UNIVERSITY OF CALIFORNIA
SANTA CRUZ

HIGH-PERFORMANCE, RELIABLE OBJECT-BASED NVRAM DEVICES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Yangwook Kang

September 2014

The Dissertation of Yangwook Kang
is approved:

Ethan L. Miller, Chair

Darrell D.E. Long

Sam H. Noh

Andy Hospodor

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

High-Performance, Reliable Object-Based NVRAM Devices

by

Yangwook Kang

Non-volatile memory (NVRAM) storage devices are increasingly used in both consumer and enterprise systems, providing high performance and low energy consumption compared to hard drives. Unfortunately, an inflexible block interface and multiple I/O subsystems designed for slow hard drives make it difficult to utilize NVRAMs in a portable and efficient manner. For example, in the OS storage stack, I/O workloads are reshaped and throttled to generate a small number of concurrent I/O requests, though NVRAMs can handle much higher workloads, and unnecessary address mapping is performed in a file system. Using NVRAM as memory can be difficult with a block interface because of small reads and writes that are common in memory workloads. At a device level, NVRAM devices need to perform a number of heuristics to attempt to recover filesystem-level semantics lost during the transmission through a block interface. As a result of these legacy interfaces, the current architecture is limited in achieving portability, efficiency, and extensibility, leaving behind more complicated problems, such as supporting heterogeneous NVRAM devices and utilizing device features.

This thesis proposes the use of the object-based storage model as a way of addressing the shortfalls of the current NVRAM architecture. More specifically, we show that the on-device NVRAM management layer can be as efficient as that of a native file system designed for a specific type of NVRAM while providing tightly coupled hardware optimizations and drop-in replacement for new types of NVRAMs. To explore the design flexibility given by this model, we investigate several data placement policies for flash memory exploiting the rich metadata coming from an object-interface. We also design an object-based byte-addressable NVRAM device to demonstrate that the object model allows more reliable, efficient NVRAM management by integrating its core data structures and a wear-leveling policy together. Our wear-leveling policy enables a low overhead, multi-granularity wear-level tracking for byte-addressable memories by exploiting the uniformity of a random function. We also investigate its extensibility and efficiency by designing Muninn, a object-based versioning key-value store for flash memory.

Realizing the object storage model requires changes in several subsystems of operating systems and hardware components, but no further system changes will be required to support new types of NVRAM. Even with the current storage stack, we hypothesize that some of the features of this model can be adopted to better utilize device hardware and improve overall efficiency of the storage system. To demonstrate these capabilities, we introduce Smart SSD, which pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying operating system code. Our system is built on a real SATA-based SSD and extends the Hadoop MapReduce framework to support in-storage processing. Our experiments show that total energy consumption is reduced by 50% due to the low-power processing inside a Smart SSD. Moreover, a system with a Smart SSD can outperform host-side processing by a factor of two or three by efficiently utilizing internal parallelism when applications have light traffic to the device DRAM under the current architecture.

Chapter 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 media increases in both consumer and enterprise markets. The recent deployment of Solid State Disks (SSDs) has accelerated this tendency by supporting backward compatibility with block devices. For example, consumer products such as laptops and smart phones are adopting flash memory to enhance their battery life and response time replacing hard drives. For enterprises, SSDs are used as a large long-term secondary cache residing between DRAM and hard drives, or replacements for 10,000/15,000 rpm hard drives and tape devices.

Beyond flash memory, several other types of non-volatile memories are currently being sold or actively developed, competing for the future storage or memory medium. For example, Phase Change RAM (PCRAM) promises high density and byte-addressability, but providing long-term resistance and high synchronous read/write performance is still challenging [1]. Ferroelectric RAM (FeRAM) provides high-performance and low power consumption, but has low density and destructive reads [28, 2]. Other types of NVRAMs such as memristors, carbon nanotube, and Spin-Torque-Transfer RAM (STT-RAM) are under development, promising superior characteristics than those currently on the market [51].

Unfortunately, despite the increasing importance of NVRAMs, storage and memory subsystems in current operating systems are not providing an efficient, portable way to adopt this technology shift yet. For example, several storage subsystems such as I/O schedulers and device queues are incurring overheads rather then contributing towards better I/O performance, and current interrupt and locking mechanisms can incur high context-switching overheads [91, 23]. At the device level, due to the limited flexibility of a block-based interface, the design of device subsystems such as mapping and wear-leveling is restricted and complicated, providing sub-optimal performance [45]. Moreover, it is not known whether this block-based access model can still be used for byte-addressable NVRAMs while efficiently hiding and handling their quirks.

In this thesis, we look towards the object-based storage model as a way of addressing the shortfalls of the current interfaces. More specifically, we show that the NVRAM management layer can be as efficient as that of a native file system designed for a specific type of NVRAM while providing tightly coupled hardware optimization and drop-in replacement for new types of NVRAMs. To demonstrate this, we explored object-based data placement and cleaning policies for flash memory, and integrated wear-leveling mechanism for byte-addressable NVRAMs. We investigate its efficiency and extensibility by designing two specialized devices: versioning flash device and smart SSD. We show that primitive object-based operations can also be served as the unified access model for these devices without sacrificing efficiency or portability, while existing protocols require dedicated command sets for each type of devices.

1.1 NVRAM Access Model

One of the biggest challenges in supporting NVRAM is the need for architectural changes including device-host communication, programming models for applications, file systems and operating system storage stack. Since NVRAMs are evolving technologies, these changes must cover a diverse range of NVRAM media without sacrificing their performance or capability. This leads to the first question we address in this thesis: *Given NVRAM hardware trends and their characteristics, what would be an efficient access model for NVRAMs?*

Some of the early flash storage systems, for example, designed their own flash-aware file system and devised a flash-friendly communication layer so it can directly access the flash memory controller [7, 89, 38]. This approach provides high efficiency but it is not cost-effective and lacks portability and compatibility, because file systems and communication layers were implemented and optimized for given hardware components. Today, many storage systems use SSDs, which internally have their own data management layer exposing a hard disk interface for compatibility with legacy software and systems. While not requiring hardware changes in this model, SSDs provide sub-optimal performance, because most of file system semantics are not delivered to SSDs and unnecessary components exist in the data path. Recently, several optimizations improving the inefficiency of SSD block management layer have been proposed [15, 94, 13]. While these approaches can eventually solve many of efficiency and architecture issues, they require significant amount of changes to the operating systems and these changes are not independent to the NVRAM medium; new changes to operating systems are required to adopt different types of NVRAMs.

To allow applications and operating systems to fully utilize the features of NVRAM devices, we thought the NVRAM storage stack needs to be efficient, reliable, extensible, and portable. More specifically, it should be capable of processing both small and large I/O requests efficiently and reliably, and provide a simple but efficient byte-addressable interface while being able to adapt to NVRAM technology shifts. To find such an efficient access model that meets the requirements, we explore the current access models for flash memory and hard disks, and other models recently proposed such as T10 object-based storage devices and nameless writes [85, 94]. Then, we evaluate the suitability and efficiency of the object-based model by designing and implementing an object-based file system and a device, and comparing the differences with the other approaches.

Through experiments on various data placement and cleaning policies in our prototype, we demonstrate that data structures in the in-device block management layer can be as efficient as that of flash-aware file systems due to the rich file system semantics enabled by an object interface. Compared to typical logical block number-to-page mapping schemes in SSDs, our object-based data allocation scheme and its cleaning policy significantly reduce the cleaning overhead. Additionally, several optimizations exploiting the existence of objects such as object-based reliability and embedding small files are adopted in our prototype, providing better reliability and space efficiency. In an operating system, since there is no block management layer in a file system, it can focus on name resolution and directory management, being independent to the underlying medium. This allows file systems to support multiple different NVRAM devices without introducing any management overhead on the host side or efficiency issues on the device. It is to be noted that unlike flash-aware file systems, the performance of this model can be further improved by tightly-coupled optimizations between the block management layer and a device hardware.

1.2 NVRAM Key-Value Store

With the growing need for data replication, fast retrieval and append performance, and scalability, more storage systems are adopting key-value stores rather than widely used relational database systems [80, 66]. Without requiring a relational schema, key-value stores provide a highly optimized insertion and retrieval performance to applications, sacrificing some of the searching capability such as a ranged search and a filtering. Since each key-value pair is independent to each other, managing replications across multiple distributed nodes and resolving conflicts become easier.

While key-value stores using hard disks are typically implemented at the user level, NVRAM key-value stores need to be in a device for better efficiency, similar to the block management layer in file systems. In addition, we notice that an object-interface can naturally support a key-value interface by considering an object id as a key and an object as a value. By doing so, we can allow object-based NVRAM devices to be used as a memory device, taking a memory address as a key. This leads to our second question in this thesis: *Can an object-based access model help system builders make more efficient and reliable NVRAM storage/memory systems?*

To answer this question, we design two NVRAM key-values stores: Muninn, a versioning object-based key-value flash device, which offers a transparent versioning using an object identifier to encode a version number, and an object-based NVRAM device that provides high reliability using probabilistic wear-leveling. More specifically, Muninn provides a hash-based key-value placement using multiple different-size Bloom filters [19], reducing metadata update overheads exploiting the copy-on-write nature of flash devices. It provides a transparent versioning to any legacy file system or user application at a low cost. While special devices, like in-device key-value stores, typically require special communication protocols, we show that without sacrificing usability or performance, the primitive object-based operations can be used as the unified access model, instead of adding dedicated SCSI commands every time a new feature needs to be added. Unlike the host-side key-value stores, which have two levels of data mappings and use host system resources, Muninn uses one efficient data management layer, providing a low write amplification factor and comparable read amplification factor compared to host-side key-value stores while not using any host resource. Users can roll back to a previous system checkpoint, and undo the changes to individual objects. Since users can control when to version and what to version in Muninn, it can be used for protecting some parts of data, *i. e.*, protecting operating system code, data generated by a specific application as well as system-wide checkpoints. This also allows users to mount the object based key-value store with different versions without having to roll back to a certain point.

The limited write endurance is one of the obstacles in using high density byte-addressable non-volatile memories as storage medium. Most proposed solutions are either constantly managing the wear-levels of all the lines, or adopting a coarse-grained wear-leveling based on a global counter, causing the lifetime of a device to be bounded by the weakest endurance cell or an inaccurate wear-leveling. We introduce a probabilistic wear-leveling technique for byte-addressable NVRAMs that enables low overhead, multi-granularity wear-leveling. Our wear-level counter requires only 1 B of space and are updated infrequently, exploiting the uniformity of a random function. We integrate this technique into each object-based NVRAM data structure, allowing various swapping mechanisms depending on its data type, and eliminating the need for additional address mapping. Compared to consistent wear-leveling mechanisms, it minimizes data movements for wear-leveling by moving objects only when needed and allows a worn-out region management at no extra cost.

1.3 In-Storage Processing

In contrast to hard drives where the performance is determined by the performance of the mechanical parts such as disk rotation and head movement speeds, the performance of NVRAM devices are determined by their processing power and the aggregate bandwidth to the underlying storage medium. More specifically, these devices tend to increase the number of independent data buses to the medium and use faster and more powerful processors to improve their ability to handle concurrent I/Os, hiding the overhead of data management. For instance, modern enterprise SSDs have multiple processors, large DRAM caches, and 8 to 16 or more flash channels, providing their peak performance when there are many large concurrent I/O requests available.

Despite the increasing hardware capability of SSDs, however, the performance at the application end is still limited due to legacy hardware and software designed for hard drives. While SSDs require a large number of concurrent I/O requests to maximize their performance, legacy storage subsystems typically throttle the number of pending I/O requests, and their locking and interrupt mechanisms sometimes introduce more overhead than processing an I/O request [92]. Block interfaces such as SAS and SATA are neither fast nor rich enough to leverage the potential of SSDs. The NVMe (Non-Volatile Memory Extension) standard for PCI Express [36] is designed to exploit the native performance of devices, but it is not yet robust, and requires applications to understand the characteristics of the underlying media to optimize their performance.

We may build new storage systems and applications for SSDs to alleviate the efficiency issues, but this requires too many changes to the current architecture, increasing complexity of each storage stack and losing portability. Instead, we note that NVRAM devices can be seen as a small computer system that has multiple processors, DRAM, and storage medium. Therefore, rather than processing simple read and write requests, devices can process complex I/O tasks defined by users, leveraging their internal parallelism and reducing the host system usage. By assigning jobs to devices, devices can carefully schedule internal I/O jobs in a way that the whole available bandwidth is utilized, maximizing the device performance. At the application end, it is necessary to generate and manage multiple independent I/O tasks so they can be executed concurrently across multiple devices. This leads to the third question we want to address in this thesis *How can we provide an efficient environment for both NVRAM devices and host systems that perform in-storage processing?*

To realize this idea, we introduce the Smart SSD model, which allows host systems to fully exploit the performance of SSDs without requiring operating systems and applications to understand the particular characteristics of SSDs by offloading data-intensive tasks from a host application to Smart SSDs. Each Smart SSD has an internal execution engine for processing locally stored data, and the host machine coordinates the sub-tasks as well as directly processing some parts of the tasks. By isolating data traffic within a device, the execution engine can schedule the I/O requests more efficiently; it can decide to fully utilize all the I/O channels and processes when the device is idle, or pause the data processing when there are many pending requests from users. This model also enables energy-efficient data processing, because power-hungry host-system resources such as host DRAMs and CPUs are not used.

In addition to the firmware enhancement, we also build a prototype of the host infrastructure to leverage Smart SSDs. The communication between a Smart SSD and a host system is handled by an object-interface implemented on top of a SATA interface to provide a compatible and flexible interface to applications and operating systems. We introduced two primitives called *select_object* and *execute_object* that select objects to be processed and execute an in-storage process on selected objects respectively. The Hadoop MapReduce framework [4] is used as an application interface to utilize Smart SSDs while hiding the details of the communications; the MapReduce framework provides independent sets of data that can be run concurrently across multiple Smart SSDs. The results show that the overhead of object-based commands implemented on top of SATA protocol is sightly slower than that of one sector write, because bi-directional communication is not allowed. With a web log analyzer example, in-storage processing outperforms host-side I/O processing while saving 98% of energy per operation.

The rest of this thesis is organized as follows. Chapters 2 provides a background on the subjects covered in this thesis. In Chapter 3, we discuss the design flexibility of the object-based access model compared to existing models, and subsequently in Chapter 4, we show the performance effects of rich file system semantics by prototyping the object-based storage model based on flash memory.Chapter 5 discusses the design goals of the key-value store we are going to provide. Chapter 6 presents the use of object-based interface on a real SSD implementing in-storage functionality, and Chapter 7 introduces probabilistic wear-leveling integrated into NVRAM data structure. Conclusions as well as possible directions for future work is given in Chapter 8.

Chapter 2

Background

Various types of non-volatile memories have become available with the rapid development of semiconductor technologies. However, the unique characteristics of these memories and the restrictions imposed by a block interface have led to many different storage system designs. In this chapter, we introduce these existing NVRAM storage system designs, and provide the background information needed to evaluate our use of the object-based storage model for NVRAM. First, we discuss the characteristics of NVRAMs, and various storage system designs to access and utilize them. This includes efficiency and portability issues that remain unsolved in current architectures. Second, we describe the current use of the object-based model in distributed systems and why it can be useful in supporting NVRAM devices. Third, we provide a discussion of in-storage processing, one way to take advantage of the full performance of an NVRAM device without requiring host operating system change. Fourth, we describe several host-side flash key-value stores, whose interfaces are natively supported by object-based devices, and versioning file systems, Bloom filters, hash functions that are relevant to the design of Muninn. The final section describes the current interfaces and wear-leveling techniques for byte-addressable NVRAMs.

2.1 Non-volatile memory

Non-volatile memory(NVRAM) is a class of memory that provides fast random I/O performance and non-volatility, including flash, FeRAM, magnetic RAM, phase-change memory (PCM), and carbon nanotube memory [51, 63, 21]. NVRAM blurs the distinction between main memory and secondary storage devices at a cheaper price than DRAM, and faster random access than disks. However, the direct use of NVRAMs in current systems requires detailed knowledge of the hardware for extracting higher performance and perhaps even proper operation.

The cell-level characteristics of these NVRAMs are well known and summarized at Table 2.1. Flash memory supports page-level access but does not support overwrites due to the need to erase a block, which consists of multiple pages, before rewriting it. While other types of NVRAMs such as PCM and MRAM provide byte-addressability and in-place update, each of them is also different in characteristics such as scalability, reliability, wear resistance, performance, and retention—differences that might favor different storage system designs. For example, flash-aware file systems usually use a log-structure to support out-of-place updates, and use a cleaner to reclaim invalidated blocks to generate free blocks for future writing [8, 43, 37]. PCM has the highest density and lowest endurance requiring wear-leveling. While other technologies such as STT-MRAM and carbon nanotube NVRAM are promising more DRAM-like characteristics, they are still early in development and except for PCM, their characteristics at a few hundred Gb density has not been revealed so we do not know what characteristics future NVRAM storage devices would have yet. For example, pushing these NVRAMs to higher densities would make endurance worse as we have seen on TLC NAND flash memory. Moreover, other technologies may have their own drawbacks such as destructive read in FeRAM. Since byte-addressable NVRAM are evolving technologies and there are a number of candidates, it is not feasible to change the host system whenever a new type of NVRAM is available. Thus, portability, reliability and performance are primary issues in designing a NVRAM-based storage system.

2.2 Existing approaches to access NVRAMs

Currently, there are two approaches to access NVRAM devices. These approaches either use a direct access model on the host side (Figure 2.1(a)) or an FTL-based model embedded in the underlying hardware (Figure 2.1(b)).

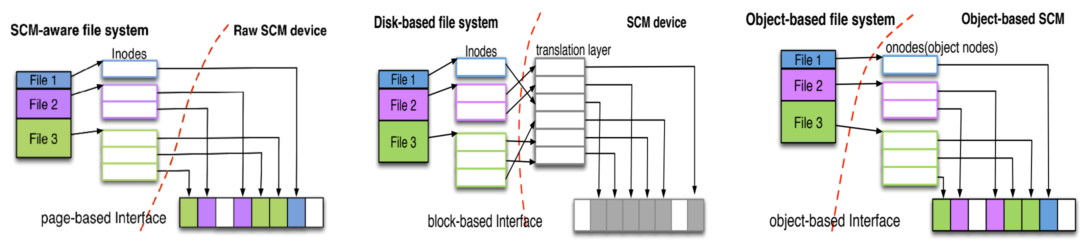
	NAND	PRAM	FeRAM	MRAM
read	15 μs	68 ns	70 ns	35 ns
write	200 μs	180 ns	70 ns	35 ns
erase	2 ms	none	none	none
in-place update	No	Yes	Yes	Yes
cleaner	Yes	No	No	No
write endurance	10^5	10^8	10^{15}	10^{15}
access unit	Page	Byte	Byte	Byte
power consumption	High	High	Low	Low

Table 2.1: NVRAM characteristics. [41]

2.2.1 Direct access model

The direct access model, shown in Figure 2.1(a), supports direct access to NVRAMs by either placing them on the main memory data path, or designing a specific file system that allows NVRAMs to work properly in the system. For instance, flash-aware file systems such as YAFFS [8], JFFS2 [90], RCFFS [43], and UBIFS [38] directly access flash memory via device drivers. Some storage systems for byte-addressable NVRAMs place them on the main memory bus, and use them as memory [58, 27], or secondary storage for metadata [49, 32].

The main advantage of this approach is that the core functionalities of the file systems such as indexing and data placement policy can be efficiently implemented for high performance. Since the characteristics of the underlying medium are understood by the file system and the raw medium can be accessed directly, the file system can fully control the medium. For example, flash-aware file systems such as YAFFS [8] and JFFS2 [90] use a log-structured mechanism [76] to meet out-of-place requirement in flash memory. While these two early flash file systems do not store indices on flash, the next version of JFFS2, UBIFS [37], uses a wandering tree, which supports out-of-place update, to store the on-flash indices, reducing memory footprint. Several approaches have been proposed to further improve the index efficiency and data placement for flash-aware file systems. Kang *et al.* developed the μ-tree [42], which is a variation of a wandering tree that packs several nodes modified by the change of a single leaf node into one page. Lim and Park place data and metadata to separate flash blocks to reduce



(a) Direct access model. (b) FTL-based model. (c) Object-based model.

Figure 2.1: Three approaches to access NVRAMs from file system

the cleaning overhead by a pseudo hot-cold separation [54].

However, since a NVRAM controller, a file system, and associated storage stacks need to be designed and developed, deploying highly optimized systems often require a long development time; while file system developers need to know the details of the underlying hardware configuration for optimizations, they do not tend to have early access to hardware, making it a longer delay to getting the whole system working. Thus, instead of relying on the features of the underlying hardware, many file systems are designed to use a fixed set of APIs — such as read_page and write_page — that are supported by lower storage stacks. However, these lower storage stacks have less information than file systems. They may lose the chance of more tightly-coupled optimizations enabled in the FTL-based model. For example, a file system may want to place data and metadata to different dies so they can be retrieved at the same time. But device drivers can not tell which request contains metadata. In addition, the migration from one type of NVRAMs to another is also problematic, because it requires both hardware and software to be re-implemented or at least a file system needs to be modified or developed. A lack of generic interfaces for NVRAMs and the need for both hardware and software changes for each NVRAM make this model difficult to adopt in commodity systems.

2.2.2 FTL-based model

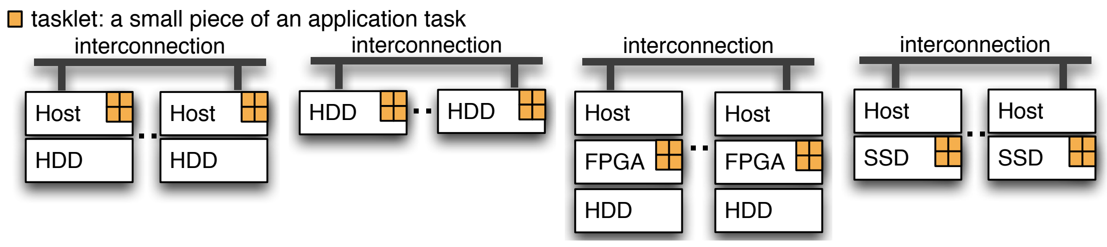
The second approach, widely used in modern SSD-based systems, is to access NVRAMs via a block translation layer. As depicted in Figure 2.1(b), the NVRAM device in this model contains a sector-to-page mapping table, and exposes a block interface to the host, allowing legacy file systems to access the NVRAM as a block-based storage device. Thus, FTL-based storage devices can replace current hard disks without modifications to the operating system.

However, compatibility with legacy systems comes at a high cost. Since the block interface is designed for disks and supports only read and write sector operations, the device cannot acquire essential information such as type of operations and data type to place or reclaim data blocks. As a result, there have been many efforts to improve the performance of the flash translation layer that propose mapping schemes, cache strategies and heuristics [25, 69, 47, 52]. For example, the use of the TRIM command in flash memory allows the device to recognize delete operations so that the device can invalidate data blocks belonging to a deleted file [79]. In spite of these efforts, they are not as efficient as the direct access model and are often very complex, due to a number of heuristics to catch the filesystem-level information lost during the transmission. Moreover, a fundamental problem of the FTL-based model is the existence of two translation layers in the data path, one in the file system, and one in the device.

Two approaches have recently been proposed to overcome the limitations of the FTL-based model by removing or minimizing the mapping table in the device. Arpaci-Dusseau, *et al.* proposed *nameless writes*, allowing the file system to directly manage the mapping table [14]. In this model, the device returns the physical page number to the file system so that it can store a mapping between a logical block number and a physical page number in an inode. DFS is a flash file system that uses a similar approach [40]. It also removes the mapping table from the device, and moves it to the virtualized flash storage layer, which is an FTL that resides in the operating system. Both approaches focus on reducing the complexity of the device and allow the operating system to control the device. However, *nameless writes* has some limitations in optimizing the flash device, since the file system has no information about the underlying medium except for a physical page number. In DFS, drop-in replacement for other types of NVRAMs are not easy because they would require a new file system for the underlying NVRAM or suffer from performance issues of using the block interface.

2.2.3 Object-based access model

The object-based access model has been used in high-performance and large-scale distributed storage systems [22, 33, 84]. These systems, including Panasas [62], Ceph [86], and Slice [12], use object-based storage devices (OSDs) to make each disk-based device work independently of the master server, and add more functionality to the device such as replica management, load-balancing, and data placement. In these systems, the object-based storage



(a) Clusters (b) Intelligent Disks (c) FPGA-aids (d) Channel Processors

Figure 2.2: Comparison of computing models

model allows devices to map IDs to variable-length blobs so they can recognize the size and type of each data locally stored, and process them without any coordination from a host system. Thus, the system can achieve high scalability and robustness.

While disk-based OSDs provide useful features in distributed systems, they have not gained much attention as a replacement for block based devices, because disks work efficiently with block interface since disks rarely remap blocks. Recently, however, Rajimwale *et al.* proposed the use of an object-based interface for flash memory [73]. In systems using NVRAMs where a richer interface than block interface is required, however, this model could alleviate several problems of the existing approaches: the lack of file system level information in the FTL-based model, portability in the direct access model, and efficiency of implementation in both. Under this model, the device can optimize the performance of the device using informed optimizations. Moreover, this approach allows a single file system to support multiple OSDs, making it easy to switch to a new type of OSD or even include multiple device types in a single file system. More advanced features exploiting the existence of objects such as object-level reliability and compression can be provided independent of the file system.

2.3 In-Storage Processing

In contrast to hard drives, where the performance is limited by the movement speed of mechanical parts, the performance of NVRAM devices depends on the performance of processors, the amount of memory, and internal I/O bandwith. This allows NVRAM devices to be capable of processing offloaded jobs from host systems without requiring additional hardware components.

The idea of disk-resident processing delegating system intelligence from host CPU to peripherals was first introduced a few decades ago for database machines. Early systems mainly focused on improving the performance of hard disks through dedicating a processor per head, track, or disk [67, 81]. However, they failed to survive because the high manufacturing cost could not be justified for marginal performance improvement. As computer systems were commoditized, however, the idea was resurrected in the late 1990s, and the intelligent disks (disk-resident processing) were actively investigated [46, 74, 75]. The studies on intelligent disks regard a disk as a complete computer system for efficient data processing and assume that the aggregation of compute capacity of less powerful but scalable processors performs better than the host-based approaches with dumb disks (Figure 2.2(a)).

Keeton *et al.* introduced the concept of IDISK (intelligent disk) for scalable decision support databases [46]. The basic idea behind this approach is to offload the data processing from desktop processors to lower-power processors to improve cost-performance. Thus, IDISK is designed as a general purpose network node that can replace costly cluster nodes. However, it was not able to be fully realized at that time because of the physical space constraint and the limited power and cooling supply issues. Riedel *et al.* proposed a similar concept called Active Disk (Figure 2.2(b)) that combines on-drive processing and large memory to allow disks to execute application level functions at the device [74, 75]. Another Active Disk [5] by Acharya *et al.* focused on the programming model and algorithm rather than the device architecture. They proposed a streaming programming model for application development, a sandbox model for secure task execution, and operating system supports both at the device level and at the host level. However, none of the previous approaches were implemented on commercially-available hardware. For example, Active Disk was designed to exploit internal disk components, but the experiments used an additional 64 MB bytes of RAM and an external processor to simulate a 4 GB disk.

More recently, there has been a resurgance in interest in in-storage processing, as exemplified by FAWN (Fast Array of Wimpy Nodes) [11], which uses low-power processors and flash to do data placement. However, FAWN KV is designed to handle PUT-GET style requests rather than coordinating and offloading the arbitrary I/O processing tasks. In this architecture, *all* data processing is offloaded to the flash nodes, requiring each node to consider data placement. There is no central coordination in the system, requiring distributed protocols to spread work around. This model works well for PUT-GET requests, but is poorly suited to Map-Reduce style processing. In addition, as it needs raw accesses to flash, a custom-designed embedded hardware is required, decreasing portability.

The concept of an intelligent disk was also adopted in the system-level solutions beyond a single device. Mueller *et al.* introduced an external module (*e. g.*, FPGA) attached to a disk (Figure 2.2(c)) that implements the system intelligence [59, 60]. IBM Netezza's Blade server [64] is a commercial version of this approach. In contrast, Oracle's Exadata is a commodity-based solution of an approach in which commodity storage servers reduce the amount of traffic by filtering data [88]. However, these solutions either require additional hardware components, or need an entire system that supports a specific type of in-storage processing.

Recently, several researchers have explored the potentials of SSD-based on-disk processing. Kim *et al.* investigated the benefits of ISP on SSD with the database scan operation [50]. They claimed the old lesson that the magnetic disk is the bottleneck of storage architecture for legacy database machines is no longer true because the fast storage medium and the increased parallelism in SSDs make the other components—embedded processors and memory—the bottleneck. Based on this intuition, a FMC (Flash Management Controller) with a scan function (Figure 2.2(d)) was simulated to filter data along the flash data path. However, due to the limited processing power, the limited amount of memory, and real-time contraints in a FMC, its applications are limited to relatively simple data processing such as filtering.

Boboila *et al.* proposed the approach of Active Flash to enable the out-of-core data analytics [20]. This study provides an analytical model that shows the performance-energy tradeoffs in moving data processing to SSDs in an HPC (High Performance Computing) context. Using a SSD simulator based on DiskSim, it shows that, with careful job scheduling that considers both the garbage collection overheads and idleness of SSDs, a significant amount of energy can be saved with only minor performance degradation. While the results are well aligned with our measurements on real SSDs, we found that the current SSD architecture is not sufficient to support the complex tasks. We also provide the in-storage processing model, which defines a host and application interface.

2.4 Key-Value Store

Key-value stores are specialized file systems optimized for insert and retrieval of small data associated with a fixed length key, replacing the needs for complex data base systems. These Key-Value stores are typically designed to work with SSDs, allowing better I/O performance. However, similar to legacy file systems using SSDs, two levels of data mapping are performed and host resources are used to manage the key-value pairs. Therefore, the memory usage and CPU usage become an important factor in achieving not only the performance, but also scalability.

FlashStore [30] 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 [31] 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 [10] maintains multiple hash tables–one in memory and the others on flash and a small set of Bloom filters to indicate whether a key might be present in a hash table.

BloomStore [56] is a recent key-value store based entirely on Bloom filters (BF). 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. Aiming to achieve low memory footprint, they store the large amounts of older BFs in Flash and read them in for lookups, hence increasing read amplification. The increased false positive rate of having multiple BFs has not been addressed properly. SILT [53] achieves a low memory footprint and a low read and write amplification using log store, hash store and sorted store. In this approach, key-value pairs are first written to a log store, then get converted to a hash store and finally to a sorted store. The sorted store gets processed and rewritten every time a hash store merges with it, requiring a large amount of free space. However, the number of flash read and write accesses can be much higher than a per-request amplification factor due to the background processes, moving data between stores. Supporting failure recovery and deletion can also be difficult.

2.4.1 Versioning Flash Systems

Versioning file systems keep a number of old copies of each file so the system can recover the previous status of each file on demand. Versioning is similar to a periodic backup, but the granularity of versioning is controlled by the file system; backup can be generated per file, device, and partition. Since out-of-place updates are required in flash memory, every flash device has multiple versions of the modified data. However, not many flash storage systems offer versioning, because it has to preserve the write order, which contains the full history of each file or data chunk, and store metadata and require additional user commands such as snapshot and rollback. It is not easy in most FTL schemes where the location of invalidated metadata is not easily trackable without reading many segments, and recognizing file boundary is not possible. For example, Lightweight Time-shift FTL (LTFTL) [82] enables to create version states at any time and go back in time to desired states. However, although it supports snapshots, finer-grained full versioning such as per-file versioning and per-directory versioning is not provided. Similar to Muninn, Tango [17] 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.

2.4.2 Hashing and Bloom filters

Hash-based data placement schemes were actively explored in 1980s [57, 55, 77, 93]. However, due to the uniformity of the hash-based distributions, these methods were not the ideal choices for hard disks where reducing the head movements is a primary design factor. 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 and metadata overhead can be minimized by not storing the actual key-page mapping.

Bloom filters are interesting alternatives for the mapping structure in hash-based placement, because it requires less space and provides high insert and look-up performance. While traditionally Bloom filters focus on how to manage static data set, there are several types of Bloom filters for dynamic data sets. Dynamic Bloom filter[34] increases or decreases the siz of the Bloom filter as the number of keys in each Bloom filter changes. Scalable Bloom filters [9] the false positive rate of each Bloom filter keeps decreasing as the number of Bloom filters increases To reduce the compounded error probability.

2.5 Byte-addressable NVRAMs

This section describes next-generation byte-addressable NVRAMs and file systems and algorithms designed for them. There have been largely two issues in adopting NVRAMs in storage systems; communication model that defines responsibilities of operating systems and devices and NVRAM characteristics. We will summarize the NVRAM characteristics first, and then discuss data structures, algorithms, and communication models designed for byte-addressable NVRAM.

Wear-leveling policies designed for byte-addressable NVRAMs can be classified into two categories: finer-grained, consistent wear-leveling and coarse-grained wear-leveling. The consistent wear-leveling techniques such as Start-Gap [72] and hardware bit-shifting [] keep the wear-level of all memory regions be within a certain range, by moving data periodically, usually based on a global counter. Depending on a frequency of data movements, they can achieve low overhead and good wear-leveling performance. However, data has to be moved even when the lines are still young and it makes it difficult to support a worn-out region management, which makes the lifetime of a device bounded by a weakest line. That is critical when there are high endurance variances between cells.

Coarse-grained wear-leveling policies are mostly used in file systems, exploiting the prior knowledge about their writing policies and data structures; they achieve low tracking overhead sacrificing some accuracy. For example, PFFS [70] uses a segment swapping and page shifting based on a log structure. However, unlike flash, data in segments can still be updated in place. When a large portion of a segment are frequently updated, it might incur a large data movement overheads, leading to a shorter device lifetime.

Unlike these two groups of policies, our policy is designed to support both fine and coarse-grained wear-leveling where a device can use different granularities on different data structures while minimizing tracking overhead as coarse-grained wear-leveling.

In flash memory, the use of a log-structure has been popular because of its requirement for out-of-place update. For byte-addressable NVRAMs, there are more data structures available to use. NV-heap implements a heap data structure that supports ACID transactions and prevents pointer corruption [26]. BPFS [27] uses a variant of B+ tree as an index structure optimized for PCM. They are detached from wear-leveling policies making it portable, but losing possible opportunities to reduce wear-leveling overheads.

Chapter 3

Object-based Storage Model for NVRAM

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

In this chapter, we describe the use of the object-based storage model as a way of addressing the shortfalls of the current interfaces. We first discuss the two main components in this model and how each storage layer is re-located in this model. Then, we discuss the problems of the current ANSI T10 OSD standard [61], and propose the primitive operations for NVRAM devices.

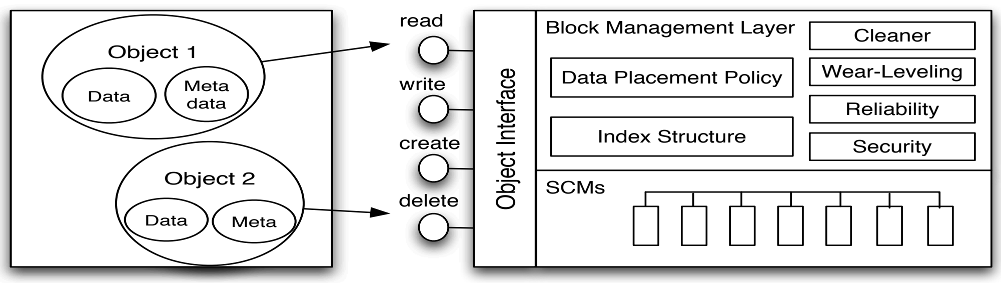
3.1 Components

The object-based model, shown in Figure 3.1, consists of two main components: the object-based file system and object-based devices. The file system and OSDs communicate via an object-interface, which exposes various types of object commands. Each operation contains an object, which includes a variable-length data and metadata associated with the data, describing one file-level or user-level I/O request. By exposing enough operations to capture file system operations as depicted in Figure 3.1(b), this eliminates the needs for host system utilities such as TRIM, which informs the devices whenever a file is deleted. Small-size requests, byte-addressable NVRAMs can be better supported.

Unlike a block-based 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 host-level file system becomes independent of the underlying storage medium while being able to deliver full file system level semantics to the devices. Thus, a single object-based file system can efficiently support multiple heterogeneous OSDs, in contrast to the current approaches that either require significant changes in the system or sacrifice I/O performance. Since the device manufacturers have better knowledge about the hardware configuration of NVRAM devices than file system designers, this model typically enables better hardware optimizations than native NVRAM-aware file systems.

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. Optimizations such as *nameless writes*, can provide such layer in the system. However, it will require a host system change whenever a target NVRAM media changes.

The block interface is limited in that it only delivers the sector number and the request type (read or write) to the device. In contrast, the object-based interface delivers objects (which contain both data and associated metadata) and recognizes all types of requests that the file system does. By doing so, the OSD is able to receive the same level of information that the file system maintains, allowing devices to provide features such as hot/cold separation to reduce cleaning overhead that have traditionally been provided by native file systems [45]. For small objects, OSDs can achieve better space efficiency than block-based devices due to the lack of a minimum allocation size. OSDs can reduce index overheads by using extent-based allocation for large, infrequently updated objects. In addition, by encapsulating data in objects, OSDs can provide more advanced features, such as object-level reliability, compression, and execution-in-place. Moreover, adding an object interface will not significantly complicate the existing FTL firmware design since NVRAM devices already need a translation layer for data placement and segment management. For example, when hybrid phase-change/flash memory is used, the file system can store data efficiently by simply sending a write-object request to the OSD; the file



(a) Object-based file system. (b) Object-based device.

Figure 3.1: System overview of object-based NVRAMs

system need not know about the two types of memory in the device.

3.1.1 Object-based File System

An object-based file system maintains files consisting of one or more data objects, which contain the file data, and a single inode object, which records metadata information about the file, such as permission and ownership. However, the inode does not point to the physical blocks, but rather the objects that make up the file, a mapping that can be done algorithmically by having the objects be identified by a combination of the unique file ID and the object's offset in the file [86].

Flat namespace Objects are identified by a 128-bit unique identifier, not by human-readable names. Thus, the identifier does not have any information about the logical relationship among objects unlike directory hierarchies, offering more flexibility when assigning and distributing objects. For example, the file system can assign a certain range of objects to each distributed node, so it can retrieve the location of object without searching and make the object sharing easy. Multiple independent sub directory hierarchies can also be built within each object range.

POSIX 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. Although the OSDs maintain a flat namespace internally, the directory support can also be implemented in the file system. 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.

When a file request comes in from the VFS layer, the file system finds objects related to the request based on an inode number and a file offset, and generates an object request for the data. Then it determines whether the requests can be deferred before sending, because the OSD can deal better with a large object, instead of multiple small objects. For directory operations, the file system can assign an object for each directory entry or store directory entries in a data object for a directory. Although an object-based model uses a flat namespace, supporting directories is not very different from that of typical file systems, since typical file systems also need to convert the directory and file name into unique inode numbers in order to maintain a hierarchy. For example, when receiving a `delete` request, the file system only needs to finds a set of object IDs, and send one or several `delete_object()` requests to the OSD, instead of generating several `read_block` and `write_block` operations to the device as is done in traditional block-based file systems.

3.1.2 Object-based Storage Devices for NVRAMs

Similar to SSDs, an OSD consists of NVRAM chips and multiple subsystems required to handle the characteristics of the medium such as a data placement policy and a wear-leveling mechanism, as shown in Figure 3.1(b). However, the design of its subsystems are more similar to that of typical native block managers for NVRAMs, because OSD subsystems can exploit the same level of information that native file systems have while SSDs require many heuristic algorithms to alleviate the problems of using the block interface.

By having rich information about the requests and a block management layer in the device, the OSD approach enables better hardware optimizations as well as simple and efficient subsystems. For example, device manufacturers can split a large write request into small concurrent write requests to multiple NVRAM chips in order to improve performance. Moreover, OSDs can further exploit the existence of objects to provide more advanced features such as object-level reliability and compression.

Object-level reliability. Most NVRAM devices used today rely on per-page error correction algorithms, which can detect and correct a certain number of bytes in a page depending on the size of the ECC. However, this approach cannot protect data against whole page failures, reading the wrong data back, or misdirected writes, which store the requested page to the wrong address and return a success code to the file system [16]. However, an object-based NVRAM can recover from this type of error by providing per-object parity. The device generates one group of parities for all data pages, and another group of parities for an object whenever it is modified. By reading all data blocks belonging to an object and comparing against the parity, it can detect and correct misdirected writes and whole page failures as well as bit-flips. The amount of parity can be adjusted based on the error rates of the medium by the device manufacturers or specified by the user on a per-object basis. In our prototype, described in Chapter 4, we maintain one parity for data pages and one parity for index nodes for each object.

Object-level compression and encryption. Since OSDs can access to object metadata, they can infer an object's type and determine whether it is a worthwhile for compression. The type of object, such as text or multimedia, can be inferred by reading a hint from a user or by reading the first few bytes of an object. Both compression and encryption can be done by specialized hardware chips in the device to improve I/O performance as well as reducing CPU utilization at the host.

Data compression and encryption can improve the space-efficiency and security as well as overall performance in NVRAM devices. In object-based NVRAMs, the device can provide either system-wide encryption or per-object basis encryption. The device can determine which object to encrypt based on its content, or users can specify that an object should be encrypted by setting a flag as a hint when sending a request. Thus, the NVRAM device could set different security levels for different types of objects and do encryption per-object, perhaps using different keys for different objects. Moreover, if the device does both compression and encryption, it can do it in the right order: compression followed by encryption.

Client library for object-based NVRAMs. A client library can allow users to customize OSD behavior by providing hints to the device. For example, users can provide an encryption key or specify the level of reliability for a specific object. Moreover, the library can provide an interface that bypasses the VFS layer to achieve a smaller memory footprint for small files by avoiding the need to have the VFS assign 4 KB pages to those small files before sending them to the file system. Moreover, an OSD is capable of managing variable-length objects efficiently; in our prototype, we have the file system send a hint to embed inode objects in onodes to reduce index records and page I/Os.

Object-level transaction. Supporting transactions in block-based devices is a difficult feature to implement because the device does not have any information about file system semantics. In object-based storage devices, however, because metadata is handled inside the device and an object can send a hint from the upper layer to the device, the devices can provide transactions without requiring modifications to the host system. For example, users can start the transaction by setting a common transaction ID to objects in the same transaction. The OSD can use this transaction ID to establish a dependency hierarchy and complete the transaction when a top-level object in the hierarchy is being created or deleted. One way to implement the transaction in the device is to use copy-on-write, which writes modifications to a different place until the commit begins, as commonly used in the process subsystem. Moreover, OSDs can seamlessly support per-object transactions, guaranteeing atomic object I/Os.

In this thesis, we explored object reliability, which adds parity pages per object, the use of client library in supporting versioning and in-storage processing. The same technique described in object-level transaction is used to deliver a version ID to a Muninn device.

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. Handling object requests and the sparse namespace are only the additional overhead to the SSD. However, since the block management layer can be much more efficient than SSDs, it may not be necessary to improve the hardware configuration. Also, considering the estimated cost of IPhone 4 processor is around $10 [35], adding more powerful processors for advanced data management would not increase the cost much.

In summary, the object-based storage model allows device manufacturers to provide NVRAM devices with various levels of functionality depending on the needs of their customers and make them fully customizable without host system changes. This model has the advantages of both FTL-based storage devices and native file systems: portability and efficiency. It also provides an abstraction of a request that can be used in various subsystems to provide intelli-

Command	**Description**
publish	publish available events and built-in functions to a host
subscribe	selects the events to be processed or parameters in a host system
list	provides a list of objects
read_object	read an object
write_object	write an object
execute_object	run an executable object in a device

Table 3.1: Primitive object commands

gence to the device.

3.2 Primitive Object Operations

The SCSI command set for object-based storage devices has been standardized since 2004. This T10 OSD standard supports objects, specified by 64-bit object and partition identifiers, using a fixed set of operations such as *read_object* and *write_object*. In addition, object-metadata can be accessed by *get_attributes* and *set_attributes*, and a list of objects can be retrieved by the *list* command. The newer version includes more functionality such as snapshots and collections.

However, the current standard is limited in that supporting various types of object-based NVRAM devices such as versioning stores and active disks, are difficult without extending the interface. Thus, instead of using T10 OSD standard, we define a small set of extensible object commands that can support an arbitrary number of device functionality, and use it as the default interface of Muninn.

Table 3.1 summarizes the primitive operations we propose. The initial handshaking is performed using *publish* and *subscribe* when a user mounts the device in the system. The publish commands provides the information about the device such as available built-in functions, device events, and other device parameters to a host. The host system can then use the subscribe command to register interesting events coming from the device, and device-wide parameters such as compression can be enabled by subscribing into this option. For example, a list of built-in function object ids can be built in a host, and the event handlers for disk capacity changes can be registered during the handshake.

The *execute* command allows users or file systems to run scripts within a device. These can be provided by a device vendor or users who want to optimize the behavior of the device for their needs. For example, in Muninn, the built-in operations for versioning such as *increase version* and *merge* are provided to manage versioned data. Lastly, the *list* operation can provide a list of objects stored in the device. It can take a object-identifier mask to allow the user to search the objects that meets the condition. By allowing vendors to provide their own sets of commands using the *execute* command, the communication layer protocols can remain small and fixed, not being frequently updated to provide more commands.

Optionally, we can consider adding *open* and *close* operations. While these operations are not required in supporting I/Os, device caches and consistency points can be better managed with them, because all the cached contents belongs to an object can be evicted or flushed upon being closed. Transactional storages may take advantage of these commands as well.

Chapter 4

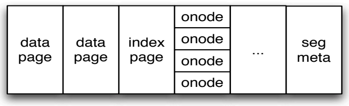
Data Placement Policies For Object-based Flash

In systems built on the object-based storage model, the device is responsible of handling the underlying NVRAM medium. The efficiency of this layer determines the performance of the device. By developing an object-based file system and an OSD based on flash memory, we demonstrate that in-device block management layer can be as efficient as that of flash-aware file systems due to the rich file system semantics enabled by an object interface. To understand the design flexibility of this in-device NVRAM management layer, we explore several design choices for the subsystems in an object-based device such as data placement policies, index structures and cleaning policies.

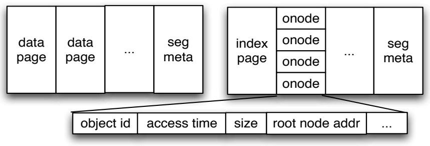
Compared to block-based mappings, our proposed data allocation scheme and its cleaning policy significantly reduce the cleaning overhead. Additionally, by adopting advanced features enabled by this model such as object-based reliability and embedding small files into a metadata structure, we show that the object-based storage model can provide better reliability and space efficiency, increasing the life-time of the device.

4.1 Design

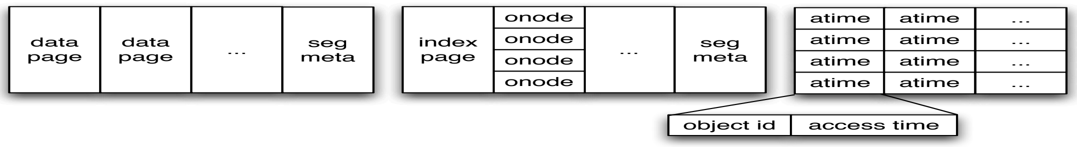
The goal of a data placement policy is the maximization of performance and lifetime by carefully placing data on NVRAMs. Typically, flash-based storage systems use a variation of a log-structure to store data because it supports out-of-place update and wear-leveling by



(a) Typical log-structured layout.



(b) Separation of data and metadata.



(c) Separation of data, metadata and access time.

Figure 4.1: Three kinds of data allocation policies

placing data and metadata together in a log, an approach we term a *combined* policy, shown in Figure 4.1(a). Since data and metadata is sequentially written to a log and never updated in this policy, a log-structure requires a cleaner to generate free space by reclaiming invalidated blocks. Cleaning overhead is known to be a primary factor determining log-structure file system performance, so reducing it is a key goal of data placement policies for flash memory. Log-structure may also be used for other NVRAMs which support in-place update for wear-leveling purpose, thus similar design choices could be made in order to reduce the cleaning overhead. The design of data placement policies for byte-addressable NVRAMs could be changed more flexibly by the manufacturer.

In object-based NVRAMs, since the type and size of the requests and other hints about data are known, various techniques to reduce the cleaning overhead can be applied. For example, hot-and-cold separation, which stores frequently accessed data blocks in a different segment, can be used since the device can identify which objects or internal data structures are more frequently accessed. Existing intelligent cleaning algorithms that have been developed for native file systems for NVRAMs can also be adopted with little modification. Our approach to optimize the data placement policy is based on the *combined* policy, with the addition that we separate frequently updated data from cold data such as object metadata and access time. Separating data and metadata was also used in other file systems such as DualFS [71] and hFS [95]. Unlike those systems and other NVRAM devices that do not manage file metadata internally, this approach can easily be accomplished in object-based NVRAMs as metadata of objects are maintained by the device itself.

Since the inode no longer maintains the physical addresses of file blocks, the OSD internally maintains the metadata of each object in a data structure called an *onode*. It contains information of each object such as size, access time and object ID as well as a pointer to the object's constituent blocks. An onode is one of the most frequently updated data structures in the device, and thus stored separately in our data placement polices, as shown in Figures 4.1(b) and 4.1(c).

Index structures. FTL-based approaches require a complex sector number to page number mapping table to determine the physical addresses of data blocks because the index structure does not recognize which block belongs to which file and the sector number is the only information they can use. Thus, various FTL schemes have been proposed to alleviate this problem. For example, the log-block mapping scheme maintains a small number of blocks in flash memory as temporary storage for overwrites [48]. However, in native file systems for NVRAMs or OSDs, an index structure utilizes full semantics of the requests and does not require a complex mapping table, thus being simpler and more efficient.

Since improving space efficiency both reduces the number of I/Os and also affects the life time in NVRAM devices, it is one of the important design issues of an index structure for NVRAM. For example, YAFFS stores the index in the spare area of each flash page to remove the index overhead [8]. However, it must perform a time-consuming scan of the entire spare area to build an in-memory index structure when mounting. Although YAFFS recently added checkpointing and lazy-loading techniques in order to improve the mounting time, it still requires scanning if the device is not properly unmounted. Thus, as capacity of flash chips grows, this approach would require more time, which can be a tradeoff between index overhead and mounting time.

Another approach to store indices in NVRAM devices is to use on-media index structures such as an inode-like structure or a variant of a B-tree. UBIFS [38] uses a wandering tree, which is a B^+ tree that supports out-of-place updates by writing all the internal tree nodes whenever a leaf node is updated, resulting in a much higher index overhead than that of a normal B^+ tree. A few attempts have been made to reduce the cleaning overhead of the index structures. For instance, the *mu*-tree places all the internal nodes in one page so that only one page needs to be written [42]. Agrawal, *et al.* proposed the lazy-adaptive tree, which minimizes the access to the flash medium by using cascaded buffers [6].

In order to reduce the overhead of an index structure in object-based NVRAMs, we used two optimizations based on a wandering tree with a dedicated cache. The first is the use of extent-based allocation. Since an object has a variable-length, and its size and request type are known to the device, an OSD can efficiently support extents, particularly if the file system can generate large object requests. This tree structure uses a combination of object ID and data type as a key, and a combination of physical address, offset and length as a value. The second approach to reduce the index overhead is to use a write buffer for small objects. Small objects whose size is less than a minimum write unit can be stored together to reduce the internal fragmentation within one unit, thus saving some page I/Os and index records. By sequentially writing large objects, high write performance can be achieved as well.

Wear-leveling and cleaning. Increasing the lifetime of the medium is another important goal in designing NVRAM devices, since many NVRAMs burn out after a certain number of writes. Therefore, most NVRAM devices need to maintain the wear-level of each erase unit and try to use them evenly. Since the storage management layer is inside the device in the object-based storage model, the device manufacturers can freely choose appropriate wear-leveling and cleaning policies for target NVRAMs.

In NVRAMs that do not support in-place updates, global wear-leveling is typically done by a log-structure cleaner, which maintains metadata for each segment such as number of erases and age, and selects victim segments based on those values. For byte-addressable NVRAMs such as PCM, other approaches can be used to track the wear-level of NVRAMs. Condit, *et al.* proposed two techniques for efficient wear-leveling; rotating bits within a page at the level of the memory controller and swapping virtual-to-physical page mappings [27]. Although some wear-leveling techniques require hardware support, the manufacturers can add more hardware without affecting the host system in the object-based storage model.

In our prototype, different cleaning thresholds are used for different types of segments. Since *atime* (access time) segments do not contain any valid data in them, they are always picked first. We set a lower threshold for metadata segments than data segments because the live data in metadata segments is more likely to be modified soon.

An object-based storage model can keep track of the status of an entire object, so more optimizations are possible. For example, by knowing the update frequency and size of

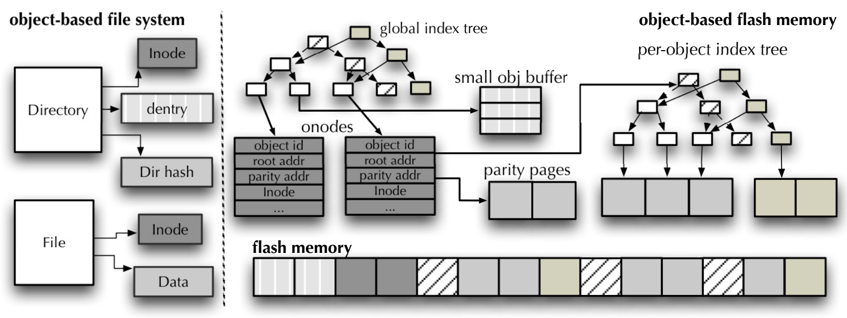


Figure 4.2: Implementation of object-based flash memory.

each object, OSDs could place cold objects in blocks where erase counts are higher than others, or group an object's blocks together.

4.2 Implementation

We built an object-based model prototype as a file system module in the Linux 2.6 kernel to investigate several design issues in the object-based model. This file system module consists of both the object-based file system and object-based storage device. The two components communicate only via an object interface, and do not share any in-memory data since they need to be independent of each other; they use memory copy instead of passing a pointer.

We picked flash memory as a target SCM for our prototype since it has more restrictions than other types of SCMs and is increasingly used in many devices such as SSDs in spite of several performance issues. The flash memory is simulated using NANDsim, the NAND flash simulator distributed with the kernel. This approach allowed us to use "raw" flash memory for the OSD; commercial flash memory typically contains an FTL and thus does not allow raw access to flash chips.

An object-based file system generates two objects per file; one for file data and one for inode, as shown in Figure 4.2. They are treated as separate objects in the device; an inode now contains only metadata information that is not frequently changed such as permission and ownership. By using a hint for inode objects, as previously discussed, inodes are stored in onodes in our prototype.

Support for hierarchical namespaces follows an approach similar to that used in disk-based systems. The directory file contains $\langle objID, dirname\rangle$, so the file system updates a data page in the directory file whenever an entry changes. One optimization we use in this prototype is a hashed directory that contains a pair of an offset of the directory entry and hash of the directory entry name. Given a directory name, it looks up the hash directory to find an offset of the entry, and tries to read the contents of data at the offset. The use of a hash directory helps to reduce lookup time when there are many subdirectories or files under one directory.

Since generating large objects helps reduce the index overhead of the device, an object-based file system maintains a cache for data pages to be written. The data page cache tries to cluster sequential writes to a file and collect them into a few large objects. The maximum size of the cache and each object is configurable to achieve the benefits of clustering multiple small pages into a large object. The object metadata operations, which usually need to be synchronously written, are directly flushed to the device. Notice that the file system does not assume flash memory in this implementation, but rather focuses on object management and the integration with the VFS, and is independent of the implementation of the underlying device.

As with most flash devices, a log-structure is used to handle the no-overwrite restriction and wear-leveling issues. As depicted in Figure 4.1(a), data and metadata are written sequentially within a segment, and segment metadata, such as timestamp, segment usage, reverse indices and superblock information, is written at the end of each segment. When there are not enough free segments, a cleaner is invoked to reclaim free segments.

Besides a typical log-structure, we implement two new data placement policies, shown in Figures 4.1(b) and 4.1(c), that are enabled by the object-based storage model. The idea behind these policies is to reduce the cleaning overhead by separating frequently accessed data from cold data so that the cleaner can pick a segment that has small live data (or metadata) in it. The *split* policy stores metadata and data separately in metadata segments and data segments respectively, based on the assumption that metadata will be more quickly invalidated than data. In this policy, the metadata of an object (its *onode*) and index structure nodes are stored in the metadata segment and other user data is stored in data segments. The *split+atime* policy further separates access time from the rest of the onode, avoiding frequent onode updates due to access time updates when an object is read but not modified. Since the size of an onode is larger than that of an access time entry, this approach can reduce the amount of metadata to written for the read request, thus reducing the cleaning overhead. In this approach, the access times of objects are journaled in the access time segment. Each entry requires only 16 B to store an object ID and access time, so a single 2 KB page can usually hold more than 128 entries, while only 10–20 onodes can be stored in one page. Access time entries are merged to the corresponding onodes when the total number of entries exceeds a certain threshold or when an object with a pending atime update is modified by a write.

Since flash memory does not support in-place updates, we use a wandering tree [38] combined with extent-based allocation in our system to show the effects of extent-based allocation in the object-based model. Each object has its own tree structure (local tree) that contains the physical location of its data blocks, and there is one global tree that maintains the physical location of onodes, small objects and reverse indices, as shown in Figure 4.2. Small objects—those less than one page long—are maintained by the global tree instead of generating a tree node that has only one entry, thus improving space efficiency.

In order to show the effects of the hints from a file system or user space, we set a flag for an inode so that it can be stored within an onode. Since inodes are very small and onodes need to be read to retrieve the inode object, embedding inodes into onodes can remove at least one page read per each inode operation.

A cleaner is invoked whenever the number of free segments is below a certain value. Unlike a combined policy, which has only one type of segment, victim segments are ordered by segment priority. For example, an access time segment contains very little live data, and the data in metadata segments is likely to be invalidated. Thus, we set a different threshold for each type of segment so that access time segments have the highest priority. Data segments have the next highest priority, and metadata segments have the lowest priority. This means a data segment will be picked first if a metadata segment has similar amount of live data, because metadata segments are likely to be invalidated sooner.

To demonstrate the advanced features enabled by the object-based storage model, we implemented object-level reliability in our prototype. For all data pages in an object, the OSD uses a Reed-Solomon code to generate a parity page and algebraic signature to detect bit corruptions [43]. The signature of each data page is stored in the spare area of a flash page, and the parity page is written as a part of an object. Since the algebraic signature and Reed-Solomon code use the same Galois field, the device can run a quick scan to check the validity of an object by the operations of taking the signatures of each page and combining them via

XOR; the parity of the signatures of data pages should be the same as the signature of a parity page. For per-object index tree nodes , the file system generates a parity page for tree nodes and stores the page as a part of an object.

There are several other data structures that are adopted to improve the efficiency of the device. For example, since onodes and tree nodes are accessed frequently, the OSD has a cache for each data structure. A one page write buffer is also used to store multiple onodes and small writes in a page. In order to reduce the mount time, the OSD uses the *next-k* algorithm, which enables the file system to quickly find the most recently written segment [43].

The object-based storage devices for byte-addressable SCMs may require different block management policies depending on the characteristics of each type of SCM. For example, the minimum unit of allocation on many SCMs can be adjustable, in contrast to flash memory, which has a fixed page size. A different indexing and data placement scheme can also be used, instead of a log-structure [83]. Although the internal data structure of OSDs might change, the object-based storage model does not require the changes of upper layers in the storage hierarchy. Thus, as SCM technologies evolve, an object-based file system can easily switch to an advanced device without requiring any modification.

4.3 Evaluation

We evaluate our object-based flash memory using two sets of performance benchmarks. First, we set the Postmark benchmark to generate a read-intensive workload and write-intensive workload, each of which contains a large number of data and metadata operations. These two workloads are used to show the effects of our two data placement policies. The second benchmark we use is a large file benchmark, which Seltzer, *et al.* used to evaluate the log-structured file system [78]. It creates one large file using four types of I/Os; sequential write, sequential read, random write and random read, and used to evaluate extent-based allocation. To measure the cleaning overhead and space efficiency, we make our file system module report the number of segment erases, the number of page I/Os for each subsystem, and the number of bytes copied during cleaning.

Our experiments were conducted on a virtual machine which has a single CPU, 1024 MB RAM, and a 10 GB hard disk. The NANDsim module is configured to model a 128 MB NAND flash memory with 2 KB pages and 128 KB erase blocks. The overall flash size is set by looking at the average response time of `readpage` and `writepage` operations. When we increased the size further, we began suffering from long latency, making the `writepage` operation take up to 4 seconds due to virtual memory reclamation. The use of a small flash size will limit the number of cold data pages we can generate and the size of the segment, making it difficult to look at long-term effects of the cleaning policies. However, each experiment in this section is designed to involve a large number of cleaning operations so that it would not underestimate the cleaning overhead due to the limited flash size; more than 500 segments were cleaned in each experiment. The size of the cache is also set proportional to the size of flash memory: the onode cache is set to 10 KB and the tree node cache to 50 KB. The segment size is set to 256 KB. In the Postmark benchmark, the read/append ratio is set to 1, the smallest number we can set for a write-intensive workload, and the number of files is increased to 1000 while the number of transactions is set to 40000. For a read-intensive workload, we set the read/append ratio to 8 and the number of transactions is increased to 90,000 in order to generate a read-dominant environment. The cleaning threshold for data segments and metadata segments is set to 80% and 60% respectively, based on the result in Section 4.3.2. The default read/append ratio of the Postmark benchmark, which generates five times more metadata than data, is used in other experiments. In each experiment, we take an average of 30 runs.

4.3.1 Cleaning Overhead of Data Placement Policies

We first measure the cleaning overhead of the three data placement policies under both read-intensive workload and write-intensive workloads using the postmark benchmark. For each policy in Figure 4.4, the left bar indicates the total number of segments cleaned and the right bar indicates the number of bytes copied during cleaning. Since more onodes and index nodes are written to update access time in the read-intensive workload, both the split policy and the split+atime policy work better than the combined policy, as shown in Figure 4.4(a). This is because each segment in the combined policy has some invalidated metadata pages and a large number of live data pages, causing only a small amount of free space per victim segment. On the other hand, each metadata segment in the split and split+atime policies contain a very small amount of metadata, thus reducing the cleaning overhead as well as the total number of pages written. The *split+atime* policy further reduces the cleaning overhead because fewer onodes are written by atime journaling.

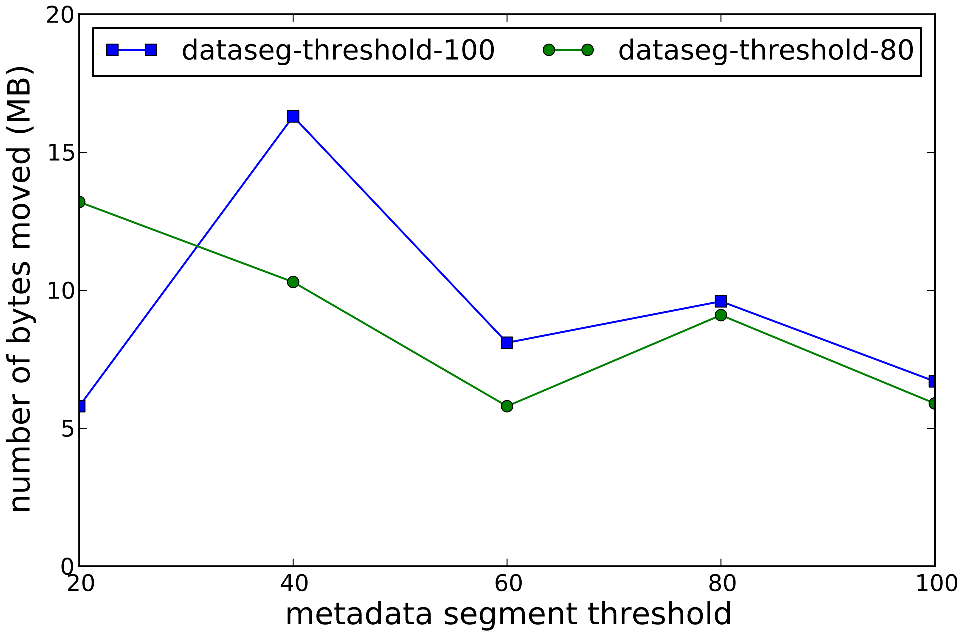
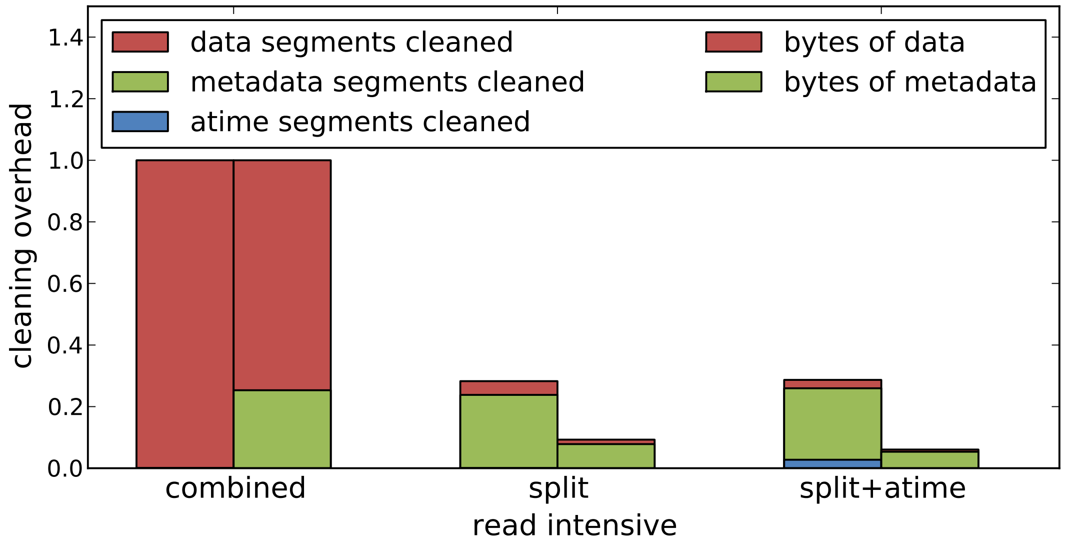
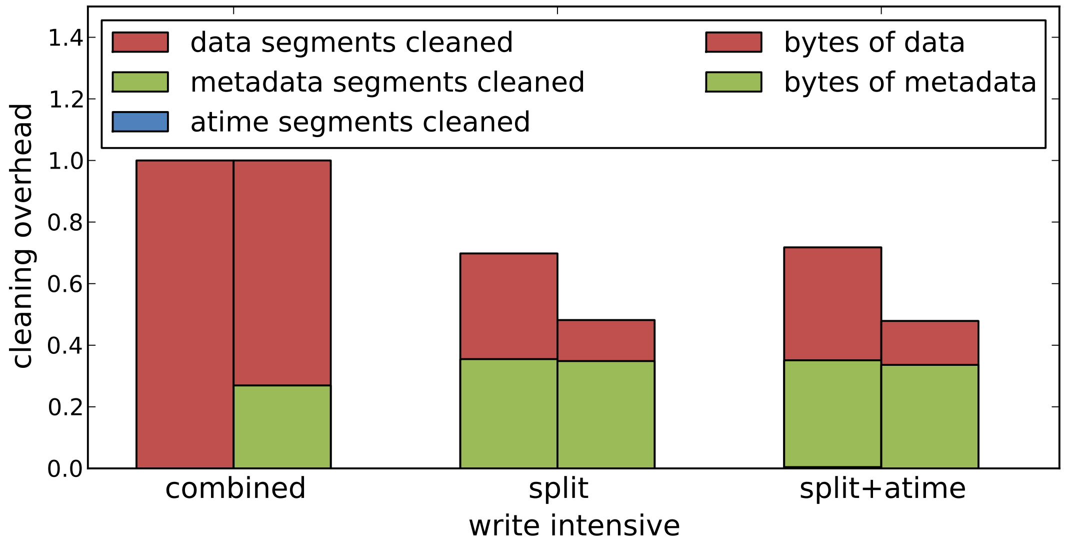


Figure 4.3: Effects of cleaning thresholds for different types of segments



(a) Read-intensive workload



(b) Write-intensive workload

Figure 4.4: Cleaning overhead of the three data placement policies. The X-axis represents three data placement policies and the Y-axis is the cleaning overhead normalized to combined policy

In the write-intensive workload, separating metadata has a smaller benefit because access times are written together when updating the fields in onodes and these dirty onodes are cached in memory. However, we still get some benefit from the onodes that are read but not written, as shown in Figure 4.4(b).

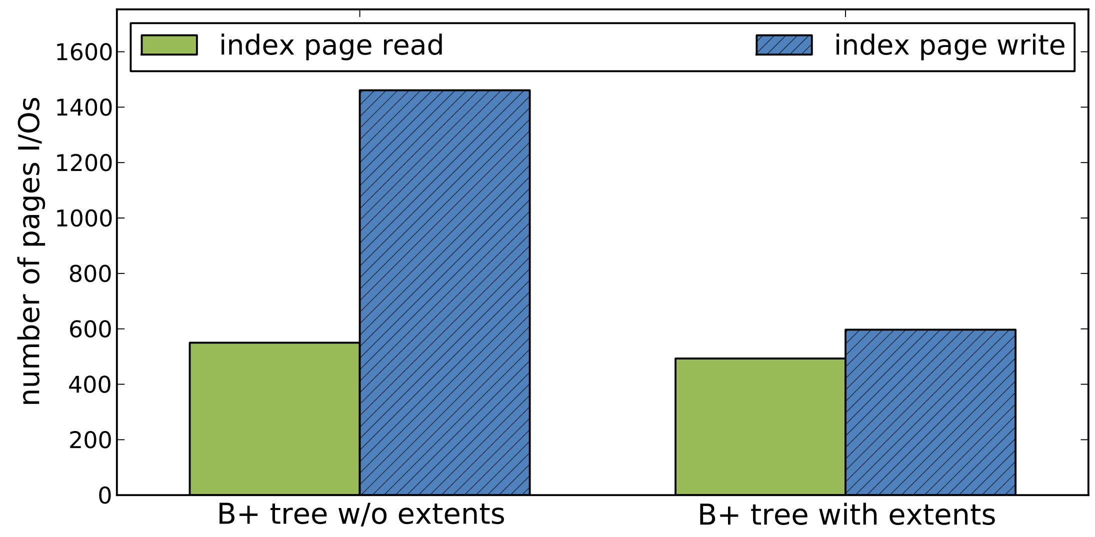
4.3.2 Effects of cleaning thresholds

Since the three different types of segments have different update frequencies, the device can set a different cleaning threshold for each segment in order to reduce the cleaning overhead. Figure 4.3 shows the cleaning overhead of the device varying the threshold for data segment and metadata segment. The number after each type of segment represents the percentage of the maximum amount of live data that the victim segments of that type can have. In a pure greedy policy, each threshold value is set to 100. Setting a lower threshold for metadata segments works well because the live data in metadata segment is likely to be invalidated quickly, so deferring cleaning often results in metadata segments containing even less live data. However, when the threshold for data segments is less than 60, the flash runs out of space, because there is insufficient space on the flash to have a large number of almost half-empty data segments. The pure greedy policy also works well in this short term cleaning overhead test due to the existence of metadata segments, which contain less live data—the segments that had the least amount of live data were the metadata segments in most cases.

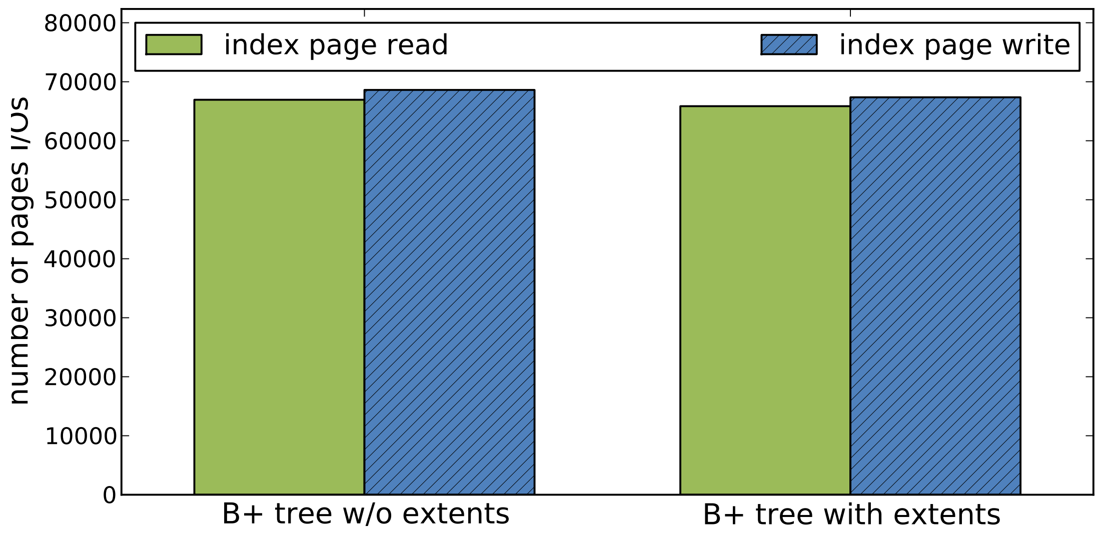
4.3.3 Index Structures

The efficiency of the index structure is critical to device performance, especially for devices that do not support in-place update. In order to show the effects of an extent-based allocation, we measure the cleaning overhead and I/O operations with and without extent-based allocation. When extent-based allocation is enabled, physical locations of objects are stored in terms of several *address-length* pairs. Otherwise, each data page is maintained by a wandering tree.

We measure the number of page I/Os issued by the index structure. In the largefile benchmark, since only one large file is sequentially written and then randomly rewritten, the result includes both the benefits of sequential writes and the overhead from random rewrites, and thus it is a good showcase for the effect of an extent-based allocation. Figure 4.5(a) shows



(a) Largefile benchmark



(b) Postmark benchmark

Figure 4.5: Effects of an extent-based allocation

	w/o inode_embed	w/o small obj buf	with all
pages read	378101	389165	364142
pages write	92006	91606	66615
seg write	718	714	288
seg clean	240	235	65

Table 4.1: Effects of informed-placement

that extent-based allocation significantly reduces both the number of page reads and the number of page writes, regardless of the overhead of random rewrites. In the Postmark benchmark, which generates many small metadata operations and a small number of large data files, the benefit of using extents are minimized, as shown in Figure 4.5(b). However, even this small file workload still realizes some benefits from extent-based allocation, which reduces the number of page writes by around 1000.

4.3.4 Effects of Informed Placement

The file system or user applications can send hints to the device to optimize the behavior of the subsystems on the OSD. For example, the write-amplification problem, which happens when the size of the request is smaller than the minimum unit of writes can be alleviated by using a hint for inode objects, since inode objects are now very small, infrequently updated, and have a fixed size. Because one tree node is generated for each object, the use of this flag reduces the generation of a separate tree node to store the inode and may eliminate some internal tree nodes as well.

The size of an object can also be used as a hint for the device. We use a one page buffer for small objects, so the device can store multiple small objects in a single page. In our implementation, a directory entry fits in this category. The size of each directory entry is around 256 B; thus, the OSD could store around 10 entries per 2 KB page.

To show the effects of these optimizations, we set the Postmark benchmark to create more files by setting the number of files is to 2000 and executing 30,000 transactions including `read`, `append`, and `delete` operations. The read/append ratio is set to 5. As shown in Table 4.1, when inode objects are not stored with onodes, they consume more pages, thus increasing the

number of files	page reads	page writes	parity reads	parity writes	with obj-reliability	without obj-reliability
500	93633	43822	2437	3388	16.50s	15.25s
750	140326	86740	5666	7865	21.67s	20.25s
1000	273931	171369	14892	33827	24.50s	23.00s

Table 4.2: Space and performance overhead of object-level reliability.

cleaning overhead due largely to the additional index overhead. If a small object buffer is not enabled, each directory entry has to use one flash page due to write-amplification, thus increasing the number of page I/Os. With both optimizations enabled, the number of page I/Os is reduced by more than 65%, and the overall cleaning overhead is also significantly reduced.

4.3.5 Object-level Reliability

The use of object-level reliability allows the device to detect more types of errors than the current error correction mechanism, which stores 4–8 bytes of ECC on the spare area of each page. It can even be used in conjunction with the current error correction mechanism to reduce the recovery time for simple errors such as a single bit-flip. In this section, we measure how much additional space is required in order to support object-level reliability. Since we generate two parity pages per each object, and the parities are updated whenever an object needs a modification, the overhead is proportional to the number of files and the number of operations.

We separately measure the parity overhead—the number of pages written by the object-level reliability mechanism—and compare it with the total number of page I/Os during the benchmark while increasing the number of files from 500 to 1000 with the same transaction size. As shown in Table 4.2, the number of pages written or read by the object-level reliability mechanism increases, but the overall overhead is less than 10% of the total I/Os, and there is no significant performance difference between the two setups. Object-level reliability thus provides detection and correction of errors that cannot otherwise be achieved while incurring a minimal performance overhead,

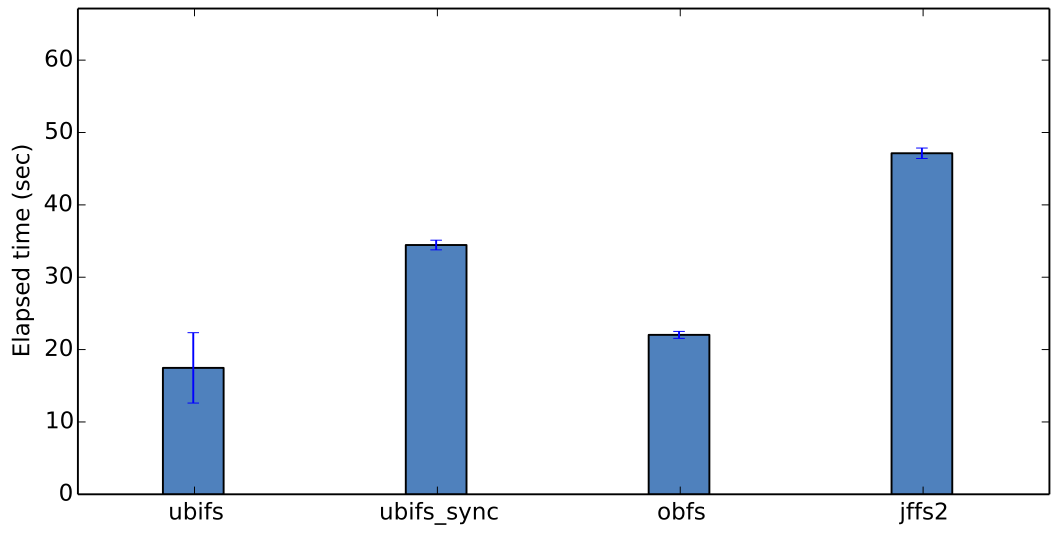


Figure 4.6: Overall performance

4.3.6 Overall Performance

Lastly, we compare the overall performance of our file system module with JFFS2 and UBIFS flash-aware file systems. Since JFFS2 does not have on-media index structures, so they have less index overhead, but both, as a result, require more mounting time, especially when the file system is not unmounted cleanly. UBIFS uses write-back, compression, and a small write buffer to improve both performance and space-efficiency. In our file system, we use the split+atime policy, B^+ tree combined with extent-based allocation, small object buffers and inode embedding.

Figure 4.6 shows the overall performance of flash-aware file systems and our file system using a postmark benchmark with 1500 files and 600000 transactions. OBFS represents our prototype, the object-based SCM for flash memory, and UBIFS_SYNC is the ubifs file system runs in synchronous mode. UBIFS shows the best performance among the file systems as it uses write-back while other file systems are using write-through.

Overall, OBFS shows performance comparable to other flash-aware file systems, even though the block management layer resides in the device and each file system uses different optimizations. For example, JFFS2 have less index overhead and UBIFS uses compression to increase the space-efficiency. Our prototype can be further improved by optimizing the implementation and adopting other features such as compression and write-back. Moreover, object-based SCMs can further improve performance by hardware optimizations in the real devices.

4.4 Summary

In this chapter, we explored several design issues in object-based SCMs: three data placement policies, two index structures, and other possible optimizations enabled in the object-based storage model such as object-level reliability and embedding inodes into onodes. Our experiments show that separating frequently accessed metadata from data can reduce the cleaning overhead in both workloads, and a B^+ tree variant with an extent-based allocation can further reduce overhead. We also showed that object-level reliability can be added to the existing reliability mechanism improving the recoverability without incurring much overhead. The performance of our object-based model prototype is comparable with other flash-aware file systems, which are much more efficient than current SSD-disk file system pairings. By using an object-based model for storage class memories, systems can realize the full performance and reliability benefits of current flash-based SCM, while avoiding the need to rewrite the file system as new SCM technologies are deployed.

Chapter 5

Versioning Flash Key-Value Store

In this chapter, we introduce Muninn, an object-based versioning key-value store that can transparently add versioning capability to an existing file system. Through the design of Muninn, we demonstrate that the object-based storage model allows storage systems to extend their features by replacing devices, and be more scalable by offloading its key-value mapping layer to devices. We show 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. Muninn is also designed to show that versioning can be added to a file system transparently with minimal host-side changes.

As a flash key-value store, it achieves better life-time 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 few 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.

5.1 Muninn

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. Its 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.

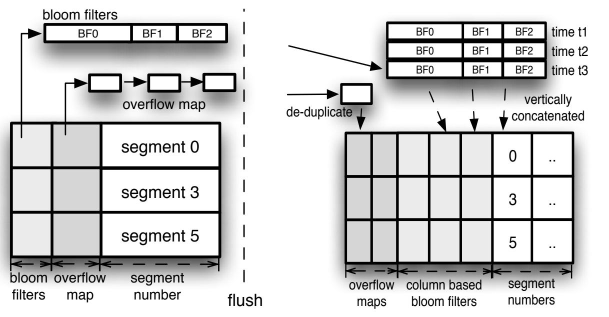
The design constraints of Muninn are different than that of most host-side key-value stores. While lowering per-key memory usage is a primary goal of most host-side key-value stores, Muninn makes this memory utilization a configurable parameter because a in-device memory is used to process key-value 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 policies can be specialized for the purpose of the device, selectively cleaning data blocks and reducing the number of reads or writes per key.

Muninn provides transparent versioning 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 5.1.2 and 5.1.3. We discuss how we preserve the write order to support versioning in Section 5.1.4. Lastly, we explain the design of a merger and the consistency of in-memory metadata.

5.1.1 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



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

Figure 5.1: Overview of Muninn

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 Chapter 3.

The version number of an object is encoded in an object ID, not stored as metadata to eliminate the need for keeping object metadata on flash. In our experiment where one instance of Muninn exists, we use a 64-bit object ID, 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. However, for better scalability in distributed storage systems, a 64-bit object identifier can be used and objects can be distributed without requiring a mapping table using stateless distribution strategies such as CRUSH [87].

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 an LBA; a partition ID becomes an identifier and an 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.

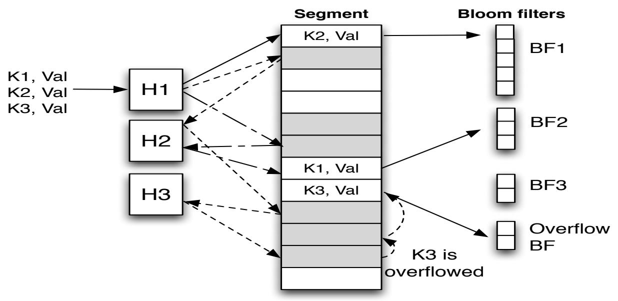


Figure 5.2: Insert a key-value pair to an active version table

5.1.2 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 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, permitting 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 5.1.5. However, using hash-based placement creates 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, 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 that of a 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.

5.1.2.1 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 the *active version table* where key-value pairs can be written as depicted in Figure 5.1(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 5.2, 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 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 was 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 an additional 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, Muninn 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.

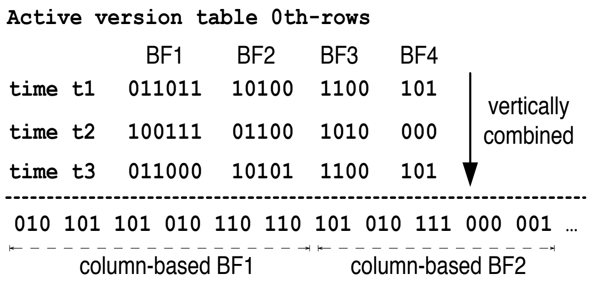


Figure 5.3: Flushing BFs to a read-only table

5.1.3 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 5.1(b).

When a key-value pair cannot be placed within a segment due to space limitations, 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 5.3 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 maximizes 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 over-

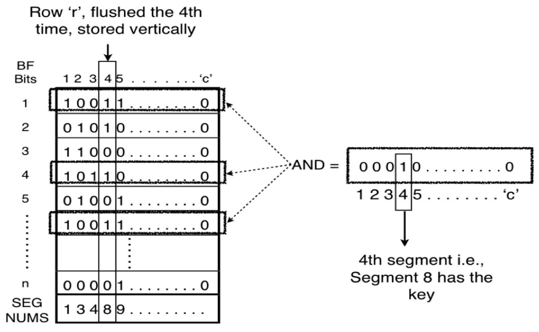


Figure 5.4: Searching a key-value pair in a read-only version table

flowed 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, we search the BFs in the read-only version tables. 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 as shown in Figure 5.4. 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 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 older key-value pairs that are 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, the system could be configured to have high false-positive rate, saving memory space

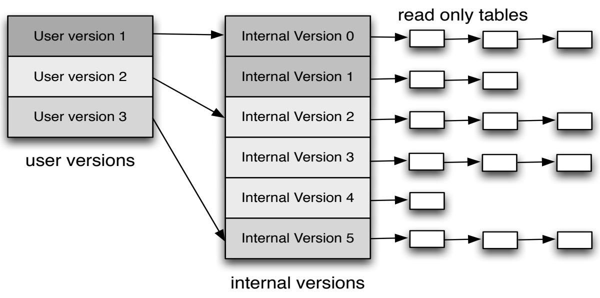


Figure 5.5: User versions and internal versions

but requiring 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 could reside in memory, resulting in a low false positive rate. The effects of each variable are analyzed in Section 5.2 and evaluated in Section 5.3.

5.1.4 Version Management Layer

To support snapshots and rollback, we have two data structures; a user version structure stores the version number given by a user and an internal version structure is a merge unit that contains a time-ordered list of a fixed number of read-only tables. A user version number increases as a version number in an object ID increases and an internal version number increases when the number of read-only tables added to the current internal version structure exceeds a threshold, as shown in Figure 5.5.

When all rows in a read-only table become full, the table is added to the head of a time-ordered list of the current internal version structure and a new read-only table is created. When searching for a key-value pair, Muninn first searches the read-only tables attached to the current internal version structure. If a key is not found, it keeps searching older version structures until a match is found.

When a user version number is increased, the active, read-only version tables, and current internal version structure are finalized to create a snapshot. A new user version structure 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. The newer versions will become invalidated and collected later by a version-aware merger.

5.1.4.1 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 a 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, Muninn 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 5.3.

One of disadvantage of a user-triggered merging is that it may require frequent user interaction to keep the free space of each device at a certain level. While this can be alleviated by allowing the device to automatically merge the oldest internal version when there is not enough free space, it may also incur a potential security issue where some of the update history of a certain object can be deleted by overwriting it. However, in any case, the overhead of merging is not greater than that of cleaning in SSDs, and merging can improve the worst case search performance by moving old live key-value pairs up to the most recent active segment.

5.1.5 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 to ensure metadata consistency. These read-only tables do not need to be updated once written so they are treated as normal data segments, and its physical page address is never invalidated. Thus, during runtime, only one active version table and one read-only version table remain dirty, and we assume that its dirty pages can be flushed to flash, and the metadata containing the locations of the current internal version structure and the active version table can be added to one of the reserved segments upon a power failure.

5.2 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 5.1.

Segment Utilization (*SU*) *SU* depends on the utilization *U* achievable for the said set of keys, *hashes*, the number of hash functions used for data placement and *oBF*, number of overflow BFs.

$$SU = \frac{data\ written\ to\ a\ flash\ segment}{segment\ size} \tag{5.1}$$

$$SU \propto hashes \cdot U_{hash} + oBF \cdot U_{overflowmap} \tag{5.2}$$

Symbol	Description
fpr_{int}	Initial false positive rate
$hashes$	Number of hash functions for data placement
oBF	Number of overflow BFs
r	False Positive Rate reduction rate
$rows$	Number of table rows
$combined$	Number of rows combined to form a read only table

Table 5.1: Design parameters

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 $1-(1-fpr)^n$ [34, 9].

The FPR of a single read only table fpr_{RO} depends on the initial FPR of a single Bloom filter fpr_{int}, the number of BFs in a single row $(hashes+oBF)$ and the number of rows combined to form a single read only table $combined$ and can be given by

$$fpr_{RO} = 1-(1-fpr_{int})^{(hashes+oBF)\cdot combined} \tag{5.3}$$

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

$$fpr_{sys} = 1-\prod_{i=0}^{nrot-1}(1-fpr_{RO}\cdot r^i) \tag{5.4}$$

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 *rows* and number of rows combined to form a single read only table *combined*. Let *s* be the factor by which the memory *m* increases as *fpr* decreases and depends on the rate *r*.

$$MO = \frac{(rows \cdot Sz_{seg}) + (m \cdot nrot \cdot (1 + s + s^2 ... + s^{nrot}))}{number\ of\ items} \tag{5.5}$$

5.3 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 read/write access). We reduced the latency below that of a typical SSD by removing the overhead of mapping and crossing communication layers and providing internal I/O bandwidth that exceeds host-device bandwidth [44]. 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. When 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, we decided to bypass ISCSI and created a benchmark tool to send requests directly to Muninn via an object interface. The workloads for this benchmark tool were generated using the Basho benchmark [18], which is a benchmark tool for key-value stores supporting various kinds of object distributions. We configure it to generate a 1 KB key-value workload using sequential, random, and pareto distributions in order to understand the efficiency of our hash-based allocation scheme under various workloads. Specifically, the pareto

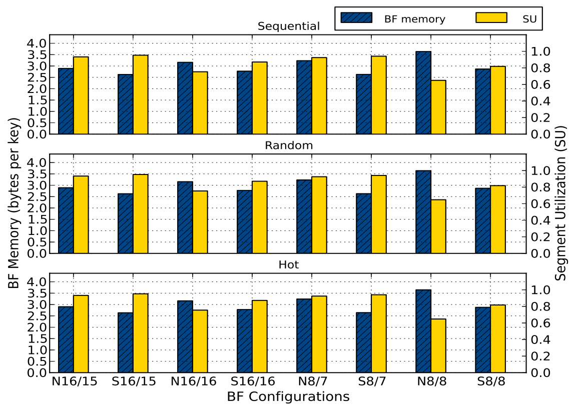


Figure 5.6: Segment utilization and memory usage; the S8/7 case used the smallest amount of memory while achieving 92% segment utilization.

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.

5.3.1 Segment Utilization and Memory Usage

Our first experiment measures the segment utilization and memory usage varying the number and types of per-segment BFs. A regular BF keeps the memory 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 5.6 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. 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 additional key-value pairs can be placed by the normal BFs. We make the first Bloom filter, which is the largest, to

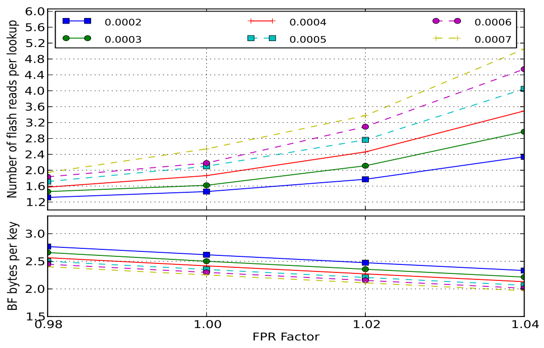


Figure 5.7: Effects of false positive rate on memory overheads and read amplification

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.

5.3.2 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 5.7 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.

5.3.3 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 5.8. 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 requests are evenly scattered across the device, it performed 34,000 operations per second. In this configuration, we tuned the system to achieve less than 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 similar lookup performance while providing much higher insert performance.

5.3.4 Versioning Overhead

Out-of-place data placement leaves all previous versions of data in place until the flash is cleared. Versioning support occurs naturally with little additional overhead. Since Muninn can retrieve the locations of any previous key-value pair without incurring additional

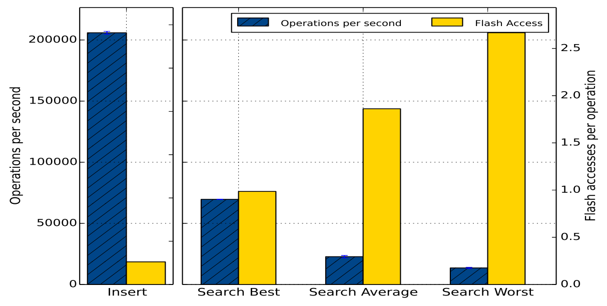


Figure 5.8: Read/Write amplification

costs compared to normal key-value retrieval, versioning does not increase operational costs.

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 only provide a fixed number of places for each key in a segment; we configure it to 24 in this experiments. Once the same key is written more than 24 times to the same segment, it can be finalized leaving the most of the space empty to limit the size of the memory used by write buffers.

We ran both random and pareto distribution with and without overwrites in the page write buffer. With pareto distribution, when 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.

5.3.5 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

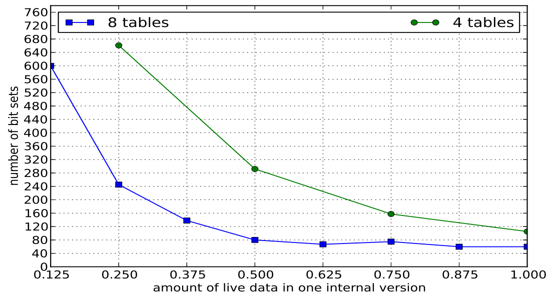


Figure 5.9: Cleaning threshold for selecting a target internal version

bit sets is measured upon a merge request to estimate the number of live data within an internal version.

Figure 5.9 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.

5.4 Summary

We introduce Muninn, a versioning key-value store using the object-based storage model, to show that through an object-based access model, it can 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 provides versioning and achieves as few as 1.5 flash page reads per look up and 0.26 flash page writes per insert on average case.

Chapter 6

In-Storage Processing

Modern NVRAM devices provide high performance on concurrent random writes, and have powerful processors, memory, and multiple I/O channels, enabling in-storage processing with almost no hardware changes. However, they do not have a flexible interface that allows users or systems to offload their I/O tasks. In this chapter, we add an object-based access model to SSDs, and explore the benefits and limitations of in-storage processing on current Solid-State Disk (SSD) architectures.

To leverage the enhanced data processing capabilities of modern SSDs, we introduce the Smart SSD model, which pairs in-device processing with a powerful host system capable of handling data-oriented tasks without modifying operating system code. By isolating the data traffic within the device, this model promises low energy consumption, high parallelism, low host memory footprint and better performance. To demonstrate these capabilities, we constructed a prototype implementing this model on a real SATA-based SSD. Our system uses an object-based protocol for low-level communication with the host, and extends the Hadoop MapReduce framework to support a Smart SSD. Our experiments show that total energy consumption is reduced by 50% due to the low-power processing inside a Smart SSD. Moreover, a system with a Smart SSD can outperform host-side processing by a factor of two or three by efficiently utilizing internal parallelism when applications have light traffic to the device DRAM under the current architecture.

6.1 Hardware Capabilities of SSDs

Rapid developments in NVRAM technology have challenged the assumption that I/O devices in storage systems are slow. In contrast to hard drives, where the performance is limited by the movement speed of mechanical parts, the performance of NVRAM devices can be improved simply by adopting more powerful processors and improving the internal bandwidth to the underlying storage media. More specifically, modern high-performance Solid State Disks (SSDs) have multiple powerful processors, large battery-backed DRAMs, and 8–16 (or more) independent I/O channels to provide high performance.

Despite the increasing hardware capability of SSDs, however, the performance at the application end is still limited due to legacy hardware and software designed for hard drives. Legacy storage subsystems typically throttle the number of pending I/O requests to accommodate disks that do not support concurrency. However, SSDs require a large number of concurrent I/O requests to maximize their performance. Equally important, locking and interrupt mechanisms can introduce more overhead than processing an I/O request [92], and block interfaces such as SAS and SATA are neither fast nor rich enough to leverage the potential of SSDs. The NVMe (Non-Volatile Memory Extension) standard for PCI Express [36] is designed to exploit the native performance of devices, but it is not yet robust, and requires applications to understand the characteristics of the underlying media to optimize their performance.

6.2 Smart SSD Model

In this chapter, we introduce the Smart SSD model, allowing host systems to fully exploit the performance of SSDs without requiring operating systems and applications to understand the particular characteristics of SSDs. We achieve this by offloading data-intensive tasks from a host application to the Smart SSD. Each Smart SSD has an internal execution engine for processing locally-stored data, and the host machine coordinates the sub-tasks as well as directly processing some parts of the tasks. By isolating data traffic within a device, the execution engine can schedule the I/O requests more efficiently; it can decide to fully utilize the I/O channels when the device is idle, or pause the data processing when there are many pending requests from users. This model also enables energy-efficient data processing, because power-hungry host-system resources such as DRAMs and CPUs are not used.

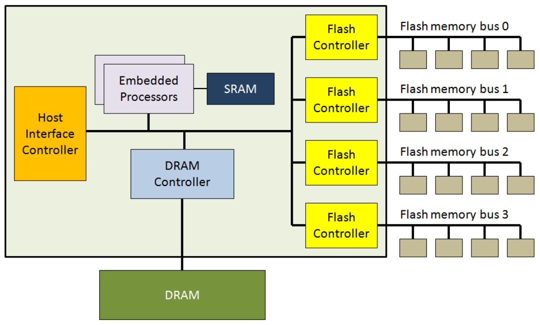


Figure 6.1: The typical architecture of modern flash-based SSDs

In addition to enhancing the firmware, we also built a prototype of the host infrastructure to leverage Smart SSDs. The communication between a Smart SSD and its host system is handled by an object-interface implemented on top of the SATA protocol to provide a compatible and flexible API to applications and operating systems. The Hadoop MapReduce framework [4] is used as an application interface to utilize Smart SSDs while hiding communication details. This chapter introduces:

- A model that demonstrates the use of an SSD as a data processing node that can achieve both higher performance and energy savings through enabling efficient data flow and consuming extremely small amounts of host system resources.
- The first evaluation of in-storage processing (ISP) on a real (not simulated) MLC (multi-level cell) SSD device.
- An end-to-end evaluation of performance and energy covering the entire system.

6.3 The Smart SSD Architecture

SSD architectures have evolved as the need for performance and capacity has increased, but this evolution has typically been limited by the need to avoid changes in the host hardware and operating system. However, the performance of modern SSDs has now reached the maximum bandwidth of the SATA interface when large burst requests are given, requiring the host system to generate more flash-optimized requests for further performance improvements.

Adding in-storage processing capabilities to SSDs is one way to reduce the demand on the data path from applications to devices while minimizing the changes required to applications and host systems. By offloading the application tasks and reducing the use of host system resources, it enables energy-efficient data processing that can fully utilize the internal resources in SSDs. Harnessing more powerful processors and interfaces with higher I/O bandwidth on a host can improve the overall data processing performance. However, it is still not easy to reduce energy consumption in a power-hungry host environment without traffic control. The Smart SSD model provides an efficient way not only to improve performance by reducing the amount of data transfers from device but also to save energy by utilizing low power processors in device. In this section, we describe the hardware capability of modern SSDs, the Smart SSD model exploiting the current architecture, and the extended host-device communication.

6.3.1 Modern SSD Architecture

Figure 6.1 illustrates the general architecture of modern SSDs [20, 50]. An SSD consists of three major components: the SSD controller, DRAM, and the flash memory array. Further, the SSD controller is composed of three subcomponents: the host interface controller, the embedded processor, and the flash memory controller.

The host interface controller implements a protocol such as SATA, SAS, or PCIe. SATA 3.0 and SAS 2.0 support up to 6 Gbps of bandwidth while PCIe 3.0 can transfer 8 Gbps per lane. Typically, 32-bit RISC processors such as the ARM series are used as embedded processors [39], providing host command handling and a flash translation layer (FTL) to map a Logical Block Address (LBA) to a physical page number in the flash memory. The embedded processors and associated SRAM (for executable code storage) require much less energy to run the SSD firmware than would a standard host CPU.

The flash memory controller (FMC) is in charge of reliable data transfer between flash memory and DRAM, which is used to cache user data and to store metadata for the FTL. Its key functionality includes Error Correction Code (ECC) and Direct Memory Access (DMA). The FMC is also responsible for exploiting the parallelism in the flash memory by chip-level and channel-level interleaving techniques to improve I/O performance.

Finally, the flash memory array is the persistent storage medium; NAND flash is currently the most popular choice. NAND flash memory is composed of blocks, each of which

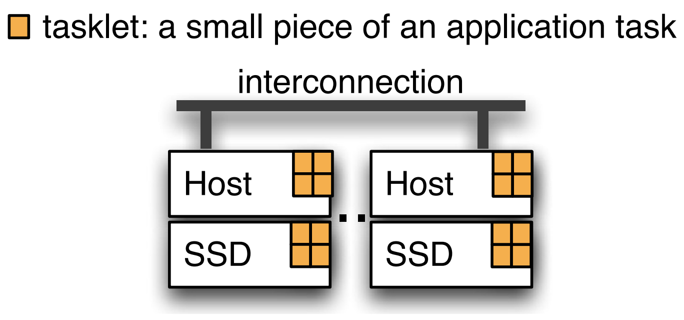


Figure 6.2: Smart SSD Model

consists of pages. A block is the unit of erase while the page is the unit of read and write. Flash memory arrays typically have multiple channels, allowing for a high degree of parallelism.

6.3.2 In-Storage Processing Model for SSD

While providing comparable or even better I/O performance and processing power than network-attached low-power cluster nodes, host system support for SSDs is still very limited. The TRIM command is the only standardized workaround to resolve sub-optimal performance issues in SSDs. In contrast to cluster nodes for which variable-length message passing is possible, current SSDs cannot handle requests smaller than a single sector, resulting in unnecessary data traffic and higher energy consumption. Maximizing the performance of SSDs is more difficult because operating systems are not designed to deliver large burst requests to devices. The need to change most of the low-level I/O subsystems makes it difficult for host systems to adapt to handle new storage devices efficiently.

In-storage processing is one way to alleviate this compatibility issue. Without changing any of the I/O subsystems in the operating systems, it allows SSDs to execute the I/O tasks internally, fully utilizing devices' internal components and their knowledge of the hardware and physical data organization. The Smart SSD model proposed in this paper is designed to support in-storage processing with minimal changes to the host system, providing the same device interface regardless of the underlying communication protocol. Instead, the details of the additional communications and task handling are hidden by a Hadoop framework, and the device exposes simple and flexible APIs that can take any in-storage processes that may need an arbitrary number of parameters to execute.

In the previous intelligent disk models depicted in Figure 2.2(b), a CPU in the disk drive is used as the main processing unit, executing the application tasks while the host conducts minimal tasks such as coordination and scheduling. In contrast, we use a CPU scavenging model where the host and devices share the workloads considering their computational and I/O processing power as illustrated in Figure 6.2. Each compute node in this model runs one or more *tasklets*, where a tasklet is a unit of an application task that can be assigned and executed on either the host or a device independently and in parallel. For example, if a database table scan operation is conducted per segment, each per-segment scan can be a tasklet, and the scan operation is an application task. The unit of computation can be adjusted based on computational complexity, resource availability in a device, or other factors.

In this approach, the host system plays two roles: computing unit and coordinator/scheduler. The host can assign a tasklet to a SSD based on its current utilization and the execution cost, which depends on the complexity of the algorithms, so highly complex tasklets can be performed at a host leveraging the powerful processors and large memory. For efficient scheduling, Smart SSDs can inform the host system about the current load of the device upon request. The mechanism for monitoring the load of each device depends on the type of host-device interface being used. The host may need to poll the devices in the SATA protocol, while additional background call-back connections can be established in SAS, and interrupts can be used for PCIe interfaces. In our prototype, which uses the SATA protocol, for example, the progress of a tasklet execution is reported to the host system as part of the return value of the read command to optimize the polling interval.

The host treats a Smart SSD as a single virtual processor, so the detailed hardware configuration of the device is not exposed to the host. If multiple embedded processors are available for ISP, the aggregated computing power of all the embedded processors is considered as the total computing power of the virtual processor. Tasklets can be assigned to one of the embedded processors of the virtual processor using cycle stealing. However, it is also possible to use dedicated processors for ISP for better performance. Regardless of the implementation, tasklets are scheduled between the processors considering priority and runtime execution costs.

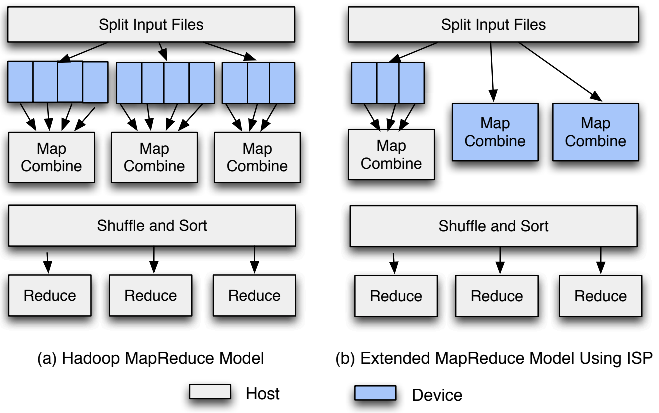

Split Input Files
Map Combine
Shuffle and Sort
Reduce
(a) Hadoop MapReduce Model
(b) Extended MapReduce Model Using ISP
Host
Device

Figure 6.3: Extended MapReduce framework

6.3.3 Host-Device Communication

In-storage processing requires some additional I/O commands to manage and execute tasklets. One of the criteria in designing these low-level device command sets is that devices should be able to provide a generic and flexible interface to applications. It requires support for tasklet parameters of arbitrary size such as input/output addresses and search keywords to avoid adding a new command for each tasklet. In addition, the interface needs to be independent from the underlying disk interface similar to the way the virtual file system layer is used for multiple different file systems.

To provide a clean and simple interface to meet those requirements, Smart SSDs use an extended object-based interface [45], which contains an *execute_object* command, and runs on top of the existing device interface such as SATA and SAS. By doing so, applications can have the same sets of APIs regardless of the underlying communication protocol. This interface allows users to add or remove a tasklet, group the logical addresses to be processed (input object), and store and read the results of the execution. Depending on the support for bidirectional communication, the host systems poll the results or get notification from the device when the results become ready. The detailed protocol implemented in our prototype is depicted in Figure 6.5 and described in Section 6.4.

The Smart SSD model uses the MapReduce programming model as an application interface because MapReduce can provide independent sets of input data that can be mapped to the tasklets directly, allowing them to be executed in parallel without any message passing or locking problems. Figure 6.3 depicts the extended MapReduce model for in-storage processing. A typical MapReduce application consists of a pair of *map* and *reduce* functions written by users. As illustrated in Figure 6.3(a), the *map* function is invoked after the partitioned data is read into the host memory, and the key and value pairs for the data are generated. A *map* function takes the generated input key-value pairs and produces a set of intermediate key-value pairs. All intermediate values associated with the same intermediate key are combined, shuffled, and sorted, then passed to the *reduce* function. The *reduce* function accepts intermediate key-value pairs, determines the set of values with the same key, and merges the values together. This model generates a large amount of disk and network traffic since the input data needs to be read into the memory before being processed, and the split data file needs to be sent over the network to the remote map nodes.

Instead of reading the raw file data into the host, the extended MapReduce model allows users to create an on-device map function, which internally calls the tasklets in the device, as shown in Figure 6.3(b). After splitting the data files, the framework sends an *execute_command* request to the device to execute the corresponding tasklet using a given range of LBAs; our on-device prototype re-uses read and write functions of the FTL so storing LBAs are maintained by our Hadoop file system and given to the device transparent to applications. However, logical addresses can be object IDs rather than simply LBAs if a device manages them on *write_object* requests, thus the same object-interface can be used with any types of physical transport layers. Each on-device map function performs combine and local reduce, before returning to the host system to minimize the disk traffic. The host system will then shuffle and sort the results from the on-device map functions, and invoke the *reduce* function. By doing so, as the size of input data increases, this in-storage processing model can save host CPU, I/O bandwidth, and memory resources. Specifically, since the amount of memory used to temporarily store input data files is dropped after the map function, it can also reduce the cache pollution at a host system.

6.3.4 Tasklet Programming

The SSD firmware does not usually provide general operating system features such as processes and dynamic memory allocations due to the limited size of SRAM and the unnecessary overhead from virtualizing hardware components. Rather, it directly accesses and

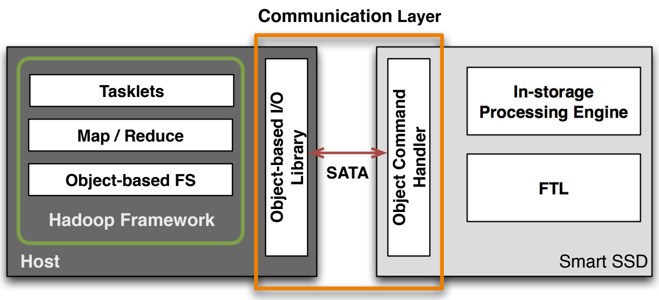


Figure 6.4: In-storage processing architecture for Smart SSD

manages the hardware resources. Therefore, porting full virtual machine-based programming languages such as Java or Python is not realistic, despite the advantage of secure execution of tasklets using sandboxing.

Instead, cross-compiled C code is the most efficient way of writing a tasklet, but it requires programmers to understand the details of the firmware implementation and hardware configuration. Moreover, the device can be exposed to many possible dangers such as overwriting of existing data and crashes, since providing a sandbox for native applications is difficult. Thus, only a small group of trusted people can write a tasklet and publicly deploy it for the target device.

Similar to CUDA [65], when the standard API for in-storage processing is defined, the tasklets can be programmed by a simple script language where only the standard APIs provided by the device can be used. The interpreter can verify illegal accesses to protected resources and execute the tasklet at the same time, or optionally compile the tasklet natively once verified.

While in this work we focus on web-log analysis, as discussed in Section 6.5.3, any applications that extract information from a large corpus of data can benefit from tasklets. For example, a desktop indexing tasklet can automatically tag incoming data, reducing the need for periodic crawling, and provide only the data blocks that a host-side application is interested in. As another example, multiple independent graphs stored across Smart SSDs can be processed concurrently. Thus, a host can collect only the matching leaf nodes from Smart SSDs, removing the need for transferring internal nodes.

6.4 Implementation

Figure 6.4 depicts the architecture of our Smart SSD prototype. It consists of the three major components: the ISP engine, the Hadoop MapReduce framework, and the communication layer. The Smart SSD implements an event-driven execution engine for ISP, which can process the tasklets assigned by the host. The Hadoop MapReduce framework is used as an application framework. Finally, the communication layer implements an ISP protocol between the application framework and the Smart SSD. This section describes the implementation details of each component.

6.4.1 In-Storage Processing (ISP) Engine

The ISP engine is an event-driven processing framework to execute tasklets using a C program cross-compiled for the ARM processors. The tasklets are executed on a dedicated processor in the device. Since dynamic memory allocation is not supported by the firmware, the tasklets are preloaded into the device, and memory space for input and output objects is reserved. Each tasklet is an object from the user's perspective, and identified by an unique identifier called an object id. To invoke the tasklet, the host passes an object id assigned to the tasklet at download time.

6.4.2 Extended Hadoop MapReduce Framework

Hadoop is an open-source project that provides a programming framework for large-scale distributed batch processing. Among various sub-components in this framework, we modified the Commons package of the framework to add an object-based file system for low-level device handling and to extend the MapReduce framework to support on-device map functions.

The goal of the object-based file system is to manage the raw SSD device by providing a resolution between an object name and an object identifier while hiding the low-level communications. Unlike the pure object-based file systems, which offload the entire block management layer to the device, it still manages the LBAs for reading and writing data blocks, because the FTL in the underlying SSDs is used. Therefore, while users can access data using an object identifier, the id and offset are translated into block numbers and sent to the device internally.

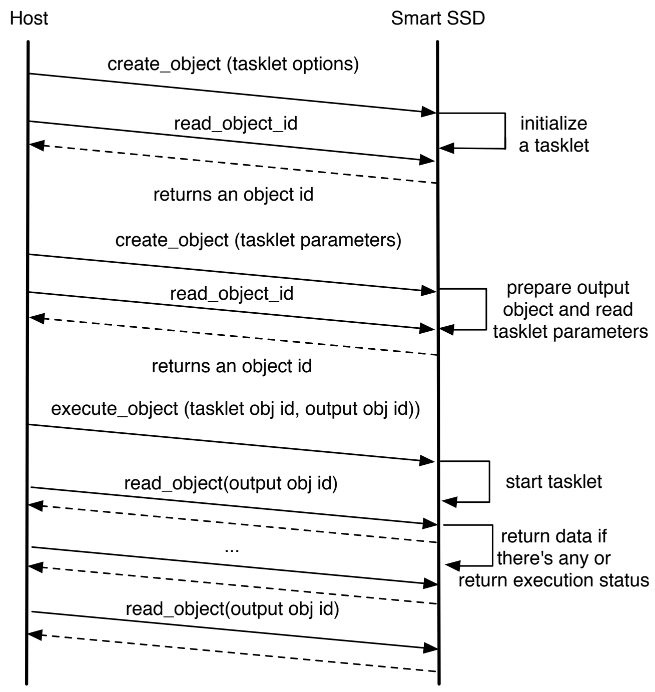

Host
Smart SSD
create_object (tasklet options)
read_object_id
returns an object id
initialize a tasklet
create_object (tasklet parameters)
read_object_id
returns an object id
prepare output object and read tasklet parameters
execute_object (tasklet obj id, output obj id))
read_object(output obj id)
start tasklet
...
return data if there's any or return execution status
read_object(output obj id)

Figure 6.5: Protocol diagram for ISP using the object interfaces

The extended MapReduce framework now supports *DeviceJob*, which internally manages the tasklets in the device, bypassing the file-read phase. Users can define a device job similar to how a normal job is configured; instead of using *InputFormat* and a host map function, applications can use *DeviceMapper* and *DeviceMapperFormat* where *DeviceMapper* represents one tasklet, and the *DeviceMapperFormat* converts the output of the *DeviceMapper* to a set of key/value pairs, which are then used by the *reduce* function.

6.4.3 Communication Layer

The communication layer is implemented on top of the SATA protocol. At the host end, the object-based I/O library handles the low-level requests and delivers the results to the object-based file system. In the device, the SATA command handler for the extended APIs is implemented in the SSD firmware. It interprets the vendor commands from the host and triggers the execution of appropriate tasklets stored in the device.

6.4.3.1 Object-based I/O library

The logical block addresses, encapsulated in an object request, are delivered to the device driver through a Unix system call, i.e., `ioctl` with the `ATA_PASS_THROUGH` command [29]. Each packet for vendor commands has a header that contains information such as command identifier, object id, data length, and others, followed by a payload containing the metadata for the commands. Since the `ioctl` operation is not supported in Java, this library is written in C and delivers the results to the Hadoop framework through the Java Native Interface (JNI), incurring the argument passing and copying overheads between a Java virtual machine and native C code. This interface interacts with a Hadoop file system and *MapReduce* job trackers, so it is not directly exposed to *MapReduce* applications. They can use the functionality by defining a device job specifying a type of in-storage processing and static parameters for the type without knowing the details of communications.

6.4.3.2 Object command handler

The object command handler in the device firmware is responsible for handling create, execute, and read requests on tasklets. Internally, these commands are implemented as vendor specific commands, reserved by the SATA standard for extensibility. In our prototype, three vendor-specific commands are implemented: *create_object*, *execute_object*, and *read_object*.

Figure 6.5 shows the protocol these vendor-specific commands use to run tasklets. The *create_object* command is first called to create and initialize a target tasklet with static parameters related to the tasklet such as the size of one key-value entry. Then, another *create_object* command is called to create an output object, which contains the outputs of the tasklet and also can deliver some runtime parameters such as search keywords and the maximum number of outputs for the tasklet. Once a tasklet object and the corresponding output object are ready, the host can invoke *execute_object* to start the tasklet, and call *read_object* to get the results.

While implementing the object command handler, we encountered several design issues because the SATA protocol does not support bi-directional communication, which can read and write data by sending one I/O request. When implementing on top of the SATA protocol, any command that has an output needs to be split into two steps under a global lock for atomicity. For instance, the *create_object* needs to return an object ID after creating an object. So the actual implementation of this command consists of two vendor-specific commands: one for object creation and another for the retrieval of the created object ID.

Similarly, the SATA protocol does not support concurrent or device-initiated connections, so polling is the only option to retrieve the outputs of operation from a device. Each polling operation has slightly less latency than a read request, but this operation is synchronous, so frequent polling would reduce the I/O throughput of an application. Therefore, we let the device return the number of processed pages whenever a polling request comes in, and the host dynamically adjusts the polling intervals to avoid a flood of requests.

6.5 Experiments

In this section, we explore the benefits and limitations of the Smart SSD using several benchmarks and applications. Since the benefits of Smart SSDs expect to come from the reduced host system resource usages and data transfers between a host and a device, the experiments are designed to characterize the in-storage processing with Smart SSD and aid us in identifying the classes of applications that can leverage the current Smart SSD architecture. First, we run a micro-benchmark that measures computing power and memory access latency of a device. Then, we measure the overhead of the object-based communication layer implemented on top of the SATA interface, focusing on the energy efficiency, data processing performance, and host resource usage. Based on the results, we identify the hardware components that limit the range of applications, and propose a future SSD architecture for in-storage processing.

6.5.1 System Configuration

Smart SSD: The Samsung SSD used in this paper has a 3 Gb/s SATA interface, 16 I/O channels, and a capacity of 200 GB. This SSD is equipped with two ARM processors, and its SLC flash memory arrays have an 8 KB page size. We used a commodity SSD that is currently on the market, and did not change any hardware components to support in-storage processing. However, we extended the firmware of the prototype to support the object command handlers and application tasklets. An internal read operation for tasklets is configured to fetch 128 pages at a time. Throughout the experiments, all I/O requests are sent sequentially to the device, and Native Command Queueing (NCQ) is turned off.

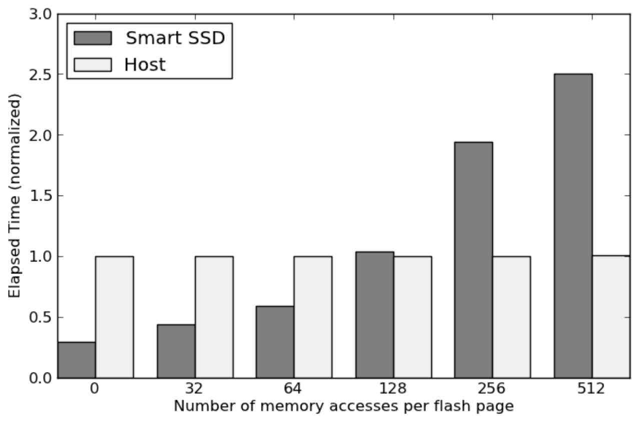


Figure 6.6: Performance of in-storage processing: for each memory access, Smart SSD reads one 4 byte integer and performs one integer comparison. The performance of the Smart SSD is normalized to the access time at the host side.

Host: Our experiments are conducted on a desktop machine with one 3.3 GHz Intel i5-2500 processor with 4 cores and 4 GB of DDR3 DRAM. We use a single SSD connected to a 3 Gb/s SATA HBA (Host Bus Adaptor) while a separate HDD is dedicated to the operating system and the Hadoop framework. The SSD and HDD do not share the HBA. This system runs Ubuntu Linux 11.04 and a modified version of Apache Hadoop 0.20.2 with default parameters.

6.5.2 Microbenchmark

Since the SSDs on the market do not target general purpose computing at all, they have the minimal hardware specifications that meet the design performance requirements. The devices have less computing power and higher DRAM access latency than the hosts while multiple DMA controllers take care of data transfers between host and device. The minimal hardware specifications, however, make the application tasklets more sensitive to the computation and memory access patterns in the case of in-storage processing. For this reason, understanding the performance characteristics of in-storage processing with our SSD device is crucial to achieving not only high energy efficiency but also high performance.

6.5.2.1 Read performance

By device specification, the internal read bandwidth of our device is around 2.5× faster than the I/O bandwidth between a host and a device. However, when including firmware overhead, our internal measurement shows the internal read performance is 1.8× faster than the I/O bandwidth between a host and a device, even when the 16 I/O channels are fully utilized. This not only means that SSDs can hide their firmware management latency from a host system, but shows the potential performance benefits of in-storage processing. When considering that it is difficult to utilize the whole I/O bandwidth to process a certain I/O task, there is a possibility that in-storage processing can improve the performance of applications by more than a factor of two.

6.5.2.2 DRAM Access Latency

To measure the internal memory access latency, we measure the read performance with 1 GB of randomly generated data varying the number of DRAM accesses and comparisons per 8 KB flash page. For each memory access, the benchmark reads one 4 byte integer and performs one integer comparison. For example, when the number of memory accesses per page is 32, it reads 128 bytes per 8 KB page, and performs 32 integer comparisons. As shown in Figure 6.6, the overall trend is that the execution time of the benchmark increases in proportion to the number of memory DRAM accesses and comparison operations while the performance of host is nearly constant. We measure the same tendency when increasing the number of comparisons, meaning that the DRAM access latency is high. This is mainly due to the limitations of the current SSD architecture where L1/L2 caches are not available and the bandwidth between CPU and DRAM is not high enough. It is to be noted that DRAMs inside the device are not designed to be accessed by the embedded processors. Instead, there is a small amount of SRAMs that contain the firmware and provide memory space for request processing, and DRAMs are used as a cache for read and write, whose contents are not directly accessed by CPUs most of the time. Although this design decision is natural for normal SSDs where processors are not involved in transferring the contents in DRAM to a host during basic read/write operations, it seems that hardware enhancements are required for Smart SSDs in order to broaden their target applications.

This tendency shows that under the current SSD architecture, Smart SSD can only benefit when the number of accesses per page is small (*e. g.*,, less then 128 for this device). The requirement for a small number of DRAM accesses can be interpreted in two ways. Applications that are interested in a small portion of data out of a large entity (*e. g.*,, one field out of large log entry or one column out of large tuple) can benefit from Smart SSDs so most data in a page are skipped. The pages transferred for such applications just pollute the caches in the host and waste host resources. The second group of applications have a small number of reads and a large amount of accesses and processing, so the processor can copy the data from DRAM to SRAM before processing. These groups of applications can benefit from the high internal read performance of SSD, amortizing the high DRAM latency penalty over the amount of accessed pages. This limitation can be easily alleviated by adopting CPU caches and increasing the bandwidth between a tasklet processor and DRAM.

6.5.2.3 Interface Overhead

Our custom Smart SSD protocol is designed and implemented on top of the SATA protocol so it can be used without modifying the current interface and the OS kernel. However our communication protocols add some overhead over the standard SATA commands because they must handle the polling interval and bi-directional communication issues.

Figure 6.7 shows the overhead of the commands supported in our prototype. We measure the turnaround time of each function at two locations: the Hadoop framework and JNI device communication module. Since the SATA protocol does not support bi-directional communication and variable-size requests, unlike object-based devices, a create command issues two I/O requests of 512 bytes each. Compared to a normal read command, which takes less than 215 us for one sector read, our results show that the *create_function* and *create_object* commands, each of which uses *create_object* commands with different metadata, take slightly more time. On the other hand, *execute_object* and *read_object* exhibit latency similar to that of a normal read command because the parameters for these operations can be embedded in one SATA command.

Due to the memory allocation and copies between Java and C, the JNI module adds 150–380 μus of latency to each function, depending on the amount of data passed to the function. Compared to the other operations, *read_object* experiences a long latency in the JNI wrapper due to the output data copied between C and Java and piggybacked information processing

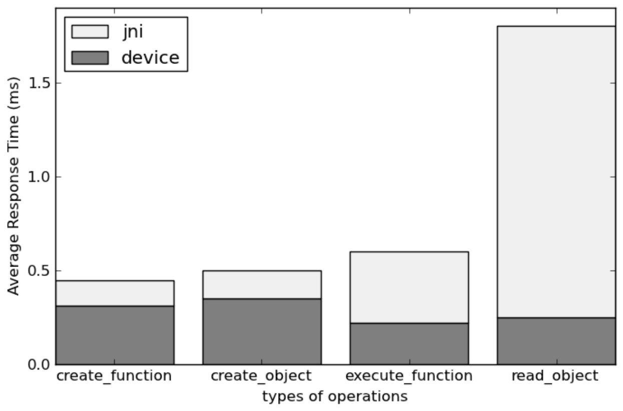


Figure 6.7: Overhead of an object interface implemented on top of SATA protocol; the raw I/O latency of the create command is higher than that in the execute and read commands, because it requires two vendor-specific commands. The JNI overhead of the read command is high due to the additional processing for polling interval and memory copy overhead.

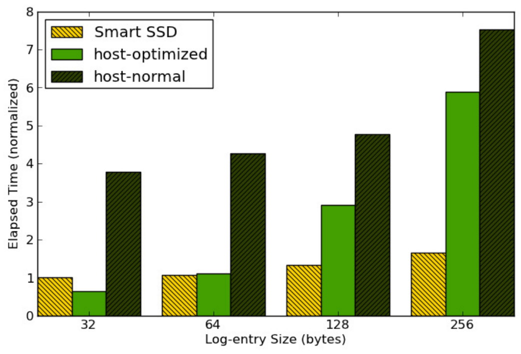
such as the polling interval.

6.5.3 Applications

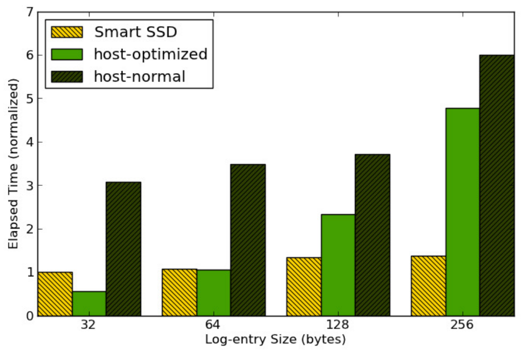
As discussed in the previous section, applications should have high data selectivity and low computing complexity to benefit from the use of our prototype Smart SSD. We understand that this guideline, due to the hardware constraints of the current SSD architecture, limits the range of applications. However, we expect that the applications that can leverage Smart SSD will grow over time because the computing capability of SSDs continues to increase. SSDs need more computing power in terms of the number of cores or clock rate and more DRAM space as the capacity and interface bandwidth of SSDs increase.

This section focuses on the potential benefits that Smart SSD can deliver with real applications. For this purpose, we evaluate Smart SSD with two applications: a web-log analyzer that uses a fixed size data structure, and a data filter that returns the only pages that contain valid data. These applications include searches through a fixed size data structure or accesses to a specific position within a flash page to retrieve information. As a result, the ISP engine can avoid a full page scan to find data of interest.

We use a real dataset named WorldCup98 which contains the requests made to the



(a) Number of accesses per geographic region



(b) Top 5 file types accessed per geographical region

Figure 6.8: Performance of log-analysis with two query scenarios. Smart SSD benefits from high internal bandwidth and no data transfer between a host and a device, losing performance slowly as the dataset size increases. However, the host-only computing models suffer from large data movements on larger datasets, decreasing performance.

1998 World Cup website [3]. We collected 7,000,000 distinct log entries combining the available dataset, and generated four data sets by adding padding to each log entry to create different sized log-entries; this changes the size of the raw workloads that need to be read into either a host DRAM or a device DRAM. In our example, the size of the smallest log entry is 32 bytes, and that of the largest entry is 256 bytes. The sizes of the data set with the smallest log entries and the largest entries are 240 MB and 1.8 GB, respectively.

We create two query scenarios that retrieve information from the log. The first scenario is to keep track of the number of accesses per geographical region, which requires one access to a field in each entry and storing of intermediate results. This query needs a reduce step to sum up the intermediate results per region. In the second scenario, a tasklet keeps track of the number of accesses per file type per region, and picks the top *N* file types per region. During the processing, it accesses one more field per entry (2 in total), and the results are not only merged but also sorted. Therefore, the second query requires more DRAM accesses than the first one.

6.5.3.1 Performance

To evaluate the performance of in-storage and in-host data processing, we implement three versions of Hadoop applications. *Host-normal* denotes a log analysis MapReduce application that uses a normal SSD and the default Hadoop protocol, and *host-optimized* is the modified version of Hadoop that minimizes the overhead of merging and sorting by storing intermediate results sorted and compacted only in in-memory data structures designed for log-analysis. With Smart SSD, the intermediate results of the *Smart SSD* application are stored and sorted inside the device, and are not sent to the host.

Figure 6.8 shows the performance of two scenarios with three types of clients. It shows that the performance gap between *Smart SSD* and the others becomes wider with larger log entries. As the log entry size increases, the number of entries per page decreases, which reduces the number of DRAM accesses accordingly. In addition, the amount of data to be sent to the host increases with larger entries. The high data transfer cost dominates the DRAM access penalty with large log entries. These results are consistent with the results of the microbenchmark in the previous section. One interesting observation in this experiment is the performance in the case of the 32 byte entry. *Host-optimized* shows better performance than the *Smart*

SSD, because the *Smart SSD* suffers from too many DRAM accesses, but receives little benefit from traffic reduction; 256 DRAM accesses are made per page while 240 MB of data are transferred through the 3 Gbps channel. While the same amount of data is transferred in *host-normal* and *host-optimized*, *host-normal* exhibits much lower performance due to the large amount of intermediate data, incurring frequently flushing of data, and high merging and sorting overhead.

The second query (Figure 6.8 (b)) shows a similar performance trend to the first query. One difference is that the *Smart SSD* catches up with the *host-optimized* slowly, and the performance gap is narrow due to the additional DRAM access per entry and the sorting process. This sensitivity to DRAM access counts can be improved by instruction-level optimizations, specifically considering the behavior of the load operation. However, we believe that small upgrades of the SSD architecture can alleviate this problem further, providing more flexibility in determining the tasks to offload. In terms of I/Os to flash, while both host-side clients need to read the whole data set from the device, *i. e.*,, 1.8 GB with a log entry size of 256 bytes, the Smart SSD client does not generate any data traffic over the system bus. This allows other devices in the host system to use the system bus while Smart SSD processes data in parallel.

6.5.3.2 CPU Usage

To evaluate the CPU overhead on the host side, we measure the CPU utilization of the host system while the three versions of Hadoop applications are running as shown in Figure 6.9. We illustrate the case with a log-entry of 256 bytes since all configurations show a similar tendency. The kernel is mostly waiting for the data from the device in the idle state, and the user state indicates the CPU time spent on the Hadoop application. The kernel is processing interrupts, and I/O and process scheduling in the kernel state.

As shown in Figure 6.9, the system with the Smart SSD client consumes very few host CPU cycles compared to the host-only versions of the applications. Even though the host CPUs occasionally wake up to check if the results from Smart SSD are available, they are mostly in the idle state waiting for a job to be finished so the CPUs can process other computation jobs or processes concurrently.

With a log-entry of 256 bytes, the host versions of the application are in the idle and kernel state for three or four times longer than the *Smart SSD* versions, waiting for the data from the device and delivering it to an application. The CPU time consumed in these periods

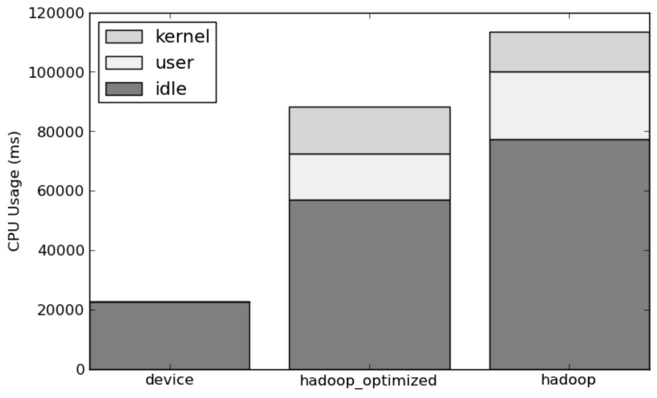


Figure 6.9: CPU usage (including all cores). The Smart SSD version of the application uses almost no host resources while the host versions of applications require kernel time to process I/Os and user time to process map processes.

increases in proportion to the size of data to be transferred and the number of other active process running in the background. Although large burst requests are preferred to maximize the throughput of SSDs, these requests may also increase the kernel and user time, creating an interval between I/O requests and thus lowering the utilization of devices' internal parallelism. For the two host versions, the large amount of intermediate data makes *host-normal* experience more idle time due to I/O waits than *host-optimized*.

While storage systems using SSDs can solve this problem by redesigning the data layout, I/O scheduler, and applications for SSDs, the Smart SSD model allows devices to handle this throughput issue by merely rewriting the applications to internally create I/O requests based on the number of available internal bandwidth at the time of the execution. The minimized use of the host CPUs further improves the overall energy efficiency as will be described later.

6.5.3.3 Energy Efficiency

Since Smart SSD uses low power processors and minimizes the use of host resources to process a tasklet, it is expected to consume less power than the host-based approaches. We plugged the host machine into a power meter device to measure its entire power consumption so we can measure the total energy consumed by the host system. Then, we ran the first query scenario for three versions of the application and measured the power consumption during the run. In order to avoid effects from other applications or the previous state of an execution, we injected a pause until the system became idle before resuming the next step. As shown in Figure 6.10, for each log-entry size, the corresponding dataset is synchronously written to the device, and the contents in the host page cache are flushed in the *wr* phase. Then, we force the application to sleep for 5 seconds, and then start the tasklet in the *exec* phase. When the tasklet terminates, we inject another pause before resuming with the next dataset.

Figure 6.10 shows the power consumption of each version of the log analyzer. In the idle state, the system consumes around 44 watts. In the exec phase, however, the power consumption of the host versions of applications increases by around 35 watts, and *host-optimized* and *host-normal* versions consume 75–82 watts in total. On the other hand, in-storage processing only increases the total energy consumption by 0.5–0.8 watts, saving more than 50% of the energy that the total host system spent. This is because the data movement is minimized in the Smart SSD model, and thus the host CPUs and I/O subsystem are not involved during the processing phase. Note that the *wr* phase is not needed when a tasklet works with existing data. In terms of energy consumption, offloading data processing or filtering jobs to the device before entering a big computation could be helpful even though the actual performance of the tasklet in the device is slower than in the host. This approach allows the combination of low-power processing in the device, as done in FAWN [11], with host-based processing for complex tasks to lower overall power consumption.

6.5.3.4 Data Filtering

To mitigate the impact of slow DRAM access in the current SSD architecture, Smart SSDs can be used to conduct a simple data filtering when excessive DRAM accesses are expected within a page. For instance, instead of identifying all matches in the page, we can send the page with the first match, and let the host do the complex computation with the page. As a result, the device can reduce the number of data transfers while the host processes less data. Since these filtering jobs can be issued to the devices before the host system starts long computations or processing other data, this tasklet could improve the overall data processing rate in a large storage system.

To show the effects of data filtering, we design a tasklet that returns the logical block addresses (LBAs) that meet a certain criteria given by a user application, instead of collecting

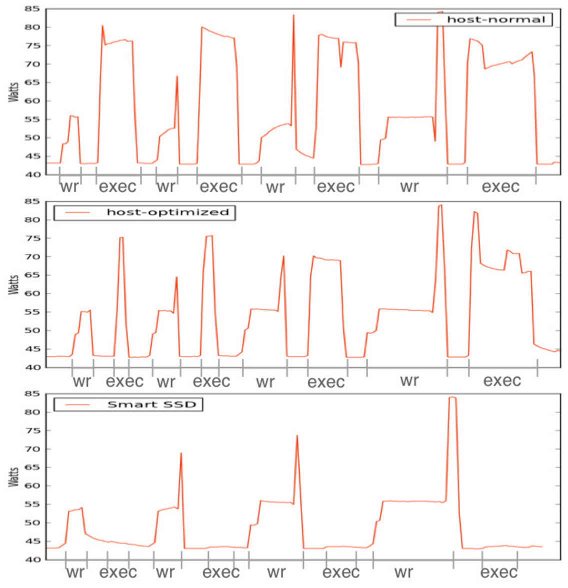


Figure 6.10: For each dataset, data to be processed is first written (*wr* phase) before executing the tasklet (*exec* phase). In the *exec* phase, the system that uses Smart SSD consumes less than 50% of the energy compared to the system that uses host-side processing, including idle power consumption, due to the internal, low-power processing.

and returning the results. Although the tasklet can also return the actual filtered data instead of the LBAs, we decided to return LBAs in order to allow the host system to choose the appropriate time to process these data. Once a match is found during the search, this tasklet stores the LBA of the page without looking at the values of other structures in the same page. Performance-wise, it can still suffer from slow memory accesses when data selectivity is very low but because this tasklet is processed in parallel with the host tasklets, it can improve the overall efficiency of the system by saving host resources.

We simulate an application that reads and compares one integer variable within a 32 byte data structure, just like the case of 256 accesses in Figure 6.6. On a 1 GB dataset with 60% selectivity, the performance of the data filtering tasklet becomes 40% faster than doing the comparisons for the entire data on the device, but it was 18% slower than doing it on the host. So, depending on the characteristics of data and computation, data filtering can be used to compensate for the low performance of the embedded processors while reducing the total amount of data transferred to the host.

6.5.4 Future SSD Architecture for In-Storage Processing

While the current SSD architecture provides enough processing power to support basic read and write operations, more hardware components and software layers are required to fully utilize the benefit of in-storage processing. Most importantly, CPU caches and high bandwidth between CPUs and DRAM need to be provided to avoid performance degradation due to the memory accesses. Adopting an application processor (or core) for tasklets would be helpful to avoid possible interferences between workloads generated by in-storage processing and user applications. This allows device tasklets and host jobs to work at the same time on the same device, thus making the system more balanced in terms of resource usage.

6.6 Future Work

We are exploring the use of Smart SSDs as a data processing engine in a large distributed environment such as social-network graph traversing or automatic replication management leveraging the internal information about the reliability of each flash cell. While the benefits from multiple Smart SSDs are expected to be similar to the aggregation of the benefits from individual Smart SSDs, one of the main issues in those systems we want to look at is the scheduling policy across multiple Smart SSDs and hosts. Since the performance of a job depends on the complexity of a job and the current load of a host and a device, we need to find a cost function for each job and assign a job to the device or the host depending on the cost. We are also looking at the use of Smart SSDs in large storage systems as a more advanced cache that has a large capacity and is capable of finding an entry quickly with extremely low power.

There remain several other optimizations that could improve the usability of Smart SSD. For example, the current object-based I/O library can be implemented inside the storage stack in the kernel, thus allowing the devices to access tasklets using a standard POSIX interface, *i. e.*, `fadvise()`. Once we find a cost function for in-storage processing, this layer can provide transparent job scheduling as well.

As for software, instead of using a cross-compiled C binary, support for scripting language in writing tasklets would be helpful in providing security mechanisms, such as sandboxing and authentication, to prevent malicious tasklets from destroying data or even devices. However, we believe that adding this feature to the current commodity SSDs is not practical because the firmware is not a general operating system. Rather, it is a state machine optimized for read and write requests, and it does not support processes, virtual memory, or even dynamic memory allocation. Therefore, for sandboxing, running a virtual machine based language, such as Java or Python, is not possible. Developing a new compiler or an interpreter that runs inside the firmware can cause high latencies in processing normal read and write requests. Our current implementation can check illegal parameters to the FTL functions, but cannot check for illegal memory manipulation that can cause the device to crash.

Having an application processor that can run a lightweight operating system could alleviate this problem, allowing application processors to run interpreters and issue I/O requests to the firmware processors. Then, an abstraction layer of hardware components and FTL functions in the device could be provided to tasklet developers so they can dynamically add or remove their own tasklets without having specific knowledge about the implementation of a SSD firmware.

6.7 Summary

Modern data centers face a common challenge that the supplied power limits their computing capacity. A trend to tackling this challenge is to consolidate servers through system optimization and low power technologies, releasing more space and reducing operating cost. Smart SSDs can contribute to this server consolidation by providing low power data processing capability. The ARM processor employed by SSDs consumes a fraction of power compared to the general purpose processors. Even though new Intel processors such as Ivy Bridge-based Intel Xeon processors and Atom-based Centerton processors are expected to consume less power, servers are still power-hungry in general. In this paper, we showed the potential of Smart SSD to not only reduce power consumption but also improve performance.

Unlike rotational hard drives where mechanical movement determines the performance, the performance of SSDs has improved with more powerful processors and increasing hardware components such as channels and memory as well as better FTL algorithms. Despite the functionality that makes it a complete low-power computing system with CPUs, memory, and storage, a host system still thinks of an SSD as a dumb I/O device, passing up the chance of optimizing its data flow. To address this shortcoming, we explored the potential benefits of SSDs as data processing nodes and identified the limitations of the current SSD architecture. We presented a multi-functional storage device, Smart SSD, that harnesses the processing power of a device using an object-based communication protocol. Smart SSDs rely upon tasklets: independent I/O tasks of an application running inside the device. To allow applications to better use SSDs, we developed a programming interface to execute tasklets based on MapReduce. We implemented the Smart SSD features in the firmware of a Samsung SSD and modified the Hadoop Core and MapReduce framework to use tasklets as a *map* or a *reduce* function. An object-based command set for Smart SSDs is created to realize in-storage processing on SSDs. To evaluate our prototype, we ran a microbenchmark and a log-analysis application on 7,000,000 entries in both a device and a host. We found that under the current SSD architecture, excessive memory accesses will make the tasklet execution slower than in the host due to the high memory latency and low processing power. However, the results with a log-analysis example show that our Smart SSD prototype consumes 2% of the energy compared to the host versions of applications in processing a given workload, saving more than 50% of the total energy while providing up to 2–3$\times$ better performance, depending on the entry size.

Chapter 7

NVRAM Wear-leveling

The limited write endurance is one of the obstacles in using high density byte-addressable non-volatile memories (NVRAMs) as a storage medium. Most proposed solutions are either constantly managing the wear-levels of all the lines, or adopting a coarse-grained wear-leveling based on a global counter, causing the lifetime of a device is bounded by the weakest endurance cell or an inaccurate wear-leveling.

We introduce a probabilistic wear-level tracking that can efficiently maintains a wear-level of any data type utilizing an infrequently updated 1B counter per type. The main idea behind this is to reduce both update frequency and storage overhead of a wear-level counter, exploiting the uniformity of a random function; for example, the counter will be updated when a randomly chosen number between 0 and 400K becomes 0, incurring an 1B access overhead per 400K overwrites. Due to the small space and access overheads, this also allows NVRAM devices to track wear-levels at various granularities; it can track a wear-level per cache line, block, segment or object, or use multiple granularities together.

Instead of using one fixed wear-level tracking granularity, we integrate the probabilistic wear-level tracking with NVRAM data structures, allowing each structure to select appropriate granularity based on its type; frequently updated metadata such as *inodes* and index nodes can use a line-level tracking and line-swapping internally, and for objects that are written as a whole, a device can reduce space and access overheads by using per-object (coarse-grained) tracking and an object swapping. By doing so, each data structure can recognize its wear-level and be swapped internally or externally with other data structures when its wear-level is higher than others. Additionally, due to the multi-granularity tracking, it makes possible to manage small weaker regions, not causing the many other adjacent young cache lines to be disabled together.

In designing NVRAM data structures for indexing and data layout, our goal was to reduce storage and access overheads while supporting object swapping efficiently. We use multiple block sizes to reduce internal fragmentation, and extent-based block management; the free blocks are organized by their wear-levels and sizes using a modified buddy allocator. To prevent their own metadata, such as free block header, to be worn out, it is either randomly placed inside its structure or probabilistic wear-level tracking and line-swapping are used. These structures are encapsulated under the object-based storage model, which enables tightly-coupled integration while allowing drop-in replacement for newer NVRAM technology, which is important for emerging technologies like NVRAM. The specific contributions of this chapter include:

- A probabilistic wear-level tracking that demonstrates the use of a uniform random function to track the wear-level of each data structure to achieve low runtime and storage overheads while allowing multi-grained wear-level tracking.
- Extent-based and wear-aware NVRAM data structures designed for efficient object swapping integrated with a probabilistic wear-level tracking.

7.1 Endurance of Byte-addressable NVRAMs

New types of high density, byte-addressable NVRAM technologies such as phase-change RAM (PCM) and magnetic RAM (MRAM) [51, 21] have received increasing attention in storage systems as flash memory is approaching their density limits. However, the limited endurance of high density NVRAM cells that provide Gb densities has been one of the major obstacles in using NVRAMs as a storage medium. Several other technologies such as Nano RAM and STT-MRAM [63, 24] are promising unlimited endurance, but their densities are still in Kb or Mb densities. Pushing NVRAMs to higher density would further decrease endurance as we have seen in TLC NAND flash memory.

Unlike flash memory where counting the number of erases per erase-block is sufficient for wear-leveling, next-generation NVRAMs require a finer-grained wear-level control due to their random accessibility and byte-addressability. Several proposed approaches [72, 70, 27] uses either a consistent, fine-grained wear-leveling, which can result in lower device lifetime under high endurance variance or incur unnecessary data movements. Coarse-grained wear-leveling [70, 27] that maintains a wear-level counter per segment or a large block, on the other hand, suffers unnecessary data movements due to their inaccurate wear-level tracking.

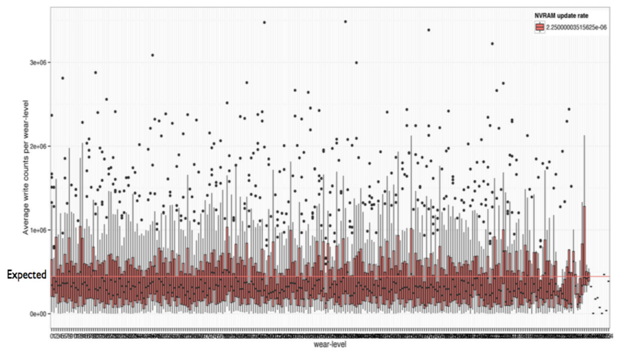
This chapter is organized as follows. Section 7.2 introduces a probabilistic wear-leveling tracking and Section 7.3 discuss the design of object-based NVRAM data structures tightly integrated with the tracking mechanism. We summarize the implementation details and the methodology for experiments and experimental results are presented in Section 7.4. Section 7.5 concludes the chapter with a discussion on future research directions.

7.2 Probabilistic Wear-leveling

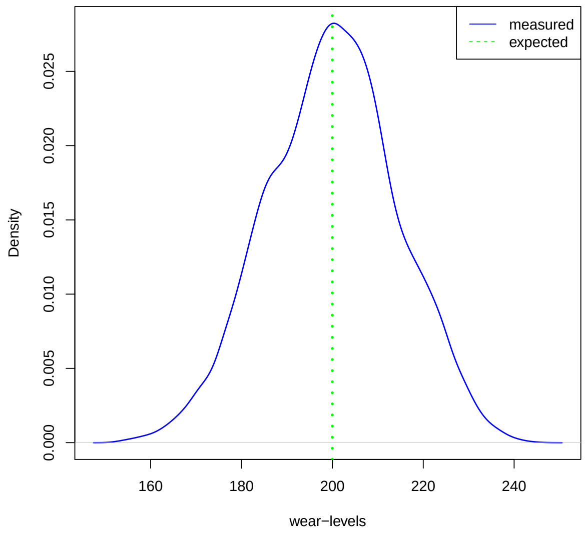
In flash devices, per-block erase counters have been sufficient information to provide wear-level tracking because of its erase-before-write nature; pages in an erase block cannot be re-programmed without cleaning the whole block. However, data in byte-addressable NVRAMs can be randomly accessed and updated, causing small, irregularly worn-out regions. Existing approaches to this issue use either a consistent wear-leveling using small sets of global counters or a coarse-grained wear-leveling exploiting file system data structures.

However, both approaches have their own issues. While consistent wear-leveling policies can incur small storage overhead, data needs to be consistently moved around even cells are still young. Moreover, the lifetime of a device is bounded by the weakest cell in NVRAMs, because the mapping is not stored on a data structure; the address of moved locations are determined either arithmetically or by hardware. Thus remapping is not easy; if one line dies in a region, the whole region will become unusable without additional worn-out region mapping requiring look up costs for every read/write operation. When one of the region is invalidated due to the weakest cell in the region, the whole device would become unusable soon.

Coarse-grained wear-leveling approaches, on the other hand, are used in systems where their access units are fixed. For example, PFFS uses a segment swapping based on a per-segment write counter similar to flash file systems, and additionally shift pages in a segment when swapping. BPFS uses a random page swapping based on a global write counter, and within a page, use a hardware bit shifter. However, in some cases, they may suffer from large data movement overheads or shorter lifetime due to inaccuracy in tracking; if more than a half of the segment is updated frequently, some pages in a segment can still be hotter or colder than



(a) Write counts per wear-level



(b) Wear-levels at endurance limit

Figure 7.1: Accuracy of a probabilistic wear-level tracking

others because multiple pages are shifted at once. Random swapping may not handle bi-modal distributions of updates.

We try to address these issues by enabling multi-grained wear-leveling exploiting the uniformity of a random function. Instead of updating a counter each time the data is overwritten, the counter is update when a random number chosen in a range of 0 to F becomes one predefined value within the range. For example, when F is 400 K, the wear-level counter W can be increased by one only when the random number becomes 0 (can be any number in the range). If the endurance of a NVRAM is around 10^8, it will generate around 245 wear-levels where each level represents 400K overwrites. By doing so, we can reduce the size of a wear-level counter from 8 bytes to 1 byte, and its access overhead becomes $1B/400K$ overwrites.

Since it's possible to use 1B small counters to keep track of wear-levels and reduce its update frequency, finer-grained wear-level tracking can be implemented at a low cost. If we assign a counter for every cache line, the total space overhead becomes around 1.5 % while keeping the access overhead of each counter near zero ($1B/(400K*64B)$) when the size of cache line is 64B.

Besides a small wear-level tracking overhead, another benefit of a probabilistic wear-leveling is that it enables a fine-grained worn-out region management, preventing a premature failure when an endurance variance of each cell is high. In previous wear-leveling techniques where wear-leveling is performed per a region or segment, the whole region is invalidated when one of its line is worn out. After the region is invalidated, wear leveling in the region ceases. The remaining data in the region must be moved to prevent dataloss, however, the lack of a mapping table makes this a difficult task.

Compared to using write counters, a probabilistic wear-leveling may incur some errors because it relies on a random function's performance. To understand its accuracy, we measure differences between the expected number of writes and actual number of writes while writing a single cache line 10^8 times with $1B/444K$ update frequency. Figure 7.1(a) shows the actual write counts of each level. Although each level is expected to represent 444 K writes, there are several instances that are much greater than that. However, when we accumulate the wear-levels until it reaches the endurance limit, as shown in Figure 7.1(b), lines are expected to be worn out at the expected wear-level with an error of 23%. Since each NVRAM cell might have different endurance, keeping a perfectly accurate counter would not be needed. Our wear-leveling policy avoids reusing lines within this range until swapping brings all other lines within the same range. Furthermore, the wear-leveling policy simply invalidates the cache lines associated with worn-out objects.

The purpose of wear-leveling is to not only find worn-out lines but also remove hot regions that are frequently updated. This is because as hot regions will be invalidated quicker than other regions, thus reducing the capacity of a device. Therefore, once a hot region is detected using a wear-level counter, it needs to be swapped with colder regions to avoid frequently updating the same hot region. To provide this, we partition wear-level space into multiple zones from the safest to most dangerous. Each zone is a checkpoint guaranteeing that the regions belong to the next zone will not be used until the regions in the current zone do not exist. The maximum wear-level of each zone is used as a threshold for swapping; an object or data will be swapped with other nodes whenever its wear-level exceeds each threshold. By doing so, instead of consistently moving data, it scatters hot writes only when needed, reducing data movement overheads and thus increasing the lifetime of a device.

For efficiency, swapping and tracking can be performed in a different granularity depending on a data type. For example, a large block or object that is written as a whole can have one counter instead of multiple per-line counters. A tree node, which contains many of slots updated independently, can have per-line counters and do line-swapping within a node to avoid irregular worn-outs. In the next section, we describe how swapping and tracking can be integrated with common NVRAM data structures, enabling an efficient multi-grained wear-leveling.

7.3 Object-based NVRAM

This section describes a prototype of an object-based storage device for NVRAM that uses a probabilistic wear-level tracking. Systems built upon our object-based storage model will offload the data management later to a device with a rich variable-length object interface delivering file system semantics. This allows co-optimizations between dedicated hardware components and NVRAM software similar to SSDs while enabling an informed data management like file systems. By isolating device-specific technology, now file systems become independent to underlying storage media, thus making it easy to support hybrid devices and future NVRAM devices without modifying the host system.

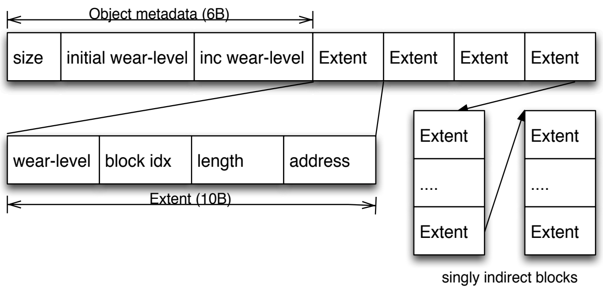


Figure 7.2: Structure of an onode

To demonstrate the effectiveness of a probabilistic wear-leveling, we design a NVRAM object store whose object-model is similar to Amazon S3 where an object is a unit of writing. It consists of two parts; a wear-aware b+ tree and a wear-aware free space management, each of which is designed to have low operation and space overheads, integrated with a probabilistic wear-level tracking. Our device uses extent-based data management for space efficiency and supports multiple block sizes to reduce internal fragmentation. Depending on data type, either line swapping or object swapping is used to scatter hot writes.

We now discuss our object model in Section 7.3.1 and the design of a wear-aware B+ tree in Section 7.3.2, and we subsequently describe our buddy allocator used for free space management in Section 7.3.3

7.3.1 User Objects

An object is a unit of writing, which can be of any size. Since not all objects are block-aligned in byte-addressable NVRAM devices, one of the primary concerns for the data management layer is to determine a block size; large blocks would incur internal fragmentations and using small blocks would increase indexing and searching overheads. To alleviate such issues, our device uses multiple block sizes to reduce internal fragmentation, and extents to avoid creating an index per block when storing objects.

Figure 7.2 shows how object data is managed in this system. Each object is stored in a data structure called an onode, which is an inode for an object. It consists of object metadata, including size, initial wear-level and incremental wear-level, and a list of extents. An initial wear-level represents a wear-level of the most worn-out block belonging to an object at the time of allocation, and an incremental wear-level counts the number of overwrites after its creation. Thus, the sum of these two values becomes a wear-level of an object. We separate them to avoid updating a wear-level of each extent whenever an object is overwritten; instead, only an incremental wear-level is updated at a probability of $1B/K object overwrites$.

An onode is designed to fit in the value field of a wear-aware B+ tree so its wear-leveling can be performed directly by an index structure; wear-leveling for extents and singly indirect extent blocks is not required because they are read-only until an object is freed.

7.3.1.1 Object Life-cycle

Create given an object create request, a set of consecutive blocks are first allocated from a free block pool in order from lowest wear-level to highest. Then an onode is constructed using the blocks and a pair of object ID and onode is stored in a wear-aware B+ tree.

Write or Update upon a write request, data is first written to the blocks allocated to an onode. Then, a random number is generated using a uniform random function to determine if wear-level updating or swapping is required. If it equals to a predefined value, which is chosen between 0 and K, the value of an incremental wear-level is increased by 1. If the wear-level of an object reaches to the zone threshold, it initiate an object swapping in the background as described in the next section, and return to the user.

Free upon an object delete request, it first calculates the wear-level of an object by adding an incremental wear-level to the wear-level of each extent. A device, then, returns the allocated extents to a free block pool, and the pool will organize blocks using its block size and new wear-level given for each extent for fast retrieval next time.

Worn-out If an error occurs during writing a block, the whole extent containing the block is copied into memory because the other blocks belong to the extent are also likely to fail. Then, the blocks will be returned to a free block pool except for the block with an error. A device will re-allocate the extent to see if it be constructed with other younger blocks. By doing so, the bad block management is done by dropping the worn-out blocks without requiring additional mapping.

If the wear-level of an object reaches to the maximum wear-level of 255 without any error, it will stop increasing the wear-level while keeping its allocation priority to the lowest.

7.3.1.2 Object Swapping

The purpose of wear-leveling is to enhance the lifetime of a device without keeping decreasing its capacity. To guarantee this, hot regions need to be either not created in the first place, or detected and removed once they become hot. We take the second approach in our device because it is common for file system workloads to have locality and bimodal distribution. Also, completely preventing hot regions is not a easy task even for consistent wear-leveling approaches. For detection, we use an incremental wear-level counter per object, and to balance the heat, object swapping is used between hot and cold objects.

For hot object identification, the wear-level or temperature of an object is calculated by $initial\ wear-level+\ incrementalwear-level$ whenever the value of an incremental wear-level increases. An object is classified as hot if the wear-level reaches the wear-level zone threshold and there are free blocks or objects that have not reached this threshold yet. If colder free blocks exist, the hot object is copied into memory and reallocated with free blocks. Otherwise, object metadata is searched to find a colder object to be swapped with.

Since object metadata (an onode) is stored in a wear-aware B+ tree as a value, finding out the wear-levels of other objects can be done by scanning the values in the leaf nodes of the tree. More specifically, we examine several other objects belong to the same leaf node where the detected hot object are stored. If the minimum wear-level of these objects is smaller than that of the current object, the hot object and the object with the minimum wear-level are swapped. If such cold objects are not found, it also expand the search area to its neighbor nodes.

In this process, a device checks a limited number of its neighbor nodes and are not trying to find an object with the minimum wear-level across all objects. This is because it is not guaranteed that the coldest object will exhibit the same infrequent update pattern in the future. Thus, our goal is to consume all the blocks belonging to the lower wear-level zone before using the blocks in the next wear-level zone.

Once hot and cold objects to be swapped are found, its content will first be copied into memory, and the two objects are freed, returning all extents back to the free block pool. Then, a hot object and a cold object are created again in order; the free block pool always returns the blocks with lower wear-level first so now a hot object as will be discussed in Section 7.3.3.

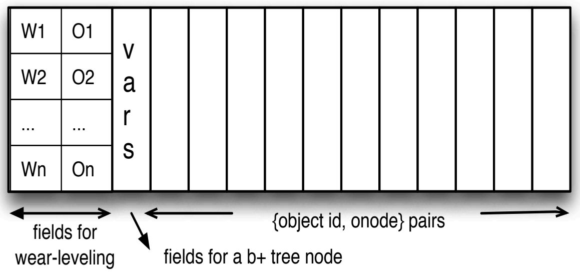


Figure 7.3: Structure of a wear-aware B+ tree node

7.3.2 Wear-aware B+ Tree

Various index structures such as hash tables, page tables and trees can be used to manage the location of data blocks and their metadata for byte-addressable NVRAMs. In this work, we choose to use a b+ tree, because it requires smaller amounts of changes to the original structure than other data structures while not incurring frequent relocations or inconsistent lookup overheads. For example, chained hashing requires the pointers of neighbor nodes to be updated when one node is added or removed. Balancing the length of chains would need more internal data movements. Cuckoo hashing [68] has a good lookup performance but may need to relocate multiple entries when hash collision occurs. Page tables can be useful when we use one fixed frame length because the address space needs to be partitioned into multiple regions, but it is not suitable for our design goals.

A pair of an object ID and an onode is stored in a b+ tree; an object ID is an unique key for an object, and an onode manages the location and wear-level of blocks belonging to an object. Unlike onodes where all of its data is updated with the same frequency, the key-value pairs in tree nodes may exhibit different access frequency depending on a workload, requiring a finer-grained wear-leveling. To realize this, we use a per-cache line wear-level tracking for tree nodes, which adds an 1B wear-level counter and an 1B offset to each cache line. Once hot lines are detected, a line swapping will be initiated to balance the wear-level between lines. An object swapping will be used when a tree node becomes hotter than other tree nodes to spread the heat across b+ tree nodes. We will now discuss the changes made to a b+ tree in detail, and swapping mechanisms in next sections.

7.3.2.1 Inner-node Wear-leveling

Figure 7.3 shows the internal structure of a leaf node where each slot is logically aligned to a CPU cache line size of typically 64B or 128B; the cache line is the minimum unit of an internal writing in our device. For wear-leveling, two additional information is added to a tree node, occupying two cache lines. The first line stores the wear-levels of all lines in a node, and the second line contains an actual offset of each line. The size of a tree node is fixed to the square of a cache line size so that all wear-level counters and offsets for a tree node can be stored in two lines. For example, if a cache line size is 64B, a node size becomes 4KB containing 64 lines. 64 1B wear-levels and 1B offsets are stored in the first and second line, respectively.

The line swapping is started when the difference between the minimum wear-level and wear-level of the current line being updated is higher than a threshold. The new locations of the two lines are stored in offset fields, which is indexed by a logical line offset and returns the physical line offset within a node. Therefore, the physical address of a line n can be calculated by $treenode\ address + O(n)$ where O(n) is a value in the n-th field in the second line.

To avoid scanning to find the minimum wear-level, it uses the fact that the two additional lines do not require wear-leveling. Because each 1B wear-level or 1B offset can be updated up to 256 times, the maximum number of updates per line can be up to is 16K or 32K depending on a cache line size, which is substantially lower than the endurance limits of most NVRAMs. Thus, we use the first two wear-levels (W1 and W2) to store the minimum wear-level and the number of lines with the minimum wear-level, eliminating the need for scanning while not requiring additional line updates.

7.3.2.2 Node Swapping

The wear-level of a tree node is then determined by $minimum\ wear\text{-}level + threshold$. Whenever the minimum wear-level increases, a device first checks if the wear-level of a tree node has reached to the wear-level zone threshold. If then, we randomly inspect multiple tree nodes by sampling to see whether a colder tree node exists or not. Swapping is not required if a tree node with the minimum wear-level is greater than the wear-level of the current tree node. Otherwise a node swapping is initiated between the two tree nodes meaning that the current object was one of the hottest objects in the current zone and needs to be swapped with colder objects to avoid a possible premature worn-out.

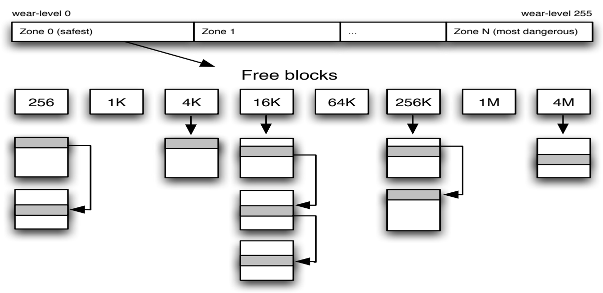


Figure 7.4: Free space manager

7.3.2.3 Wear-leveling Overhead

The storage overhead for tree node wear-leveling is 3% and 1.5% when the cache line size is 64B and 128B respectively, since 2B are used per cache line. Its access overhead depends on the workload, but since updating a wear-level happens infrequently at a rate of $1/K$ *line writes*, line swapping happens even less frequently only when hot/cold workload is given. On a uniform workload, line swapping will not be started because the wear-level differences between lines would be less than a threshold, not incurring unnecessary data movements.

Node swapping can be occurred up to N times where N is the number of wear-level zones. If 64B cache lines are used, each 4K tree node can be copied up to N times so the swapping overhead per node is $4K * N$. Because the number of node changes as more objects are stored, the total swapping overhead can only be measured under the context of workloads. We will show the access and storage overheads for the wear-aware B+ tree under various workloads in Section 7.4.

7.3.3 Wear-aware Buddy Allocator

This section describes our memory allocator that is designed to support an multi-block, extent-based allocation and a wear-leveling. By using multiple size blocks, it tries to minimize its internal fragmentation and with an extent-based free space management scheme, it can provide an efficient merging and splitting without having many small blocks.

Among various memory allocation schemes such as page tables, log-structures, we choose to use a buddy memory allocator, which divides memory into multiple different size blocks and use splitting and merging to satisfy a memory request. Per different size block, it maintains a doubly linked list of free blocks of the same size. Its metadata called a free block header is typically located in the top of a free block containing state flags and pointers to the previous and next blocks.

There are several issues to be addressed to use a buddy allocator as a storage allocator. First, while a binary buddy algorithm is typically used in *malloc* implementations, it may incur frequent merging due to a large number of required block sizes, and thus increasing the need for the wear-leveling. For example, when using a small minimum block size of 256 B, 15 different block sizes are needed to allocate up to a 8MB consecutive block. Using a large minimum block size, on the other hand, would generate high internal fragmentation on small writes. Second, using a high order buddy algorithm would require link traversal because more blocks need to be checked to see if merging is required. Lastly and most importantly, the wear-leveling for free-block header is required that is updated when allocating, freeing or merging.

To alleviate these issues, we design a wear-aware free space management scheme that uses randomly placed headers and extents based on a buddy memory allocator. As depicted in Figure 7.4, each wear-level zone has its own buddy allocator so free blocks in a lower wear-level zone can be easily found and consumed first. Each buddy allocator has an order of 2 where 4 consecutive blocks can be merged. To avoid a link traversal for merging, we make a free block header contain an extent that includes a pointer to a next block and a length. By doing so, 8 different blocks can cover blocks from 256 B to 4 MB, and allocate up to a 16 MB consecutive block. The wear leveler determines whether merging is required by simply checking the length of each extent.

The free block header of each free block and the head of per-block link list are hot objects in this buddy allocator. For wear-leveling, we place the list heads into a separate node, called an address node, and apply the same line-level tracking and line-swapping used for tree nodes. For per-free block headers, we place the header randomly within a free block to avoid updating the same location repeatedly. We can easily adopt the same technique we used for tree nodes, but we choose to use a random placement not to increase its storage overhead.

Tree nodes are allocated from the separate pool to allow object swapping between tree nodes. The pool is first initialized by allocating a fixed-length consecutive memory region from the buddy allocator. The allocated region is then divided into multiple tree node blocks, each of which will be initialized as a tree node; each block becomes a free tree node with an appropriate header: wear-levels and offsets. The free tree nodes are connected via a singly linked-list within each wear-level zone; a pointer to a next node resides in a tree node and its wear-level is managed, and the address of the head of each linked list is stored to the address node for recovery. When allocated, the node is evicted from the head of the free list and managed by a wear-aware B+ tree.

New memory regions are allocated when there is no free tree node. The allocated regions are released when a no swap target is found during a tree node swapping, and younger free blocks exist in the buddy allocator; this will require any live tree node in the region to be copied to a new region, but they will not incur additional overheads because they will be eventually copied to a new region using object swapping as their levels are above the average and there are younger free blocks.

7.3.3.1 Operations

Allocation Extents for user objects are directly assigned from the buddy allocator. Upon a request, a device divides the requested size into the smallest number of extents. Then, it tries to allocate blocks starting from the safest wear-level zone, and return the generated extents. If the requested free blocks are not available, it tries to either merge or split blocks to generate the requested blocks.

Free When an object is freed, its free blocks are inserted to the appropriate buddy allocator, using a block size specified in an extent containing the block and a newly calculated wear-level. Merging can be followed when the length of a free block extent becomes 4.

7.4 Evaluation

In this section, we explore the accuracy and overhead of probabilistic wear-leveling. Our experiments are designed to understand the effects of random wear-level counters, wear-aware data structures and two swapping algorithms. We measure the lifetime and write distribution of a device, and the overheads of line-swapping and object-swapping mechanisms with or without endurance variance.

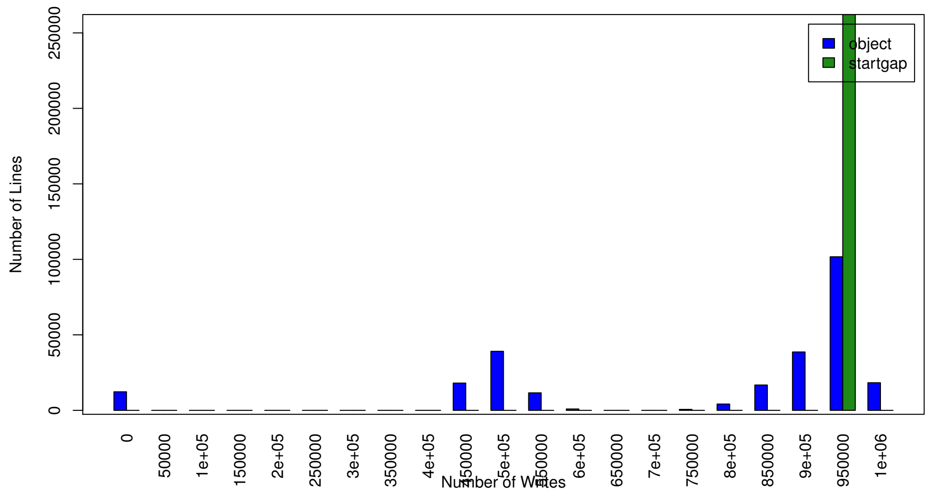
Our object device simulator is implemented in c++ as an user-level application. It consists of workload generators, NVRAM data management layer, pluggable wear-leveling policies, and byte-addressable NVRAM simulator. There are two types of workload generators; uniform and pareto. The pareto distribution is configured to meet the 80-20 rule where 80% of the workload consists of 20% of the objects, and the uniform workload represents the ideal workload that does not require any wear-leveling. The NVRAM data management layer contains the common data structures for NVRAM key-value stores, such as free space management and data allocation policy, and is shared by wear-leveling policies. Two wear-leveling policies are implemented; start-gap and object wear-leveling. Finally, our device is designed to simulate NVRAM using DRAM recording the actual number of writes per cache line and the overhead incurred by each wear-leveling policy. Endurance variance is simulated using a normal distribution having a mean of the endurance limit, and a standard deviation of up to 10 % of the endurance limit; currently, endurance variance within a chip is not known so we assumed in this evaluation that the variance would not exceed 10 %.

The experiments were conducted on a Linux machine with 128 GB of memory. 16 MB of NVRAM is simulated using DRAM. The endurance limit of a device is set to 1 M for faster simulation. Other parameters for wear-leveling are set according to this endurance limit; the update frequency of a random counter is configured to 5000, meaning that the device is expected to be worn out at the wear-level of 200. We configured it to have 6 wear-level zones; the highest wear-level in each zone is 140,158,165,180,190 and 256, respectively. Start-gap is configured to have one region and move gap lines every 100 writes as recommended in the article. In each experiment, unique 4 KB objects are generated to fill the 70 percent of the device capacity. We write the objects until a device becomes unusable following either a uniform or pareto distribution.

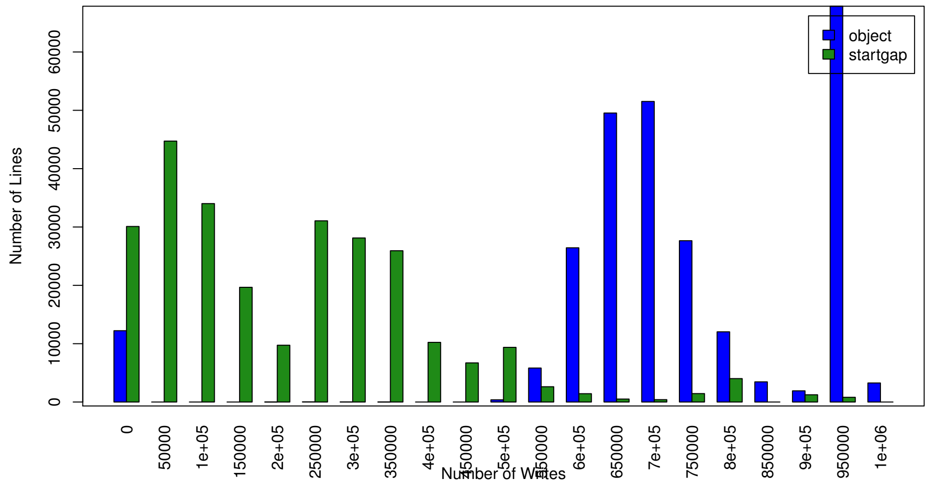
7.4.1 Write Distribution and Lifetime

One of the major goals of wear-leveling is to distribute writes across a device to prevent some hot lines to be worn-out earlier than others, reducing the capacity and lifetime of a device. To show the effectiveness of probabilistic wear-leveling, we measured the actual number of writes per cache line while running a uniform and pareto distribution with or without an endurance variance.

Figure 7.5(a) shows the histogram of a number of actual writes with no endurance variance under a uniform workload. The y-axis represents the number of lines, and the x-



(a) uniform



(b) pareto

Figure 7.5: Write distribution

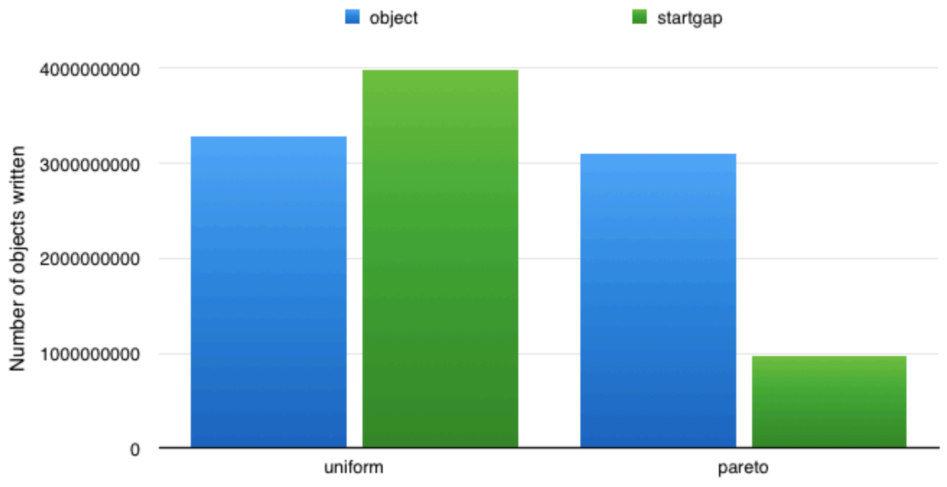


Figure 7.6: Number of objects processed

axis represents the bins. As expected, a start-gap device is worn out when one of its lines reaches the endurance limit. Our object device, on the other hand, has two groups of bars near 480 K and the endurance limit. This is because it performs wear-leveling on data nodes and metadata nodes separately. The first group of bars near 480 K belong to metadata nodes and the other group contains the lines belong to objects. Since we use fine-grained wear-leveling for metadata nodes, metadata lines are expected to be younger, thus more reliable than data lines. Due to this separation and the requirement for some free space in probabilistic wear-leveling, startgap is performing 18 % better in terms of the number of processed objects under a uniform workload as shown in Figure 7.6. However, under a pareto workload, as we can see in Figure 7.5(b), startgap failed to redistribute the updates to a hot region, and thus making the entire region unusable early. Our object devices can handle around the same number of objects in both pareto and uniform workloads.

When we set the endurance variance to 10 % of the endurance limit as shown in Figure 7.8, the highest bar in a start-gap policy moved to the middle because when a device fails when its weakest-line is worn-out while other lines are still young. However, since worn-out objects can be identified using a random counter, lines belong to the object are dropped and replaced with other free lines in probabilistic wear-leveling. As a result of fine-grained wear-leveling, probabilistic wear-leveling processed 65 M more 4 K write requests than start-gap.

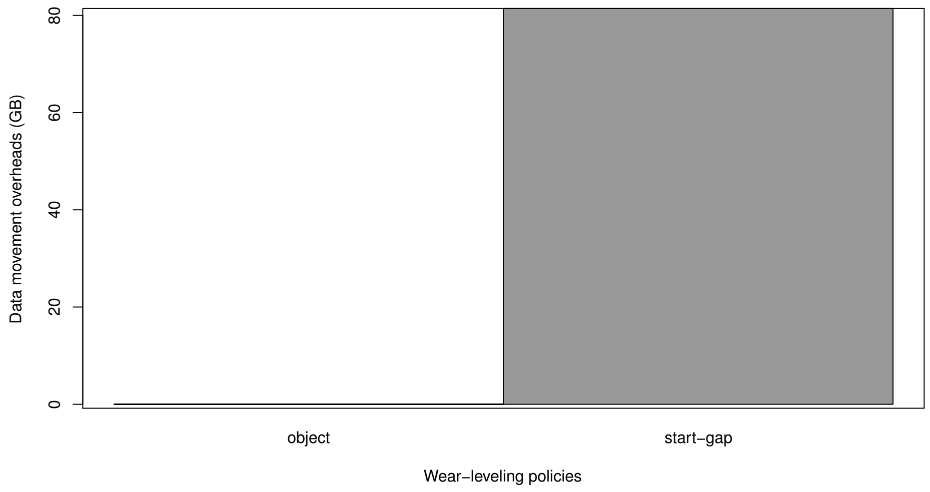


Figure 7.7: Number of bytes written for wear-leveling

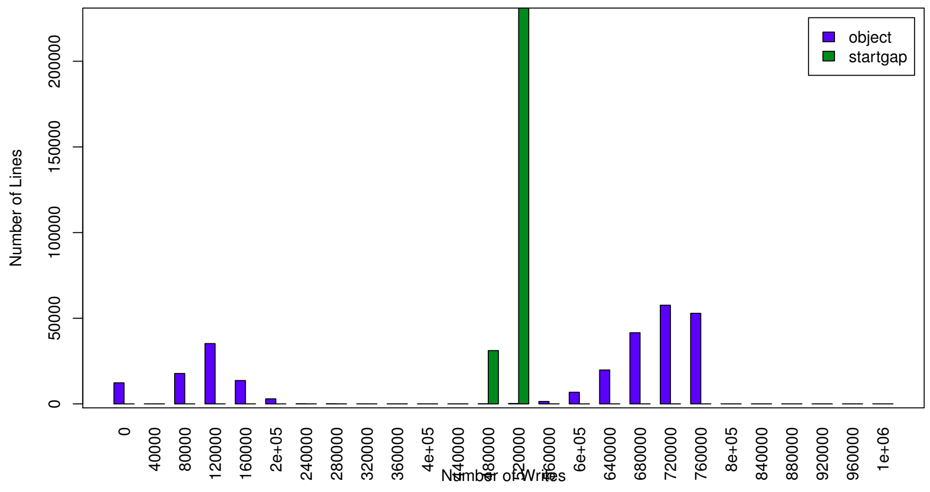


Figure 7.8: Write distribution under a uniform workload with endurance variance of 10 %

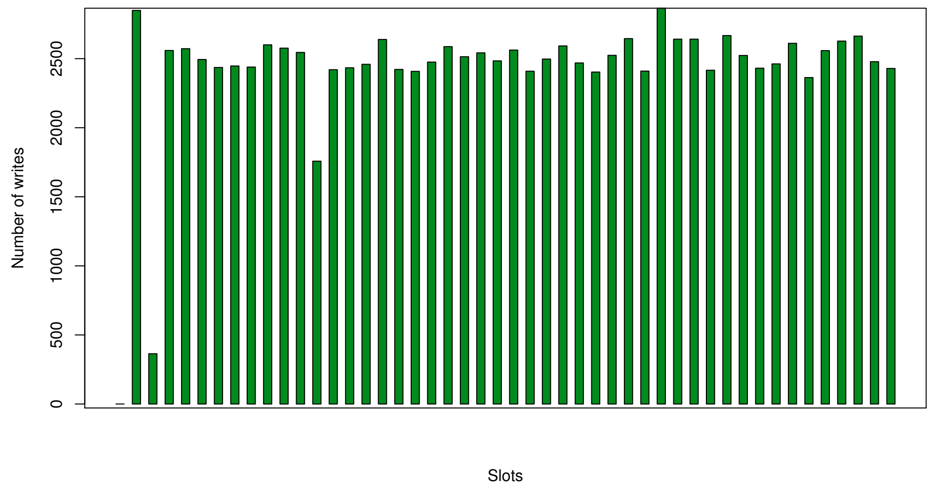


Figure 7.9: Effects of line-swapping

7.4.2 Wear-leveling overhead

In start-gap wear-leveling, data is relocated based on a global counter instead of fine-grained counters. This helps reduce the space required for wear-leveling, but it requires data to be moved even when a device is young. Figure 7.7 shows how small random counters can prevent this data copy, moving data only when it is needed. Given 14 TB of workload, start-gap moved 151 GB of data for wear-leveling while our policy moves 0.26 GB of data.

7.4.3 Internal wear-leveling

The slots for key and values in B+ tree nodes are internally wear-leveled based on a per-node wear-level counter for object swapping. Thus, the goal of internal wear-leveling is to keep the wear-level difference between slots low. To show the gap between the actual write counts and wear-levels, we picked one of the tree nodes and examined the write counts per slot after running the previous experiments. Figure 7.9 shows the actual number of writes per slot in one B+ tree node. Other tree nodes show the similar pattern. a variance of write counts between lines was around 0.403145, and standard deviation was about 0.146479. As the internal wear-leveling threshold is set to 4 in this experiments, the difference between the lowest and highest number of writes, are expected to be less than 20 K, and we can see that all lines are inside this boundary.

7.5 Summary

The advent of a fast, byte-addressable NVRAM with limited endurance has challenged the design of storage system, requiring a low latency, high throughput, and a fine-grained wear-leveling. Several solutions have been introduced to handle wear-leveling issues, such as start-gap and random page swapping, but they have been either in-accurate or its lifetime is bounded by the weakest endurance cell.

We alleviate this problem by adopting an object-based storage model and integrating probabilistic wear-leveling tracking with internal data structures. Our wear-level tracking method based on a uniformity of a random function reduces the update frequency and space overhead while providing comparable wear-leveling performance compared to existing algorithms. Through multi-grained wear-leveling, our scheme works more efficiently under high endurance variance or hot-cold workloads. Because it easily detect and reallocate worn-out objects, instead of invalidating the whole region. We also introduce space-efficient free space manager based on a buddy-algorithm, that aware of wear-levels, and a wear-aware B+ tree to manage byte-addressable NVRAM. Our results show that probabilistic wear-leveling provides comparable lifetime while incurring less overhead, and can improve a device lifetime by a factor of three, compared to start-gap when an endurance variance exists.

Chapter 8

Future work and Conclusion

With the advent of fast, non-volatile memory devices, the OS storage stacks designed for hard drives have been challenged; interrupt-based I/O mechanisms and concurrent I/O throttling are not contributing towards better performance, and a block interface is limited in delivering necessary information from/to NVRAM storage devices. Today, most NVRAM devices expose a block interface for compatibility with legacy storage devices. However, host applications must maintain their own heuristics in order to recover information lost during transmission and optimize workloads for the underlying storage devices. This, unfortunately, increases complexity and reduces efficiency and extensibility. The NVM-e interface is designed to provide a memory-like interface to NVRAM devices based on a PCI-e interface, but its standard is focused on efficient data transmission, not on a right boundary between NVRAM storage devices and storage systems, based on system-wide efficiency and portability.

This thesis proposed the use of the object-based storage model as a way of addressing the shortfalls of the current NVRAM architecture. By offloading the NVRAM management layer from file systems to devices with an object interface, we show that an on-device NVRAM management layer can be designed as efficient as that of NVRAM-aware file systems while improving scalability and portability. The features provided by devices can be easily exposed to host systems with minimal changes. To explore its design flexibilities, we first examined the object-based data placement policies for flash memory and studied their effects on cleaning and write performance. In Chapter 5, throughout the design of Muninn, we show that adding device features to existing file systems can be done with small host system changes. We also discussed the effects on hash-based key-value placement and a Bloom filter-based search used to manage data in this object store. Then, we extended the commodity SSD architecture to support object-interface and in-storage processing to show that a flexible object interface can offload not only the data management layer, but also I/O tasks from applications, improving both performance up to 2 times and energy consumption by 50 %. Lastly, we discussed our probabilistic wear-leveling mechanism for byte-addressable memory that exploits the uniformity of a random function to reduce the update frequency and size of random counters. The object-based storage model allows NVRAM devices to integrate its data management layer with probabilistic wear-leveling.

As a part of this research, I have looked at the use of object-based NVRAM devices as main-memory devices. While the references to main memory are still key-value type requests like NVRAM storage devices, there are several areas to be considered in the future design of NVRAM operating systems and devices; file systems, memory subsystem, and process subsystem may be merged and simplified, requiring additional operations like *make_persistent*. The OS memory subsystem can offload page tables to devices and become a part of the on-device mapping structure. Since NVRAMs are slower than DRAMs, the on-device NVRAM management layer may need to be designed for low-latency reads and high-bandwidth writes. Additional features such as atomic writes and compression can be added as well. We have discussed wear-leveling policy and data management policy for byte-addressable NVRAM in Chapter 7, but it would be useful to examine more swapping policies and data structures for internal data placement and cache policies.

In the design of Smart SSDs, support for scripting languages in writing tasklets would be helpful in providing security mechanisms, such as sandboxing and authentication, to prevent malicious tasklets from destroying data or even devices. Running MapReduce on multiple Smart SSDs would need a job distribution strategy that considers the capability and current load of both host systems and Smart SSDs. An obvious piece of future work is to implement a MapReduce framework that can manage multiple concurrent device jobs.

Over the years of our research, we have seen the rapid development of NVRAM technologies and their interfaces. Flash memories become denser and a variety of NVRAMs are under development, promising superior characteristics than flash memory. One can now purchase high-performance SATA SSDs for desktops and laptops. For the next step, we are switching a block interface to more memory-like interface such as PCI-e and NVM-e for better performance and extensibility, making it easier to implement object-based storage devices. Because every component specific to NVRAM is in object-based devices, we can perform co-optimization between NVRAM management and hardware layers, while making the file systems be independent from the underlying media, and thus allowing drop-in replacement for new types of devices. As a processing power of NVRAM devices increase, we expect more and more NVRAM devices are trying to provide auxiliary features such as in-storage processing and versioning. We believe the object-based storage model can be a good access model for these future NVRAM devices.

Bibliography

[1] Challenges in phase-change memory. http://en.wikipedia.org/wiki/Phase_change_ram#Challenges.

[2] Ferroelectric ram. http://en.wikipedia.org/wiki/Ferroelectric_RAM.

[3] Worldcup98 internet traffic archive. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[4] A. Bialecki and M. Cafarella and D. Cutting and O. Malley. Hadoop: A framework for running applications on large clusters built of commodity hardware. http://lucene.apache.org/hadoop/, 2005.

[5] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms, and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1998.

[6] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2:361–372, August 2009.

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

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

[9] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, March 2007.

[10] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Pro-*

ceedings of the 7th USENIX conference on Networked systems design and implementation, NSDI'10, pages 29–29. USENIX Association, 2010.

[11] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, MT, October 2009.

[12] Darrell C. Anderson, Jeff S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[13] Raja Appuswamy, David C. Van Moolenbroek, and Andrew S. Tanenbaum. Flexible, modular file volume virtualization in loris. In *Mass Storage Systems and Technologies (MSST)*, may 2011.

[14] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Vijayan Prabhakaran. Removing the costs of indirection in flash-based SSDs with nameless writes. In *Proceedings of the 2nd Workshop on Hot Topics in Storage and File Systems (HotStorage '10)*, June 2010.

[15] Anirudh Badam and David W. Nellans. Enabling application directed storage devices. In *3rd Annual Non-Volatile Memories Workshop*, 2012.

[16] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 223–238, February 2008.

[17] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad 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.

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

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

[20] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST 2012)*, April 2012.

[21] Hans Boeve, Cristophe Bruynseraede, Jo Das, Kristof Dessein, Gustaaf Borghs, Jo De Boeck, Ricardo C. Sousa, Luís V. Melo, and Paulo P. Freitas. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics*, 35(5):2820–2825, September 1999.

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

[23] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395. IEEE Computer Society, 2010.

[24] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S.A. Wolf, A. W. Ghosh, J.W. Lu, S. J. Poon, M. Stan, W.H. Butler, S. Gupta, C. K A Mewes, T. Mewes, and P.B. Visscher. Advances and future prospects of spin-transfer torque random access memory. *Magnetics, IEEE Transactions on*, 46(6):1873–1878, 2010.

[25] Hyun Jin Choi, Seung-Ho Lim, , and Kyu Ho Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Trans on Storage*, 4(4), January 2009.

[26] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[27] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick 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, October 2009.

[28] Ramtron International Corporation. `http://www.ramtron.com`.

[29] Curtis E. Stevens. ATA command pass-through. ftp://ftp.t10.org/t10/document.04/04-262r1.pdf, 2005.

[30] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, 3:1414–1425, Sept 2010.

[31] Biplob Debnath, Sudipta Sengupta, and Jin 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.

[32] In Hwan Doh, Jongmoo Choi, Donghee Lee, and Sam 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.

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

[34] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The Dynamic Bloom Filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, January 2010.

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

[36] Amber Huffman. Nvm express, revision 1.0c. www.nvmexpress.org, Feb. 2012.

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

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

[39] Samsung Electronics Inc. Samsung solid state drive basics. http://www.samsung.com/global/business/semiconductor/product/flash-ssd/overview, 2010.

[40] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage*, 6(3), September 2010.

[41] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6(1):1–25, 2010.

[42] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-Tree : An ordered index structure for nand flash memory. In *7th ACM & IEEE Conference on Embedded Software (EMSOFT '07)*, pages 144–153, 2007.

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

[44] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *Mass Storage Systems and Technologies (MSST)*, may 2013.

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

[46] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3), Aug. 2004.

[47] Hyojun Kim and Seongjun Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[48] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[49] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi, and Kyoung Il 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.

[50] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Int'l Workshop on Accelerating Data Management Systems (ADMS) jointly held with Int'l Conference on Very Large Data Bases (VLDB)*, Sep. 2011.

[51] M.H. Kryder and Chang Soo Kim. After hard drives – what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406 –3413, Oct 2009.

[52] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. on Embedded Computing Systems*, 6(3), 2007.

[53] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23nd ACM Symposium on Operating Systems Principles (SOSP '11)*, October 2011.

[54] Seung-Ho Lim and Kyu-Ho Park. An efficient nand flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906 – 912, july 2006.

[55] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the sixth international conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 212–223. VLDB Endowment, 1980.

[56] Guanlin Lu, Young Jin Nam, and David H.C. 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.

[57] G. N.N. Martin. Spiral Storage: Incrementally Augmentable Hash Addressed Storage. Technical report, University of Warwick, Coventry, UK, UK, 1979.

[58] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2009.

[59] R. Mueller and J. Teubner. Fpga: What's in it for a database? In *SIGMOD*, 2009.

[60] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. In *PVLDB*, 2009.

[61] 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.

[62] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of SC '04*, November 2004.

[63] Nantero, Inc. Nano-ram. http://www.nantero.com/.

[64] Netezza Corporation. The Netezza Data Appliance Architecture: a Platform for High Performance Data Warehousing and Analysitics, 2010.

[65] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[66] Facebook Engineering Notes. Mcdipper: A key-value cache for flash storage. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920`.

[67] E. A. Ozkarahan, S. A Schuster, and K. C. Smith. Rap - associative processor for database management. In *AFIPS*, 1975.

[68] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[69] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proc. of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, 2006.

[70] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. Pffs: A scalable flash memory file system for the hybrid architecture of phase-change ram and nand flash. In

Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08, pages 1498–1503, New York, NY, USA, 2008. ACM.

[71] Juan Piernas, Toni Cortes, and José M. García. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95, New York, NY, 2002.

[72] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM.

[73] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 USENIX Annual Technical Conference*, June 2009.

[74] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6), 2001.

[75] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Int'l Conference on Very Large Data Bases (VLDB)*, Aug. 1998.

[76] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[77] Michel Scholl. New file organization based on dynamic hashing. *ACM Transactions on Database Systems*, 6(1):194–211, March 1981.

[78] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 249–264, 1995.

[79] Frank Shu and Nathan Obr. Data set management commands proposal for ATA8-ACS2. http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-

Data_Set_Management
_Proposal_for_ATAACS2.doc.

[80] Swaminathan Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[81] Stanley. Y. W. Su and G. Jack Lipovski. Cassm: A cellular system for very large data bases. In *International Conference on Very Large Data Bases*, pages 456–472, Sep. 1975.

[82] 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. In *8th ACM & IEEE Conference on Embedded Software (EMSOFT '08)*, pages 51–58, 2008.

[83] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[84] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300, College Park, MD, April 2004. IEEE.

[85] Ralph O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, August 2002.

[86] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[87] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[88] Ronald Weiss. A technical overview of the oracle exadata database machine and exadata storage server. http://www.oracle.com/us/products/database/exadata-technical-whitepaper-134575.pdf, March 2012.

[89] D. Woodhouse. The journalling flash file system. *Ottawa Linux Symposium*, 2001.

[90] David Woodhouse. The journalling flash file system. In *Ottawa Linux Symposium*, July 2001.

[91] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[92] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Usenix Conference on File and Storage Technologies (FAST 2012)*, February 2012.

[93] Wei-Pang Yang and M. W. Du. A Dynamic Perfect Hash Function Defined by an Extended Hash Indicator Table. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 245–254, 1984.

[94] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi 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.

[95] Zhihui Zhang and Kanad Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of EuroSys 2007*, March 2007.