Muninn: a Versioning Flash Key-Value Store Using an Object-based Storage Model

Yangwook Kang Rekha Pitchumani Thomas Marlette Ethan L. Miller
Storage Systems Research Center, University of California, Santa Cruz
{ywkang,rekhap,tmarlette,elm}@cs.ucsc.edu

Abstract

While non-volatile memory (NVRAM) devices have the potential to alleviate the trade-off between performance, scalability, and energy in storage and memory subsystems, a block interface and storage subsystems designed for slow I/O devices make it difficult to efficiently exploit NVRAMs in a portable and extensible way.

We propose an object-based storage model as a way of addressing the shortfalls of the current interfaces. Through the design of Muninn, an object-based versioning key-value store, we demonstrate that an in-device NVRAM management layer can be as efficient as that of NVRAM-aware key-value stores while not requiring host resources or host changes, and enabling tightly-coupled optimizations. As a key-value store, Muninn is designed to achieve better lifetime and low read/write amplification by eliminating internal data movements and per-object metadata updates using Bloom filters and hash functions. By doing so, it achieves as little as 1.5 flash page reads per look up and 0.26 flash page writes per insert on average with 50 million 1 KB key-value pairs without incurring data re-organization. This is close to the minimum number of flash accesses required to read and store the 1 KB key-value pairs, thus increasing performance and lifetime of flash media.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management

General Terms Design, Experimentation, Measurement

Keywords Object-based storage device, Flash memory, Key-value store, Versioning

1. Introduction

Non-Volatile Random Access Memories (NVRAMs) are becoming increasingly important in the storage hierarchy as the need for energy-efficient and high performance storage increases in both consumer and enterprise markets. Consumer products, such as laptops and smart phones, are adopting flash memory to enhance their battery life and response time instead of hard drives. Enterprise SSDs (Solid State Disks) are used as replacements for 10,000/15,000 RPM hard drives. Beyond flash memory, several other types of non-volatile memories such as Phase Change RAM (PRAM), Spin-Torque Transfer RAM (STT-RAM), and memristors are competing to be the future storage and/or memory medium [26].

Despite the increasing importance of NVRAMs, storage and memory subsystems in current operating systems are not yet fully ready to adopt this technology shift, because of the assumption of slow block-based I/O devices in the design of core I/O subsystems [8]. At the device level, the design of device subsystems such as mapping and wear-leveling is restricted and complicated due to the limited flexibility of a block-based interface, providing sub-optimal performance [22]. Several optimizations such as *nameless writes* [33] and DFS [20] are proposed to alleviate these efficiency issues, but they require changes in the data management layer in the host systems whenever a new type of NVRAM becomes available. Supporting heterogeneous devices would be even more difficult, because it may require multiple management layers or extended file systems for the NVRAM.

We propose the use of the object-based storage model for NVRAMs to address the shortcomings of current NVRAM interfaces. This model offloads the NVRAM data management layer from a file system to a device and provides an object interface, which supports variable-length requests. By isolating the NVRAM specific technology behind a rich object interface, it allows a file system to be independent from the underlying storage medium, enabling an easy transition between different NVRAM technologies. Heterogeneous NVRAM devices and hybrid NVRAM devices such as PRAM-Flash hybrid and Flash-HDD hybrid can also be

transparently supported without redesigning or modifying a local file system. Performance-wise, the in-device NVRAM management layer can be as efficient as native file systems designed for a specific type of NVRAM, while allowing tightly-coupled hardware optimizations and eliminating the duplicate translation layers in the data path. In addition, its capability of handling variable-length objects and the associated metadata allows storage systems to support a wide variety of device types including key-value stores and active disks without altering a device-host interface.

We designed Muninn to demonstrate the design flexibility, efficiency, and extensibility of the object based storage model as a new storage interface for non-volatile storage devices. For extensibility, we show Muninn can add new features to existing file systems or applications without altering them. To demonstrate this, we add a versioning support to Muninn that can be transparently applied to existing file systems. For efficiency and flexibility, in the design of key-value management policies, we try improve read and write amplifications of a key-value store by eliminating on-flash per-object metadata and internal data movements using Bloom Filters and hash-based data placement. This may incur more read operations when compared to the mapping layer in FTLs due to the possibility of false positives in the bloom filter, however we expect it could increase the lifetime of devices by reducing the number of writes and frequent cleaning. Additionally, Muninn can help a host system achieve better scalability; the host no longer needs to manage the exact mappings between a key and a logical block number per device.

Compared to host-side key-value stores like SILT [27], Muninn can be configured to fit in a dedicated in-device memory, requiring no internal data movement on flash. By directly managing flash memory, it can avoid the overheads from having two index structures; one for key-value store, and another for SSDs. Our results show that without requiring any background operation, Muninn achieves 1.5 page reads per look up and 0.26 page writes per insert on average while inserting 50 million 1 KB key-value pairs and reading 25 million pairs back on a flash memory with 4 KB pages. Considering that the minimum numbers of flash accesses for just reading and storing one key-value pair are 1 page read and 0.25 page write in this configuration, it shows that the average flash storage management overhead can be minimized while providing additional features such as versioning and compression. We make two contributions in this paper:

- We design Muninn, which is a versioning key-value store built on the object-based storage model. We show that Muninn can add versioning transparent to existing file systems while achieving low read/write amplifications.
- We introduce the hash-based data placement policy for flash memory, which eliminates the need for per-object metadata and a direct logical to physical mapping, reducing write amplification.

2. Background and Related Work

We first describe the evolution of NVRAM storage systems, focusing on their design issues and interface changes. We then discuss versioning file systems, key-value stores, and Bloom filters that influence the design of our Muninn.

2.1 Evolution of NVRAM System Designs

As NAND flash memory became cheaper and large enough to replace programmable read-only memory and battery-backed SRAMs in late 90s, and eventually data storages in mid 2000s, many flash-aware file systems such as YAFFS [1], UBIFS [19], and RCFFS [21] were designed and implemented for embedded systems. Since both the file system and hardware are deployed together, flash-aware file systems provided tightly-coupled optimization, an efficient placement and cleaning policies. To further improve the lifetime and performance of these devices, several file system specific extensions using byte-addressable NVRAM have been studied [14, 24]. Condit *et al.* propose a PRAM file system that directly places byte-addressable NVRAMs such as PRAM and MRAM on the main memory bus, and uses them as a backing store [11]. While these systems have the most efficient architecture to handle NVRAMs, they are mostly used in embedded systems or custom designed systems due to their limited portability, compatibility, and dependency between a file system and a device.

In modern SSDs, the compatibility issue has been solved by adding an indirection layer between a legacy file system and storage medium in a device. This sector-to-page mapping layer, called the Flash Translation Layer (FTL), allows legacy file systems to access an SSD as a block-based storage device. However, due to the lack of file system semantics, the efficiency of data structures in SSDs is typically lower than that in flash-aware file systems. In the data path, while the data allocation layer in a legacy file system is still processed incurring some overhead, its results are remapped and not used inside the device.

There have been many efforts to improve the efficiency of FTLs, mostly focusing on detecting and handling various types of workloads [10, 30]. Some of the recent approaches also look at the contents of the workload to further optimize the device [9, 17].

While these optimizations can improve the inefficiency of FTLs, the problems with duplicate translation layers and limited file system semantics cannot be solved without an interface change. Nameless writes [33] remove a file system address translation by extending a block interface to inform the file system whenever the location of data changes. DFS [20] uses an another approach to this problem, which moves the flash translation layer to the virtualized flash layer in a host operating system. In these approaches, however, significant changes in the operating systems code are required, and more importantly, these changes are not independent from the NVRAM medium.

The object-based storage model has been used mostly in large-scale distributed storage systems to improve scalability by adding intelligence to the devices [7, 15]. For use as a NVRAM interface, Rajimwale *et al.* [31] suggest some optimization techniques such as informed cleaning, and priority I/O scheduling. Kang *et al.* [22] study the effects of various data placement policies in this model, and other flash-oriented optimizations. We show that this model can support various types of NVRAM I/O devices including a specialized device like a key-value store, providing a new feature to the system in a portable and efficient way.

2.2 Versioning Flash Systems

Since flash memory requires out-of-place updates, every flash device has multiple versions of the modified data at least for some time. However, not many flash storage systems offer versioning, because they must preserve the write order, which is not easy in most FTL schemes. They also need to store metadata and provide additional user commands such as snapshot and rollback. For example, Lightweight Time-shift FTL (LTFTL) [32] can create version states at any time and go back in time to desired states. However, because it is based on block-based FTL, snapshots can be supported, but finer-grained full versioning such as per-file versioning and per-directory versioning is not possible. In contrast, our object-based versioning system provides both a per-object versioning and a system-wide snapshot transparently using the flash space that was already used. Similar to Muninn, Tango [4] provides snapshots, but it is designed to support consistent replication without using complex distributed protocols, not focusing on directly managing flash memory or providing per-object versioning.

2.3 Key-Value Stores

Key-value stores are specialized file systems optimized for insert and retrieval of small data associated with a fixed length key, replacing the need for complex database systems. These key-value stores are designed to run on a host using SSDs, so reducing host system resource usage is one of the main design factors in this system to achieve not only performance, but also scalability. FlashStore [12] is a key-value store used as a cache for hard-disk based key-value storage system. It uses a single in-memory hash table to index all keys on flash and hence achieve one read per lookup. SkimpyStash [13] indexes the flash via a hash table with linear chaining to achieve low memory footprint. However, it requires on average 5 flash reads per lookup. BufferHash [3] maintains multiple hash tables—one in memory and the others on flash memory— and uses a small set of Bloom filters (BF) to indicate whether a key might be present in a hash table. BloomStore [28] is a recent key-value store based entirely on Bloom filters. They append incoming key-value pairs to Flash page and maintains one BF per flash page. Hence, the system contains many BFs and lookups have to search all the BFs in parallel and in batches to locate a key.

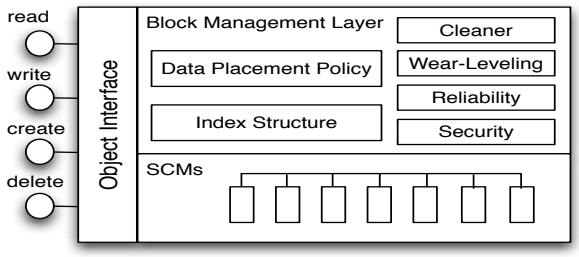


Figure 1. NVRAM OSD exposing T10 object interface

Aiming to achieve low memory footprint, they store large amounts of older BFs in flash and read them in for lookups, increasing read amplification. The increased false positive rate of having multiple BFs has not been addressed properly. SILT [27] achieves a low memory footprint and a low read and write amplification using log store, hash store and sorted store. However, the number of flash read and write accesses can be much higher than a per-request amplification factor due to background processes, moving data between stores. Also, its read/write amplification factor does not consider the page accesses by an SSD. Supporting failure recovery and deletion can be also difficult. In contrast, our system stores each key-value pair only once and does not use any background process to manage the written key-value pair.

2.4 Hashing and Bloom filters

The characteristics of flash memory and byte-addressable NVRAMs make hashing an interesting choice today, because the access latency is the same regardless of its address. In addition, it can minimize the metadata overhead by not storing the actual key-page mapping. Our scheme uses multiple hash functions associated with multiple BFs [6] to achieve better space utilization.

Among various types of BFs, our Bloom filter structures are similar to Dynamic BFs [16] and Scalable BFs [2] in a way that multiple small-size BFs exist instead of a large BF. Dynamic BFs increase or decrease their sizes as the number of keys in each BF changes. To reduce the compounded error probability, the false positive rate of each Bloom filter keeps decreasing as the number of BFs increases in Scalable BFs. Muninn's per segment BF is most similar in design to the BF based summaries described in [25].

3. Object-based Storage Model

The object-based model [15] consists of two main components: an object-based file system and an object-based storage device (OSD). Unlike a typical file system, an object-based file system only provides the name resolution, offloading the storage management layer to the OSD. By isolating device-specific technology behind a metadata rich object interface, the file system becomes independent of the underlying storage medium while being able to deliver full file

system level semantics to the devices. Thus, the NVRAM management layer in the device can be designed similar to that of a native NVRAM-aware file system while enabling co-optimization between the data management layer and NVRAM hardware. In addition, a single object-based file system can support multiple heterogeneous OSDs, enabling a drop-in replacement for new types of NVRAM devices without altering any host subsystem.

When a hybrid NVRAM device becomes available, for example, both NVRAM-aware systems and FTL-based systems would require a host system change. A host file system and its I/O subsystems need to understand the characteristics of multiple NVRAM medium in NVRAM-aware systems. FTL-based systems would require an intelligent layer in a host system that can deliver file-system or user-level semantics to the device, otherwise, the use of NVRAMs is limited to a write buffer for file system blocks. *Nameless writes* [33], is one way of delivering semantics. However, it will still require a host system change whenever a target NVRAM media changes.

Interface The file system and OSDs communicate via an object-interface, which exposes various types of object commands as depicted in Figure 1. Each operation takes an object, which includes variable-length data and metadata associated with the data, describing one file-level or user-level I/O request. By exposing various operations to capture file system operations, this eliminates the needs for host system utilities such as TRIM, which informs the devices whenever a file is deleted. The format of an object and types of object commands are standardized by ANSI T10 [29].

However, one of the major issues of the current standard is that basic object I/O operations such as read, write and collection are supported, limiting the range of available device features. For example, an in-device search operation, as introduced in [23], cannot be represented easily with the standard command sets. Therefore, we suggest allowing devices to *publish* their features to a host system when mounting, and letting a host *subscribe* the features to be used. Additionally, an *execute* command are required to use a feature on demand.

Flat namespace Objects are identified by a 128-bit unique identifier, not by human-readable names. There is no hierarchy between objects, offering more flexibility when assigning and distributing objects. For example, the file system can assign a certain range of object identifiers to each distributed node so it can retrieve the node that stores an object by looking at its identifier, making searching and sharing easy. However, this does not prevent a host system from creating a hierarchy on top of OSDs, because directory hierarchies and name resolution is still managed by a host system. Our versioning system does not use a hierarchy, but we encoded a version information in an object identifier so users can easily control versions of key-value pairs without any support from a host file system or an additional library.

User interface The object-based storage model can support both POSIX interface and an object-interface as a user interface. When the POSIX interface is used, the Virtual File System (VFS) in an operating system passes an inode number and an offset to an object-based file system. Then, the file system generates an object identifier, and sends data, and necessary metadata for the given type of request to the OSD via an object interface. When an application directly accesses the OSD via an object interface, the application can send higher level information to optimize its data path. For example, full text searching can be executed within a device, sending only the results of the search to the host, not the whole contents of an object [23].

Cost While providing advanced and efficient data processing in this model, the hardware manufacturing cost of an object-based device can remain as low as SSDs, because SSDs already have multiple powerful embedded processors, large memory, and multiple high-bandwidth independent I/O channels to the underlying medium to process mapping and wear-leveling [23]. Handling object requests and the sparse namespace are the only additional overhead to the SSD. Also, considering the estimated cost of an iPhone 4 processor is around $10 [18], adding more powerful processors for advanced data management would not increase the cost of OSDs much.

4. Muninn: Versioning Key-Value Store

We designed Muninn to demonstrate the design flexibility, efficiency, and extensibility of the object based storage model as a new storage interface for non-volatile storage devices. To show the extensibility of the object interface, we transparently bring new features to existing file systems or applications in an efficient way. We show design flexibility and efficiency through our design of key-value management policies that forego traditional logical-to-physical mapping layers in favor of bloom filters and hash based data placement. It data management layer is designed to reduce the average read and write amplification and improve life-time of a device by reducing metadata updates sacrificing some read performance for not frequently accessed objects.

The design constraints of Muninn are different than that of most host-side key-value stores. While lowering per-key memory usage is a primary goal of most host-side key-value stores, Muninn makes this memory utilization a configurable parameter because a in-device memory is used to process key-value pairs, not shared by other devices or processes. Thus, instead of moving and sorting data around the multiple internal stores for reduced memory usage, in-device key-value stores can focus on increasing life-time by reducing unnecessary data movements. Additionally, data placement and cleaning polices can be specialized for the purpose of the device, selectively cleaning data blocks and reducing the number of reads or writes per key.

Muninn adds versioning transparent to the existing file system while being accessible from file system utilities through an object interface for advanced management. A versioning feature was chosen because a history of updates can be maintained at a low cost in flash memory where overwritten data is remained until it is cleaned due to out-of-place update requirements. We use a chain of BFs to preserve the update history, and find an object. Similarly, instead of storing the physical address of an object, we use a hash function to place and find data in a flash block.

We first describe how the host systems and device applications communicate, and then explain how we insert and search data in flash memory using hash functions and Bloom filters in Section 4.2 and 4.3. We discuss how we preserve the write order to support versioning in Section 4.4. Lastly, we explain the design of a merger and the consistency of in-memory metadata.

4.1 Host-Device Communications

Since an OSD can be thought of as a key-value store that supports a fixed-length key and a variable-length value, file systems or applications can use existing *read* and *write* operations to access key-value pairs. However, we need some extra commands to manage versions. For example, mounting a device with a specific version or undoing some changes on a specific object requires a special command. To support this, we make use of an *execute* as described in Section 3.

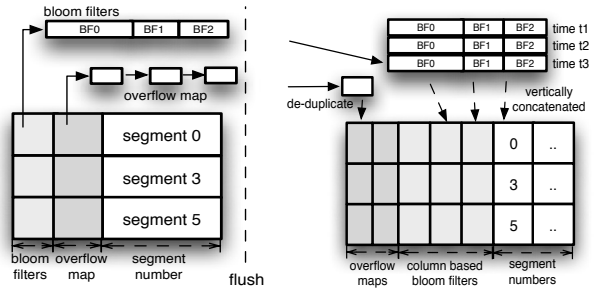
A version of an object is encoded in an object ID, not stored as metadata to eliminate the need for keeping object metadata on flash. We use a 64-bit object ID in Muninn, which consists of a 32-bit identifier, a 16-bit version number, and a 16-bit offset; it can hold up to 4 billion objects and provide up to 64K different versions to each object. The offset is used by the device to split large objects into multiple pieces; each object can have up to 64K flash pages.

To snapshot and revert, file systems can increase or decrease the version number in an object ID; it can maintain a global version number that is incremented periodically so all objects that are written in the same period of time can have the same version number. To rollback some changes for an object, users can set the negative version number, which represents the number of modifications to be cancelled.

We support legacy file systems by constructing an object ID from a partition ID and a LBA; a partition ID becomes an identifier and a LBA is considered as an offset of an object. This conversion is done by the kernel module that runs between a file system and a device. Additionally, it takes an *ioctl* command that allows applications to modify the current version number.

4.2 Hash-based Data Placement

Muninn places key-value pairs using hash functions to eliminate a direct mapping between logical and physical addresses and the need for per key-value pair metadata. When initializing a device, a flash memory is logically split into



(a) Active version table (b) read-only version table

Figure 2. Overview of Muninn

fixed-length segments whose size is a multiple of a flash erase block. Each segment consists of a fixed number of flash pages, which is a minimum unit of writing in flash memory. This ensures that no key-value pairs are written across segments, allowing segments to be erased independently. Therefore, the physical address of a key-value pair can be represented by a segment number, a page offset (within a segment), and an offset (within a page).

On writes, the physical address of a key-value pair is determined by hash functions and only the raw key-value pair is written to flash memory. In-memory dirty data for searching will be flushed later for consistency as described in Section 4.5. However, using hash-based placement brings several issues in flash storage system design. First, when a hash collision happens, the system needs a way to relocate the key-value pair and remember the new location. Second, distributing key-value pairs across the entire flash device is not practical. This is because it would require an in-memory write buffer for every flash page if the size of a key-value pair is not exactly the same as flash page. Third, segment utilization can be low under a hot-cold workload where some objects are more popular than others, generating many collisions in the same location.

4.2.1 Insert and Delete

To address the issue with key-value pair distribution, we maintain a small group of active segments in the data structure called *active version table* where key-value pairs can be written as depicted in Figure 2(a). Each row contains information about a segment, and the segment that stores a key-value pair is determined by using the last one byte of a hash of a key. Having multiple rows has two purposes: distributing hot keys across a small set of segments lowering the chance of collisions, and reducing the search space to find a key-value pair.

Within a segment, Muninn addresses collisions using multiple BFs and associated hash function to give multiple locations to frequently updated keys within a segment. More specifically, as shown in Figure 3, if a BF becomes full or the corresponding page offset is already written, the hash function associated with the next Bloom filter is used to

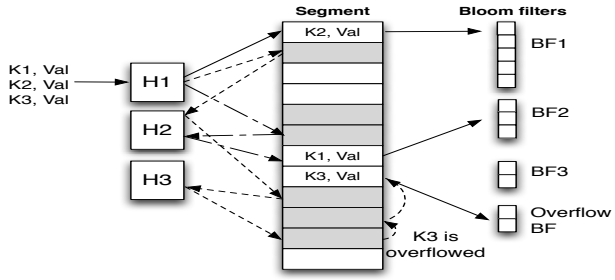


Figure 3. Insert a key-value pair to an active version table

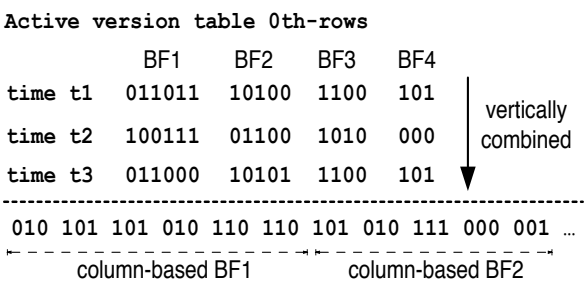


Figure 4. Flushing BFs to a read-only table

place data. For example, if a key exists in the first and second BFs of a segment, we know that the same key-value pair has written twice, and the hash function associated with the second bloom filter contains the latest version. Once the hash function to use is determined, we take the modulo operation between the output of the hash function and the number of pages in a segment to determine a page offset.

When a key cannot be placed by using hash functions associated with BFs, the key is considered an *overflowed* key. Muninn stores a hash of a key and a n-bit bitmap for each overflowed key to provide additionally n different hash functions each overflowed key can use; the position of a bit in a bitmap represents the ID of hash functions.

By giving multiple locations to hot keys, it can keep the segment utilization high even under hot-cold workloads, however since the overflowed keys require at least 6 bytes per key, the maximum size of an overflow map is fixed, and the size of normal BFs is configured to minimize the size of an overflow Bloom filter and an overflow map.

To preserve the write order, we sequentially allocate key-value pairs in a bitmap so the last bit set indicates the last hash function ID for the key. Similarly, key-value pairs are written sequentially within a page so a larger offset represents later in time. When storing a variable-length value, it additionally stores the size of the data into the page.

Deleting a key is the same as inserting the key with a special value indicating that the key is deleted. Thus, when a search function finds the pair with this special pattern as the most recent result, it returns not-found, instead of this special value.

4.3 Search

To find the physical location of a key-value pair, Muninn needs to check multiple BFs to identify the hash function originally used to place the key-value pair. However, searching individual BFs is an inefficient operation because up to one word access might be required for each bit checked. Therefore, we adopt another data structures called the *read-only version table*, which is optimized for searching multiple BFs at a time as shown in 2(b).

Once there is not enough space to place a key-value pair in a segment in an active version table, the corresponding row is flushed to the read-only version table. Since the key-value pair is already written to the segment, only the BFs and an overflow map are moved to the table. First, the flushed standard BFs are added to the column-based BFs where multiple BFs can be searched at once with a minimum number of memory word accesses. Figure 4 shows the process of combining the the three sets of BFs of row 0, flushed at time $t1, t2$, and $t3$. The bits of BF1 at position k, get combined to form a 3-bit word at position k in the column-based BF1.

The column-based BFs improve search performance by converting bit accesses to byte or word accesses. The number of BFs to be combined is determined by considering the memory access efficiency; the size of each column can be byte-aligned (8) or word-aligned (32 or 64) depending on the target architecture. In Muninn, we use a word size column, because it is the most efficient in terms of a cache-line efficiency.

To find a key-value pair, Muninn searches the active version table and the read-only version tables. Muninn first determines the row index using the hash of the key, and then searches the multiple BFs of the row in a reverse order. This is because a Bloom filter with higher index contains a newer version of the key. When the key is found in the BFs for overflowed keys, it retrieves the physical address of the key from the overflow map. If the key is in the BFs for non-overflowed keys, it uses the corresponding hash function to calculate the page address.

When the key is not found in the active version table, BFs in read-only version tables are searched. It is similar to search standard BFs, but each bit in a standard bloom filter becomes a set of bits that came from each standard BF, added to the combined BF. Checking whether a key is present in a single Bloom filter involves examining whether k bit positions determined by k hash functions are all set. Hence, for each read-only version table, k columns determined by k hash functions are read, and a bitwise AND operation is performed. The 1s in this result vector represents the indices of the BFs that may contain the key-value pair. It becomes then straightforward to retrieve the segment number and the page offset for the key-value pair from the table.

Performance-wise, searching for a non-existing key would show the worst case performance, because Muninn has to

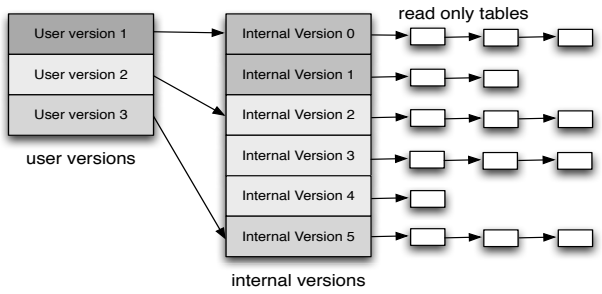


Figure 5. User versions and internal versions

read all read-only only tables belongs to the current version. The best case performance can be achieved when the keys are found in one of the recent read-only tables. This means newer key-value pairs can be quickly retrieved, and a older key-value pairs that never updated would require multiple flash page reads.

The number of unnecessary reads is also affected by the false-positive rate of each BF, and the number of BFs to be searched to find a key. Thus, the size of the memory, number of table rows, and a decreasing factor of a false positive rate should be carefully set, considering the trade off between the number of reads and the size of memory. For example, in consumer products, it could be configured to have high false-positive rate, saving memory space but sacrificing more reads. Caching some old, but not updated objects can be cached or rewritten for better performance and memory usage. In enterprise products, all version tables can reside in memory with low false positive rate. The effects of each variable are analyzed in Section 5 and evaluated in Section 6.

4.4 Version Management Layer

To support snapshots and rollback, we have two data structures; a user version provides a snapshot and an internal version is a merge unit that contains a time-ordered list of a fixed number of read-only tables. User version increases as a version number in an object ID increases, and internal version increases when the number of read-only tables added exceeds a threshold, as shown in Figure 5.

When all rows in a read-only table become full, it is added to the head of a time-ordered list of the current internal version and a new read-only table is created. When searching for a key-value pair, it searches the read-only tables following the list of the current internal version. If a key is not found, it keeps trying to search the previous versions until a match is found.

When a user version increased, the active, read-only version tables, and current internal version are finalized to create a snapshot. A user node is then created to store the internal, just finalized, version number. This represents the highest internal version number users can access in the version; any internal version that is lower than this can be accessed by this user version. For example, if the user version number is 2, it can access internal versions from 0 to 2.

Reverting to one of the previous snapshots would just require changing a user version number. If a user wants to undo N changes to an object, a device searches the key-value pair skipping the N objects, and then rewrites the object so it can be searched first next time.

4.4.1 Version-aware Merge

Muninn does not automatically reclaim space because it needs to keep the old versions of key-value pairs, including deleted ones. However, it can merge old versions upon a user request to free some space and achieve better search performance.

The unit of merging in Muninn is the internal version node, which is designed to have a fixed number of read-only tables that are written around the same period. The merge process tries to check the liveness of key-value pairs and reclaim space for old versions except for the newest one in an internal version. We use this fixed partitioning because the liveness of the key-value pair cannot be determined without looking at the other segments that might contain the same key, but searching the entire segment for merging would incur lots of read overhead.

The version-aware merger uses two thresholds to select the target internal version and the segments: internal-version utilization and segment utilization. The internal-version utilization is calculated by performing the bitwise AND operation among all the combined BFs in the same internal version, and counting the bit sets. Since the same key will set the same bit position in BFs, the number of bit sets can indicate the number of the same keys across multiple read-only tables.

After selecting the target internal version, it searches the live keys from the read-only version tables belong to the target internal version considering the utilization of each segment. The segment utilization is estimated, when the corresponding row is flushed to a read-only table, by the sum of the number of keys inserted to each Bloom filter, and the number of hash collisions in the row. The number of keys will indicate the empty space not used by any keys, and the number of hash collisions represents the possibility of the existence of duplicated keys. The effects of these two merging thresholds are evaluated in Section 6.

4.5 Consistency

If capacitors in a device are not big enough to dump all in-memory data structures to flash on a power failure, Muninn can be configured to store a read-only version table, a user version, and an internal version as soon as they become fully written for metadata consistency. Thus, during runtime, only one active version table and one read-only version table remained dirty. Since they are smaller than most FTL data structures, we can safely assume that they can be dumped to flash upon a power failure.

Symbol	Description
fpr_{ini}	Initial false positive rate
nh	Number of hash functions for data placement
no	Number of overflow BFs
r	False Positive Rate reduction rate
nr	Number of table rows
nrc	Number of rows combined to form a read only table

Table 1. Design parameters

5. Analysis

We discuss the three important design parameters in Muninn: segment utilization (SU), false positive rate (FPR), and memory usage (MU). The total amount of physical memory of a Muninn device needs to be chosen depending on a desired FPR and SU. We explain the relationship between these parameters in this Section, and show the sensitivity of these variables with the experimental results in the next Section. The adjustable parameters of Muninn are summarized in Table 1.

Segment Utilization (SU) SU depends on the utilization U achievable for the said set of keys, nh, the number of hash functions used for data placement and no, number of overflow BFs.

$$SU = \frac{data\ written\ to\ a\ flash\ segment}{segment\ size} \tag{1}$$

$$SU \propto nh \cdot U_{hash} + no \cdot U_{overflowmap} \tag{2}$$

False Positive Rate (FPR) As more read-only version tables are generated, Muninn needs to search more BFs. Lets look at a case where fpr is the false positive probability of one Bloom filter and there are n such BFs, and determine the false positive probability of an item x not present in all n BFs. Then, the probability that not all the address bit positions of x in each of the n BFs are set to a non-zero value is $(1 - fpr)^n$. The combined fpr is given by the probability that all the address bit positions of the item x are set to a non-zero value in at least one of the n BFs is $1 - (1 - fpr)^n$ [2, 16].

The FPR of a single read only table fpr_{RO} depends on the initial FPR of a single Bloom filter fpr_{ini}, the number of BFs in a single row $(nh + no)$ and the number of rows combined to form a single read only table nrc and can be given by

$$fpr_{RO} = 1 - (1 - fpr_{ini})^{(nh+no)\cdot nrc} \tag{3}$$

Since we tighten the FPR of a single Bloom filter for each read only table created by a factor r, the system FPR fpr_{sys} depends on r and the number of read only tables $nrot$ and can be given by

$$fpr_{sys} = 1 - \prod_{i=0}^{nrot-1} (1 - fpr_{RO} \cdot r^i) \tag{4}$$

Memory Overhead (MO) The total memory consumed by the system is divided into two parts: the memory required for buffering the write requests and the memory consumed by the BFs. Let m be the memory consumed by a read only table and it can be easily calculated given the Bloom filter fpr, the number of table rows nr and number of rows combined to form a single read only table nrc. Let s be the factor by which the memory m increases as fpr decreases and depends on the rate r.

$$MO = \frac{(nr \cdot Sz_{seg}) + (m \cdot nrot \cdot (1 + s + s^2 ... + s^{nrot}))}{number\ of\ items} \tag{5}$$

6. Evaluation

In this section, we explore the performance and memory usage characteristics of Muninn. The experiments are designed to understand the benefits and limitations of hash-based allocation and the use of BFs in terms of read/write amplification factors and the number of operations per second.

Our experiments were conducted on a Linux machine with 128 GB of memory. 64 GB is used to simulate flash memory with 4 KB pages and 256 KB erase blocks (15 μs are added to each flash access). For Muninn, the size of a segment is set to 1 MB. The number of table rows are set to 128, meaning at maximum 128 MB of memory is used as a page write buffer; this amount does not increase in proportion to the number of keys. When full-versioning is not used, the page write buffers are also used as a write cache to be able to absorb the extreme burst updates of the same key, *i. e.* 100 updates of the same key-value pair within a second. Otherwise, it is used to buffer the small writes to fill flash pages.

Muninn has been implemented as an ISCSI OSD target device that supports a subset of the T10 OSD commands to make our device usable under the current architecture. However, during evaluation, we noticed that the delay of the ISCSI protocol layer was often bigger than the delays of our object operations, polluting the results: it was not easy to accurately exclude the ISCSI delays. Thus, instead of using the ISCSI layer, we create a benchmark tool that directly sends the requests via an object interface. The workloads for this benchmark tool were generated using Basho benchmark [5]. It is configured to generate 1 KB key-value workload using sequential, random, and pareto distribution in order to understand the efficiency of our hash-based allocation scheme under various workloads. Specifically, the pareto distribution is designed to be the worst case, because 20% of the keys are updated 80% of the time. We use a single thread to execute each type of workloads of 50 million key-value pairs.

6.1 Segment Utilization and Memory Usage

The first experiment we conducted was to measure the segment utilization and memory usage varying the number and types of per-segment BFs. A regular BF keeps the memory

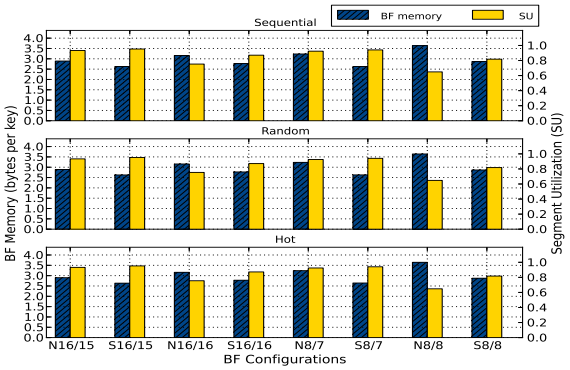


Figure 6. Segment utilization and memory usage; the S8/7 case used the smallest amount of memory while achieving 92% segment utilization.

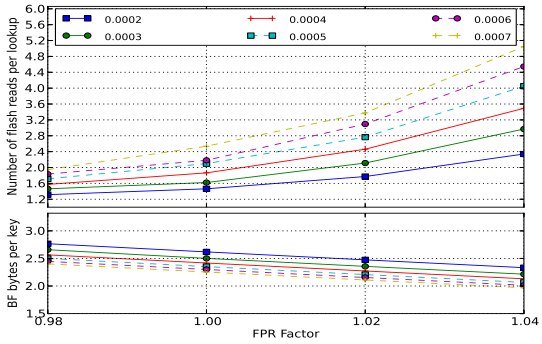


Figure 7. Effects of false positive rate on memory overheads and read amplification

overheads low, but offers only one bucket for a key, limiting the key's chances of finding free space. In contrast, the overflow BF gives additional places for key-value pairs, but increases the memory overheads. Our goal is to achieve a balance between the two.

Figure 6 shows the results of our experiments to find the balance using the three different distributions. The x-axis represents the number of BFs for each segment and the skewness of their size; one BF was dedicated for storing overflowed keys in the 16/15 and 8/7 cases and no separate overflowed key processing was done for 16/16 and 8/8 cases. The prefix S and N indicate whether the maximum number of keys per BFs is skewed or not.

Both 8/8 and 16/16 cases suffer from a low segment utilization due to the limited number of locations available for collided keys, but we can see that in the 8/7 and 16/15 cases, an overflow map is successfully alleviating the issue without increasing the memory usage much. It also shows that using different size BFs uses less memory. This is because more key-value pairs can be placed by the normal BFs; we make the first Bloom filter, which is the largest, to serve the most of the keys and adjust the size of the other smaller BFs to serve the rest. Among various configurations, we found that the S8/7 case used the smallest amount of memory while achieving more than 92% segment utilization similar to N16/15 and S16/15, and used it for the rest of the experiments.

6.2 False Positive Rate

We measured the number of flash reads and the number of bytes used per key to see the effects of the initial false positive rate and its increasing or decreasing factor. Each trace is configured to generate the worst-case search performance; only the key-value pairs in the oldest table are accessed. Figure 7 shows the number of flash reads per lookup and the memory occupied by the BFs, varying the initial FPR and FPR factor. We found that the number of reads is more sensitive to the initial FPR than memory usage, so lowering initial FPR would not increase its memory usage proportionally. Therefore, to save memory, our policy was to pick the highest initial FPR that provides less than 2 pages per read and increase FPR slightly over time so old read-only tables can have a lower FPR then recent read-only tables. More specifically, we use an initial FPR of 0.0005 and a FPR factor of 1.01 for the rest of the evaluation.

6.3 Read/Write Amplification and Performance

Using the design parameters chosen from the above experiments, we measured read and write amplification, and the number of flash accesses per operation in four categories, as shown in Figure 8. When inserting a key-value pair, it shows that Muninn requires 0.26 flash writes per operation, which is near optimal because the size of the key-value is 1 KB and the flash page is 4 KB; a write of a 1 KB value *must* write at least 1/4 of a 4 KB page, absent data compression. The high insert performance clearly shows the benefit of a hash-based allocation. The key-value pairs can be placed with two hash function calculations and a few Bloom filter operations; no index structure is maintained and flushed to flash like SSDs and other key-value stores, and no background operations exist as in SILT. In this configuration adjusted to achieve the worst-case read amplification of 2 and a nearly optimal write amplification, Muninn used around 2.5 bytes per key, requiring 250 MBs of in-device DRAM for 100 M keys.

Our lookup performance, on the other hand, varies depending on the location of data to be retrieved. In the best case where all requested key-value pairs are located in the first few read-only version tables, Muninn achieves very high read performance over 69,000 operations per second on a single I/O thread with no read cache. However, in the worst case where all data is in the oldest read-only table, it suffers from false-positive reads and lots of Bloom filter operations, and achieves only 20,000 operations per second. In the average case where the requested are evenly scattered across the device, it performed 34,000 operations per second. In this configuration, we tuned the system to achieve less than

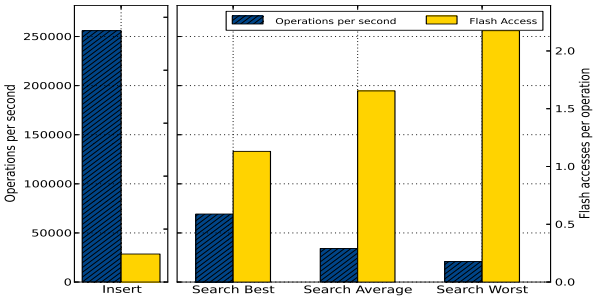


Figure 8. Read/Write amplification

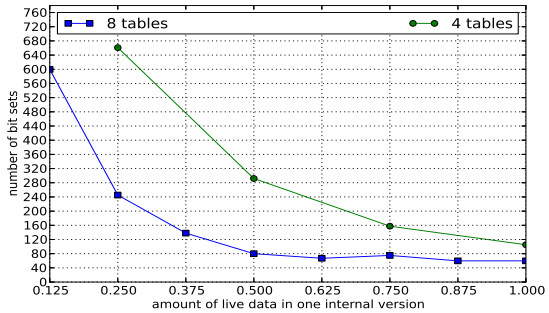


Figure 9. Cleaning threshold for selecting a target internal version

2 page reads per look up, but the performance can be optimized further by reducing the number of BFs or increasing the number of rows per table.

Although the latency of lookup operations varies, the average performance of Muninn is compatible to the existing key-value stores such as SILT and Bloomstore while providing full-versioning. With 4 instances of SILTs, it performs 23 K inserts/second and 46 K lookups/second for 1 KB key-value pairs. Bloomstore performs 25k to 77k operations per second using 64 B key-value pairs, retrieving 4.8 MB of data per second, which is similar to SILTs in terms of throughput. Muninn also achieves the similar lookup performance while providing much higher insert performance.

6.4 Versioning Overhead

Supporting versioning does not add much additional overhead to the flash device because of its out-of-place data placement; all the previous data is left in flash until it is cleaned. Since Muninn can retrieve the locations of any previous key-value pair without incurring additional costs compared to normal key-value retrieval, versioning does not add an operational cost.

However, when more than 20-30 updates to the same key are given in a very short period of time, storing every single update of the key will increase the space usage and merge overhead, because Muninn can provide a fixed number of places for each key per segment; we configure it to 24 in this experiments. To minimize the size of the memory used by the write buffers, if it exceeds 24, segments can be finalized leaving the most of the space empty due to the high collisions in the Bloom filter.

We ran both random and pareto distribution with and without overwrites in the page write buffer. With pareto distribution, when the overwrites are enabled, it shows 94.1% segment utilization and uses 2.143 bytes per key. After disabling the overwrites, the segment utilization comes down to 65.2% and the bytes per key is increased to 3.1. This can be solved by fixing the size of the write buffer and use the buffer until it becomes fully filled. Turning the overwrites on does not increase the memory usage, but loses some update histories of the very hot keys. When using the random distribution, regardless of the page write buffer, both shows around 94.1% utilization and around 2.14 bytes per key.

6.5 Cleaning Threshold

When a user requests a merge, Muninn scans the internal versions from the oldest to the newest to select a target internal version to be cleaned, using threshold values; segment utilization and the number of bit sets in the combined Bloom filter. The segment utilization is calculated by counting the unused space in normal BFs when flushing a row, and the number of bit sets is measured upon a merge request to estimate the number of live data within an internal version.

Figure 9 shows the relationship between the number of bit sets in the combined Bloom filter and the amount of live data in each read-only version table. In both internal version sizes, it shows the number of bit sets gets higher as the number of live data reduces. Based on this, we set the threshold to skip the internal version whose expected amount of live data is around 80%. Once the internal version is selected, we investigate the segment utilization of each segment to determine whether it needs to be cleaned or not.

7. Conclusion

We introduce Muninn, a full-versioning key-value store using the object-based storage model. By offloading the key-value pair management into the device with a rich interface, it achieves scalability, extensibility and efficiency. Muninn is designed to demonstrate those properties. It can add versioning to existing applications without altering their design using a version number is encoded in an object ID. For efficiency, Muninn shows the use of bloom-filters and hash functions to place and search key-value pairs, eliminating the need of per-object metadata or a direct mapping between LBAs and physical addresses. Our results show that Muninn achieves as low as 1.5 flash page reads per look up and 0.26 flash page writes per insert on average case, providing versioning.

References

[1] Aleph One Ltd. YAFFS: Yet another flash file system. http://www.yaffs.net.

[2] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, Mar. 2007.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 29–29. USENIX Association, 2010.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, New York, NY, USA, 2013. ACM.

[5] Basho Technologies Inc. Basho benchmark. http://docs.basho.com/riak/latest/cookbooks/Benchmarking.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[7] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 0:385–395, 2010.

[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)*, 2011.

[10] H. J. Choi, S.-H. Lim, , and K. H. Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Trans on Storage*, 4(4), Jan. 2009.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Oct. 2009.

[12] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, 3:1414–1425, Sept 2010.

[13] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 25–36. ACM, 2011.

[14] I. H. Doh, J. Choi, D. Lee, and S. H. Noh. Exploiting non-volatile RAM to enhance flash file system performance. In *7th ACM & IEEE Conference on Embedded Software (EMSOFT '07)*, pages 164–173, 2007.

[15] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[16] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The Dynamic Bloom Filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, Jan. 2010.

[17] 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)*, Feb. 2011.

[18] A. Hesseldahl. Apple iphone 4 parts cost about $188. `http://www.businessweek.com/technology/content/jun2010/tc20100627_763714.htm`.

[19] A. Hunter. A brief introduction to the design of UBIFS. http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

[20] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage*, 6(3), Sept. 2010.

[21] Y. Kang and E. L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *9th ACM & IEEE Conference on Embedded Software (EMSOFT '09)*, Oct. 2009.

[22] Y. Kang, J. Yang, and E. L. Miller. Object-based SCM: An efficient interface for Storage Class Memories. In *Mass Storage Systems and Technologies (MSST)*, pages 1–12, may 2011.

[23] Y. Kang, Yangsuk-Kee, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST)*, may 2013.

[24] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *8th ACM & IEEE Conference on Embedded Software (EMSOFT '08)*, pages 31–40, 2008.

[25] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with choices. *IEEE/ACM Transactions on Networking*, 16(1):218–231, Feb. 2008.

[26] M. Kryder and C. S. Kim. After hard drives - what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406 –3413, Oct 2009.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23st ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[28] G. Lu, Y. J. Nam, and D. H. Du. BloomStore: Bloom-Filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST)*, pages 1 –11, april 2012.

[29] D. Nagle, M. E. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4):401–411, 2008.

[30] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proc. of the 2006 International Conference on Compilers, Archi-*

tecture and Synthesis for Embedded Systems, pages 234–241, 2006.

[31] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 USENIX Annual Technical Conference*, June 2009.

[32] K. Sun, S. Baek, J. Choi, D. Lee, S. H. Noh, and S. L. Min. LTFTL: Lightweight Time-shift Flash Translation Layer for Flash Memory based Embedded Storage. In *8th ACM & IEEE Conference on Embedded Software (EMSOFT '08)*, pages 51–58, 2008.

[33] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.