BVSSD: Build Built-in Versioning Flash-based Solid State Drives

Ping Huang, Ke Zhou✉, Hua Wang and ChunHua Li

School of Computer Science & Technology, Huazhong University of Science & Technology
Key Laboratory of Data Storage Systems, Ministry of Education of China
Wuhan National Lab for Optoelectronics
pinghp.hust@gmail.com,{k.zhou,li.chunhua,hwang}@hust.edu.cn

Abstract

Time-traveling ability, which enables storage state to be reverted to any previous timepoints, is a highly desirable functionality in modern storage systems to ensure storage continuity. Continuous Data Protection (CDP) is a typical time-traveling implementation mechanism. CDP can guard well against software bugs, unintentional errors, malicious attacks, all of which are often beyond the capabilities of traditional periodical backup schemes. Broadly speaking, CDP can be implemented in two different ways, i.e., either integrate it seamlessly to the target file systems or more generally make it sit at the device block level. However, the state-of-the-art CDP implementations suffer from various limitations, e.g., huge implementation complexity, non-trivial performance interference. In this study, we introduce BVSSD, a new block level versioning system specifically designed for the emerging flash-based SSD. BVSSD realizes CDP functionality through positively and usefully exploiting the inherent idiosyncrasies of flash, which is the well-known "no-overwritten" property. Specifically, BVSSD simply keeps track of the SSD FTL metadata changes, which essentially represent the dynamics of the SSD storage state, and restores them to past timepoints to perform recoveries. Compared with existing block-level CDP schemes, BVSSD is much more light-weight, less performance-interfering, easier to realize, and more importantly, it requires no intrusive modifications to the upper file systems and applications. Our trace-driven simulation results with a number of different realistic enterprise-scale workload traces have shown that BVSSD only incurs marginal performance overheads, somewhere between 3% and 8% performance degradation, while with minimum additional RAM requirement, which is an acceptable price for the high reliability that BVSSD can provide. Furthermore, given most of the typcial SSDs deployment scenarios and their ever-increasing capacity trend, BVSSD is realistically poised to be feasible to be deployed in actual situations.

SYSTOR'12 June 4–6, 2012, Haifa, Israel.

Categories and Subject Descriptors D.4.2 [*OPERATING SYSTEMS*]: Storage Management-Secondary storage; B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids

General Terms Design, Experimentation, Verification, Performance

Keywords Continuous Data Protection, Flash Memory, Solid-state Drives, Storage Recovery

1. Introduction

Data durability and system resiliency are of paramount importance to data owners for various reasons. Traditionally, companies and corporations have been employing a variety of data protection schemes, e.g. periodical backups, snapshots [11] and Continuous Data Protection (CDP) [23], to ensure accessibility to historical storage versions for potential recovery purposes. While backup and snapshot techniques preserve a fixed number of versions (which correspond to the timepoints at which the backups and snapshots were taken) of past storage image, CDP can provide theoretically unlimited number of versions to which the storage state could be reverted. The commonly deployed scenario is a combination of backup or snapshot and CDP, which performs backups or snapshots at pre-configured regular intervals(e.g. once a day, at midnight) as full-copy storage image and provides CDP functionality between two consecutive intervals, as incremental changes. By doing this way, users can access older full-copy data versions as well as every single incremental change, thus it is possible to get the exact state corresponding to any timepoint by applying the incremental changes to those full-copy versions. *Continuous recovery* is particularly important for certain critical applications requiring business continuity or compliance with regulations and laws [29].

Broadly speaking, CDP can be implemented in two different schemes, i.e., at file system layer and at block level,

named as *VersioningFS* and *VersioningDisk*, respectively. *VersioningFS* [17, 25, 29, 34, 37] achieves CDP functionality through monitoring and logging changes occurring to the file system to distinct locations by trapping related system calls. Typically, *VersioningFS* is file system-specific and tightly coupled with the target file system. Rollback operations are accomplished by traversing the log history and retrieving the log entries which correspond to the requested previous point-in-time image, and then overwriting the disk with the corresponding data content. One advantage of *VersioningFS* is that it is able to provide the flexibility to protect entities of different granularities, e.g., a single file, a directory or the entire file system. However, developing *VersioningFS* is a huge and no easy task, since it requires the designers to be quite familiar with the targeted file systems. For example, Ext3cow[29] is geared toward the linux popular ext3 file system and its implementation necessitates implementers' excellent understanding of the ext3 file system internals. Additionally, being file system-specific, *VersioningFS* lacks portability which is one of the key factors which hinder its wide deployment. *VersioningDisk* [13, 23, 40, 43] is file system independent and implemented underneath the file systems. It captures the I/O activities at the block level and can only rollback at volume granularity, relatively less flexible than *VersioningFS*. Existing block level CDP schemes are all implemented at the device driver layer, typically by registering a pseudo-device in a module which replicates and splits arriving block write requests [23, 43] and store them in a separate logging pool. Those block level CDP schemes are primarily flawed in two aspects. One is that, they suffer from non-trivial performance degradation due to the tremendous overheads associated with the request-replicating operations imposed on the original I/O path. The other one is that, they are implemented on top of and agnostic to the underlying devices and thus can not fully exploit the peculiar characteristics of them. On the whole, the underlying mechanism of both *VersioningFS* and *VersioningDisk* is either using *Copy-on-Write(COW)* or *Indirection* [17], meaning when a write request arrives, they can adopt two alternatives, copying out the previous data residing at the write destination (thus COW) before overwriting them and store them on a separate repository or redirecting the write request to new locations (thus Indirection) and update the mapping logic to reflect its new state, leaving the old data there. Ultimately, the central point to CDP philosophy is to preserve the historical data sitting at the to-be-overwritten locations and provide necessary management facilities to sort out the correct data set corresponding to a specific timepoint.

As the semiconductor industry steadily advances and manufacturing processes become sophisticated, the capacity of flash-based SSDs are getting much larger than before with the price continuously decreasing at the same time. It's been demonstrated that SSDs are playing an important role in the storage systems[7, 33, 39]. In contrast to their conventional mechanically rotating hard disk drivers(HDDs) counterparts, flash-based SSDs exhibit widely disparate operational characteristics. There are three kinds of operations that can be performed on SSDs, which are *read*, *write/program* and *erase*. Read and write operations are performed at page granularity, while erases can only be performed in units of block which consists of several hundreds of pages. Erase takes several milliseconds and is much more expensive than read and write operations which only need several hundreds of microseconds. Pages can only be reused after being erased again. In order to avoid the costly erase operations severely impacting performance, an intermediate software named Flash Translation Layer(FTL) (discussed in Section 3) has been introduced in SSDs to manage dynamic address translations. Thanks to FTL, page erases are rendered out of the I/O path and are carried out in batches later when the available free(erased) pages reach a preset low watermark threshold. This inherent idiosyncrasies of SSDs would cause substantial remanent *superseded* [3, 41] data to linger in SSDs until they are selected for *garbage collection*. Those lingering *superseded* data exactly represent the history of the SSD. Leveraging this idiosyncrasy, we propose BVSSD, which usefully exploits those lingering *to-be-garbage-collected* data to implement CDP functionanlity by augmenting a mechanism which tracks those data and provides interfaces to manage them. By restoring those *superseded* data appropriately, BVSSD is able to be reverted to the storage state of some previous timepoint. The experimental results have shown that BVSSD is much less intrusive than other block level CDP schemes, causing only 3%-8% performance slowdowns compared with the original non-time-travelable SSDs, at the additional cost of minimal additional RAM. Another important advantage is that it needs virtually no modification to upper layer applications, operating systems or file systems, since the whole CDP functionality is completely implemented within the SSD itself.

In this study, our main contributions are three-fold: (1) We design a new lightweight device-built-in block-level CDP scheme for SSDs by leveraging the potential opportunity, particularly transforming the long-perceived negative aspect to a merit. Especially, to the best of our knowledge, we are the first to implement CDP for SSDs, which are getting more and more prevalent in the storage hierarchy; (2) We experimentally demonstrate that it outperforms the existing CDP schemes and embraces a number of other salient features, such as non-intrusive, high portability. (3) We propose a novel buffer scheme which sort the write requests by their logical addresses in order to reduce the garbage collection overheads. The general ideal behind this scheme can be extrapolated to other cases, e.g., sorting write requests by block hotness. Additionally, our research experience with BVSSD demonstrates that emerging technologies with their own advantages and shortcomings would provide us with

new opportunity to re-explore or re-think old techniques, possibly in a much more efficient manner.

The remainder of this paper is structured as following. For the convenience of exposition, we first give some background and our motivation in Section 2 and present related work in Section 3. Then, in Section 4 we discuss two important observations obtained from the real-world traces. After that, we elaborate on the design and implementation BVSSD in Section 5 and make a thorough evaluation of it in Section 6. In Section 7, we give a brief discussion of BVSSD. Finally, we end the paper with a concluding remark and planned future work in Section 8.

2. Background and Motivation

2.1 Walking Back the Storage System

Storage systems are constantly facing a wide variety of risks, including hardware malfunctions, equipment breakdowns, site disasters, virus intrusions, malicious attacks, unintentional errors,etc. In order to well protect the data, it's critical to build robust storage systems that are reasonably resilient to those faults and failures. Periodical backups and CDP are the most commonly deployed techniques for organizations and corporations. Backups provide full storage images, while CDP enables the storage state to be reverted to any previous point-in-time between backup intervals. There are already many existing CDP systems implemented at both file system layer [17, 29, 37] and block-level [13, 40, 43] which we will discuss in Section 3.

2.2 Flash Memory

Flash chips are a kind of non-volatile storage media and the basic building blocks of SSDs. They are composed of a number of *planes*, each of which contains several thousands of *blocks*. Each block in turn consists of hundreds of *pages*. Each page contains a data region and a metadata region. The data region stores the actual data, while the metadata region holds auxiliary information, such as the corresponding logical page number, Error Correction Code(ECC), etc. They only support read, write and erase operations. Read and write operations can only be performed at page granularity, while erase operations are block-only. Flash differs from HDDs in many important aspects, which result from the fundamentally different underlying information recording mechanisms, i.e., semi-conductor VS magnetic surface. Due to the absence of mechanical latencies, flash exhibits better performance than HDDs. However, it also suffers from several restrictions. The most excruciating one is that they does not allow "in-place updates" operations, resulting in *"erase-before-write"* problem [3]. Another limitation is that flash is *depletable* [30], which implies that it has limited life cycles. To overcome those peculiar restrictions, many algorithms and mechanisms have been proposed. Especially, different Flash Translation Layer(FTL) schemes combing with *Garbage Collection(GC)* and *Wear Leveling (WL)* have been proposed. Differing in the mapping granularity, FTL schemes are categorized into *page mapping* [15], *block mapping* [22] and *hybrid mapping* [42]. FTL dynamically manages the upper layer (file system) virtual address to physical page translations. When the same virtual address is overwritten, FTL does not immediately erase the corresponding physical page. Instead, it writes the new content to a free erased page, and then updates the mapping information to reflect the virtual address's newly mapped location. Those *superseded* pages will be at sometime later selected by *Garbage Collection* process to be erased for reuse. In this paper, we focus on page mapping scheme to implement BVSSD to avoid involving unnecessary complexity (e.g. expensive merge operations) associated with hybrid schemes, since our goal in this paper is to explore the feasibility of implementing CDP within SSD devices. Another reason is that page mapping scheme has been shown to be pragmatic and even more efficient than other schemes for a variety of applications [15].

2.3 Current Deployment of Flash-based SSDs

Despite of numerous excellent advantages, such as high performance, small form factor, lower power consumption, shock resistance, SSDs are still not widely deployed due to their high manufacturing cost disadvantage [26]. Though there exist data centers which are based solely on SSDs [7], at present, SSDs are much more often used as caches for HDDs in hybrid architectures to fully take advantage of the complementary benefits of HDDs (i.e. low cost, large capacity) and SSDs (i.e. high performance) [32, 33] or utilized to only store the performance-critical data to achieve better overall performance [8, 12] or deployed to support throughput-oriented applications, like OLTP and database applications [24]. Generally speaking, in most of those usage scenarios, systems are designed and optimized to make SSDs be presented with read-dominant access patterns and small to moderate total write capacity. As the manufacturing cost continues to decrease, SSDs of hundreds of GB are now available on the market [2], making it possible to support reasonably long CDP protection window,as we will see in the next section.

Considering the facts that the SSDs' inherent peculiarities can potentially provide the necessary requirements for CDP implementation, the promisingly anticipated SSDs-integrated future storage stack and write-avoiding usage scenarios of SSDs, we are motivated by the idea of integrating CDP with flash techniques to make the future storage systems more reliable from the time dimension perspective. Specifically, we implement CDP functionality within SSDs to make them more useful and applicable. Hopefully, by implementing BVSSD, we demonstrate the feasibility of an inherently much more reliable and resilient storage device that would be considered by the industry to turn them into real products.

3. Related Work

BVSSD benefits from numerous insightful lessons and experiences learned from previous works.

Flash Memory and SSDs: Previous researches on Flash and SSDs were centered around on how to disclose SSDs internals, overcome their inherent shortcomings and expand their usage spectrum. Agrawal et al.[3] discussed a range of design tradeoffs in implementing NAND-flash solid-state storage and developed an SSD simulator that can be seamlessly integrated into the DiskSim [6] simulation framework. Grupp el at.[14] empirically investigated various characteristics of flash memory from the external performance perspective, while Yoo el at.[44] also attempted to peek the SSD internals but from the consumed energy behaviors viewpoint. Boboila and Desnoyers [5] specifically investigated the write endurance problem of flash drives. All of these researches are of great help to us in understanding the flash/SSDs internals and coming up with the idea of BVSSD. Griffin [38] utilizes HDDs as write caches for SSDs to reduce the write traffic to SSDs, aiming to prolong SSDs lifespan. Their basic idea is to fully leverage HDD and SSD's complementary characteristics. BPLRU [21] is an effective write buffer management scheme that greatly improves SSD random write performance by buffering them to create sequentiality. IPL [24] greatly enhances the performance of database servers on SSDs, by logging page update deltas in dedicated logging regions as opposed to immediately overwriting destination pages. Similarly, HybridSSD [39] also logs page updates but uses PCM as log regions, leveraging the fact that PCM bears a relatively longer lifetime than flash and can be updated in an in-place manner at byte granularity. CAFTL [9] and CA-SSD [16] integrate duplicate-detecting function into the FTL to eliminate duplicate writes to SSDs. Berman *et al.*[4] formally verified the potential space savings when only differences between consecutive file versions are stored. Hystor [8] deploys SSD to store semantically critical and frequently accessed data in an SSD and HDD hybrid architecture to improve the performance. Similarly, SieveStore [32] uses SSDs to filter storage accesses by tracking the dynamic popular data blocks in a large-scale storage system to improve storage performance and energy savings. More recently, ICASH [33] intelligently couples HDDs with SDDs, with SSDs storing seldom-changed and mostly read reference data blocks and HDD holding a log of deltas relative to the corresponding reference blocks. All of the work, though address different peculiar problems of Flash/SSDs, do not positively use the Flash's negative characteristics.

Versioning Systems: Researchers have proposed a number of versioning systems. Elephant [34] thoroughly discussed various file retention policies in a versioning file system. Ext3cow [29] is a time-shifting modified version of ext3. It achieves time-shifting function through deeply involving in carefully changing the main file system data structures and introducing a number of mechanisms to trace and preserve modifications to ext3. CVFS [37] introduces the concept of *protection window* to a file system, which is a period of time in which any point-in-time image can be accessed particularly for post-intrusion analysis. It focuses on methods of efficient management of versioning metadata. Laden el at. [23] proposes four block level CDP implementation schemes and extensively analyzes and compares the overheads/tradeoffs associated with each of them. Peabody [40] is a time-traveling disk implemented in iscsitarget in a distributed storage management system. It logs every change to the virtual iscsi disks. TRAP [43] is also a block level CDP implementation but focuses on improving space-efficiency by only storing the differential contents (XORed results) of consecutive updates to the same block. Oh *et al.*[27] implements snapshot in flash memory file systems to reduce the interference associated with mobile data backups. However, their proposed PosFFS2 can only maintain two snapshots.

Exploiting Workloads Locality: Workload locality has long been observed and exploited in many previous SSD researches. SuperBlock FTL [20] leverages both spatial and temporal locality in workloads to create "SuperBlock" to reduce the FTL RAM footprint. *Block-level* spatial locality is utilized to formulate *superblock* with consecutive logical blocks, while temporal locality is used to identify hot and cold data within the superblock. DFTL [15] exploits workload locality to dynamically cache only part of the page-level address mapping information into the constrained on-flash SRAM. DFTL has been experimentally demonstrated to outperform other FTL schemes with improved performance, reduced garbage collection overheads. More recently, S-FTL[19] exploits the spatial locality of workloads to reduce FTL table size by compacting consecutive mappings to a single mapping entry. Similarly, as we will see, *the Garbage Collection* process in BVSSD also greatly benefits from the locality exhibited by the workloads thanks to our locality-aware optimized write scheme.

Treating with the Superseded Data: There are also several related papers that specifically focus on the *superseded* data in SSDs. In contrast to usefully exploiting the remnant data in SSDs as done in BVSSD, Wei *el at.* [41] and Chhabra *el at.* [10] perceive those remnant data as potential security threat, with which they dealt by immediately reliably erasing and encrypting to avoid information leakage. TxFlash[31] deploys the *superseded* data to implement atomic-write semantic where the transaction consists of multiple-pages writes. It focused on the development of a commit protocol in the software. Similarly, BeyondI/O [28] also leverages the remnant data to implement atomic-write semantic, but implement that within the SSD itself. By slightly changing the MySQL store engine InnoDB to be aware of the device-provided atomic-write facility, it's been demonstrated to improve database performance and reduce write storage to SSDs, while maintaining ACID transaction semantics. At a high level, BVSSD is similar to TxFlash

and BeyondI/O, with the difference being that how long the remnant data should be preserved in order to realize atomic-write semantics or CDP. In BVSSD, the data should be attempted to be preserved until the last possible moment, while for atomic-write, the data can be discarded as soon as the corresponding transaction has been committed.

4. Workload Observations

In this section, we investigate two important observed properties of two real world workloads traces. The purpose is try to gain some insightful understandings of them to guide our design. We choose two real world enterprise-scale workloads traces, i.e., Financial1 and Web-Search1 [1], as our study examples. Financial1 is a write-dominant (91%) I/O trace from an OLTP application running at a financial institution and Web-Search1 is a read-dominant (99.98%) Web Search engine trace. Both of them are made available by Storage Performance Council(SPC).

Observation 1: BVSSD is realistically useful. One of the key features of CDP systems is the length of *protection window*, i.e how long the target system can be CDP-protected consecutively. We use the *Total Write Capacity (TWC)* metric to measure how much written data that SSDs should tolerate for a specific workload in order to obtain the longest possible *protection window* . As opposed to HDDs, every overwritten data should also be accounted into TWC for SSDs, since new data do not immediately overwrite old data, but are written to new locations. And old *superseded* data would remain there until being erased and thus consume storage capacity. By analysis, Financial1 trace, which lasts for 12 hours, only generates about 15GB data totally, including writes and overwrites. Thus, an advertised 60GB SSD can guarantee a 2-days-long($12 \times \frac{60}{15} = 48hours$) *protection window*, i.e., it can be continuously written for 2 days without triggering garbage collection process, which is assumed to be reasonably long for CDP which is supplemented by periodical full backups in reality. It is even more promising for Web-Search1 trace whose TWC is only less than 2MB for a period of one hour running time. Taking upper layer optimization techniques as BPLRU buffer [21] and IPL [24] which significantly reduce write traffic at block level into account, modern SSDs would support even longer *protection windows*.

Observation 2: Workload temporal locality is beneficial to GC in BVSSD. By analyzing the relationship between accessed locations and their accessing timepoints in the two traces, we found that the workload exhibits interesting temporal locality characteristics. More specifically, blocks are accessed periodically in an intervened manner, i.e. they are accessed consecutively for a period of time interval and then left unvisited for another interval, then are visited again afterwards, and so on. Such temporal locality has important beneficial implications for GC process in BVSSD. In SSDs, the erased pages are allocated to new requests sequentially within blocks. Thus, due to the temporal locality, most pages of erased blocks are very likely to be allocated to the same logical page or a small region, which contributes greatly to reducing *write amplification* [3] overheads associated with BVSS GC process. We will elaborate on the GC process in BVSSD and how the temporal locality is exploited to optimize performance in the next section.

5. System Design and Implementation

Simply saying, the basic and straightforward principle behind BVSSD is to keep the change history of the FTL as it changes. The FTL maintains the whole logical to physical address mappings, hiding the SSD peculiar characteristics from the host's view and enabling the SSD to emulate conventional block-interface based HDDs. Every time the FTL is modified due to overwrite operations, we firstly copy out the current corresponding mapping entry, timestamp it and then store it into *CDP_store*, which is a key-value store engine added into SSDs to manage *superseded* FTL entries. Also, *CDP_store* exports interfaces to support CDP related management, including storing, deleting, destroying, recovery, etc.

5.1 BVSSD Overview

SSDs are composed of a number of flash chips, a dedicated processor, some RAM, buffer manager and the internal interconnection circuitry. Sitting on top of those flash packages is the FTL software handling address translations. Besides of those normal components, a *CDP_store* module is integrated into BVSSD to manage those *superseded* page mapping entries. *CDP_store* can be entirely implemented in the on-flash SRAM or stored on a dedicated flash region [15], since they are only read out when CDP recovery is needed. Currently, we store the *CDP_store* in memory. The CDP functionality can be configured to be enabled or disabled according to specific requirements. Figure 1 shows a simplified schematic view of BVSSD working mechanism.

To be simple but without missing the essential point, in Figure 1 we assume that the SSD only consists of two flash chips and each chip comprises of two blocks, each of which in turn contains four pages. The bottom of the figure represents the physical flash chips for storing data. Those *superseded* pages are shadowed. The left top is the FTL mapping table. To facilitate our exposition, we have also shown the evolution process of the mapping table. In reality, the mapping table only keeps the latest mapping relationship. The right top figure shows *CDP_store* engine which stores every *superseded* FTL mapping entry. For each entry, it records the mapping relationship between logical and physical addresses, the time when it was *superseded* and some other housekeeping information indicating whether it is valid(i.e. the *superseded* pages have not been *garbage collected*) and other information. For example, for the logical block number 3, it was initially mapped to physical page 4, and afterwards

it was *superseded* and mapped to physical page 5 at time T_1, causing a corresponding entry to be inserted in *CDP_store*.

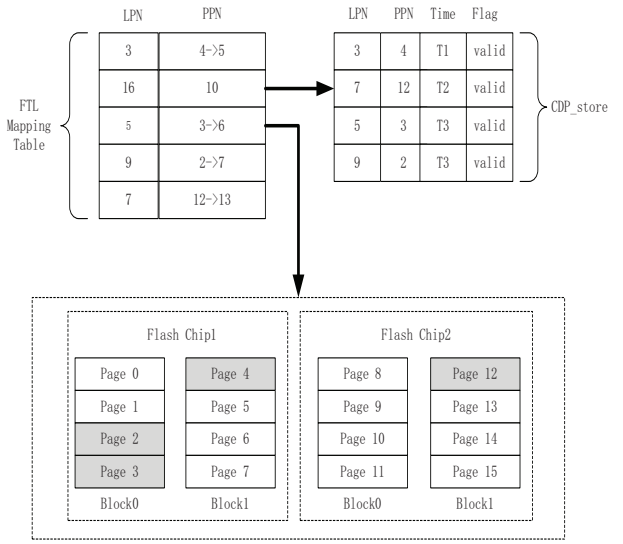


Figure 1. A simplified schematic view of BVSSD

The workflow has been changed slightly due to the presence of *CDP_store*. In this subsection, we discuss the read and write processes in BVSSD. In the next subsection, we will detail on the GC process and erase operations.

Read: Reads are carried out just as originally, irrespective of *CDP_store*, since only write operations would cause *superseded* pages. For a read request, it first looks up the write requests buffer or the FTL mapping table to get the corresponding physical address, and then fetch the data directly from that location. Thus, BVSSD imposes virtually no negative impacts on read operations, as we will see.

Write: Overwrite operations would cause *superseded* pages in SSDs. Thus for write requests, BVSSD stores the corresponding to-be-invalidated mapping entries into *CDP_store* before proceeding with the normal handling process as the original SSD does, which firstly allocates an erased page, then writes the new content to the allocated page and finally updates the corresponding FTL mapping entries. To avoid excessive direct accesses to flash to store the *superseded* mapping entries for each overwrite, the *CDP_store* could be implemented as a two-tier store engine, with RAM acting as a buffer for invalidated mapping entries and a dedicated portion of the SSD flash memory being the final hosting place. The RAM buffer is periodically flushed to the backend flash memory in batches. Thanks to the RAM buffer, the introduced overheads of logging invalidated FTL mapping entries would not get too overwhelming. However, in our current BVSSD prototype, we store the *CDP_store* in memory totally.

5.2 Design Challenges

As observed in Section 4, the current SSD raw capacity [2] is potentially able to sustain a reasonable length of *protection window* for certain applications. However, it is still likely that the *protection window* is expected to dynamically change and move forward due to the accumulated space consumption as time proceeds. Such usage scenarios necessitate a mechanism that recycles the history CDP data correctly, which is the most challenging problem that must be well taken care of. In normal SSDs, the *Garbage Collection(GC)* process is triggered when the amount of available free pages reach the preset low watermark. Typically, the watermark is set for every individual chip. When the watermark is reached, SSD scan through the chips in parallel to erase those blocks having the largest cleaning efficiency (i.e. the ratio of the number of invalid pages to the number of valid pages inside a block) [3] to free space until the free space is above a high watermark again. The GC process may cause *write amplification* which comes from copying out remaining valid pages in the victim block to migrate them to other clean blocks. Thus, there are two key points that need to be well addressed in BVSSD: (1) keep the CDP history data as long as possible in order to support longer protection window; and (2) minimize the GC overheads imposed by the specific requirement of preserving history data.

To tackle the above two challenges, we proposed two corresponding techniques. One is that instead of setting the low and high watermarks for individual chips, we promote the watermarks as global thresholds. Thus, only when the whole SSD available free space falls below the low watermark will the GC process be triggered. By doing that, the time that the history data can reside in BVSSD is prolonged. However, promoting global thresholds has negative implications as well. For example, the GC process is much harder, since it might migrate pages across chips even packages which is costly due to the possible serial contention of the interconnecting circuitry.

The other technique is that we add an additional filter called *Write Request Queue Buffer (WRQB)* in BVSSD. WRQB is a work queue that can buffer a configurable number of write requests and is flushed periodically to process the buffered requests. The buffered requests in WRQB are organized in different buckets according to their logical page numbers and flushed in a bucket-by-bucket manner. Because of the sequentially page-allocating heuristic in SSDs, all the write requests having the same logical page number would be very likely to be eventually written to the same block when WRQB is flushed, significantly reducing the potential extra amounts of write amplification in BVSSD.

In addition to the same write amplification causes with ordinary SSDs[3], BVSSDs suffer from other causes. In BVSSD, we write the last write information(e.g., the timepoint) onto the metadata region of blocks when the block is sealed, i.e., when all of their pages have been used up. When the *protection window* moves forward, causing the oldest revertible timepoint to advance from T_1 to $T_2(T_2 > T_1)$, those blocks whose last write timepoint is older than T_2 become

victim block candidates. However, there are still one typ of pages in those victim candidates that can not be recycled and should be copied out, causing additional write amplification. Those are the pages which have never been overwritten after T_2. Since once they are recycled, any future recovery to timepoint which is after T_2 would not be able to obtain the correct page content. Thus, those pages have to be preserved and copied out as well during the GC process as well, causing additional write amplifications. By aggregating the mapped pages of the same logical page in one block, this kind of write amplification can be significantly reduced, since once a logical page is shifted outside of the protection window, the large part of or even the whole block corresponding to the logical page are entirely invalidated and can be safely recycled altogether.

5.3 CDP_store Interfaces

In this subsection, we present several principal interfaces provided by *CDP_store* and give a brief description of each of them.

key_store(CDP_entry): key_store is used to store a superseded mapping entry into *CDP_store*. Every entry contains five pieces of information: Logical Page Number(LPN), Physical Page Number(PPN), Timestamp, Flags and auxiliary pointer link information .

CDP_drop(Time_interval): CDP_drop accepts a time interval in the form of [start_time, end_time] as its input parameter. It is called when the *protection window* moves forward. For instance, when the oldest revertible timepoint advances from T_1 to T_2, its passed-in argument is [T_1,T_2]. This function checks out and marks those victim pages for future GC process.

CDP_recovery(recovery_time): CDP_recovery takes a timepoint as its input argument and recovers the storage state to the *recovery_time* image. It searches CDP_store and obtains the corresponding history FTL mapping entries, restores the mapping table and updates the meta regions of all the pages. Specifically, it finds out those entries whose timestamp is the nearest-before the desired *recovery_time* and used them to correspondingly update the mapping table. The heuristic here is that the SSD state visible to outside is only determined by the FTL mapping table, regardless of the actual contents on the individual physical flash pages. Thus, if the mapping table is correct, then the storage state should be correct.

5.4 Overall Workflow

Summarizing the discussions in the previous sections, we can get an overview of the whole workflow of BVSSD. When a request arrives at BVSSD, it at first determines whether it is a read request or a write request. For a read request, it tries firstly to return its content buffered in WRQB. If not found, it then consults the FTL mapping table to get the physical page, and reads the data out, and then returns to the upper application. For a write request, it firstly checks the WRQB to see whether it can be buffered in the queue, i.e., is the queue already full? If the queue is not full, it enqueues the current request into WRQB. If the queue is full, it flushes out the bucket which corresponds to the address of the incoming request, making free space for the new request. And then it enqueues it into WRQB. The operations of preserving *superseded* FTL entries into *CDP_store* happens with the flushed writes. Figure 2 illustrates the discussed workflow process.

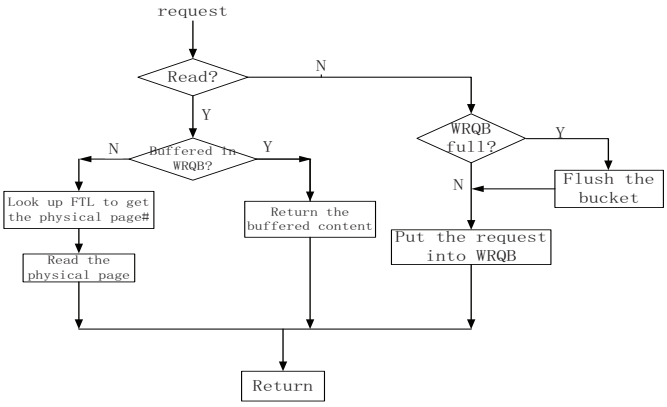


Figure 2. VBSSD workflow process

6. System Evaluation

6.1 Experimental Setup

We implement a BVSSD prototype by extending the SSD simulator from Microsoft [3]. The SSD simulator in turn is an extension of the DiskSim [6], which is an efficient, accurate, highly-configurable and widely used disk storage system simulator from PDL lab of Carnegie Mellon University. In all of the evaluation experiments, otherwise noted, we set the request length of WRQB to be 256. We use the traces of four enterprise-scale real-workloads to evaluate the efficacy of BVSSD. They are Financial1, Cello99, TPC-H and Web-Search1 [15]. As mentioned in Section 4, Financial1 and Web-Search1 are write-dominant and read-dominant applications, respectively. Cello99 was a disk trace collected from a time-sharing server running HP-UX operating system at Hewlette-Packard Laboratories. TPC-H is an ad-hoc, decision-support benchmark (OLTP workload) executing complex database queries against large volumes of data set. These workloads represent a wide range of read-write ratios. Table 1 and Table 2 summarize the workloads characteristics and simulation parameters,respectively. All the experiments were conducted on a simulated 60GB SSD[1]. Also, we compare BVSSD with two representative state-of-the-art CDP systems, including Ext3cow [29] which is a file

[1] The advertised capacity does not include the over-provisioning space. Over-provisioning space is used to improve the garbage collection process. We use the default over-provisioning space as in [3]

level CDP system and a modified TRAP [43] system, which is a block level CDP scheme.

Table 1. Workloads Characteristics

Workloads	Description	AvgReqSz(KB)	Read(%)
Financial1	OLTP	4.38	9.0
Cello99	HP-UX OS	5.03	35
TPC-H	OLAP	12.82	95
Web-Search1	Search App	14.86	99.9

Table 2. Simulation Parameters

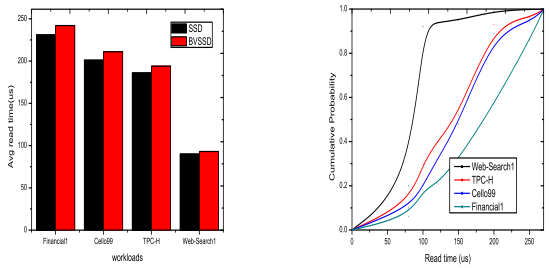
Operation	Latency
Page read	25μs
Page write	200μs
Block erase	1.5ms
Serial Access to Register	100μs
RAM access	25ns

6.2 Performance Impacts

One important principle that should be strictly respected is that the introduced extra functionality into existing systems should be non-destructive, meaning that it can't affect the original systems too much. In this subsection, we evaluate the performance impacts of BVSSD relative to the original SSD architecture. We ran those four traces both on the original SSD and BVSSD, and then report and compare their average read latency, average write latency and average response time metrics. Before each experiment run, we deliberately set up the whole mapping table to emulate that the SSD is fully used. We believe that doing this way would reveal the worst-case performance. Figure 3(a) shows the average read time. We can make two interesting observations from that figure. One is that the BVSSD read performance, as expected, remains virtually the same as that of original SSD, with only 4.7%,4.9%,4.3% and 3.3% slowdowns for Financial1, Cello99, TPC-H and Web-Search1, respectively. We assume that the trivial slowdowns are not caused by changed read path, but instead, they may be caused by resource contentions such as RAM and processor cycles imposed by *CDP_store*. To be precise, the *CDP_store* consumes certain amount of RAM, which can be otherwise used by the original architecture for other operations. The other one is that, the results also show that the higher the write ratios, the worse the slowdowns, due to the more resources consumed by *CDP_store*.

Figure 4(a) and Figure 5(a) show the number of the average write time and average response time, respectively. One apparent observation is that write times and overall response times are significantly larger than read latencies, amounting to more than one order of magnitude of the corresponding read response time. Besides of the main reason that the write operations are inherently much more expensive than read operations, the overheads are contributed by another two additional factors: accessing the buffer RAM to store *superseded* mapping entries and possible GC write amplifications. Another observation is that as write ratios increase, the slowdowns are generally getting bigger. Take the average response time for example, slowdowns of Web-Search1, TPC-H,Cello99 and Financial1 are 5.1%,6.2%,8.2% and 8.06%, respectively, with their corresponding write ratios being 0.1%, 5%,65% and 91% (see Table 1). It should be noted that the Cello99 is exceptional and exhibits a reverse trend, i.e., it has a smaller write ratio than Financial1(65% VS 91%), but suffers a slightly more severe slowdown(8.2% VS 8.06%). We suspect its smaller average block size may be the culprit behind the screen. Because smaller block sizes would cause more mapping entries to be handled by *CDP_store*.

Figure 3(b), Figure 4(b) and Figure 5(b) show a more detailed story by presenting the Cumulative Distribution Functions (CDF) of average read time, average write time and average response time, respectively. As it is shown, read time CDF is relative smoother, since the read time does't vary much in BVSSD, while both write CDF and response time CDF exhibit similarly a long tail due to a few severely impacted write requests, which are possibly hindered by the hard garbage collection process. But on the whole, the performance impacts of BVSSD are rather minimal, varying from 3% to 8% for a wide range of realistic enterprise-grade workloads, which is an acceptable trade-off for the continuous protection.

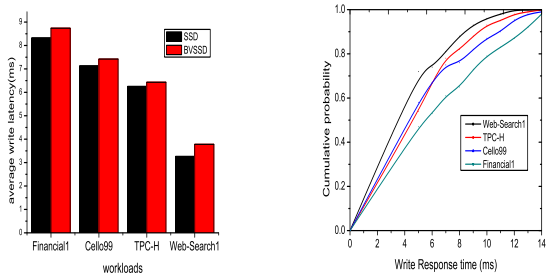


(a) Average read access time (b) CDF of read time

Figure 3. Read latencies. Figure 3(a) and Figure 3(b) show the average read latencies and their Cumulative Distribution Functions(CDF). The units in the figures are µs.

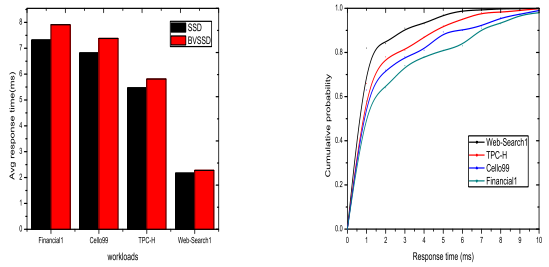
6.3 GC Behavior

Garbage Collection(GC) is needed to erase the previously programmed flash memory before reuse them in SSDs and it has important impacts on SSD performance. In this section, we are going to study and compare the GC behaviors in both SSD and BVSSD. We adopt an indirect approach to investigating the GC behavior of BVSSD, specifically focusing on the additional overheads imposed by the requirement



(a) Average write access time (b) CDF of write time

Figure 4. Write latencies. Figure 4(a) and Figure 4(b) show the average write latencies and their Cumulative Distribution Functions(CDF). The units in the figures are ms.



(a) Average response time (b) CDF of average response time

Figure 5. Response times. Figure 5(a) and Figure 5(b) show the average response latencies and their Cumulative Distribution Functions(CDF). Please note the different x-axis values.

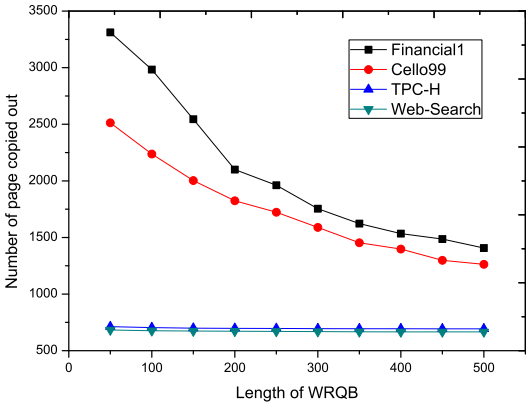


Figure 6. The write amplification relative to WRQB length

of preserving history data and the effectiveness of our proposed method in reducing the overheads. We use the number of copied pages during the GC process with respect to the queue length of WRQB to measure the write amplification impact, which is a good efficiency indicator of the GC process. In order to trigger GC process, we ran each of the four traces for a long period of warmup time(it's worth noting that it's different from initially deliberately setting up the mapping table, since page migrating is content-determined), e.g. running Financial1 for four times, to use up the simulated 60GB and varied the queue length from 50 to 500 to study the efficacy of WRQB in reducing write amplification. Figure 6 shows the relationship between them. As clearly demonstrated, WRQB is efficient in reducing the write amplification. From that figure, we know that increasing the length of WRQB is much less effective in contributing to the reduction of write amplification for read-dominant workloads than that of write-dominant applications. For example, increasing the queue length from 50 to 500 only reduces 19 and 17 page copying operations for TPC-H and Web-Search,respectively, but reduces 1905 and 1250 page copying operations for Financial1 and Cello99. There are three possible reasons for that. First, read-dominant workloads generate much less write requests than write-dominate workloads, leaving limited optimization space for WRQB. Since only write requests would cause write amplification and WRQB only buffers and reorganizes write requests. Secondly, read-dominate and write-dominate have differences in write access patterns. Examining the WRQB, we found that the average number of requests in the WRQB buckets of the write-dominant workloads are larger than that of read-dominant workloads, meaning the write-dominate are more write-concentrated, which would cause more logical pages to be resided in the same erase block, improving the efficiency of WRQB. Third, the write temporal locality observed in Section 4 is also an important contributing factor in clustering the same logical page to the same physical flash block. Another observation is that with reasonably large queue length, the number of write amplification can be reduced to the same level as the original SSD. Take Financial1 and Cell99 for example, the write amplifications are 1407 and 1263, almost approaching the level of SSD, which are 1216 and 1090, respectively. To sum up, our proposed WRQB technique is effective in suppressing the introduced overheads with reasonable cost, i.e., a limited RAM consumption for the buffered requests[2].

6.4 Compare with existing approaches

In this section, we make compare BVSSD with other state-of-the-art CDP schemes. We choose Ext3cow [29] and TRAP as our target schemes. Ext3cow is a file version-

[2] For all of the experiments, we have observed a maximum of about 200MB additional RAM requirement, which is an acceptable trade-off. Alternatively, we can adopt a two-tier approach as mentioned in Section 5.1 to further reduce the RAM requirement if that is not affordable.

ing system implemented in linux ext3 file system and has been made publicly available by the authors. TRAP is a block CDP scheme as mentioned previously. Due to the lack of the availability of its source code, we implemented a TRAP scheme according to the original paper[43]. Our goal is to uncover their respective relative performance impacts caused by the introduced CDP functionality into the original systems. To make the comparison be fair, we used postmark and Bonnie++ benchmarks, which are two widely used file system benchmarks as our workloads, and did not trigger GC process since neither Ext3cow nor TRAP has implemented garbage collection[3]. We ran the benchmarks on Ext3/Ext3cow and TRAP/Non-TRAP and report transactions per second and throughput, respectively. At the same time, we traced the block requests of the whole running processes, then fed the traces into SSD/BVSSD simulators. We reported the average response time for SSD/BVSSD. Then we compare the relative performance degradations between their respective versioning systems and the original counterparts. Figure 7 shows the comparison results. As can be seen from the figure, BVSSD is much less intrusive than Ext3cow and TRAP. That's because ext3cow involves a great deal of file system metadata operations and TRAP copies data blocks and performs encoding and decoding computations when performing CDP, which makes them degrade more than BVSSD.

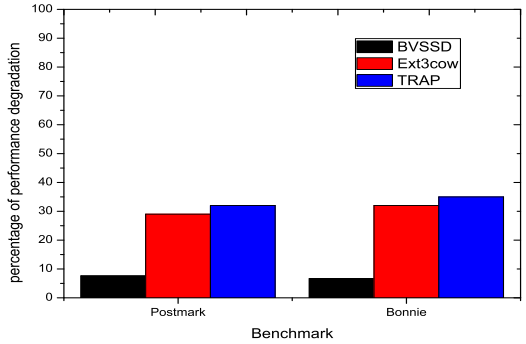


Figure 7. Comparison of performance impacts of different CDP schemes

7. Discussion

In this section, we are going to discuss some of the concerns of BVSSD. Power failures may be a big concern of BVSSD, since the *superseded* history FTL entries are stored in memory or buffered in memory before they are persisted. However, the risks caused by sudden power outage are universal for all systems deploying buffer/cache and not unique to BVSSD. For example, the FTL mapping information which are stored in on-flash memory themselves are also inherently subject to such risks. BVSSD is totally implemented within the SSD and those history FTL entries are also kept in the on-flash memory, thus it would not introduce added reliability problem. Furthermore, tricks can be deployed to remedy this concern, e.g., frequent flushing/synchronization can minimize data at risk and battery can be installed to prevent buffered data loss. In the extreme case, every FTL entry could be directly persisted when it is *superseded*. Obviously, there is a trade-off here.

Recovery consistency is another concern of BVSSD. Block content can only be interpreted by appropriate upper layer systems and block recovery is the first step to realize full system recovery. Systems often provide accompanying consistency check tools to examine their consistency state. For example, file systems and database systems have their specific tools. In the case of BVSSD, we can use the upper layer systems' tools to verify recovery consistency and to find a consistent state. Alternatively, systems can integrate *agents* which are system-specific(e.g., database systems agents are able to identify committed transactions) to facilitate the recovery process to consistent states. We are going to explore that in the future.

8. Conclusion and Future Work

In this paper, we present BVSSD, a new built-in, light-weight block CDP scheme, which is made realistically viable and applicable by the current SSDs industry trend that promises ever-growing SSD capacities and continuous-dropping prices. BVSSD outstands itself from existing CDP schemes in two primary aspects. One is that it is the first CDP scheme that is particularly proposed for SSDs, which are steadily gaining their position in the storage stack in recent years. The other one is that it is much more light-weight than other schemes. It implements CDP functionality by leveraging the specific idiosyncrasies of SSDs. Extensive experiments have shown that BVSSD is more efficient and less intrusive than the state-of-the-art CDP systems. We also demonstrate that BVSSD is pragmatic by investigating the current SSDs usages scenarios. Our experience in developing BVSSD has told us two important lessons. Firstly, new technology may open new opportunities for already existing techniques. Rethinking and re-implementing existing techniques in emerging technologies may be a good research option. Secondly, if perceived in perspective, weakness and negative sides of technologies can potentially be exploited to turn them useful.

Several future directions have been planned with BVSSD. Realistically, the length of protection window that a SSD can support is highly workloads-dependent, especially highly related to the write ratio of the workloads. In this paper, we have shown that modern hundreds of GB SSDs can support a reasonable length of protection window for the workloads we examined. But there remain more thorough evaluations to

[3] Here, it exactly means the recycling of those history data that are outside of the protection window due to the variation of protection window.

be carried out with more various workloads. Semantically-smart disk (SSD) [36] technique enables the data blocks somewhat semantically-aware and thus can be deployed to make more intelligent decision to improve performance or reliability. For instance, D-GRAID [35] uses block semantics to organize the data blocks belonging to the same file to one disk(i.e. the same fault domain) and replicate several copies of metadate blocks, significantly improving the availability of RAID in the event of failures. In the case of BVSSD, block level semantics can be used to define more flexible data retention policies. For example, BVSSD may prioritize reserving file system metadata blocks rather than data blocks when conducting GC, as opposed to our current simple retention policy, which is pruning the oddest blocks. We plan to integrate block semantics into BVSSD to make it somewhat semantic-aware to provide more flexible data retention policies for users. Block level recovery is inherently subject to the consistency problem, and thus it is not a unique problem to BVSSD. We are currently devising a consistency guarantee scheme from both application and SSD interface extension perspectives. Finally, we are also interesting in extending the CDP functionality to SSD RAIDs[18]. For example, an agent may be implemented in the RAID controller to coordinate all of of the constituent versioning SSDs and realize RAID-level versioning functionality.

Acknowledgments

The authors would like to thank the anonymous reviewers and Gokul Kandiraju (our shepherd) for their tremendous feedback and comments, which have substantially improved the quality of this paper. This paper was supported in part by National Basic Research Program (973 Program) of China under Grant No.2011CB302305 and National Key Technology R&D Program under Grant No.2012BAH35F03. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporters and sponsors.

References

[1] Available at: http://traces.cs.umass.edu/index.php/storage.

[2] http://www.seagate.com/www/en-us/products/enterprise-ssd-hdd/pulsar/pulsar-xt-2.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of USENIX Annual Technical Conference*, 2008.

[4] A. Berman and Y. Birk. Integrating de-duplication and write for increased performance and endurance of solid-state drives. In *Proceedings of the 26th Convention of Electrical and Electronics Engineers in Israel(IEEE'2010)*, 2010.

[5] S. Boboila and P. Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'2010)*, 2010.

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual. http://www.pdl.cmu.edu/disksim. 2010.

[7] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'09)*, 2009.

[8] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the 25th ACM International Conference on Supercomputing(ICS'2011)*, 2011.

[9] F. Chen, T. Luo, and X. Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies(FAST'11)*.

[10] S. Chhabra and Y. Solihin. i-nvmm: A secure non-volatile main memory system with incremental encryption. In *the 38th International Symposium on Computer Architecture(ISCA'2011)*, 2011.

[11] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation(OSDI'02)*, 2002.

[12] B. Debnath, S. Senguptaz, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX Annual Technical Conference*, 2010.

[13] M. D. Flouris and A. Bilas. Clotho: Transparent data versioning at the block i/o level. In *Proceedings of the 12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies(MSST'04)*, 2004.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, and S. Swanson. Characterizing flash memory: Anomalies, observations,and applications. In *Proceedings of the 42nd International Symposium on Microarchitecture(MICRO'09)*, 2009.

[15] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'09)*, 2009.

[16] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies(FAST'11)*, 2011.

[17] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.

[18] N. Jeremic, G. Mühl1, A. Busse, and J. Richling. The pitfalls of deploying solid-state drive raids. In *In proceedings of SYSTOR'2011*, 2011.

[19] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen. S-ftl: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies(MSST'2011)*, 2011.

[20] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT'06)*, 2006.

[21] H. Kim and S. Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[22] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[23] G. Laden, P. Ta-Shma, E. Yaffe, and M. Factor. Architectures for controller based cdp. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies(FAST'07)*, 2007.

[24] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[25] K.-K. Muniwamy-Reddy, C. P.Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, 2004.

[26] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating enterprise storage to ssds: analysis of tradeoffs. In *Proceedings of 4th EuroSys Conference(EuroSys'2009)*, 2009.

[27] Y. Oh, W. Lee, W. Park, S. Park, and C. Park. Posffs2: A new nand flash memory file system supporting snapshot in embedded linux. In *The 8th USENIX FAST'10 Poster Session*, 2010.

[28] X. Ouyangyz, D. Nellansy, R. Wipfely, D. Flynny, and D. K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *the 17th IEEE International Symposium on High-Performance Computer Architecture(HPCA'2011)*.

[29] Z. N. J. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[30] V. Prabhakaran, M. Balakrishnan, J. D. Davis, and T. Wobber. Depletable storage systems. In *Proceedings of HotStorage'2010*, 2010.

[31] V. Prabhakaran, T. L. Rodeheffer, and LidongZhou. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation(OSDI'2008)*, 2008.

[32] T. Pritchett and M. Thottethodi. Sievestore: A highly-selective, ensemble-level disk cache for cost-performance. In *the 37th International Symposium on Computer Architecture*, 2010.

[33] J. Ren and Q. Yang. I-cash: Intelligently coupled array of ssds and hdds. In *the 17th IEEE International Symposium on High-Performance Computer Architecture(HPCA'2011)*, 2011.

[34] D. S. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitchy. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, 1999.

[35] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with d-graid. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*.

[36] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau. Semantically-smart disk systems. In *Proceedings of 2nd USENIX Conference on File and Storage Technologies(FAST'03)*, 2003.

[37] C. A. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies(FAST'03)*, 2003.

[38] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending ssd lifetimes with disk based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'2010)*, 2010.

[39] G. Sun, Y. Joo, Y. C. Y. Chen, and H. Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *the 16th IEEE International Symposium on High-Performance Computer Architecture(HPCA'2010)*, 2010.

[40] C. B. M. III and D. Grunwald. Peabody: The time travelling disk. In *Proceedings of the 11th NASA Goddard/20th IEEE Conference on Mass Storage Systems and Technologies(MSS'03)*, 2003.

[41] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies(FAST'11)*, 2011.

[42] S. won Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer based flash translation layer using fully associative sector translation. *IEEE Transactions on Embedded Computing Systems*, 6(3), 2007.

[43] Q. Yang, W. Xiao, and J. Ren. Trap-array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture(ISCA'06)*, 2006.

[44] B. Yoo, Y. Won, S. Cho, S. Kang, J. Choi, and S. Yoon. Ssd characterization: From energy consumption's perspective. In *Proceedings of HotStorage'2011*, 2011.