Snapshots in a Flash with ioSnap

Sriram Subramanian,
Swaminathan Sundararaman,
Nisha Talagala
Fusion-IO Inc.
{srsubramanian,swami,ntalagala}@fusionio.com

Andrea C. Arpaci-Dusseau,
Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison
{dusseau,remzi}@cs.wisc.edu

Abstract

Snapshots are a common and heavily relied upon feature in storage systems. The high performance of flash-based storage systems brings new, more stringent, requirements for this classic capability. We present ioSnap, a flash optimized snapshot system. Through careful design exploiting common snapshot usage patterns and flash oriented optimizations, including leveraging native characteristics of Flash Translation Layers, ioSnap delivers low-overhead snapshots with minimal disruption to foreground traffic. Through our evaluation, we show that ioSnap incurs negligible performance overhead during normal operation, and that common-case operations such as snapshot creation and deletion incur little cost. We also demonstrate techniques to mitigate the performance impact on foreground I/O during intensive snapshot operations such as activation. Overall, ioSnap represents a case study of how to integrate snapshots into a modern, well-engineered flash-based storage system.

1. Introduction

Modern disk-based storage systems need to support a rich array of data services. Beyond a bevy of reliability machinery, such as checksums [4, 28], redundancy [17], and crash consistency techniques [10, 14, 23], customers expect systems to provide time and space saving capabilities, including caching [6, 16], deduplication [33] and archival services [22].

An important feature common in disk-based file systems and block stores is the ability to create and access *snapshots* [11, 15, 30]. A snapshot is a point-in-time representation of the state of a data volume, used in backups, replication, and other functions.

However, the world of storage is in the midst of a significant change, due to the advent of flash-based persistent memory [2, 5, 9]. Flash provides much higher throughput and lower latency to applications, particularly for those with random I/O needs, and thus is increasingly integral in modern storage stacks.

As flash-based storage becomes more prevalent by the rapid fall in prices (from hundreds of dollars per GB to about $1 per GB in the span of 10 years [8]) and significant increase in capacity (up to several TBs [7]), customers require flash devices to deliver the same features as traditional disk-based systems. However, the order of magnitude performance differential provided by flash requires a rethink of both the requirements imposed on classic data services such as snapshots and the implementation of these data services.

Most traditional disk-based snapshot systems rely on Copy-on-Write (CoW) or Redirect-on-Write (RoW) [1, 11, 13, 18] since snapshotted blocks have to be preserved and not overwritten. It is a well known fact that SSDs also rely on a form of Redirect-on-Write (or Remap-on-Write) to deliver high throughput and manage device wear [2]. Thus, at first glance, it seems natural for snapshots to seamlessly work with an SSD's flash translation layer (FTL) since the SSD does not overwrite blocks in-place and implicitly provides time-ordering through log structured writes.

In this paper, we describe the design and implementation of ioSnap, a system which adds flash-optimized snapshots to an FTL, in our case the Fusion-io Virtual Storage Layer (VSL). Through ioSnap we show how a flash-aware snapshot implementation can meet more stringent performance requirements as well as leverage the RoW capability native to the FTL.

We also present a careful analysis of ioSnap performance and space overheads. Through experimentation, we show that ioSnap delivers excellent common case read and write performance, largely indistinguishable from the standard FTL. We also measure the costs of creating, deleting, and accessing snapshots, showing that they are reasonable; in addition, we show how our rate-limiting machinery can re-

EuroSys 2014, April 13 - 16 2014, Amsterdam, Netherlands.
Copyright © 2014 ACM 978-1-4503-2704-6/14/04. . . $15.00.
http://dx.doi.org/10.1145/2592798.2592825

duce the impact of ioSnap activity substantially, ensuring little impact on foreground I/O.

The main contributions of this paper are as follows:

- We revisit snapshot requirements and interpret them anew for the flash-based storage era where performance, capacity/performance ratios, and predictable performance requirements are substantially different.
- We exploit the RoW capability natively present in FTLs to meet the stringent snapshot requirements described above, while also maintaining the performance efficiency for regular operations that is the primary deliverable of any production FTL.
- We explore a unique design space, which focuses on avoiding time and space overheads in common-case operations (such as data access, snapshot creation, and snapshot deletion) by sacrificing performance of rarely used operations (such as snapshot access).
- We introduce novel rate-limiting algorithms to minimize impact of background snapshot work on user activity.
- We perform a thorough analysis depicting the costs and benefits of ioSnap, including a comparison with Btrfs, a well known disk-optimized snapshotting system [1].

The rest of this paper is organized as follows. First, we describe background and related work on snapshots (§2). Second, we motivate the need to revisit snapshot requirements for flash and outline a new set of requirements (§3). Third, we present our design goals and optimizations points for ioSnap (§4). We then discuss our design and implementation (§5), present a detailed evaluation (§6), followed by a discussion of our results(§7), and finally conclude (§8).

2. Background

Snapshots are point-in-time representations of the state of a storage device. Typical storage systems employ snapshots to enable efficient backups and more recently to create audit trails for regulatory compliance [18]. Many disk based snapshot systems have been implemented with varied design requirements: some systems implement snapshots in the file system while others implement it at the block device layer. Some systems have focused on the efficiency and security aspects of snapshots, while others have focused on data retention and snapshot access capabilities. In this section we briefly overview existing snapshot systems and their design choices. Table 1 is a more extensive (though not exhaustive) list of snapshotting systems and their major design choices.

Block Level or File System?

From the implementation perspective, snapshots may be implemented at the file system or at the block layer. Both systems have their advantages. Block layer implementations let the snapshot capability stay independent of the file system above, thus making deployment simpler, generic, and hassle-free [30]. The major drawbacks of block layer snapshotting include no guarantees on the consistency of the snapshot, lack of metadata storage efficiency, and the need for other tools outside the file system to access the snapshots [15].

System	*Type*	*Metadata Efficiency*	*Consistency*
WAFL [11]	CoW FS	-	Yes
Btrfs [1]	CoW FS	-	Yes
ext3cow [18]	CoW FS	-	Yes
PersiFS [20]	CoW FS	-	Yes
Elephant [24]	CoW FS	Journaled	Yes
Fossil [22]	CoW FS	-	Yes
Spiralog [29]	Log FS	Log	Yes
NILFS [13]	Log FS	Log	Yes
VDisk [30]	BC on VM	-	No
Peabody [15]	BC on VD	-	Yes
Virtual Disks [19]	BC on VD	-	Yes
BVSSD [12]	BC	-	No
LTFTL [27]	BC	-	No
SSS [26]	Object CoW	Journaled [25]	Yes
ZeroSnap [21]	Flash Array	-	Yes
TRAP Array [32]	RAID CoW	XOR	No

Table 1. Versioning storage systems. *The table presents a summary of several versioning systems comparing some of the relevant characteristics including the type (Copy-on-Write, log structured, file system based or block layer), metadata versioning efficiency and snapshot consistency. (BC: Block CoW, VD: Virtual Disk)*

File system support for snapshots can overcome most of the issues with block level systems including consistent snapshots and efficient metadata storage. Systems like PureStorage [21], Self Securing Storage [26] and NetApp filers [11] are standalone storage boxes and may implement a proprietary file system to provide snapshots. File system level snapshotting can give rise to questions on the correctness, maintenance, and fault tolerance of these systems. For example, when namespace tunnels [11] are required for accessing older data, an unstable active file system could make snapshots inaccessible. Some block level systems also face issues with system upgrades (e.g., Peabody [15]) but others have used user level tools to interpret snapshotted data, thus avoiding the dependency on the file system [30].

Copy-on-Write vs. Redirect-on-Write

In Copy-on-Write (CoW) based snapshots, new data writes to the primary volume are updated in place while the old data, now belonging to the snapshot, is explicitly copied to a new location. This approach has the advantage of maintaining contiguity for the primary data copy, but induces overheads for regular I/O when snapshots are present. Redirect-on-Write (RoW) sends new writes to alternate locations, sacrificing contiguity in the primary copy for lower overhead updates in the presence of snapshots. Since FTLs perform Remap-on-Write, which for our purposes is conceptually similar to Redirect-on-Write, we use the two terms interchangeably in this paper.

Metadata: Efficiency and Consistency

Any snapshotting system has to not only keep versions of data, it must also version the metadata, without which accessing and making sense of versioned data would be impossible. Fortunately, metadata can be easily interpreted and thus changes can be easily compressed and stored efficiently.

Maintaining metadata consistency across snapshots is important to keep older data accessible at all times. The layer at which snapshotting is performed, namely file system or block layer, determines how consistency is handled. File system snapshots can implicitly maintain consistency by creating new snapshots upon completion of all related operations, while block layer snapshots cannot understand relationships between blocks and thus consistency is not guaranteed. For example, while snapshotting, a file, the inode, the bitmaps, and the indirect blocks may be required for a consistent image. Recently, file systems (such as Ext4, XFS, and Btrfs) have added support for freeze (that blocks all incoming I/Os and writes all dirty data, metadata, and journal) and unfreeze (unblocks all user I/Os) operations to help block devices to take file-system-level consistent snapshots.

Snapshot Access and Cleanup

In addition to creating and maintaining snapshots, the snapshots must be made available to users or administrators for recovery or backup. Moreover, users may want to delete older snapshots, perhaps to save space. Interfaces include namespace tunnels (e.g., an invisible subdirectory to store snapshots, appending tags or timestamps to file names as used in WAFL [11], Fossil [22] or Ext3cow [18]) and user level programs to access snapshots (e.g., VDisk [30]), etc.

3. Flash Optimized Snapshots

We now discuss the differences between disk and flash-based systems and outline areas where snapshot requirements should be changed or extended to match the characteristics of flash-based storage systems.

3.1 Flash/Disk differences and Impacts on Snapshots

Flash devices provide several orders of magnitude performance improvements over their disk counterparts. A typical HDD performs several hundreds of IOPS, while flash devices deliver several thousands to several hundreds of thousands of IOPS. Increasing flash densities and lower cost/GB are leading to terabyte capacity flash devices and all-flash storage arrays with tens to hundreds of terabytes of flash.

Flash has a much higher IOPS/GB, implying that an average flash device can be filled up much faster than an average disk. For example, a multi-threaded workload of 8KB I/Os (a common database workload) operating at 30K IOPS, can fill up a 1TB flash device in a little over an hour. The same workload, operating at 500 IOPS, will take almost three days to fill up an equivalent sized disk. Due to the difference in IOPS/GB, the rate of data change in flash-based systems is much greater than disk. We anticipate greater numbers of snapshots would be used to capture intermediate state.

The performance requirements of flash-based systems are also much more stringent, with predictable performance frequently being as critical if not more critical than average performance. As such, we anticipate the following performance requirements with regard to snapshots: (i) the performance of the primary workload should not degrade as snapshots accumulate in the system, (ii) the creation of a snapshot should minimally impact foreground performance, and (iii) the metadata (to recover/track snapshot data) should be stored efficiently to maximize the available space for storing user data. These requirements, if met, also enable frequent creation of snapshots, which enables fine grained change tracking as suitable for storage systems with a high IOPS/GB, while still retaining lower cost/GB ratio.

4. Design Goals and Optimization Points

In this section, we discuss the design goals and optimizations points for our flash-based snapshotting system.

4.1 Design Goals

The requirements listed in (§3) lead us to define the following goals for our flash-based snapshot design.

Maintain primary device performance: Since the primary goal of any flash device is performance, we strive to ensure that the primary performance of the device is minimally impacted by the presence of the snapshot feature.

Predictable application performance: Common case operations of snapshots should be fast and have minimal impact on foreground operation. In other words, foreground workloads should see predictable performance during and after any snapshot activity (i.e., creation, deletion, and activation).

Unlimited snapshots: Given that the rate of change of data is much higher with flash-based storage than with disk, one should focus on a snapshot design that limits snapshot count only to the capacity available to hold the deltas.

Minimal memory and storage consumption: To support large numbers of snapshots, it is important that both system memory and snapshot meta-data in flash does not bloat up with increasing numbers of snapshots.

4.2 Optimization Points

To achieve the above design goals, we exploit a few characteristics of both flash and snapshots.

Remap-on-Write: We leverage the existing remap-on-write nature of flash translation layers and exploit this wherever possible for snapshot management.

Exploit known usage patterns of snapshots: To optimize our snapshot performance, we exploit known usage patterns of snapshots. First, we assume that snapshot creation is a common operation, rendered even more frequent by the high rate of change of data enabled with flash performance. We further assume that snapshot access is (much) less common than creation. Typically, snapshots are activated only when the system or certain files need to be restored from backup (disaster recovery). Thus, activations do not match in frequency to creation. We also assume that many (not all) snapshot activations are schedulable (e.g., predetermined based

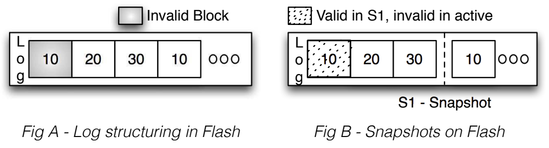
Invalid Block
Valid in S1, invalid in active
Log
10
20
30
10
ooo
S1 - Snapshot
Fig A - Log structuring in Flash
Fig B - Snapshots on Flash

Figure 1. Snapshots on the log. *This figure illustrates how a log-based device can naturally support snapshots. Each rectangle is a log, with squares indicating blocks (logical block address (LBA) mentioned inside). Figure A shows a log with four blocks with LBAs 10, 20, 30 and LBA 10 get overwritten. The log in Figure B, shows how the log can be leveraged to implements snapshots. If we were to write blocks at addresses 10, 20 and 30 and then create a snapshot (call it S1, indicated by the dashed line) and overwrite block 10, then according to snapshot S1, the original block at address 10 is still valid while the active log only see the new block at address 10. Thus by selectively retaining blocks, the log can support snapshots.*

on backups) and that this knowledge can be exploited by the snapshot system to manage activation overheads.

Exploit the performance of flash: We should keep as little snapshot-related metadata on flash as possible when it is not actively accessed. The majority of the snapshot metadata can be constructed in memory when the snapshot is activated. Also, for metadata that is best kept in memory, we can use a CoW model to ensure that duplicate snapshot metadata is not stored in memory.

5. Design and Implementation

In this section, we describe ioSnap's design and implementation in detail, focusing on the APIs, the base FTL design, and FTL components that were extended to support snapshots.

5.1 Snapshot API

The basic snapshot operations are minimal: snapshot create, delete, activate, and deactivate. Snapshot create and delete allow the user to persistently create or delete a snapshot (ioSnap implicitly retains all snapshots unless explicitly deleted). Activation is the process of making a snapshot accessible. Not all snapshotting systems require activations. We make activation a formal step because ioSnap defers snapshot work until activation in many cases. Finally, ioSnap also needs to provide the ability to deactivate a snapshot. The snapshot APIs are meant to provide a set of mechanisms, on top of which workload-specific creation and retention policies may be applied.

5.2 The FTL: Fusion-io Virtual Storage Layer

In this section, we describe the basic operations of the Fusion-io Virtual Storage Layer (VSL) which provides flash management for Fusion-io ioMemory devices. The VSL is a host-based flash management system that aggregates NAND flash modules, performs flash management and presents a conventional block device interface. We provide a high-level overview of the FTL; our discussion covers a previous-generation (still high performance) FTL into which we later integrate ioSnap.

5.2.1 Remap-on-Write

The VSL maps all incoming writes to fresh NAND locations since NAND flash does not support efficient overwrites of physical locations (requiring an expensive read, erase and write cycle [2]). Also, the NAND flash chips can be erased and programmed only a certain number of times before the raw bit error rates becomes too high [8]. To circumvent this costly process, data is never immediately overwritten, but instead appended in a log-structured manner [23]. Thus, the address exposed by the block device maps to a new address on the log every time the block is modified (see Figure 1). The VSL manages this remapping by threading through the log-structured NAND segments in the device.

5.2.2 Basic Data Structures and Operations

The *FTL* and the *validity bitmap* are the primary data structures that manage the RoW operations. The Fusion-io VSL (i.e., FTL) uses a variant of a B+tree, running in host memory, and translates Logical Block Addresses (or LBAs) to physical addresses in flash.

The validity bitmap indicates the liveness of each physical block on the log. Block overwrites translate to log appends followed by invalidation of the older blocks. Thus, an LBA overwrite results in the bit corresponding to the old block's physical address being cleared in the validity map and a new bit being set corresponding to the new location as illustrated in Figure 2.

A read operation involves looking up the FTL to translate a range of LBAs to one or more ranges of physical addresses. The driver reads the data in physical addresses and returns the data to the requesting file system or user application.

A write request requires a range of LBAs and the data to be written. New blocks are always written to the head of the log. The FTL is used to look up the range of LBAs to figure out if a portion of the write involves an overwrite. If blocks are being overwritten, in addition to modifying the FTL to indicate the new physical addresses, the driver must alter the validity bitmap to invalidate older data and validate the new physical locations.

5.2.3 Segment Cleaning

Segment cleaning is the process of compacting older segments, so as to get rid of all invalid data, thus releasing unused space [23]. Over the life of a device, data on the log is continuously invalidated. Overwrites (and subsequent invalidation) can lead to invalid data being interspersed with valid data. Without the ability to overwrite in place, the driver must erase one or more pages which contain invalid data to regain lost space while copy-forwarding valid data to ensure correctness. The segment cleaner also modifies the FTL to indicate the new location of the block after copy-forward.

The segment cleaner can also have significant performance implications. The erase operation is relatively expensive (erase times are in the order of a few milliseconds for

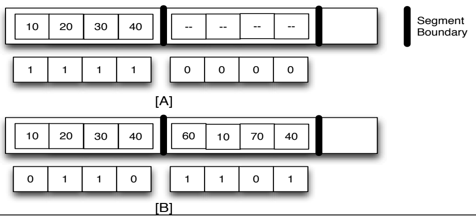

Figure 2. Validity maps. *The figure above describes the use of validity maps to help the segment cleaner. The example shows a log with 8 blocks spread over two segments. The validity map shown in Figure A represents 4 blocks 10, 20, 30 and 40. The remaining blocks are unused and the validity bits are cleared. Figure B represents a point in the future where 4 more blocks have been written to the log namely 60, 10, 70 and 40. As we can observe, blocks 10 and 40 have been overwritten and so the corresponding bits from the validity map are cleared.*

currently available NAND modules), and must be performed in bulk to amortize latency. The log is divided into equal-sized segments and the segments form the unit of the erase operation. The segment to erase is chosen on the basis of various factors, such as the extent of invalid data contained in the segment and the relative age of the blocks present (degree of hotness or coldness of data). The choice of the segment to clean determines the volume of copy-forward (write-amplification) and how frequently a segment gets erased (wear-leveling), and thus indirectly impacts the overall performance of the system.

5.3 Adding Snapshot Capability

5.3.1 Exploiting Remap-on-Write

The RoW capability naturally supports snapshots. New blocks are appended to the head of the log, leaving behind the invalidated data to be reclaimed by the segment cleaner. Thus, supporting snapshots requires selectively retaining older blocks and allowing the segment cleaner to discern snapshot blocks from invalid blocks. Figure 1 shows a detailed example.

5.3.2 Tracking Snapshots: Epochs and Snapshot Tree

The log structuring in the VSL inherently creates time-ordering within the blocks and the time ordering of blocks can easily allow us to associate a group of blocks to a snapshot. However, this time ordering is not completely preserved in the base FTL. The segment cleaner moves valid blocks to the head of the log and disrupts the time-ordering. Once blocks from distinct time frames get mixed with active writes, it becomes hard to distinguish blocks which belong to one snapshot from another. Figure 3(A) illustrates the impact of segment cleaning on a sample log.

In order to completely preserve time ordering, ioSnap adds the notion of *Epochs* [18, 22]. Epochs divide operations into log-time based sets segregating operations that occurred between subsequent snapshot operations. Every epoch is assigned an epoch number, which is a monotonically increasing counter that is incremented after every snapshot operation. ioSnap stores the current epoch number in the data block header (i.e., out-of-bound area) when blocks are written to the media. By associating every block with an epoch number, ioSnap can (loosely) maintain the notion of log-time despite intermixing induced by the segment cleaner. The segment cleaner also then has the option of keeping epochs co-located after cleaning. Figure 3(B) shows how associating epoch numbers to blocks can help manage the block intermixing. Epochs are incremented when snapshots are created or activated. Every snapshot is associated with an epoch number, which indicates the epoch number of blocks that were written after the last known snapshot operation.

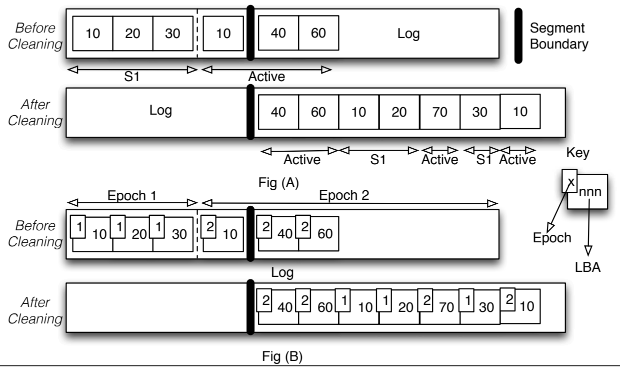

Figure 3. Epochs. *The figure illustrates the impact of the garbage collector on the time order of the log and the need for Epochs. In Figure A, we write blocks at LBAs 10, 20, 30, create a snapshot S1 and continue writing blocks to LBAs 10, 40 and 60. The segment boundaries are indicated by the thick black line. While blocks are moved, newer writes may also be processed resulting in active data (LBAs 40, 60, 10 and 70) getting mixed with blocks from S1 as shown by the log at the bottom of Figure A and makes distinguishing blocks impossible. Figure B shows the use of epochs to delineate blocks belonging to various snapshots. Epoch numbers are assigned to all blocks as indicated by the number in the small box on the left top.*

A snapshot created at any point in time is related to a select set of snapshots created earlier. The moment a snapshot is created, every block of data on the drive is (logically) pointed to by two entities: the *active tree* (the device the user issues reads and writes to) and the snapshot which was created. In other words, the active tree implicitly inherits its blocks from the snapshot that was created. As subsequent writes are issued, the state of the active tree diverges from the state of the snapshot until the moment another snapshot is created. The newly created snapshot captures the set of all changes that took place between the first snapshot and the present. Finally, as more snapshots are created and activated, ioSnap keeps track of the relationships between the snapshots through a snapshot tree [3]. Figure 4 illustrates an example depicting four snapshots and their relationships are created through a sequence of creates and activates.

The combined approach of epochs and snapshot tree enables us to meet our design goals in the following two ways. First, minimal impact on normal operations regardless of how many snapshots are created. Since the snapshot tree is

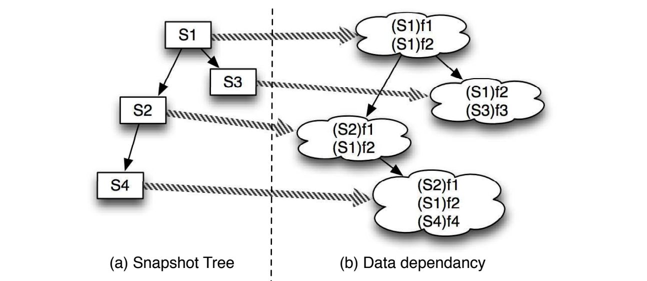

Figure 4. Snapshot tree. *The figure illustrates the relationship between snapshots based on data they share. The tree to the left shows the relationship between snapshots and the tree to the right shows the data associated with each snapshot (as files for simplicity with the snapshot it originated from in parentheses). Snapshot S1 has two files, f1 and f2. After updating file f1, snapshot S2 is created followed by S4 which adds a file f4. S4's state contains files from S1 and S2. At some point, S1 was activated and file f1 was deleted and a new file f3 was created. Deleting file f2 does not affect the file stored in S1. Activating S1 creates a fork in the tree and after some changes, S3 is created.*

only updated during snapshot operations, the presence of the snapshot tree has close to no impact on normal user performance. Also, unlike most disk-based snapshot systems, this approach of using a separate tree to track snapshots combined with the log-embedded epochs to track blocks for each snapshot ensures that no matter how many snapshots exist in the system, the data path of the primary workload remains the same. For example, no CoW for device blocks, no map updates on writes beyond what is normal for flash, or no need to check multiple levels of maps to see where the most current block is. Previous approaches all have the effect of slowing the system down as the snapshot count increases (even if those snapshots are not being used) and are inherently unfriendly to high snapshot count configurations.

Secondly, since all snapshot tracking for blocks is embedded in the log through epochs, its unnecessary to create a separate tracking map in memory for the blocks associated with the snapshot (done in most disk-based systems). This renders the snapshot creation extremely fast and in-memory snapshot overhead is minimal for unactivated snapshots (which we assume to be the majority).

5.4 Snapshot-aware Segment Cleaner

The segment cleaner is responsible for reclaiming space left by invalidated data. Not surprisingly, adding snapshots to the FTL necessitates changes to the cleaner.

There are two major issues that the cleaner must address to operate correctly in the presence of snapshots. Originally, as mentioned in (§5.2.3), the cleaner simply examines the validity bitmap to infer the validity of a block. Unfortunately, when snapshot data is present in the log, a block which is invalid in the current state of the device may still be used in some snapshot. The first challenge for the segment cleaner is thus to figure out if there exists at least one older snapshot which still uses the block (as seen in Figure 3). Second, in

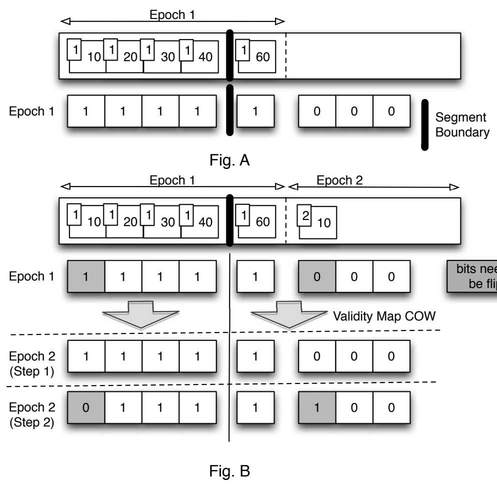

Figure 5. Epochs and validity maps. *The figure describes the use of per epoch validity maps to help the segment cleaner. The example shows a simple case with two epochs spread over two segments. The validity map named Epoch 1 corresponds to the validity map of the log at snapshot creation. The validity map in Figure A shows 5 valid blocks on the log. After snapshot creation , block 10 is overwritten, involving clearing one bit and setting another. The bits that need to be cleared and set are shown in gray in Figure B and clearly these belong to Epoch 1 and hence read-only. Thus, the first step is to create copies of these validity blocks (shown in step 1). Next, the bits in the newly created epoch 2 are set and cleared to represent the updated state. It is important to note that we are only selectively copying the portion of the validity bitmap that was modified as part of this operation and not the full bitmap.*

the presence of snapshots, the segment selection algorithms must be optimized for not just basic cleaning, but also to ensure that snapshots are optimally located on flash media after the cleaning operation.

5.4.1 Copy-on-Write Validity Bitmap

Validity bitmaps are used to indicate the liveness of a block with respect to the active log. In the presence of snapshots, a block that is valid with respect to one snapshot may have been overwritten in the active log, thus making maintenance of validity bitmaps tricky.

Since one design goal of ioSnap is to avoid arbitrary limits on the size of snapshots, we avoid reference counters which are the cause of snapshot limits in many past designs. ioSnap solves the validity bitmap problem by maintaining bitmaps for each epoch. During the course of an epoch, the validity bitmap is modified in the same manner as described in Section 5.2.2. When a snapshot is created, the state of the validity bitmap at that point corresponds to the state of the device captured by the snapshot. Having copies of the validity bitmap for each snapshot allows us to know exactly which blocks were valid in a snapshot.

A snapshot's validity bitmap is never modified unless the segment cleaner moves blocks (more details in Section 5.4.3). Like epochs, the validity bitmap for a snapshot inherits the validity bitmap of its parent to represent the state of the blocks inherited. The active log can continue to modify the inherited validity bitmap.

A naive design would be to copy the validity bitmap at snapshot creation, which would guarantee a unique set of bitmaps which accurately represent the state of the blocks belonging to the snapshot. Clearly, such a system would be highly inefficient since every snapshot would require a large amount of memory to represent its validity bitmaps (e.g., for a 2 TB drive with 512 byte blocks we need 512 MB of validity bitmap per snapshot).

Instead, ioSnap takes an approach that relies on CoW for the validity bitmap blocks. It leverages the fact that time ordering of the log typically guarantees spatial collocation of blocks belonging to the same epoch. So, until blocks are moved by the segment cleaner to the head of the log, the validity bitmaps corresponding to an epoch would also be relatively localized.

ioSnap uses this observation to employ CoW on validity bitmaps after a snapshot operation where all validity bitmap blocks are marked CoW. When an attempt is made to modify a block marked CoW, a copy is created and is linked to the snapshot (or epoch) whose state it represents. The validity bitmap pages that were copied are read-only (until reclamation) and can be de-staged to the log. Figure 5 illustrates an example where validity bitmaps are inherited and later modified by the active log.

5.4.2 Policies

A snapshot-aware segment cleaner has to minimize intermixing of blocks across different epochs to help achieve a low degree of write amplification [23]. Lower intermixing of blocks also helps reduce the overheads of copy-on-write validity bitmaps and hence improve overall cleaner performance. Systems such as HEC [31] have explored implications of segment cleaning policies on write amplification and have shown that the choice of the cleaning policy could have significant impact on the overall performance of the system.

Intermixing of blocks across epochs can be minimized by collocating blocks belonging to an epoch. Snapshotted data also represents cold data: so they can be trivially segregated [23]. Such policies can be applied during active (concurrent with user I/O) and idle-time segment cleaning. Unfortunately, in this version of ioSnap we do not delve into the policy aspect of the segment cleaning.

5.4.3 Segment Cleaning: Putting It All Together

Segment cleaning in the presence of snapshots is significantly different from the standard block device segment cleaner. The segment cleaner needs to decide if a block is valid or not and the validity of a block cannot be derived just by looking at one validity bitmap. The following steps need to be followed to clean a segment:

Merge validity bitmaps: The validity information for each block is spread across multiple validity bitmaps. To obtain a global view of the device across epochs, ioSnap merges the validity bitmaps (via logical OR) and produce a cumulative map for the segment. Epochs corresponding to deleted snapshots need not be merged unless there exists at least one descendant epoch still inheriting the validity bitmap. Figure 6 illustrates this process.

Check for validity of blocks: The merged validity bitmap for a segment represent the globally valid blocks in that segment. The blocks which are invalid are the ones which have been overwritten within the same epoch or the ones which may belong to deleted snapshots.

Move and reset validity bits: For every valid block copy-forwarded, the validity bits need to be adjusted. One or more epochs may refer to the block, which means ioSnap needs to set and clear the validity bitmap at more than one location. In the worst case, every valid epoch may refer to this block, in which case ioSnap may be required to set as many bits as there are epochs.

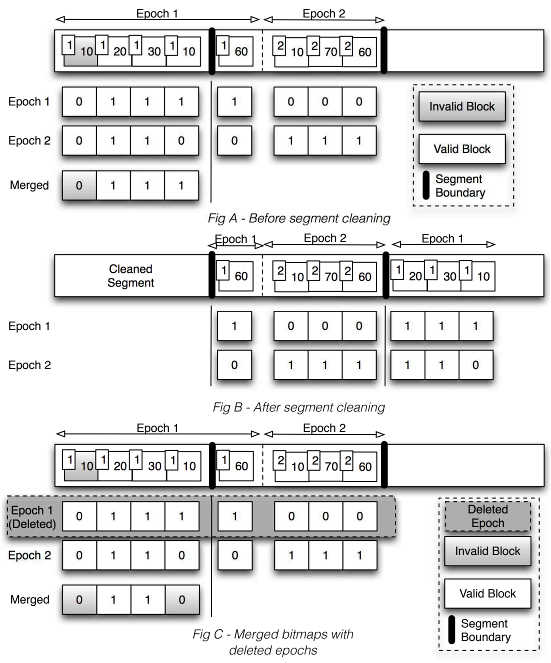

Figure 6. Segment cleaner in operation. *The figure illustrates the segment cleaner operation in the presence of snapshots. Figure A and B show the state of the drive before and after segment cleaning. The log in Figure A uses two segments with 4 blocks each and has one snapshot (indicated by the epochs 1 and 2) with the corresponding validity maps shown below. The segment cleaner merges the validity maps to create the merged map. Based on the validity of blocks in the merged maps, blocks 20,30,10 (from Epoch 1) are copy forwarded to Segment 3 as shows in Figure B. While moving blocks forward, the segment cleaner has to re-adjust the validity maps to reflect block state in Segment 3. Finally, Figure C shows an example with a deleted epoch (Epoch 1). The merged bitmap is equivalent to Epoch 2 (the only valid epoch) , thus automatically invalidating blocks from Epoch 1.*

5.5 Crash Recovery and Reconstruction

The device state needs to be reconstructed after a crash. In the version of the driver we use, the device state is fully checkpointed only on a clean shutdown. On an unclean shutdown, some elements of in-memory state are reconstructed from metadata stored in the log.

5.5.1 Forward map reconstruction

The forward map of the active tree is reconstructed by processing logical to physical block translations present in the log. In our prototype, we only reconstruct the active tree and do not build trees corresponding to the snapshots. This goes back to our design choice of keeping operation on the active tree fast and unobstructed. Attempting to reconstruct snapshot trees would not only slow down the reconstruction process, it would put pressure on the system memory and eventually hinder segment cleaning.

The reconstruction of the forward map happens in two phases. In the first phase, the snapshot are identified and the snapshot tree is constructed. Once the snapshot tree is constructed, we have the lineage of the active tree whose forward map needs to be constructed. In the second pass, we selectively process the translations that are relevant to the active tree. The relevant packets for the active tree are those that belong to the epoch of the active tree or any of its parent in the snapshot tree. We also process the snapshot deletion and activation packets in the second pass and update the snapshot tree. Once all packets are processed, we sort the entries on their logical block address and reconstruct the forward map in a bottom up fashion.

5.5.2 Validity bitmap reconstruction

Validity bitmap reconstruction happens in multiple phases. In the first phase, we sort data entries based on the epoch in which the data was created (or written in). We then eliminate duplicate (or older) entries and uniquely identify packets that are valid and active in the epoch under consideration. Once the valid entries are identified, we construct the validity bitmap for each epoch starting from the root of the snapshot tree until we reach the leaf nodes in a breadth first manner. The final validity map for each epoch is reconstructed by merging the epochs that could contribute packets to this epoch (namely, the parent epochs). While merging we eliminate duplicates or invalidated entries. The snapshot tree is traversed in a breadth first manner and validity maps constructed for every epoch in the manner described above. The number of phases needed to reconstruct the validity bitmap depends on the number of snapshots created in the device. For example, in Figure 4, packets belonging to epochs/snapshots S1, S2, S3 and S4 are first segregated and sorted. The next phase would involve constructing validity maps for S1. Following the breadth first path along the tree, we merge packets belonging to S2 and S1 and construct the validity maps for the merged set. Similarly, we merge S3 and S1 and finally S4, S2 and S1 to reconstruct the entire state.

The validity map reconstruction for individual snapshots is done in a breadth first manner. For each snapshot, the forward map entries are compared with the forward map entries of its parent epoch and then create the validity bitmap of the physical packets they represent.

5.6 Snapshot Activation

In order to access a snapshot, we need to activate it. The primary reason for an activation operation is due to the fact that we only maintain the forward map of the active drive prior to an activation. In order to access a snapshot, we need the forward map corresponding to the snapshot, and keeping multiple forward maps in memory (even under CoW) is expensive. Moreover, having multiple maps may require multiple updates to the map when the packet is moved by the segment cleaner as packets may be shared between snapshots. Finally, we believe snapshots are activated to restore lost or corrupted data, which is a rare event. Hence, the overheads associated with activation of a snapshot does not make the system unusable.

Snapshot activations can impact foreground user performance. Activating a snapshot requires reading packet headers from the log and recreating the forward map. The snapshot activation traffic competes with user I/Os for device bandwidth and as a result could introduce jitters (or spikes) in foreground user workloads. As mentioned earlier, variability in user performance is unacceptable and to alleviate the problem we rate-limit the snapshot activation process to minimize variability in performance.

Device traffic due to snapshot activations are rate-limited in the driver based on the foreground user traffic. We intersperse read traffic due to snapshot activation with user traffic and the degree of interspersing is varied based on the instantaneous user traffic. We also provide a knob to the user to control the rate-limiting where users need to trade-off latency and bandwidth for faster snapshot activation.

The steps to construct the forward map of a snapshot are the same as that of forward map reconstruction of the active tree. The only difference is that we start from the epoch number of the snapshot that needs to be activated instead of the epoch number of the active tree.

Upon activation, ioSnap produces a new writable device which resembles the snapshot (but never overwrites the snapshot). Writability of snapshots is achieved by creating a new epoch that absorbs writes to block device (which represents the state of the snapshot).

Finally, ioSnap in theory, does not impose any limit on the number of snapshots that may be activated in parallel at any given time. One may derive any number of new block devices from existing snapshots and perform I/O.

5.7 Predictable Performance

Since the design has focused on performing minimal work during snapshot create, the primary focus of predictable performance is during activation and segment cleaning as activation/cleaning can impact foreground I/O performance-and could introduce unacceptable jitters (or spikes) in foreground workloads.

To alleviate the problem, ioSnap rate-limits background I/O traffic generated by snapshot activation and segment cleaning. Background activation traffic is rate-limited by providing a tunable knob that determines the rate at which snapshots are activated at the expense of activation time. In the case of segment cleaning, ioSnap improves the vanilla rate-limiter by providing a better estimate of the amount of work that needs to be done to clean the segment as default structures do not account for snapshotted data. For example, the number of valid blocks in a segment is computed after merging the validity bitmaps of all epochs instead of only the current epoch.

5.8 Implementing Snapshot Operations

Given our understanding of the log, epochs, and the snapshot tree, we now describe the actions taken during various snapshotting activities. For *snapshot creation*, the following four actions take place. First, the application must quiesce writes before issuing a snapshot create. Second, ioSnap writes a snapshot-create note to the log indicating the epoch which was snapshotted. Third, it increments the epoch counter, and finally, adds the snapshot to the snapshot tree.

Snapshot deletion require two steps. First, ioSnap synchronously writes a snapshot-delete note to the log. The presence of the note persists the delete operation. Second, it marks the snapshot deleted in the snapshot-tree. This prevents future attempts to access the snapshot. Once a snapshot is marked deleted, the blocks from the epoch corresponding to the snapshot are eventually reclaimed by the segment cleaner in background as described earlier in Figure 6. Thus, deleting a snapshot does not directly impact performance.

The process of *snapshot activation* is more complicated as a result of our design decision to defer work to the rarer case of old-snapshot access, and requires five steps. First, ioSnap validates the existence of the requested snapshot in the snapshot tree. Second, ioSnap synchronously writes a snapshot-activate note to the log. The note ensures accurate reconstruction in the event of a crash by ensuring that the correct tree would be reconstructed to recreate the state of the system. Third, it increments the epoch counter. Activating a snapshot creates a new epoch which inherits blocks from the aforementioned snapshot. Fourth, ioSnap reconstructs the FTL and validity bitmap (see Section 5.5). Though our design permits for both readable and writable snapshots, we have only prototyped readable snapshots and do not allow parallel activations. *Snapshot deactivation* only requires writing a note on the log recording the action.

	Vanilla (MB/s)	ioSnap (MB/s)
Sequential Write	1617.34 ± 1.63	1615.47 ± 5.44
Random Write	1375.16 ± 84.6	1380.46 ± 88.9
Sequential Read	1238.28 ± 10.8	1240.51 ± 0.24
Random Read	312.15 ± 1.05	310.23 ± 0.71

Table 2. Regular operations. *The table compares the performance of the vanilla FTL driver and ioSnap for regular read and write operations. We issued 4K read and writes to the log using two threads. Writes were performed asynchronously and 16 GB of data was read or written to the log in each experiment (repeated 5 times).*

6. Evaluation

In this section, we evaluate ioSnap to determine how well we met our initial goals of (a) minimal impact on base performance, (b) minimal impact on foreground performance during snapshot creation and (c) minimal impact on foreground performance due to previously created snapshots. We also compare ioSnap with a well-known disk optimized snapshot implementation, Btrfs.

All experiments were performed on a quad core Intel i7 processor, with a 1.2 TB NAND Flash drive and 12 GB of RAM, running Linux 2.6.35 and an older generation of the VSL driver. The flash device was formatted with a sector size of 4KB (unless otherwise explicitly stated).

6.1 Baseline performance - Regular I/O operations

We employ microbenchmarks to understand the impact of snapshot support in ioSnap during regular operations. We use sequential and random read/write benchmarks to measure the performance of the vanilla VSL driver and ioSnap. Table 2 presents the results of these microbenchmarks. From the table, we see that the performance impact of running ioSnap when there is no snapshot-related activity is negligible. This is in line with our design goal of being close to the baseline device performance.

6.2 Snapshot Operations

The design of ioSnap introduces multiple overheads which may directly or indirectly impact user performance. Users may observe performance degradation or increased memory consumption due to snapshot-related activity in the background. We now discuss the implications of snapshot operations on user performance and FTL metadata.

6.2.1 Snapshot Create and Delete

A key assumption made in our design is that snapshot creation and deletion are invoked much more frequently than activation. These two operations have to be extremely fast to avoid user-visible performance degradation. To measure the performance of snapshot creation and deletion, we use the microbenchmarks shown in Table 2 and vary the amount of data before the snapshot create or delete operation. In all our experiments, we observed a latency of about 50 μs, with 4KB of metadata written to the log. This is due to the fact

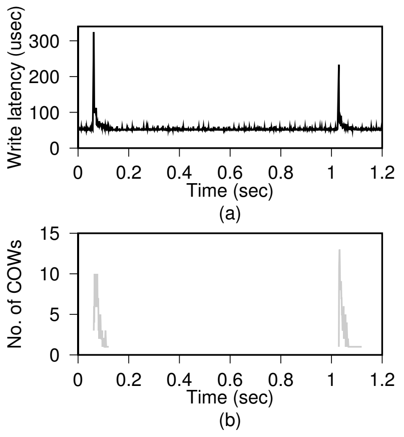

Figure 7. Impact of snapshot creation. *The figure above illustrates the impact of a snapshot creation operation on the latency observed by the user. In this experiment, we first write 512 byte blocks to arbitrary logical addresses (total of 3GB). Then we create a snapshot and continue writing blocks to random locations (totaling 8 MB). The process of creating a snapshot and writing 8 MB is then repeated. We depict the latency by the solid black line on the primary Y axis and the number of validity bitmap copy-on-write occurrences with the solid grey line on the secondary Y axis.*

that only a snapshot creation (or deletion) note is written to the log and is independent of the amount of data written. The metadata block of 4KB per snapshot is insignificant (on a 1.2 TB drive).

Though snapshot creation is extremely fast, it could impact the performance of subsequent writes. After a snapshot is created, requests which overwrite existing data result in the corresponding validity bitmap being copied (Section 5.4.1). On the other hand, read operations are not impacted after snapshot creation and snapshot deletes do not impact read or write performance.

We wrote a microbenchmark to understand the *worst-case* performance impact of CoW. In our microbenchmark, we first wrote 3GB of data to the log to populate the validity bitmaps. We then created a snapshot and issued synchronous 512 byte random-writes of 8MB data to overwrite portions of the 3GB data on the log and repeated this process a second time. Also, for this experiment, we formatted the device with 512 byte sectors to highlight the *worst-case* performance overheads. Figure 7 illustrates the impact on user observed write latency.

Latency: From Figure 7a we can see that the write latencies shoots up (to at most 350 μsec) for a brief period of time ($\approx$ 50 msec) before returning to the pre-snapshot values. The period of perturbation obviously depends on the amount of copy-on-write to be performed, which in turn depends on the total amount of data in the previous epochs. We observe similar behavior upon the creation of a second snapshot. Note that this is the *worst-case* performance (validity bitmap copy for every write) and the latency spike would be *smaller* for other workloads. The spike in this test is due to the fact that bitmaps are copied in a bursty fashion due to the lack of spatial locality in the writes.

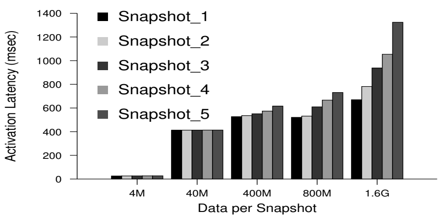

Figure 8. Snapshot activation latency. *The figure illustrates the time spent in activating snapshots of various sizes. Each cluster represents a fixed volume of data written between snapshots. For e.g., the first cluster labeled 4M represents creation of five snapshots with 4MB of data written between each snapshot create operation. The five columns in each cluster indicates the time taken to activate each of the five snapshots. Within each cluster, every column (i.e, snapshot) requires data from all the columns to its left for its activation.*

Space Overheads: Snapshot creation directly adds a metadata block in the log (described earlier) and indirectly causes metadata overheads due to additional validity bitmap pages created as part of CoW (bitmaps are kept in-memory). The validity bitmap CoW overhead is directly proportional to both data present on the log and data overwritten to distinct logical addresses after a snapshot. Figure 7b displays the count of the validity bitmaps that were copied during the execution of the micro benchmark described above. In the experiment, we observed 196 validity bitmap blocks being copied after the first snapshot, incurring an overhead of 784 KB per snapshot (or about 0.024% per snapshot). Note that the validity bitmap CoW overheads will decrease with larger block sizes as fewer validity bitmaps are copied.

In summary, snapshot creation and deletion operations are lightweight (50 usec) and add a metadata block in the log. But snapshot creation could also impact subsequent synchronous write performance because of CoW of validity bitmap pages. The amount of validity bitmaps created is directly proportional to data on the log and the distinct logical addresses that get overwritten.

6.2.2 Snapshot Activation

Snapshot activation requires a scan of the device to identify blocks associated with the snapshot being activated to recreate that snapshot's FTL. The log scan and FTL recreation incur both memory and performance overheads. Also, foreground operations could be impacted because of the activation process. We now evaluate the cost of snapshot activation.

Time overheads: The time taken to activate a snapshot directly depends on the size of the snapshot. To measure snapshot activation time, we first prepare the device by writing a fixed amount of data and then create a snapshot. We then repeat the process for 4 more times (i.e., 5 snapshots are created with equal amount of data). The amount of data written to the device was varied between 4M and 1.6GB. Figure 8 present the time spent in activating snapshots of various sizes. In this figure, each column within the cluster indicates the time spent in activating a specific snapshot.

From the figure, we make two important observations. First, the more data on the log, the longer it takes to activate a snapshot. Second, more time is required to activate snapshots that are deeper (i.e., are derived from other snapshots) in the snapshot tree. The increase in activation time is due to the fact that to create the FTL of a snapshot, all of its ancestors in the snapshot tree have to be processed in a top-down manner (Section 5.6). Upon looking at the time spent at various phases of activation, we observed that for fixed log size, irrespective of the number of snapshots present, the time taken to identify blocks associated with a snapshot is constant. The constant time is due to the fact that the segment cleaner could have moved blocks in the log and hence, the entire log needs to be read to ensure all the blocks belonging to the snapshot are identified correctly. For example, to read a log containing 8 GB of data (the column named 1.6 GB), we spend around 600 msec scanning the log followed by the reconstruction phase which may vary from 60 msec (snapshot 1, no dependencies) to 750 msec (snapshot 5, dependent on all 4 snapshots before it).

Snapshot no. activated	Size of tree at snapshot creation (MB)	Size of tree after snapshot activation (MB)
1	1.38	0.84
2	4.41	3.63
3	7.91	7.09
4	11.20	10.51
5	14.44	13.72

Table 3. Memory overheads of snapshot activation. *The table above presents the memory overheads incurred when snapshots are activated. In the experiment we created five snapshots with 1.6 GB of of random 4K block writes being issues between each subsequent snapshot create operations. After each snapshot create operation we measure the size of the tree in terms of the number of tree nodes present. After five snapshots are created, we activate each of those snapshots and measure the in-memory sizes of newly created FTL.*

Space overheads: Activation also results in memory overheads as we need to create the FTL of the activated snapshot. To measure the in-memory overheads, we first create a log with five snapshots each with 1.6 GB worth of data (using random 4K block writes). We activate each of the 5 snapshots and measure the in-memory FTL size of the activated snapshot. Table 3 presents the results.

From the table, we make two observations: first, with an increase in data present in the snapshot, the memory footprint of the new tree also increases. Second, the tree created by activation tends to be more compact than the active tree with exactly the same state. The better compaction is due to the fact that the original tree is fragmented, while the tree created by the activation is as compact as the tree can be.

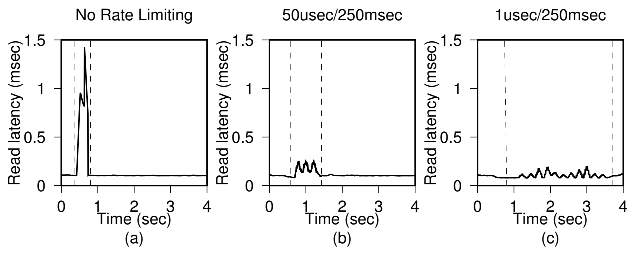

Figure 9. Random read performance during activation. *The figure above illustrates the performance overhead imposed by activation on random read operations and how rate-limiting activation can help mitigate performance impact. We perform random read operations of 4K sized blocks on the drive with 1 GB of data spread across two snapshots. About 500 msec into the workload we activate the first snapshot which contains 500 MB of data. The random reads average about 100 usec before activation. The naive performance is show in (a) and two rate-limiting schemes are shown in (b), and (c). In each of the rate-limiting scheme, the parameters shown as "x usec/y msec", meaning for every x usec of activation work done, the activation thread has to sleep for y msecs. The dashed lines in each plot indicates the time when activation started and completed.*

Impact on foreground requests: Activating a snapshot may also interfere with on-going operations. To quantify the interference, we performed a simple experiment where we issued 4K random reads with 1 GB of data spread over two snapshots. Figure 9 shows the results of the experiment. From the figure, we see that the random read latencies shoots up during activation and then stabilizes. From Figure 9a, we observe that latency has gone up by almost 10x. Such latency spikes are unacceptable for many performance sensitive enterprise applications. Next we test the impact of our rate limiting algorithms which control the amount of activation work done per unit time and trade-off activation time for foreground workload latency. Figures 9b(50usec/250msec) and 9c(1usec/250msec) show the effect of rate-limiting on activation process. From these figures, we can clearly see that the impact on read performance falls significantly with rate-limiting (worst case read latency decreases from 10x to 2x) but the time to activate increases (from 0.3 sec to 3.5 sec). The variation in latency could be further reduced by increasing the activation time.

6.3 Segment Cleaning

The segment cleaner copy-forwards valid blocks from a candidate segment. The amount of valid data may increase in the presence of snapshots since blocks invalidated in a snapshot may still be valid in its ancestors. The increase in data movement could impact both segment cleaning time and foreground user requests. To measure the impact of segment cleaning, we use a foreground random write benchmark with 4K block size that writes 5GB of data spreading across multiple segments while a background thread creates two snap-

No. of Snapshots	Overall Time (sec)	Validity Merge (msec)
Vanilla (0)	10.42	113.07
0	10.48	127.9
1	10.14	140.65
2	10.8	205.15

Table 4. Overheads of segment cleaning. *The table presents the overheads incurred while reclaiming a segment. In our experiment, a foreground thread issues 4K random writes filling up multiple segments (around 5 GB of data). Also, a background thread creates snapshots at arbitrary intervals. We force the cleaner to pick up the segment which was just written in order to measure the overheads. We present the overall time spent in cleaning the segment and the time spent in merging validity bitmaps when computing validity of data.*

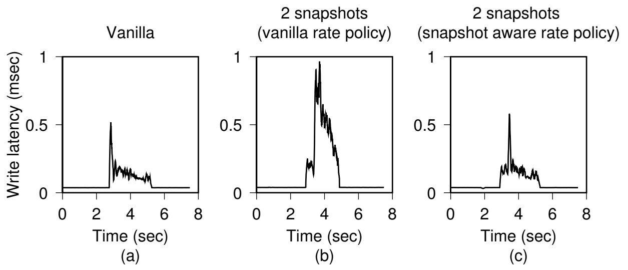

Figure 10. Impact of segment cleaner on user performance. *The figure illustrates the performance overhead imposed by segment cleaning on foreground random writes and how snapshot-aware rate-limiting can help mitigate performance impact. We perform random writes of 4KB blocks on the drive with 5 GB of data spread across two snapshots. The impact on random read performance due to vanilla segment cleaner is show in (a), ioSnap with two snapshots in (b), and ioSnap with snapshot aware rate-limiting in (c).*

shots (still within the first segment). Once the first segment is full, we force the segment cleaner to work on this segment while foreground writes continue to progress.

Table 4 show the overheads of snapshot-aware segment cleaning in ioSnap. In the presence of snapshots, the cleaner has to perform additional data movement to capture snapshotted data (568, 628, and 631 MB respectively for vanilla/zero, one and two snapshots respectively). We do not consider the additional data as an overhead. Interestingly, the segment cleaning time neither increases with the number of snapshots nor with the amount of valid data moved. The vanilla FTL has rate-limiting built into it where the segment cleaner aggressively performs work when there is more data to move. Upon closer inspection of various stages of the segment cleaner, we observed that with more snapshots present within a segment, the validity bitmap merging operation tends to get more expensive. Even in the presence of zero snapshots, we incur an overhead of 12% and this progressively grows.

Data movement introduced by the segment cleaner imposes overheads on the application visible performance and Figure 10 illustrates this impact on the aforementioned user workload. From the Figure 10a, we can see that vanilla driver

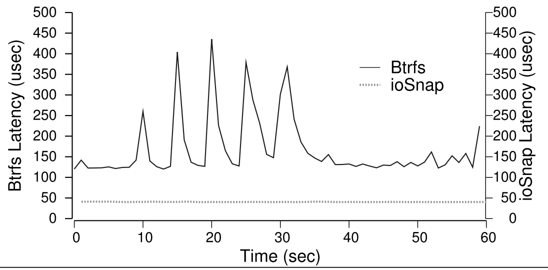

Figure 11. Impact of foreground writes latency upon snapshot creation. *The figure illustrates impact of snapshot creation on foreground write latency. After an initial sequential workload of 8 GB, a random write process is started and snapshots of the volume are create every 5 seconds. The solid line indicates the latency observed by the random write when run on top of a Btrfs file while the dashed line represents directly running on top of the ioSnap device. Though strictly not comparable, we do observe the variation relative to the baseline performance. Btrfs writes are significantly degraded (upto 3x latency increase) while ioSnap performance is fairly stable.*

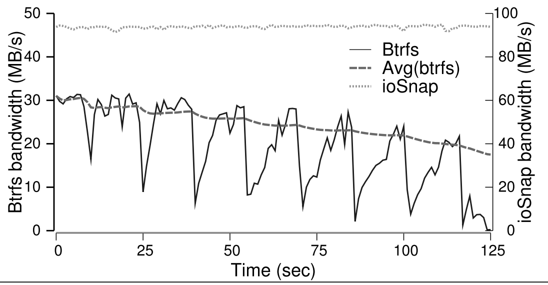

Figure 12. Impact of snapshots on sustained bandwidth. *The figure above illustrates the impact of (non-activated) snapshots on sustained write bandwidth. Initially, around 200 GB of sequential data was written and this was followed by a random workload interspersed by a snapshot once every 15 sec. The graph depicts the bandwidth observed by the writer process. The solid line (indicating Btrfs performance) depicts the slow recovery in bandwidth after a snapshot is created. The dashed line (ioSnap) does not suffer from such a problem, delivering consistent bandwidth throughout.*

does impact the write latency which is an artifact of the version of the driver we are using. We observed similar behavior in ioSnap with zero snapshots (not shown in the figure). Figure 10b shows that ioSnap increases write latency by a factor of 2. We mitigate the increase in latency by introducing snapshot awareness to the rate-limiting algorithm in the segment cleaner by providing a better estimate of the number of valid blocks in the segment containing snapshotted data. As we can see in Figure 10c, the overheads are brought back to the original levels shown in Figure 10a.

6.4 Comparison with a disk optimized snapshot system

We now compare ioSnap performance with that of Btrfs, a well known disk optimized file system which has built-in snapshots. Both systems are run on the same flash hardware as outlined earlier in the section. The first figure shows the performance of Btrfs and ioSnap for a foreground workload while snapshots are being created in the background. Since the two systems are very different architectures, we cannot compare their baseline performances. However, we can compare how much deviation from baseline occurs during a snapshot creation. With Btrfs, the foreground latency impact of a snapshot create is very visible (Figure 11, with performance degrading as much as 3x from the baseline. For ioSnap however, the performance variation is negligible, about 5% from the baseline.

The second graph (Figure 12) shows the impact of created (but not actively accessed) snapshots in both cases. In this experiment, after an initial write of 200GB, snapshots are created every 15 seconds while random write traffic runs in the foreground. The disk optimized snapshot in Btrfs shows increasing time to recover from snapshot create as the number of snapshots build up (as indicated by the gradually declining sustained bandwidth), while ioSnap shows virtually no degradation as snapshot count increases.

7. Discussion and Future Work

In this section, we discuss the implications of our design choices, and additional areas for improvement revealed by our evaluation.

Unlimited snapshots: ioSnap is able to support as many snapshots as the physical media size allows. We did however learn that the interdependency between successive snapshots, while not impacting the performance of creation or foreground workloads, can increase the latency of snapshot activation. With increasing snapshots, validity bitmap CoW and merge overheads may also increase. We believe this problem can be handled by employing snapshot-aware segment selection policies (outlined in Section 5.4.2) to help minimize intermixing of data belonging to different snapshots.

Maintaining primary device performance: Performance of regular I/O operations with ioSnap were virtually indistinguishable from the performance of the same operations on the baseline device. We also saw that once snapshot-aware segment cleaning was introduced, the performance of regular I/O operations during segment cleaning was largely similar with and without ioSnap.

Predictable performance: Snapshot operations are fast and have minimal impact on foreground operation. Furthermore, the presence of dormant snapshots should not impact normal foreground operation. Both of these goals were met, with the comparison with Btrfs showing that the flash-optimized design can deliver far superior performance predictability than a disk-optimized design.

Tolerable memory and storage consumption: Memory consumption increases as snapshots are activated, but remains low for dormant snapshots. This implies that the presence of snapshots does not really increase memory consumption, since the FTL in the absence of snapshots could consume the same amount of memory if all of the physical capacity was actively accessed as a normal volume. Also, the snapshot metadata storage consumption is fixed (4K or a block for the snapshot note) and does not impose constraints on the number of snapshots.

The design choices we made to separate snapshot creation from activation and focus heavily on the optimization of snapshot creation helped us meet the above goals. However, the evaluation shows clearly that deferring activation comes at a price. In the worst case, activations may spend tens of seconds reading the full device and in the process consume memory to accommodate a second FTL tree (which shares no memory with the active tree). Activations can be further optimized by selectively scanning only those segments that have data corresponding to the snapshot. Also, rate-limiting can help control the impact on foreground processes (both during activation and segment cleaning) making these background tasks either fast or invisible, but not both. Improving the performance and memory consumption of activation is a focus of our future work.

Generality: While we integrated our flash aware snapshots into a specific FTL, we believe most of the design choices we made are applicable to FTLs in general. We mostly relied upon the Remap-on-Write property, which is required for all NAND Flash management. The notion of an epoch number to identify blocks of a snapshot can still be used to preserve time ordering even if the FTL was not log structured. Other data structures such as the snapshot tree and CoW validity bitmaps, are new to the design and could be added to any FTL. The snapshot aware rate limiting and garbage collection algorithms were not specifically designed for the VSL's garbage collector. However, since each FTL has its own specialized garbage collector, if similar techniques were applied to another FTL, we expect that they would need to be integrated in a manner optimal for that FTL.

ioSnap's design choices represent just one of many approaches one may adopt while designing snapshots for flash. Clearly, some issues with our design, such as the activation and validity-bitmap CoW overheads, need to be addressed. Most systems allow instantaneous activation by always maintaining mappings to snapshots in the active metadata [1, 11, 22, 27], but this approach does not scale. Activation overheads may be reduced by precomputing some portions of the FTL and checkpointing the FTL on the log. The segment cleaner may also assist in this process by using policies like hot/cold [23] to reduce epoch intermixing, thereby localizing data read during activation. Finally, keeping snapshots on flash for prolonged durations is not necessarily the best use of the SSD. Thus, schemes to destage snapshots to archival disks are required. Checkpointed (precomputed) metadata can hasten this process by allowing the backup manager to identify blocks belonging to a snapshot.

8. Conclusions

In summary, ioSnap successfully brings together two capabilities that rely upon similar Remap-on-Write techniques, flash management in an FTL and snapshotting. Given the RoW nature of an FTL, one could easily assume adding snapshots would be trivial, which unfortunately is far from the truth. Consumers of flash-based systems have come to expect a new, more stringent level of both baseline predictable performance. In ioSnap, we have explored a series of design choices which guarantee negligible impact to common case performance, while deferring infrequent tasks. Unfortunately, deferring tasks like activation does not come for free. Thus, ioSnap reveals performance trade-offs that one must consider while designing snapshots for flash.

9. Acknowledgements

We thank the anonymous reviewers and Robert Morris (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the Advanced Development Group at Fusion-io for their insightful comments.

References

[1] Btrfs Design. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html`, 2011.

[2] Nitin Agarwal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[3] S. Agarwal, D. Borthakur, and I. Stoica. Snapshots in hadoop distributed file system. *UC Berkeley Technical Report UCB/EECS*, 2011.

[4] Joel Bartlett, Wendy Bartlett, Richard Carr, Dave Garcia, Jim Gray, Robert Horst, Robert Jardine, Doug Jewett, Dan Lenoski, and Dix McGuire. The Tandem Case: Fault Tolerance in Tandem Computer Systems. In Daniel P. Siewiorek and Robert S. Swarz, editors, *Reliable Computer Systems - Design and Evaluation*, chapter 8, pages 586–648. AK Peters, Ltd., October 1998.

[5] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 217–228, Washington, DC, March 2009.

[6] FusionIO. Fusion-io directCache. http://www.fusionio.com/data-sheets/directcache/, 2011.

[7] FusionIO. IO-Drive Octal. http://www.fusionio.com/products/iodrive-octal/, 2011.

[8] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. FAST'12, 2012.

[9] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[10] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[11] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[12] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. Bvssd: build built-in versioning flash-based solid state drives. SYSTOR '12, 2012.

[13] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS OSR*.

[14] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

[15] Charles B. Morrey III and Dirk Grunwald. Peabody: The time travelling disk. In *Proc. of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, 2003.

[16] NetApp. NetApp Performance Acceleration Module. http://www.netapp.com/us/communities/tech-ontap/pam.html/, 2008.

[17] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[18] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[19] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. NSDI'06.

[20] Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine. Persifs: a versioned file system with an efficient representation. SOSP '05.

[21] PureStorage. ZeroSnap snapshots. http://www.purestorage.com/flash-array/resilience.html, 2013.

[22] S. Quinlan and JMKR Cox. Fossil, an archival file server.

[23] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[24] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding When To Forget In The Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Kiawah Island Resort, South Carolina, December 1999.

[25] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. FAST '03.

[26] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. OSDI'00.

[27] Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, and Sang Lyul Min. Ltftl: lightweight time-shift flash translation layer for flash memory based embedded storage. EMSOFT '08.

[28] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration for Load Balancing in Self-managing Storage Systems. In *The 3rd IEEE International Conference on Autonomic Computing (ICAC '06)*, Dublin, Ireland, June 2006.

[29] Christopher Whitaker, J. Stuart Bailey, and Rod D. W. Widdowson. Design of the server for the Spiralog file system. *Digital Technical Journal*, 8(2), 1996.

[30] Jake Wires and Michael J. Feeley. Secure file system versioning at the block level. EuroSys '07.

[31] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. Hec: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, page 10. ACM, 2013.

[32] Qing Yang, Weijun Xiao, and Jin Ren. Trap-array: A disk array architecture providing timely recovery to any point-in-time. *SIGARCH*, May 2006.

[33] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.