StableBuffer: Optimizing Write Performance for DBMS Applications on Flash Devices

Yu Li, Jianliang Xu, Byron Choi, and Haibo Hu
Dept. of Computer Science
Hong Kong Baptist University
Hong Kong SAR, China
{yli, xujl, bchoi, haibo}@comp.hkbu.edu.hk

ABSTRACT

Flash devices have been widely used in embedded systems, laptop computers, and enterprise servers. However, the poor random writes have been an obstacle to running write-intensive DBMS applications on flash devices. In this paper, we exploit the recently discovered, efficient write patterns of flash devices to optimize the performance of DBMS applications. Specifically, motivated by a *focused write pattern*, we propose to write pages temporarily to a small, pre-allocated storage space on the flash device, called *StableBuffer*, instead of directly writing to their actual destinations. We then recognize and flush *efficient* write patterns of the buffer to achieve a better write performance. In contrast to prior log-based techniques, our StableBuffer solution does not require modifying the driver of flash devices and hence works well for commodity flash devices. We discuss the detailed design and implementation of the StableBuffer solution. Performance evaluation based on a TPC-C benchmark trace shows that StableBuffer improves the response time and throughput of write operations by a factor of 1.5−12, in comparison with a direct write-through strategy.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Information Storage; H.2 [**Database Management**]: Physical Design

General Terms

Algorithms, Design, Performance

Keywords

Databases, write performance, flash devices

1. INTRODUCTION

Flash devices have been widely used in embedded systems such as PDAs and smartphones for many years, and recently have been gaining increasing popularity in laptop computers and enterprise servers in the form of Solid State Drives (SSDs). International Data Corporation (IDC) predicts that the growing needs of large-scale Web-based and cloud computing companies will drive up the sales of enterprise flash devices by an average of 165% annually until 2013 [15]. It is expected that more and more DBMS applications will run on flash devices.

Compared with hard disk drives, flash devices offer a smaller form factor, lighter weight, lower power consumption, and higher shock resistance. Moreover, without mechanical moving parts, flash devices are faster in random reads. But they are less competitive in writes, especially random writes, due to an erase-before-write limitation. Thus, many research studies have been carried out to optimize the write performance of flash devices. One pioneer work is the in-page logging technique [16]. The idea is to log changes made to a data page in a reserved area of the flash block, instead of updating the page directly. In this way, the in-page logging technique turns random writes into change logs which are sequentially written to the log area. When the log area is full, the change logs are merged into their data pages. In a more recent study [14], page-differential logging suggests writing only the content difference between the original page and the up-to-date page as so to reduce the page read overhead incurred in in-page logging. In common, these two techniques require modifying the driver or firmware of flash devices, e.g., augmenting FTL[1] with logging support. However, this may not be always possible for commodity flash products as their hardware and software designs are often proprietary. Thus, this requirement makes log-based techniques have limited applications.

Recent studies on flash I/O performance [4][8] reveal that the write performance of flash devices depends on not only simple parameters such as access types (*i.e.*, sequential or random) and granularity (*i.e.*, page size), but also write patterns. More specifically, *uFLIP* [4] shows that a general random write that writes pages randomly to the entire disk space performs badly. However, there are several "less random" write patterns that are indeed efficient by leveraging write locality and write parallelism of flash devices. Among these patterns, two are particularly interesting from an algorithm designer's perspective:

- The first pattern is "random writes limited to a focused area", or in short, *focused writes*. A *focused area* is a

[1]FTL (acronym for Flash Translation Layer) is an internal software layer in flash devices to support normal block-device I/O functionalities.

Table 1: Write Response Time on Two SSDs (unit: ms, page size: 32KB) [4]

SSD Brand/Model	Sequential Write	Random Write		
		G	F	P
Mtron SATA7035-016	0.4	9	0.8	0.6
Samsung MCBQE32G5MPP	0.6	18	0.9	1.2

G: General Random Write
F: Focused Random Write within 8MB locality
P: Random Write with 4-Way Partition

small area of logical addresses. For instance, the space inside a pre-allocated file smaller than 4MB can be regarded as a focused area.

- The second pattern is *partitioned sequential writes*, or simply *partitioned writes*. For example, a write sequence "`1,50,2,51,3,52,`$\cdots$"[2] is an instance of partitioned write pattern. The sequence appears similar to random writes. In fact, it is a mixture of two sequential write sequences (*i.e.*, "`1,2,3,`$\cdots$" and "`50,51,52,` $\cdots$"), which correspond to two partitions of the address space.

Table 1 gives the response time on the two write patterns, together with sequential and random writes of two flash devices evaluated in [4]. In brief, these flash devices possess three *efficient* write patterns (*i.e.*, sequential, focused, and partitioned writes), in contrast to inefficient *general* random writes.

In this paper, we exploit these unique write behaviors of flash devices to optimize the performance of write-intensive DBMS applications, such as OnLine Transaction Processing (OLTP). Arising from small insert/delete/update requests, the writes in OLTP are often random and scattered. As a result, the (relatively) inefficient random writes can easily become a performance bottleneck on flash devices. By taking advantage of the focused write pattern, we propose to write pages temporarily to a small, pre-allocated storage space on the flash device, called *StableBuffer*. We then recognize and flush *efficient* write patterns of the buffer to achieve a better overall performance.

Our solution can be implemented as an add-on module of any existing DBMS buffer manager. It is optimized for flash devices that exhibit efficient focused write patterns. It is also transparent to the underlying flash technology which may often be proprietary. On the other hand, we note that StableBuffer is not proposed as a substitute to the log-based techniques such as [16] and [14], rather a complement that can work with them when modifications to the device driver or firmware are allowed.

There are several issues and challenges in the design of StableBuffer: (i) how to organize and manage the space inside the StableBuffer; (ii) how to recognize efficient write patterns in the StableBuffer at a low cost; and (iii) how to decide when and which write patterns to be flushed from the StableBuffer to their destinations on the flash device. We present the detailed design of StableBuffer and propose solutions to address these issues in this paper. To summarize, the contributions we make in this paper are as follows:

- We propose a StableBuffer solution for optimizing the performance of write-intensive DBMS applications running on flash devices. It does not need to modify the driver or firmware of flash devices; and it is orthogonal to other log-based optimization techniques.
- We present two write pattern recognition methods, namely on-demand and incremental recognition. The former improves the throughput of database applications, while the latter aims at a shorter response time by incrementally maintaining some auxiliary structures in the StableBuffer.
- We design two strategies for deciding when to flush pages inside the StableBuffer, namely passive and proactive page flushing. These strategies attempt to achieve different levels of tradeoffs between the throughput and response time.
- We conduct a performance evaluation of StableBuffer on several real flash devices. The results demonstrate that StableBuffer improves the response time and throughput of write operations by a factor of 1.5−12, in comparison with a direct write-through strategy, and reveal interesting observations on the performance improvements with different types of flash devices.

[2]For simplicity, we use a small integer to denote the logical address of a page.

The rest of this paper proceeds as follows. We present the design and implementation of StableBuffer in Sections 2 and 3, respectively. We evaluate StableBuffer and analyze the experimental results in Section 4. The related work is surveyed in Section 5. Finally, we conclude the paper in Section 6.

2. STABLEBUFFER

This section presents the proposed solution — *StableBuffer*. An overview is given in Section 2.1. Section 2.2 describes the data structures used to support StableBuffer. Finally, we detail the read/write/flush operations of StableBuffer in Section 2.3.

2.1 Overview

To take advantage of the *efficient* write patterns of flash devices (*i.e.*, sequential, focused, and partitioned write patterns), one idea is to allocate a *write buffer* in main memory. It buffers (random) writes temporarily and subsequently outputs them to the flash device following efficient patterns. That is, the write buffer serves as a "hub" to collect writes and reorder their sequence for performance optimization. However, this approach has two limitations. First, it needs to take up a considerable amount of buffer space that would be otherwise allocated for the DBMS. This would obviously sacrifice the read performance and complicate the buffer management. Second, the main memory is volatile. Thus, with a *force* buffer policy, writes that are buffered have to be *forced* to the stable storage when a transaction is committed. On the other hand, with a *no-force* buffer policy, some additional mechanism (e.g., *logging*) has to be provided to achieve write durability.

Fortunately, as reported in [4], random writes to a focused area (sized 4MB∼16MB) are almost as fast as sequential writes for most flash devices. Thus, to address the above issues, we propose *StableBuffer* to buffer writes on some pre-allocated space on the (stable) flash device. To achieve efficient write performance, the size of the StableBuffer is limited to that of a focused area. As shown in Figure 1, the existing DBMS architecture is not changed except that after

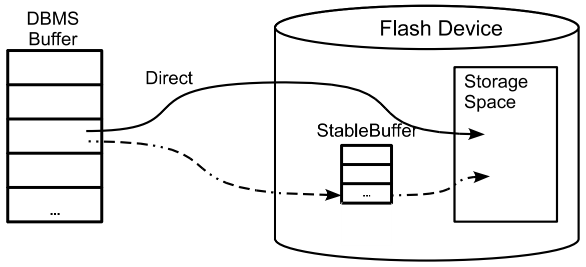


Figure 1: Overview of StableBuffer

the writes are output by the DBMS buffer, they will be temporarily stored in the StableBuffer, instead of directly writing to their actual locations. When the StableBuffer collects enough writes (*e.g.*, when it becomes full), we identify some instances of *efficient* write patterns and then *flush* them to their actual locations. To do so, each page of the identified instance will be read out of the StableBuffer and written to its actual destination. Since all writes involved follow an *efficient* write pattern, they are expected to perform well in the overall performance.

To show the potential benefit of employing the StableBuffer, let us consider a single page write on an Mtron device. To flush a page to its destination through the StableBuffer, as discussed, there are two writes following *efficient* write patterns and one random read. Based on the measurements of Table 1, the worst cost of two efficient-pattern writes is estimated as $0.8 + 0.8 = 1.6$ ms. The random read generally costs less than a sequential write (*i.e.*, < 0.4 ms). Thus, the total cost is estimated as $1.6 + 0.4 = 2.0$ ms. On the other hand, if we directly flush the page to the flash device, it is very likely to be a *general* random write, whose cost is as high as 9 ms. Clearly, although the StableBuffer writes the same page twice, by following *efficient* write patterns, its overall performance is improved 4.5X against the direct write-through strategy. We remark that the StableBuffer is a *logical* area and the intensive writes to it would not cause an endurance issue with the support of wear-leveling techniques [7]. Moreover, the long write endurance of flash devices today makes the doubled writes of StableBuffer an acceptable compromise. For example, the lifetime of some enterprise-class flash devices already reaches 140 years at a rate of 50GB writes per day [2]; even with doubled writes, the lifetime can still reach 70 years.

Discussions. In DBMS, it is possible that a transaction updates a series of sequential pages. A wise DBMS buffer manager should avoid to write such pages through the StableBuffer, since they already form an efficient write patten. It would be the responsibility of buffer manager to identify these transactions and decide how to efficiently respond their read/write requests. In the literature, research work like DBMIN [10] has demonstrated that it is possible to capture such transactions within the DBMS buffer manager. Therefore, collaborating with advanced DBMS buffer managers, the possible performance degradation on processing sequential writes through the StableBuffer could be avoided. In practice, transactions that update a large number of sequential pages, such as loading data to tables and transforming data columns, are not likely to mix with OLTP processing. In this paper, to simplify the discussion, we assume that the DBMS buffer manager would efficiently process the transactions that update sequential pages.

2.2 Data Structures

The StableBuffer is a small pre-allocated area on the flash device. It could be implemented on some space reserved inside the DBMS storage, or simply as a pre-allocated temporary file if the DBMS storage is organized based on files. We format the StableBuffer into a number of slots, each of which can accommodate a disk page. For example, given a 4MB StableBuffer and a page size of 4KB, 1,024 slots are allocated in the StableBuffer.

To support the operations of the StableBuffer, several main-memory data structures are needed (Figure 2):

- **StableBuffer Translation Table**. It is an in-memory table that maintains the mappings between the slot number of the StableBuffer space and the page destination address. For example, if a page with destination address `0x123456ab` is stored in the `32`nd slot of StableBuffer, a mapping entry <`0x123456ab`, `32`> will be maintained in the StableBuffer Translation Table. To support efficient page lookup based on destination address, this table is implemented as a hash table whose key is the destination address.
- **Bitmap for Free Slots**. To keep the list of free slots in the StableBuffer, a bitmap is used. Each bit in the bitmap represents the status of a StableBuffer slot: "1" means that the corresponding slot is free; "0" means that the slot is occupied. Furthermore, to facilitate the lookup of free slots, a *free-slot pointer* is maintained to point to the next free slot, if available.

These data structures are maintained in main memory. It is noted that their size can be insignificant. Consider a 1,024-slot StableBuffer. Suppose each mapping entry takes 8 bytes (4 bytes for each element), the size of the StableBuffer Translation Table is *# of slots* $\times$ *mapping entry size* = 8KB and the size of the bitmap is *# of slots* / 8 bytes = 128 bytes only.

Recovery. Since the main memory is volatile, some extra metadata (*i.e.*, the destination address and the timestamp) is kept with each page stored in the StableBuffer, so that the data can be recovered in case of a system crash. Specifically, before writing a page to the StableBuffer, the actual destination address and the timestamp will be embedded in the page header. After a system crash, we use these information to rebuild the StableBuffer Translation Table. In detail, we scan the StableBuffer space slot by slot. For each page in slot O whose destination address is D, we compare its timestamp ts_O to the latest update time ts_D of the current page at destination D. If $ts_O \leq ts_D$, the page must have been flushed to its actual destination. Hence, we mark the O-th slot of the StableBuffer as free. Otherwise, $ts_O < ts_D$, which indicates that the page is not yet written to its destination before the crash. Thus, we mark the slot as occupied and insert an entry, $<D, O>$, to the StableBuffer Translation Table.

2.3 Page Operations

As illustrated in Figure 2, the StableBuffer Manager consists of three main components: *Reader*, *Writer*, and *Flush Manager*. They support operations in the granularity of

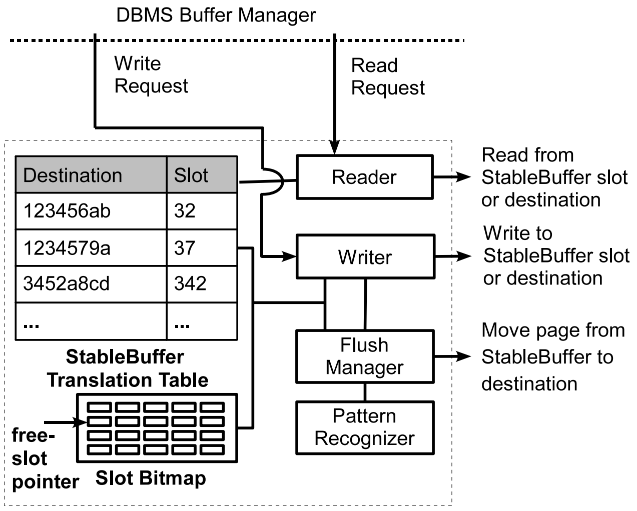


Figure 2: Architecture of StableBuffer Manager

```
Input   : Read Request(Destination $D$)
Output: Page requested
look up StableBuffer Translation Table (SBTT) for destination $D$ ;
if an entry <$D$, $O$> is found then
    read page $P$ from slot $O$ and return ;
else
    read page $P$ at $D$ and return;
```

Algorithm 1: Read Through StableBuffer

pages, *i.e.*, *page read*, *page write*, and *page flush*, respectively. Upon receiving a read request for the page with destination D, the Reader looks up the StableBuffer Translation Table by hashing (see Algorithm 1). If there is an entry $<D, O>$ found, we read in the page stored in slot O of the StableBuffer. Otherwise, we issue a normal read request to the flash device. Note that in either case, only one flash I/O is incurred. The only overhead is the table lookup. Thanks to hashing, such overhead would be negligible.

For page writes, upon receiving a dirty page with destination D evicted by the DBMS buffer manager, the Writer first adds some metadata to the page header as discussed above. Then, it looks up the StableBuffer Translation Table. If the page destination D is found in some slot O of the StableBuffer, we update the page. Otherwise, the Writer attempts to locate a free slot in the StableBuffer. This is done by checking the *free-slot pointer*. In case the *free-slot pointer* is *null* (*i.e.*, the StableBuffer is full), a *flushing* procedure (to be detailed in Section 3.2) will be invoked to free some slot(s). After a free slot O' is located, the page is written to slot O', and a new entry, $<D, O'>$, is inserted to the StableBuffer Translation Table. Finally, the free-slot pointer is advanced to the next free slot, if available, in the bitmap; otherwise it is set to *null*. The write operation is summarized in Algorithm 2. Again, only one flash I/O is incurred in writing a new page, but some additional I/Os might be needed to free space in the StableBuffer.

Finally, the *Flush Manager* is responsible for flushing the pages temporarily stored in the StableBuffer to their actual destinations following efficient write patterns. Two crucial

```
Input: Write Request(Page $P$, Destination $D$)
embed $D$ & current time into page $P$ ;
look up StableBuffer Translation Table (SBTT) for destination $D$ ;
if an entry <$D$, $O$> is found then
    write page $P$ to slot $O$ ;
    respond to the requester and return ;
else
    if free-slot pointer == null then
        wait the flush manager to free slot(s) ;
        set free-slot pointer to a free slot ;
    $O'$ := no. of slot pointed by free-slot pointer ;
    write page $P$ to slot $O'$ ;
    respond to the requester ;
    insert an entry <$D, O'$> into SBTT ;
    mark $O'$ as occupied in the slot bitmap ;
    count := 1 ;
    while the slot pointed by free-slot pointer is not free do
        if count == bitmap_size then
            free-slot pointer := null ;
        free-slot pointer := (free-slot pointer + 1) mod bitmap_size ;
        count := count + 1 ;
    return ;
```

Algorithm 2: Write Through StableBuffer

issues here are how to recognize efficient write patterns and when to flush pages. We will present in the next section several solutions with different performance tradeoffs between the throughput and response time.

3. STABLEBUFFER MANAGEMENT

Having presented the design of the StableBuffer, we now focus on its page management. The problem can be described in terms of when and how to (i) detect and (ii) flush *efficient* write patterns. In Section 3.1, we address the first issue with on-demand and incremental methods for write pattern recognition. In Section 3.2, we discuss the second issue with passive and proactive strategies for page flushing.

3.1 On-Demand v.s. Incremental Write Pattern Recognition

We present two alternative methods to recognize *efficient* write patterns. In a nutshell, an *on-demand* recognition method determines the *efficient* write patterns in the StableBuffer upon a page flush request, whereas an *incremental* recognition method maintains auxiliary information about write patterns with each write operation to the flash device. Upon receiving a page flush request, the auxiliary information is used to determine the efficient write patterns.

On-Demand Method. The on-demand method can be easily implemented with a sorted scan on the StableBuffer Translation Table. We provide its details for different write patterns as follows:

- **Sequential Writes.** Sequential writes can be recognized by sorting and scanning the destination addresses maintained in the StableBuffer Translation Table. The write operations that form the longest continuous address sequence are identified.

- **Partitioned Writes.** Partitioned writes can also be recognized by a scan on sorted destination addresses. Each continuous address sequence forms a partition. The partitions of the same size are grouped. The group with the most number of write operations is then identified.

- **Focused Writes.** Focused writes can be detected by scanning the sorted page destination addresses with a sliding window. The size of the sliding window is identical to the size of the focused area of the flash device. When scanning, we maintain (i) the start address of the sliding window and (ii) the number of destination addresses that reside inside the sliding window. Finally, the start address with the densest destination addresses is identified.

Recall that the StableBuffer Translation Table is implemented with a hash table. Hence, the scans mentioned above can be implemented in $O(n\log n)$ time, where n is the number of pages in the StableBuffer.

Incremental Method. The objective of the incremental method is to efficiently amortize the computation of determining efficient write patterns over voluminous write operations. It maintains some auxiliary data structures for incrementally recognizing candidate instances of efficient write patterns. Whenever a page is inserted into/deleted from the StableBuffer, the auxiliary data structures will be updated. With the help of these auxiliary data structures, the efficient write patterns can be found immediately at any time. In detail, each write pattern is incrementally maintained as follows:

- **Sequential Writes.** We maintain a set $SL = \{S_1, S_2, \cdots, S_i, \cdots\}$ of continuous address ranges, where $S_i = (addr_{min}, addr_{max})$ is defined as a range representing the addresses from $addr_{min}$ to $addr_{max}$. The set SL is sorted in ascending order of $addr_{min}$ of each S_i. Whenever we insert (or delete) an address $addr$, we invoke a binary search on SL for the closest S_i to $addr$ (or the S_i covering $addr$). For an insertion, we may merge $addr$ into the closest S_i if a longer range can be formed; otherwise a new address range with $addr$ only will be created. For a deletion, the S_i covering $addr$ will be split into two. It is easy to see that each S_i is a candidate instance of sequential write pattern.

- **Partitioned Writes.** To incrementally recognize candidate instances of partitioned write pattern, we maintain an extra data structure $\{P_1, P_2, \cdots, P_l, \cdots\}$ based on the set SL (used for sequential writes), where P_l keeps track of all S_i's in SL that have a size of l. For instance, P_2 contains a set of pointers pointing to the S_i's with a size of 2. Each P_l represents a candidate instance of partitioned write pattern. Upon a page insertion/deletion, after updating the SL, we continue to update the affected P_l's.

- **Focused Writes.** For the focused write pattern, we maintain a set $FL = \{F_1, F_2, \cdots, F_i, \cdots\}$ of focused write clusters, where $F_i = (addr_{min}, addr_{max}, set_{addr})$ is defined as a set of addresses set_{addr} falling in the range $[addr_{min}, addr_{max}]$. The distance between $addr_{min}$ and $addr_{max}$ is less than the size of the focused area of the flash device. Thus, each F_i is a candidate instance of focused write pattern. F_i's are sorted in ascending order of their $addr_{min}$ addresses. When inserting a new address $addr$, we invoke a binary search on FL to find the closest cluster F_i. We then merge $addr$ into the found F_i if possible, or otherwise create a new F_i with $addr$. In contrast, a deletion will remove the address from the corresponding cluster. Note that here the focused write clusters are recognized in a greedy manner, which minimizes the computational overhead but could sacrifice the quality of clustering results. Nevertheless, as we will see in the performance evaluation, this compromise works well in practice for balancing the throughput and response time.

The time complexity of updating the data structures for each page insertion/deletion is bounded by $O(\log n)$, where n is the number of pages stored in the StableBuffer. The space complexity of each data structure is $O(n)$. Since the size of the StableBuffer is usually small, the space requirement would be acceptable.

Discussions. Compared with the on-demand method, the incremental method identifies an efficient write pattern more quickly upon a page flush request. Thus, the incremental method may achieve a shorter write response time. However, the incremental method introduces a certain overhead into each write operation. The overall computation of the incremental method is often higher than that of the on-demand method. Moreover, the incremental method recognizes instances of focused write pattern with local information only, and it may not return results as good as the on-demand method that has global information available at the time of the page flush request. As such, we expect that the throughput of the incremental method would be lower than that of the on-demand method.

3.2 Passive v.s. Proactive Page Flushing

The second issue of page management is to decide when and how to flush pages from the StableBuffer to their actual destinations. A straightforward method is to flush pages of the best write pattern instance when there is no available space in the StableBuffer for a write operation. We call this a *passive* method. An alternative method is to *proactively* flush write pattern instances when it is *qualified* to maintain some free space during any write operations.

Passive Method. Upon being triggered, the passive method selects an instance of some write pattern from a set of candidates and then flush the pages contained in the selected instance. With the on-demand write pattern recognition method, the candidate instances are formed on-the-fly by scanning the StableBuffer Translation Table; with the incremental write pattern recognition method, the candidate instances have been incrementally maintained and are immediately available for the selection. Denote each candidate by WP_i^x, where x represents the type of the write pattern (*i.e.*, sequential, partitioned, or focused writes). Suppose that the average time for writing a page of write pattern x is T_x, which can be observed from the benchmark results such as uFLIP [4]. We select to flush the instance with the largest value of $\frac{|WP_i^x|}{T_x}$, which is the longest instance that is expected to be written fastest. Intuitively, this selection criterion strikes a balance between the amount of space reclaimed and the page flushing speed.

Proactive Method. In the proactive method, the flush manager keeps running *in background* to detect good enough instances of efficient write patterns. It naturally requires the pattern recognizer working in an incremental fashion. The flush manager is triggered whenever there is a change to any write pattern instance that is dynamically maintained. We check whether the affected write pattern instance is *qualified* for flushing. In particular, the qualification of an instance is determined based on its size and the type of write pattern. Specifically, for each type of write pattern x, a flushing threshold θ_x is used and only the instances with a size larger than θ_x are qualified for flushing. In case more than one affected write pattern instances become qualified at the same time, we select the best one according to the selection criterion used in the passive method.

Deciding θ_x with Benchmark Results. We start our discussion with the sequential write pattern. Consider a sequential write instance consisting of m pages each sized ℓ. Since all pages in the instance are sequentially addressed, we estimate the cost of writing these m pages by the cost of writing a large page sized $m \cdot \ell$. For example, for a sequential write instance consisting of 8 pages each sized 4KB, the cost of writing them is estimated by the cost of writing a single page of size 8×4KB=32KB. With a conservative estimation, we may think that writing pages sized $m \cdot \ell$ is a random operation. When m is small, the time of writing a random page sized $m \cdot \ell$ may vary in a wide range, which is influenced by possible flash erasure operations. When m is large enough, the write time becomes more stable since the erasure operations are more uniformly distributed. This observation motivates us to decide the flushing threshold as follows. We consider the random write performance and note down the minimal page granularity ℓ_{min} after which the worst write performance is about to become stable. Then the flushing threshold θ_{seq} is calculated as ℓ_{min}/ℓ. This makes the page flushing performance more predictable.

Next, we derive the flushing thresholds for other write patterns from θ_{seq}. Let T_{seq}, T_{par}, and T_{focus} be the average time to write a page in a sequential write pattern, in a partitioned write pattern, and in a focused write pattern, respectively. To make the instances of other write patterns as efficient as sequential write instances, we set their thresholds in direct proportion to that of the sequential write. In detail, we set the threshold for flushing partitioned writes as $\theta_{par} = \theta_{seq} \cdot T_{par}/T_{seq}$, and set the threshold for focused writes as $\theta_{focus} = \theta_{seq} \cdot T_{focus}/T_{seq}$.

As an example, we show how to decide the flushing thresholds for an Mtron SSD based on the uFLIP benchmark results [4]. First, we take the benchmark test results for random writes with different page granularities.[3] We observe that when the page granularity is larger than 32KB, the write performance becomes stable. Hence, ℓ_{min} = 32KB. Given a page size of 4KB, we have θ_{seq} = 32KB / 4KB = 8. As reported in [4], the time of a partitioned write is about 1.5 times of that of a sequential write, and the time of a focused write is about twice of that of a sequential write. Thus, $\theta_{par} = 1.5\theta_{seq} = 12$, $\theta_{focus} = 2\theta_{seq} = 16$.

[3]The results are available online at http://uflip.inria.fr/~uFLIP/results/index.php?device=mtron16&mb=GranularityRW& show=results.

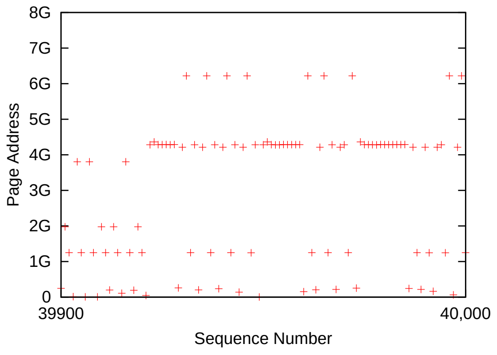


Figure 3: A Segment of Traced Write Requests

4. PERFORMANCE EVALUATION

4.1 Experiment Setup

We have developed a simulator to evaluate the performance of StableBuffer. The simulator can access a trace file and process read/write requests archived in the file. In the simulator, the StableBuffer is implemented as a pre-allocated temporary file which is accessible through a page-oriented interface. We implemented three combinations of pattern preconization and page flushing methods, *i.e.*, *ondemand+passive*, *incremental+passive* and *incremental+proactive*. For comparison, we also implemented the direct write-through strategy (denoted as *direct*) which writes pages directly to their actual destinations.

We conducted performance evaluation on a Windows desktop PC with an Intel-Core 2 Quad Q6600 CPU. The evaluation replayed a write trace on three different flash devices — an SSD (Mtron MSD-SATA-3525, 16GB), a USB flash memory (Toshiba, 8GB), and an SD card (Kingston SDHC, Class 4, 8GB). The write trace was obtained by running the TPC-C benchmark [1] on PostgreSQL 8.4 with default settings. The TPC-C benchmark simulates online transaction processing of record insertions, deletions and updates in an enterprise database. We ran the benchmark with 20 warehouses and 20 clients for over 30 minutes, and collected a trace of 122,150 write I/O requests. The page destination addresses of the write requests range from 0 to 7GB. Figure 3 shows a small segment of these write requests. As can be observed, the writes exhibit a random distribution in a short time period but are clustered in a relatively long time period.

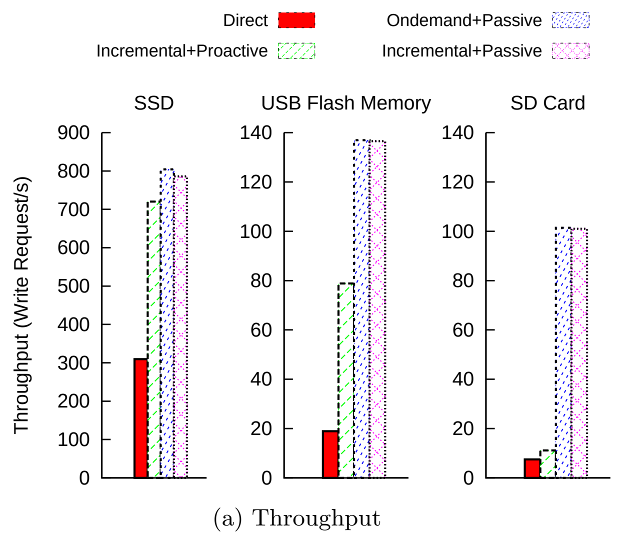
For the system setting, the default page size is 4KB, and the StableBuffer size on each flash device is set based on their respective benchmark results.[4] On the SSD, the StableBuffer size is set at 4MB (*i.e.*, 1,024 slots). On the USB flash memory and the SD card, the StableBuffer size is set at 2MB (*i.e.*, 512 slots).

4.2 Performance Results

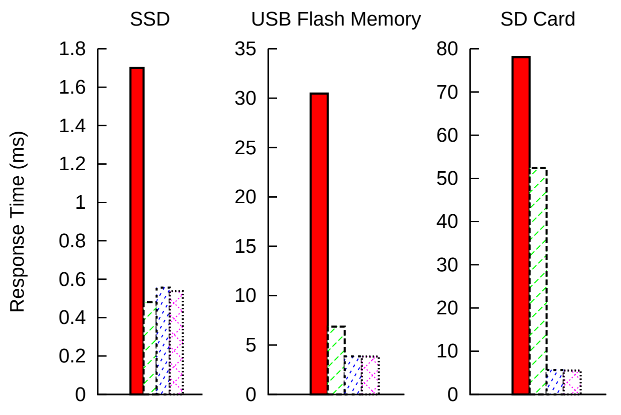
4.2.1 Performance Comparison With Direct Writes

Figure 4 shows the evaluation results of throughput and response time on the three flash devices. The throughput

[4]We used uFLIP [4] as the benchmark tool.



(a) Throughput



(b) Response Time

Figure 4: Throughput & Response Time under Default Setting

is calculated as dividing the number of completed write requests by the total execution time. For the response time, we collect the response time of each write request and report the average value. The response time includes both the I/O time and the CPU time.

We can see that, compared with the direct write-through strategy, the write performance is significantly improved by using the StableBuffer in terms of both performance metrics. For the SSD, the throughput is increased by 132% (employing incremental+proactive) to 159% (employing ondemand+passive), and the response time is reduced by 67% (employing ondemand+passive) to 71% (employing incremental+proactive). For the USB flash memory, the throughput is increased by 317% (incremental+proactive) to 624% (ondemand+passive), and the response time is reduced by 77% (incremental+proactive) to 87% (incremental+passive). For the SD card, the throughput is increased by 49% (incremental+proactive) to 1,256% (ondemand+passive), and the response time is reduced by 33% (incremental+proactive) to 97% (incremental+passive). Overall, the incremental+passive method achieves the best balance between the throughput and response time.

It is seen from Figure 4 that the relative performance of the incremental+proactive method for the USB flash memory and SD card is not as good as that for the SSD. To investigate the reason, we designed a test for measuring the parallel write performance on these devices. The test accepts write streams and outputs them to the flash device in two different modes: (1) *serial*: processing the write streams one by one in a single thread; (2) *parallel*: using two threads to process the write streams in parallel, with each write stream processed by one thread. The serial mode simulates the page writing in the incremental+passive and ondemand+passive methods, while the parallel mode simulates the page writing in the incremental+proactive method where the writing of StableBuffer and page flushing might be paralleled. We feed the test with focused write streams and plot the results in Figure 5. As can be seen, for the SSD, the parallel mode works almost twice as fast as the serial mode. As a result, the incremental+proactive method is benefited from this parallel ability of SSD to make up the degradation caused by incremental write pattern recognition. For the USB flash memory, the parallel mode works almost same as the serial mode. Hence, the incremental+proactive method is not benefited from parallel I/Os and suffers from the relatively poor pattern recognition results. For the SD card, the parallel mode is much worse than the serial mode. Allowing parallel I/Os will deteriorate the I/O efficiency and cancel most benefits brought by the StableBuffer, making the throughput of the incremental+proactive method close to the direct-write strategy. Regarding the response time, the incremental+proactive method performs even worse than the passive methods for the USB flash memory and SD card. This is mainly because of the lengthened waiting time of write operations due to a low throughput.

We also observe in Figure 4 that the two passive methods perform very close to each other on the USB flash memory and SD card, which is different from the case on the SSD. To gain more insights, we look into the time spent in I/O and CPU for the two passive methods. As shown in in Figure 6, while 24.3% and 31.5% of time are spent in CPU for the ondemand+passive and incremental+passive methods on the SSD, only 5.1% and 5.3% of time are spent in CPU on the USB flash memory and only 4.2% and 5.5% of time on the SD card. This indicates that the write response time on the USB flash memory and the SD card is dominated by the I/O cost. Consequently, the differences in CPU time between the ondemand+passive and incremental+passive methods do not differentiate the overall performance much.

We make some comments on the performance differences between the two passive methods for the SSD (see the leftmost subfigure of Figure 6). The ondemand+passive method spends less time in CPU. The reason is two-fold. First, the StableBuffer is not large, so is the StableBuffer Translation Table. Therefore, the write pattern recognition task can be performed efficiently. Second, the incremental+passive method spends more time on computation than the ondemand+ passive method, as it needs to incrementally recognize *efficient* write patterns (discussed in Section 3.1). By sacrificing some CPU time and throughput, the incremental+passive method achieves a shorter write response time than the ondemand+passive method (see the leftmost subfigure of Figure 4(b)).

4.2.2 *The Impact of StableBuffer Size*

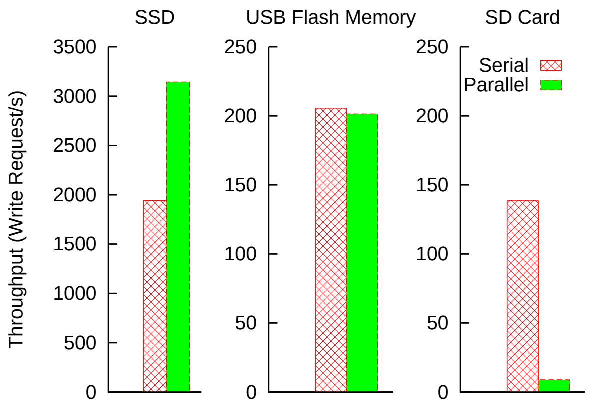
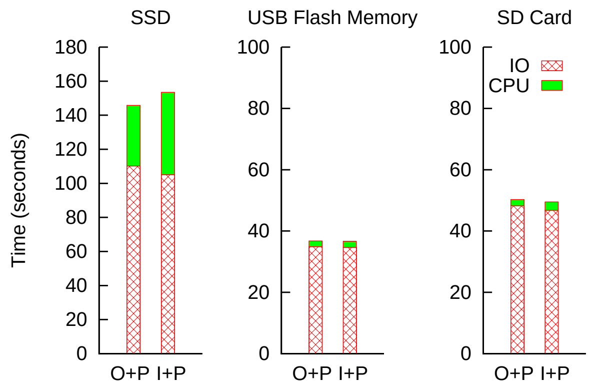


Figure 5: Parallel Write Test Results



O+P: Ondemand+Passive; *I+P*: Incremental+Passive

Figure 6: IO v.s. CPU Time under Default Setting

Next, we investigate the impact of the StableBuffer size on the overall performance. The results are plotted in Figure 7. We vary the StableBuffer size from 25% of the default size (1MB for the SSD and 512KB for the USB flash memory and SD card) to four times of the default size (16MB for the SSD and 8MB for the USB flash memory and SD card).

First, we observe that when the Stable Buffer size is at either extreme, the performance of the StableBuffer methods degrades (*i.e.*, the throughput drops and the response time increases). This can be explained as follows. When the StableBuffer size is too small, the quality of write patterns recognized is not good enough. Hence, optimizing write patterns to improve write performance becomes less effective. On the other hand, when the StableBuffer size is too large, writing to the StableBuffer itself may no longer be focused writes. Thus, it takes more time to write a page to the StableBuffer, which cancels out the time saved in optimizing write patterns from the StableBuffer to the flash device.

Second, as with Figure 4, the incremental+proactive method does not perform well on the USB flash memory and SD card. Regarding the SSD, its performance remains more stable after the StableBuffer size exceeds the default one. When we select a size larger than the default one, as the flush manager in the incremental+proactive method proactively recognizes write pattern instances and flushes them out, the ratio of the utilized space in the StableBuffer remains low. In this experiment, we observed that when the StableBuffer size is 16MB, the utilization ratio is only 27% (about 4.3MB) on average. Since there are a lot of free slots in the StableBuffer, the time for locating a free slot is fast, and the writes to the StableBuffer are very likely to be sequentialized. As a result, the overall system performance remains good.

We also remark that the results above justify the setting of StableBuffer size to that of the focused area. For a general flash disk device, the latter parameter can be obtained either through microbenchmarks such as uFLIP[4] or from the flash disk manufacturers in the future.

5. RELATED WORK

Database management on flash-based storage media has attracted increasing research attention in recent years. Early work focused on assembling flash chips to simulate traditional hard disks [6][11][13] and enhancing the write endurance [5][7]. Recent research has investigated opportunities to optimize DBMS performance by exploiting the unique characteristics of flash devices. In view of the asymmetric read/write speed and the erase-before-write limitation, Wu *et al.* [25] proposed a log-based indexing scheme for flash memory. To improve this scheme for read-intensive workloads, Nath and Kansal [20] developed an adaptive indexing method that is aware of the workload and storage device. Lee and Moon [16] studied the erase-before-write limitation of DBMS storage and designed a novel in-page logging (IPL) technique. More recently, Kim *et al.* [14] proposed a page-differential logging technique to overcome the page read overhead of IPL. The basic idea is to write only the difference between the original page in flash memory and the up-to-date page in main memory. Stoica *et al.* [23] proposed a flash-aware data layout called "append and pack", which enhances transactional DBMS performance by eliminating random writes. Whereas these techniques of optimizing write performance require modifying the driver or firmware of flash devices, our proposed StableBuffer solution does not have this requirement and works well for commodity flash devices.

Bouganim *et al.* [4] proposed uFLIP as a microbenchmark to help researchers systematically understand the I/O patterns of flash devices. Through benchmarking on various flash storage devices, they discovered several good page-writing practices for algorithm designers. Chen *et al.* [8] also conducted intensive experiments and measurements on different types of state-of-the-art SSDs. They observed several unanticipated performance issues and dynamics of SSDs (e.g., fragmentation could cause dramatic performance degradation). Our work has been inspired by these interesting flash performance reports.

For flash-based database applications, Lee *et al.* investigated how the performance of standard DBMS algorithms is affected when a magnetic disk is replaced by an SSD [17]. By exploiting fast random reads and sequential writes on SSDs, efficient DBMS scanning and joining algorithms have been reexamined in [19] and [24]. Observing that small sequential writes are well supported by flash devices, Chen [9] suggested improving the synchronous logging performance by using USB devices. In addition, novel index structures have been proposed to support efficient query processing on flash devices [3][18]. New buffer management policies (e.g., [12][21][22]) have also been proposed for flash devices. To

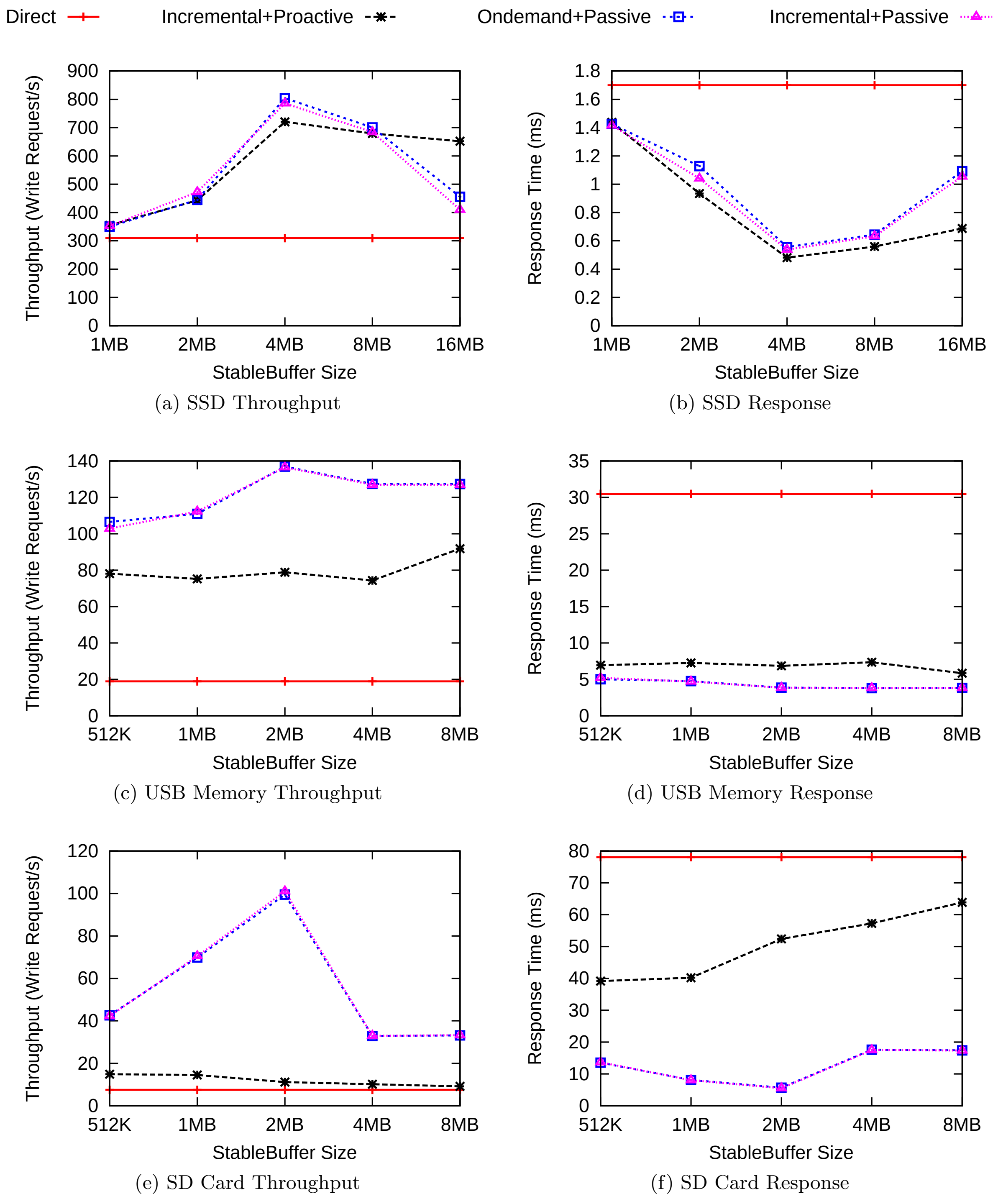


Figure 7: Throughput & Response Time v.s. StableBuffer Size

address the read-write asymmetry, they divide the buffer pool into a clean pool and a dirty pool, and give priority to choosing clean pages as victims over dirty pages. These policies are designed for main-memory buffers. In contrast, our proposed StableBuffer resides on the flash device so that it achieves write durability effortlessly. It works efficiently by exploiting the *efficient* write patterns that were recently discovered in [4][8].

6. CONCLUSIONS

In this paper, we have investigated how to overcome the poor write performance of flash devices for DBMS applications. Inspired by recent findings on flash I/O performance, we proposed a StableBuffer solution for flash devices that exhibit efficient focused write patterns. This solution does not require modifying the driver or firmware of the flash device. The basic idea is to archive page writes into the StableBuffer and then flush them to the actual destinations following efficient write patterns. The detailed design and implementation of the StableBuffer have been discussed. We have also presented several write-pattern recognition and flushing methods for managing the pages stored in the StableBuffer. The performance evaluation based on a TPC-C benchmark trace show that, by employing the StableBuffer, both of the throughput and response time are improved by a factor of 1.5−12, in comparison with a direct write-through strategy. In particular, while the passive and proactive flushing methods achieve different levels of tradeoffs between the throughput and response time for the SSD that supports parallel I/Os, the passive flushing methods are preferred for the USB flash memory and SD card where parallel I/Os are not well supported.

As for future work, we are going to optimize the write pattern recognition and page flushing algorithms. In particular, we are interested to investigate how to further improve the online algorithm for clustering focused writes. We also plan to theoretically model the page flushing problem in order to study its optimal solution.

Acknowledgement

This work was supported by the Research Grants Council of Hong Kong (Grants HKBU210808 and HKBU211510) and Natural Science Foundation of China (Grant 60833005).

7. REFERENCES

[1] TPC Benchmark C: Standard Specification. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[2] Solid State Drive MSD-SATA3035: 3.5-inch Product Specification. Mtron Co. Ltd., Jan. 2008.

[3] D. Agrawal, D. Ganesan, R. Sitaraman, and Y. Diao. Lazy-adaptive tree: An optimized index structure for flash devices. In *VLDB '09*, 2009.

[4] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding flash io patterns. In *CIDR*, 2009.

[5] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07*, pages 1126–1130, 2007.

[6] L.-P. Chang, T.-W. Kuo, and S. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[7] Y. Chang, J. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC '07*, pages 212–217, 2007.

[8] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*, pages 181–192, 2009.

[9] S. Chen. FlashLogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD '09*, pages 73–86, 2009.

[10] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB '85*, pages 127–141. VLDB Endowment, 1985.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[12] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST '08*, 2008.

[13] H. Kim and S. Lee. A new flash memory management for flash storage system. In *COMPSAC '99*, 1999.

[14] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-Differential Logging: An efficient and dbms-independent approach for storing data into flash memory. In *SIGMOD '10*, 2010.

[15] S. Lawson. Cloud computing could be a boon for flash storage. http://www.businessweek.com/technology/content/aug2009/tc20090824_219491.htm, 2009.

[16] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07*, pages 55–66, 2007.

[17] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD*, 2008.

[18] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. In *VLDB '10*, 2010.

[19] Y. Li, S. T. On, J. Xu, B. Choi, and H. Hu. DigestJoin: Exploiting fast random reads for flash-based joins. In *MDM '09*, pages 152–161, 2009.

[20] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for nand flash. Technical Report MSR-TR-2006-168, Microsoft Research, 2006.

[21] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. FD-Buffer: A buffer manager for databases on flash disks. In *CIKM '10*, 2010.

[22] Y. Ou, T. Haerder, and P. Jin. CFDC—A flash-aware replacement policy for database buffer management. In *DaMoN '09*, 2009.

[23] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMoN '09*, 2009.

[24] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD '09*, pages 59–72, 2009.

[25] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Trans. on Embedded Computing Sys.*, 6(3):19, 2007.