash disk with the I/O settings specified in the Samsung data sheet [39]. We summarize the default parameter settings in Table 2. The life cycle of an erase block is 10,000 cycles. The simulator has implemented the basic strategies in FTL, including wear leveling and basic block mapping policies [1]. The threshold on the free disk space to trigger the garbage collection is set to 5 percent. The wear-leveling algorithm migrates a block when the block is with less than 85 percent remaining lifetime or the variance of the remaining lifetime among all blocks is above 20 percent. In our experiments, we vary the parameters that are mostly relevant to the overhead of fragmentation, including the number of chips, I, and the reserved disk space for cleaning, R, in terms of the percentage of the disk capacity.

Evaluating the impact of our algorithms on real flash disks is a challenging task. Our replication manager is currently designed as a runtime component in FTL. However, on real flash disks, FTL is a piece of firmware that is hidden by SATA interfaces, and we cannot modify FTL. Moreover, the multi-chip parallelism is also hidden from applications. Thus, we adopt an approximated approach in the previous study [11] for multi-chip parallelism within a flash disk. The replication manager is implemented as a software library. We divide the flash disk into P equal-sized partitions and treat each partition as a chip for replication. P is set to be the number of chips in the flash disk. We perform the replication by picking a free page from a random chip that does not have the page replica to be accessed. While the replicated request may also trigger internal activities, we find that this overhead is small in our experiments with benchmark workloads. Particularly, we ran our experiments on a Linux workstation with an Intel 2.4 GHz quad-core CPU, 8 GB main memory, and SATA flash SSDs. The five SSDs include three relatively legacy ones (an Mtron MSD-SATA3035 64 GB, an Intel X25-M 80 GB and an Intel X25-E 64 GB) and two recent ones (an OCZ Vector 256 GB and an Intel 335 series 240 GB). We calibrate P in the flash disk, and set $P=16$ for all SSDs (except $P=8$ for the Mtron SSD). We use direct I/O to remove the impact of operating system page cache.

Workloads. We use three benchmarks on transactional processing, namely TM-1 [29], TPC-B [44], and TPC-C [45]. The default settings for simulations are 2.4, 2.2 and 2.4 GB for TPC-C, TPC-B and TM-1, respectively. We use relatively small databases for simulations. The settings are sufficiently large for assessing the fragmentation behavior of the simulated 4 GB flash memory and meanwhile allow us to study the behavior in a detailed manner. The impact of varying different databases sizes will be studied in the real flash disk experiments. On real flash disk experiments, the default settings are 48.0, 22.4, 22.4 GB for TPC-C, TPC-B and TM-1, respectively.

We run the workload on PostgreSQL 8.4 with default settings, e.g., the page size is 8 KB. We collect the simulation trace from all the page accesses to the buffer manager of the PostgreSQL execution. For each real-disk experiment or collecting the simulation trace, we run a sufficient period of time around 3 hours, including a 30 minutes warm-up period. The number of clients is 10 for all benchmarks without thinking time. For other system parameters, the ratio of the buffer pool size to the database size is set to 1 percent by default. We also study the impact of the recency region size. The default setting is that the *recency region ratio* (i.e., the ratio of the recency region size to the size of the replication space) is 0.5. The default epoch size for calculating the suitable threshold values is

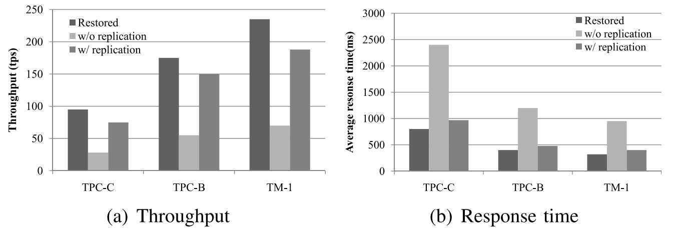


Fig. 4. The throughput and average response time of the three workloads.

1 minute. We evaluate the impact of different epoch sizes, and find that the performance is rather stable as long as the epoch is between 30 seconds and 5 minutes.

More detailed experimental setup and results can be found in Appendix D of the supplementary file, available online.

5.2 Simulation Evaluation

We use simulation to study the fragmentation/aging behavior of flash memories as well as the internal impact of our proposed approach, which cannot be exposed in the real-disk experiment. In particular, we compare the sustained performance with PBLRU, with the restored performance and the sustained performance without replication.

Fig. 4 shows the throughput and the average response time of the three workloads. We study the restored performance as well as the sustained performance with and without replication (denoted as "w/ replication" and "w/o replication," respectively). We make two major observations. First, compared with the restored state, fragmentation significantly degrades the transaction processing throughput and increases the average response time. The performance degradation is similar for all these three workloads. For example, on TPC-C, the throughput decreases by 71 percent and the average response time increases by 100 percent. Second, our replication techniques are helpful on fragmented flash disk, exploiting the chip parallelism within the flash disk. Replication improves the throughput by 176 percent and reduces the average response time by 59 percent. The sustained throughput with replication is only 26 percent lower than the restored throughput, and the response time is 25 percent higher.

The significant performance degradation is due to the increasing overhead of the internal activities, as the flash disk ages. Fig. 5 shows the time breakdown per run when the buffer contains 1,024 pages. We divide the total time into two parts, *Busy* and *Other*, which represent the time when the chip handles I/O requests and time components other than the busy time, respectively. The *Busy* time is

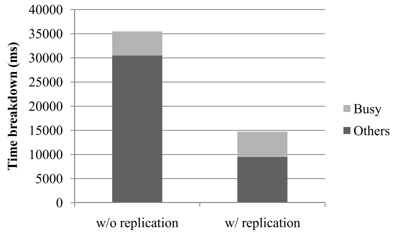


Fig. 5. The time breakdown with and without replication per run.

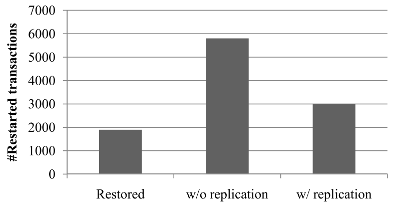


Fig. 6. The number of restarted transactions.

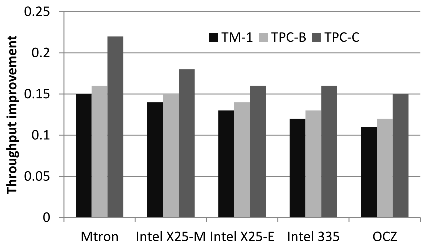
directly obtained from the simulator. The *Other* component measures the overhead when transaction executions are blocked by internal activities. Without replication, the busy time is only 14 percent of the total time. Due to the overhead of internal activities, the concurrency level of transaction processing is reduced. This is validated by the dramatic increase in the response time of transactions. Replication helps on exploiting the portion of idle time. With replication, the *Busy* time contributes to 33 percent of the total time. Since replication exploits the chip parallelism in the flash disk, it issues more reads/writes to the flash disk. This results in a higher bandwidth utilization.

We next investigate the overhead of fragmentation on concurrency control. We also observe an increasing number of restarted transactions. Fig. 6 shows the number of restarted transactions per run when the flash disk is in the sustained state. Due to fragmentation, the time for holding a lock by a transaction becomes long, which increases probability of transaction conflicts. Specifically, fragmentation increases the number of restarted transactions by 200 percent, and replication reduces this number by 50 percent.

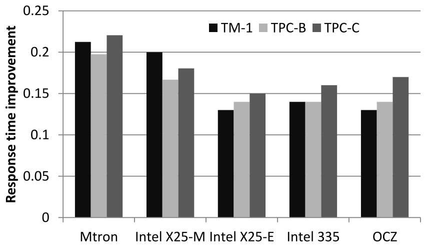
5.3 Results on Real Flash Disks

We first study the performance of running the three benchmarks on PostgreSQL with and without replication. We integrate the replication manager into PostgreSQL, as an extension to the storage manager. Fig. 7 shows performance improvement of our replication approach when the disk space budget for the replication manager is set to be 10 percent of the database size. We measure the response time with default settings, e.g., 10 clients and without thinking time. The policy used in replication manager is PBLRU. Replication improves the average transaction response time on real flash disks, with an improvement of 14-22 percent. We observe similar improvement on the throughput comparison. The largest improvement is 22 percent for TPC-C on Mtron disk. The main memory consumption of the replication manager is less than 20 MB for all the benchmarks (2-3 percent of the main memory buffer size).

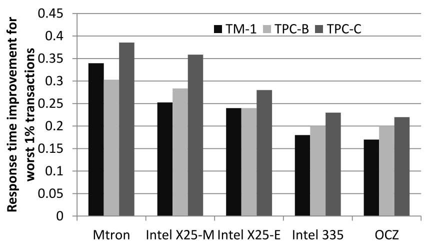
Let us study the overhead of replication in more details. The overhead includes the storage overhead as well as the interference to the normal I/O operations. In our experiments, the maximum number of replicas per replica group (K) is three on all real flash disks. Moreover, most replica groups have two replicas. For example, on the Mtron disk, the number of replicas in over 95 percent replica groups is two, and the remainder 5 percent are three. Table 3 shows the ratios categorized by the number of I/O per operation in *RRead* and *RWrite*. Over 80 percent of *RRead* and 54 percent of *RWrite* calls issue a single I/O operation. A very small ratio of *RRead* and *RWrite* issues three I/Os within



(a) Throughput



(b) Response time



(c) 1% worst transactions

Fig. 7. Overall results on performance improvement (#users $= 10$, no thinking time).

one call. This result indicates that chip parallelism is available with a high probability for our simple approach.

As for the interference to normal I/O operations, we calculate the interference in a chip to be the total time that an extra access is being performed and at the same time there is at least one normal I/O request in the queue of the chip. In our measurements, the interference is smaller than 3 percent of the total execution time on all chips.

Fig. 7c shows the average response time improvement for the 1 percent worst transactions of our replication based approach. Replication significantly reduces the average response time of the 1 percent worst transactions. The improvement of TPC-C is 20-39 percent. The improvements are more significant than those on the average response time of all transactions. This is because replication effectively hides the overhead of internal activities, reducing the response time outlier.

Table 4 compares the ratio of extra writes compared with the approach without replication on Intel X25-E and OCZ disks. The ratio of extra writes caused by page replications in PBLRU is lower than 5.4 percent for the three benchmarks. Due to those extra writes, our replication introduces a reasonable small increase on the occurrences of internal activities and a small degradation on the life time. By differentiating the pages with blocking probabilities, PBLRU always has a smaller number of writes than LRU. We further analyze the extra writes caused by *RWrite* or *RRead*, and find that over 90 percent of extra writes are caused by *RWrite*. Additionally, our estimation on the number of extra writes is close to the measurement (with the difference of 10-20 percent).

TABLE 3
Ratio of *RRead* and *RWrite* Calls Categorized by the Number of I/O per Operation on Mtron

	1	2	3
RRead	80.2%	18.5%	1.3%
RWrite	54.2%	39.9%	5.9%

TABLE 4
Ratio of Extra Writes Compared with "w/o Replication" Approach on Intel X25-E and OCZ Disks

X25-E	TPC-C	TPC-B	TM-1
LRU	8.2%	7.4%	6.5%
PBLRU	5.4%	5.2%	4.9%
OCZ	TPC-C	TPC-B	TM-1
LRU	7.3%	6.2%	5.4%
PBLRU	3.9%	4.1%	3.5%

5.4 Comparisons with Other Approaches

We further adopted two recent algorithms [2], [47] to our scenario for comparison. Overall, we observe similar results on the evaluated disks: 1) while those two individual techniques can also reduce the overhead of internal activities, they are generally inferior to our replication approach; 2) combining those individual techniques and replication often leads to larger performance improvements. For space interests, we present the detailed results on comparing CFLRU/E [47], with a focus on the results from TPC-C on the OCZ flash disk only. The comparisons with differential RAID [2] can be found in Appendix D of the supplementary file, available online.

CFLRU/E [47] is originally designed as a buffer management algorithm, which chooses the victims with the priority firstly on the clean pages and secondly the dirty pages with the lowest block erase frequency. We approximate the block erase frequency with our blocking probability. Current PostgreSQL uses a LRU-like algorithm, and a direct comparison with CFLRU/E is unfair. Therefore, we further implemented CFLRU/E into our replication approach, and denote this algorithm to be "CFLRU/E +Replication." Table 5 shows the throughput and response time improvement over the baseline (without replication) and the normalized number of writes of TPC-C on the OCZ disk. We normalize the number of writes to the number of writes in the baseline. The number of writes is a direct measurement on the lifetime of the flash disk. We make two major observations. First, CFLRU/E has better performance than the baseline approach. This is due to the significant reduction on the number of writes by the flash-aware buffer management in CFLRU/E. Second, the combined approach on replication and CRLRU/E further improves the throughput.

6 Concluding Remarks

In this paper, we study the transaction processing as one typical and important update-intensive workloads on flash disks. We experimentally evaluate the impact of costly

TABLE 5
Comparing Replication and CFLRU/E [47] of Running TPC-C on the OCZ Disk

	Throughput improvement	Response time improvement	Normalized #writes
Replication	16%	15%	103.9%
CFLRU/E	5%	7%	90.5%
CFLRU/E + Replication	18%	19%	97.8%

internal activities with two complementary approaches, simulations and real flash disks. Both approaches demonstrate significant performance degradation of transaction processing on fragmented flash disk. We further develop a dynamic replication approach to improve the performance of update-intensive workloads on flash disks. Our replication exploits the data localities to maximize the effectiveness of replication, as well as the idle periods to reduce the overhead. Our simulation results show that, under various flash disk configurations, the replication manager improves the response time by 59 percent, and the experimental results on real flash disks demonstrate an improvement by 22 percent on the response time. Our work is relevant to big data and enterprise applications for improving the performance of update-intensive workloads with SSDs.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. The work was in part supported by Research Grants Council of the Hong Kong SAR, China No. 418512 and SUG Grant M4080102.020 of Nanyang Technological University, Singapore. The author Amelie Chi Zhou is also with Nanyang Environment and Water Research Institute (NEWRI).

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigraphy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf.*, 2008.

[2] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD Reliability," *Proc. Fifth European Conf. Computer Systems (EuroSys)*, 2010.

[3] P.A. Bernstein and N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. Database Systems*, vol. 9, pp. 596-615, Dec. 1984.

[4] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *ACM SIGOPS Operating Systems Rev.*, vol. 41, no. 2, pp. 88-93, 2007.

[5] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," *Proc. Eighth USENIX Conf. File and Storage Technologies (FAST)*, 2010.

[6] P. Bonnet, L. Bouganim, I. Koltsidas, and S. Viglas, "System Co-Design and Data Management for Flash Devices," *Proc. VLDB Endowment*, vol. 4, pp. 1504-1505, 2011.

[7] L. Bouganim, B.T. Jónsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," *Proc. Fourth Biennial Conf. Innovative Data Systems Research (CIDR)*, 2009.

[8] Y.-H. Chang and T.-W. Kuo, "A Management Strategy for the Reliability and Performance Improvement of MLC-Based Flash-Memory Storage Systems," *IEEE Trans. Computers*, vol. 60, no. 3, pp. 305-320, Mar. 2011.

[9] F. Chen, D.A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," *Proc. 11th Int'l Joint Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[10] F. Chen, R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-Speed Data Processing," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA)*, 2011.

[11] S. Chen, "Flashlogging: Exploiting Flash Devices for Synchronous Logging Performance," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[12] M.-L. Chiao and D.-W. Chang, "Rose: A Novel Flash Translation Layer for NAND Flash Memory Based on Hybrid Address Translation," *IEEE Trans. Computers*, vol. 60, no. 6, pp. 753-766, June 2011.

[13] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A Survey of Flash Translation Layer," *J. Systems Architecture*, vol. 55, nos. 5-6, pp. 332-343, May 2009.

[14] B. Debnath, S. Sengupta, J. Li, D.J. Lilja, and D.H.C. Du, "BloomFlash: Bloom Filter on Flash-Based Storage," *Proc. 31st Int'l Conf. Distributed Computing Systems (ICDCS)*, 2011.

[15] S. Gurumurthi, "Architecting Storage for the Cloud Computing Era," *IEEE Micro*, vol. 29, no. 6, pp. 68-71, Nov./Dec. 2009.

[16] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance," *IEEE Trans. Computers*, vol. 62, no. 6, pp. 1141-1155, June 2013.

[17] S. Im and D. Shin, "Flash-Aware Raid Techniques for Dependable and High-Performance Flash Memory SSD," *IEEE Trans. Computers*, vol. 60, no. 1, pp. 80-92, Jan. 2011.

[18] Intel, Understanding the Flash Translation Layer (FTL) Specification, Intel Corporation, 1998.

[19] H. Jeon, K. El Maghraoui, and G.B. Kandiraju, "Investigating Hybrid SSD FTL Schemes for Hadoop Workloads," *Proc. ACM Int'l Conf. Computing Frontiers (CF)*, 2013.

[20] I. Koltsidas and S. Viglas, "Flashing up the Storage Layer," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 514-525, 2008.

[21] W. Lai, Y. Fan, and X. Meng, "Scan and Join Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives," *Proc. 14th Int'l Conf. Web-Age Information Management (WAIM)*, 2013.

[22] S. Lee, D. Shin, and J. Kim, "BAGC: Buffer-Aware Garbage Collection for Flash-Based Storage Systems," *IEEE Trans. Computers*, vol. 62, no. 11, pp. 2141-2154, Nov. 2013.

[23] S.-W. Lee and B. Moon, "Design of Flash-Based DBMS: An in-Page Logging Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2007.

[24] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pp. 1075-1086, 2008.

[25] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, pp. 1303-1306, 2009.

[26] Y. Li, B. He, R.J. Yang, Q. Luo, and K. Yi, "Tree Indexing on Solid State Drives," *Proc. VLDB Endowment*, vol. 3, nos. 1-2, pp. 1195-1206, 2010.

[27] Microsoft, *SSD Extension for DiskSim Simulation Environment*, Microsoft Corporation, 2008.

[28] V. Mohan, T. Siddiqua, S. Gurumurthi, and M.R. Stan, "How I Learned to Stop Worrying and Love Flash Endurance," *Proc. Second USENIX Conf. Hot Topics in Storage and File Systems (HotStorage)*, 2010.

[29] Nokia, *Network Database Benchmark*, http://hoslab.cs.helsinki.fi/homepages/ndbbenchmark/, 2014..

[30] Y. Oh, J. Choi, D. Lee, and S.H. Noh, "Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage System," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, 2012.

[31] S.T. On, S. Gao, B. He, M. Wu, Q. Luo, and J. Xu, "FD-Buffer: A Cost-Based Adaptive Buffer Replacement Algorithm for Flash Memory Devices," *IEEE Trans. Computers*, vol. PP, no. 99, p. 1, 2013.

[32] S.T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu, "FD-Buffer: A Buffer Manager for Databases on Flash Disks," *Proc. 19th ACM Conf. Information and Knowledge Management (CIKM)*, pp. 1297-1300, 2010.

[33] S.T. On, J. Xu, B. Choi, H. Hu, and B. He, "Flag Commit: Supporting Efficient Transaction Recovery in Flash-Based DBMSs," *IEEE Trans. Knowledge and Data Eng.*, vol. 24, no. 9, pp. 1624-1639, Sept. 2012.

[34] Y. Ou, T. Härder, and P. Jin, "CFDC: A Flash-Aware Replacement Policy for Database Buffer Management," *Proc. Fifth Int'l Workshop Data Management on New Hardware (DaMoN)*, 2009.

[35] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[36] V. Prabhakaran, T.L. Rodeheffer, and L. Zhou, "Transactional Flash," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2008.

[37] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 286-297, Dec. 2011.

[38] K.A. Ross, "Modeling the Performance of Algorithms on Flash Memory Devices," *Proc. Int'l Workshop Data Management on New Hardware (DaMoN)*, 2008.

[39] *K9XXG08UXA Datasheet*, http://www.samsung.com/Products/Semiconductor/, Samsung Corporation, 2014.

[40] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpointing System," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[41] A.L. Shimpi, *The SS Relapse Understanding and Choosing the Best SSD*, 2009.

[42] S.H. Son, "Replicated Data Management in Distributed Database Systems," *ACM SIGMOD Record*, vol. 17, pp. 62-69, Nov. 1988.

[43] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and Repairing Write Performance on Flash Devices," *Proc. Int'l Workshop Data Management on New Hardware (DaMoN)*, 2009.

[44] TPC, TPC *Benchmark B: Standard Specification*. http://www.tpc.org/tpcb/spec/tpcb_current.pdf, 2014.

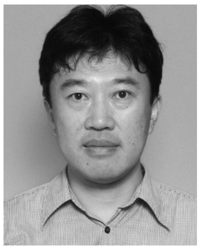
[45] TPC, *TPC Benchmark C: Standard Specification*. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2014.

[46] D. Tsirogiannis, S. Harizopoulos, M.A. Shah, J.L. Wiener, and G. Graefe, "Query Processing Techniques for Solid State Drives," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[47] Y.-S. Yoo, H. Lee, Y. Ryu, and H. Bahn, "Page Replacement Algorithms for Nand Flash Memory Storages," *Proc. Int'l Conf. Computational Science and Its Applications (ICCSA)*, 2007.

[48] D. Zeinalipour-Yazti et al., "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices," *Proc. Fourth USENIX Conf. File and Storage Technologies (FAST)*, 2005.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2008. He is currently an assistant professor in the Division of Networks and Distributed Systems, School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems.



Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science, from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He is currently a professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include graph mining, graph database, keyword search, and query processing and optimization. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of the ACM.

Amelie Chi Zhou received the bachelor's degree in 2009, and the master's degree in 2011, both from Beihang University. She is currently working toward the PhD degree from the School of Computer Engineering, Nanyang Technological University, Singapore. Her research interests include cloud computing and database systems.