# Block Management in Solid-State Devices

Abhishek Rajimwale[+], Vijayan Prabhakaran[⋆], John D. Davis[⋆]
[+]*University of Wisconsin, Madison*
[⋆]*Microsoft Research, Silicon Valley*

## Abstract

*Solid-state devices (SSDs) have the potential to replace traditional hard disk drives (HDDs) as the de facto storage medium. Unfortunately, there are several decades of spinning-media assumptions embedded in the software stack as an "unwritten contract" [20]. In this paper, we revisit these system-level assumptions in light of SSDs and find that several of them are invalidated by SSDs, breaking the unwritten contract and resulting in poor performance and lifetime. The underlying cause is the incorrect division of labor between file systems and storage. Block management must be removed from the file system and delegated to the SSD to prevent further accumulation of storage-specific assumptions. We find that object-based storage is an appropriate way to achieve this.*

## 1 Introduction

Storage systems export a simple abstraction of a linear block-level interface that has worked well for cases ranging from a single disk to the aggregation of disks such as RAID arrays and logical volumes. In fact, the simplicity of the interface has helped to hide the complexity of the underlying device from higher-level systems.

Unfortunately, the interface has also hidden device-specific details, so that the file system is forced to make assumptions about the underlying storage, referred to by Schlosser and Ganger as an "unwritten contract" [20]. Not surprisingly, most of these assumptions are regarding block management such as allocation, layout, scheduling, and cleaning, all of which could benefit from device-specific knowledge. For example, file systems assume that random accesses are much slower than sequential ones, and hence the block management layer is optimized for this. While this assumption is valid for disks, when the properties of the underlying device change, such assumptions may either hold or fail. For example, for MEMS-based storage, Schlosser *et al.* find that the existing abstractions are mostly valid [20].

We reexamine the existing storage abstraction and the resulting assumptions in light of solid-state devices (SSDs). SSDs are different from disk drives in many aspects such as unique semiconductor properties, internal architecture, and controller firmware, which affect the performance, reliability, lifetime, power, and security properties of the SSDs. Overall, SSDs are substantially complex and self-managing and require more information than is provided by the standard storage interface.

To find out if the disk-specific assumptions hold for SSDs, we list six system-level assumptions (three of which are from the unwritten contract as originally stated) and explain how each of them fail for SSDs, resulting in poor performance and lifetime. The underlying problem is the incorrect division of labor: file systems perform block management, which for a device such as an SSD is best done internally because of its knowledge of intricate device-specific properties, policies, and algorithms.

A more expressive interface such as object-based storage (OSD) [10, 11, 22] can improve the current state of the art. First, OSD delegates finer details of block management to the SSD, thereby preventing any new storage-specific assumptions. Second, OSD expresses the intentions of the higher layers clearly, thereby improving the internal SSD operations.

## 2 Background

**Storage Interface.** Storage access protocols such as SCSI and ATA/IDE export a narrow, block-based interface with simple `read` and `write` APIs to access the logical block number (LBN) (a 512-byte sector). A storage controller internally maps the LBN to a physical sector, which is hidden and fixed for most cases.

The main disadvantage of the block-based interface is

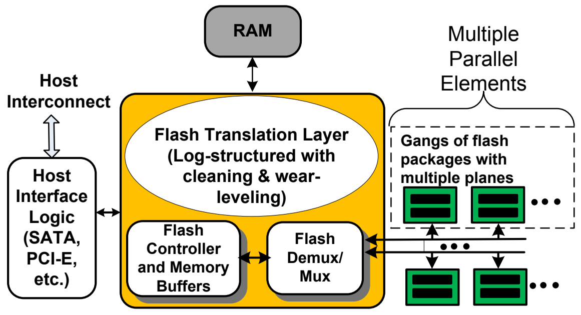
RAM
Host Interconnect
Multiple Parallel Elements
Flash Translation Layer (Log-structured with cleaning & wear-leveling)
Gangs of flash packages with multiple planes
Host Interface Logic (SATA, PCI-E, etc.)
Flash Controller and Memory Buffers
Flash Demux/ Mux

Figure 1: **SSD Architecture.**

the absence of higher-level semantics. Several previous works note this deficiency and propose more expressive interfaces [5–7, 9]; some even allow programming the storage controller [1, 16, 21]. One approach is to use an object-based interface [10, 11, 22], which exports the abstraction of an object as a collection of bytes. Structures such as trees, tables, files, and directories can be represented as objects, reflecting the higher-level semantics better than a block-based interface; the device controller performs block allocation and layout for the objects.

**Solid-State Devices.** An SSD consists of a set of flash memory packages that are connected to a controller (Figure 1). Each package has one or more dies; each die has multiple planes, which in turn have many blocks; each block consists of many 4 KB pages [18].

SSDs differ from HDDs on 3 main properties. The first obvious difference is the absence of mechanical moving parts. Second, flash pages are non-overwrite in nature and must be erased before being overwritten. To hide the high erase overhead and create the abstraction of an in-place write, modern SSDs implement a log-structured design [17] in the flash translation layer (FTL) [15]. To uniformly spread the block usage, wear-leveling is also implemented. Finally, SSDs have several layers of parallelism that is dictated by the flash packages and the way they are connected to the controller.

There are two types of NAND flash memory: single-level cell (SLC) and multi-level cell (MLC). SLC flash stores a single bit of data per cell, while MLC flash stores multiple bits per cell. MLC flash has some drawbacks such as shorter lifetime (10K erase cycles vs 100K erase cycles of SLC), slower write, and erase operations.

## 3 Failed Assumptions

In this section, we discuss the system-level assumptions that fail when applied to SSDs, and the reasons behind these failures. In Table 1, we list the original (1-3) and extended terms of the unwritten contract and state whether they are satisfied or violated by different devices. This list is by no means complete because we focus only on block-management issues; more assumptions may be added as our experience with SSDs grows. For comparison, we list RAID arrays and MEMS-based storage, but for the rest of the paper we focus only on how the assumptions fail on SSDs.

### 3.1 Sequential vs. Random

In a disk, the latency and bandwidth of sequential access are several tens of times better than random access. However, on SSDs that use a log-structured FTL [15], both sequential and random writes are likely to take similar time. Table 2 lists the ratio of sequential-to-random bandwidth for an HDD (a Seagate Barracuda 7200.11 drive) and several SSDs. One of the SSDs is simulated ($S4_{slc\_sim}$) using the simulator from our previous work [2], while others are real ($S1_{slc}$, $S2_{slc}$, $S3_{slc}$, $S5_{mlc}$). We anonymize the real SSDs because they are engineering samples and pre-production models. To help the reader understand the results better, we specify whether the devices use SLC or MLC flash memory.

From the table, we can observe that SSDs (using SLC or MLC memories) have random-read performance that is only a few times smaller than their sequential-read performance. This is even true for writes on certain SSDs ($S1_{slc}$, $S4_{slc\_sim}$, $S5_{mlc}$), but not on all of them; in fact, some of the SSDs ($S2_{slc}$, $S3_{slc}$) have random-write performance that is worse than HDDs. One of the reasons for this poor performance is write amplification, which we will discuss later (§3.4).

From the above results, we can see that the gap between sequential and random accesses is narrowing on SSDs. File systems that run primarily on SSDs must reconsider the need for complex policies to achieve block-level sequentiality. Instead, a file system must focus on higher-level operations such as object management, consistency, and recovery, and move the low-level block management to the SSD, using say, the OSD interface.

### 3.2 Logical-to-Physical Mapping

The second term of the unwritten contract considers the relation between logical and physical sectors, and understanding it is important for I/O scheduling. On an HDD, nearby LBNs translate well to physical proximity. However, this contract fails on an SSD because of the log-structured design, cleaning, and wear-leveling, all of which make it harder to estimate the location of a logical sector. In fact, the physical location is irrelevant if the ratio of sequential to random accesses approaches 1. This further motivates the conclusion that the file system accesses must be in terms of objects (or parts of objects) and the SSD must handle the low-level sector-specific scheduling.

| Contract | Disk | RAID | MEMS | SSD |
|---|---|---|---|---|
| 1. Sequential accesses are much better than random accesses | $T$ | $T$ | $T$ | $F$ (flash memory, no mechanical parts) |
| 2. Distant LBNs lead to longer seek times | $T$ † | $F$ | $T$ | $F$ (log-structured writes) |
| 3. LBN spaces can be interchanged | $F$ | $F$ | $T$ | $F$ (integration of SLC and MLC memory) |
| 4. Data written is equal to data issued (no write amplification) | $T$ | $F$ | $T$ | $F$ (ganging, striping, larger logical pages) |
| 5. Media does not wear down | $T$ | $T$ | $T$ | $F$ (semiconductor properties) |
| 6. Storage devices are passive with little background activity | $T$ † | $F$ | $T$ | $F$ (cleaning and wear-leveling) |

Table 1: **Unwritten Contract.** *Terms of the unwritten contract and whether they are satisfied ($T$) or not ($F$) by various devices; a † means that the contract is only approximately satisfied because the device has grown more complex. For SSDs, a brief reason is also given.*

| | Read | | | Write | | |
|---|---|---|---|---|---|---|
| Device | Seq | Rand | Ratio | Seq | Rand | Ratio |
| HDD | 86.2 | 0.6 | 143.7 | 86.8 | 1.3 | 66.8 |
| $S1_{slc}$ | 205.6 | 18.7 | 11.0 | 169.4 | 53.8 | 3.1 |
| $S2_{slc}$ | 40.3 | 4.4 | 9.2 | 32.8 | 0.1 | 328.0 |
| $S3_{slc}$ | 72.5 | 29.9 | 2.4 | 75.8 | 0.5 | 151.6 |
| $S4_{slc\_sim}$ | 30.5 | 29.1 | 1.1 | 24.4 | 18.4 | 1.3 |
| $S5_{mlc}$ | 68.3 | 21.3 | 3.2 | 22.5 | 15.3 | 1.5 |

Table 2: **Ratio of Sequential to Random Bandwidth.**

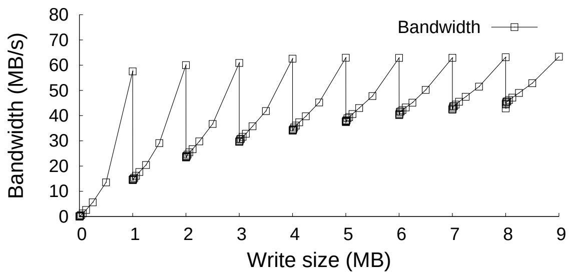

Figure 2: **Write Amplification.** *In $S2_{slc}$, maximum bandwidth is achieved when the write size aligns with the stripe size (1 MB).*

We performed a preliminary analysis with a new algorithm for SSD, called shortest wait time first (SWTF), which uses the queue wait times of all the parallel elements in an SSD and schedules an I/O that has the shortest wait time. On a synthetic workload that issues random I/Os (with 2/3 reads and 1/3 writes), we found that SWTF improves the response time by about 8% when compared to FCFS. More thorough analysis is required to find the effectiveness of such SSD-specific algorithms.

## 3.3 Interchangeable Address Space

The third term of the contract assumes that the logical address space is uniformly spread over the device. This is invalidated by disks because of zoned recording, where the outermost tracks accommodate more logical pages than innermost ones; that is, outer-track bandwidth is greater than the inner-track bandwidth. Today, SSDs are homogeneous, using only a single type of memory (either SLC or MLC), keeping the contract valid. However, we believe that in the future, SSDs might be constructed with multiple types of memories (SLC/MLC). In such systems, this contract will be violated because MLC-type memories can hold more data and have different timing characteristics than SLC. Such heterogeneity in the address space can be better utilized if the device performs block allocation for higher-level objects. For example, an SSD can choose to co-locate all the data belonging to a root object in SLC memory for faster access.

## 3.4 Write Amplification

Operating systems typically assume that the time taken to complete an I/O is proportional to the I/O size. However, in an SSD, writes may be amplified into a larger I/O due to several reasons: first, when the logical page is larger than the physical page size; second, when a write is issued in-place using a read-modify-erase-write cycle. Write amplification is not a new phenomenon; it happens on RAID arrays that need to update parity blocks.

We measured the effect of write amplification on one of the engineering samples (a low-end SSD, $S2_{slc}$); Figure 2 shows the results. We plot the bandwidth against the write size. One can observe that the bandwidth is poor on small write sizes (*e.g.*, 512 bytes). As we increased the write size, the bandwidth improved and reached its maximum at 1 MB (the SSD's stripe size). As we increased the write size further (*e.g.*, 1 MB + 512 bytes), the bandwidth again dropped, and this behavior repeated to give a saw-tooth pattern. We believe that this behavior is due to striping the logical page across a gang of flash packages that share the buses [2]. A similar saw-tooth pattern was noticed by Schindler *et al.* for disk drives on track-aligned accesses [19]. However, in their case it was due to the effect of track switches and rotational latencies. It is important to note that write caches might not always mask the write amplification; for example, $S3_{slc}$has a write buffer cache of 16 MB, but it is ineffective in masking the write amplifications, as can be seen from the random-write performance in Table 2.

Write amplification can be reduced by merging writes and aligning them to stripe sizes. Since it is harder to estimate the stripe size and alignment boundaries from a file system (especially in the presence of a write cache

| Probability of sequential access | 0 | 0.2 | 0.4 | 0.6 | 0.8 |
|---|---|---|---|---|---|
| Unaligned | 10.6 | 10.6 | 10.5 | 10.2 | 10.5 |
| Aligned | 10.6 | 10.4 | 8.9 | 7.6 | 5.6 |

Table 3: **Improved Response Time with Write Alignment.** *Average I/O response time (in ms) for unaligned and stripe-aligned 4 KB writes with varying degrees of sequentiality.*

| | Postmark | TPCC | Exchange | IOzone |
|---|---|---|---|---|
| Improvement (%) | 1.15 | 3.08 | 4.89 | 36.54 |

Table 4: **Macro Benchmarks with Stripe-aligned Writes.**

and background activity), an SSD must be responsible for sector allocation and layout according to the stripe sizes. Table 3 shows results from stripe-aligned and unaligned writes. We simulated a 32 GB SSD with one gang of eight 4 GB flash packages. A single 32 KB logical page spanned over all the packages. We ran a synthetic workload that issued a stream of writes with varying degrees of sequentiality. We compared two schemes: one, issuing the writes as they arrive; two, merging and aligning writes on logical page boundaries. On a completely random workload, both schemes worked similarly because of the small chance to merge the writes into stripe sizes. As the sequentiality increased, aligning writes paid off well, resulting in an improvement of over 50%. Table 4 presents the improvement in response time for various workload traces. Of all the workloads, IOzone benefits the most (over 36% improvement) due to its large write sizes.

## 3.5 Block Wear

File systems assume that the media wear-down does not depend on the number of writes to particular sectors. However, flash memory blocks have limited erase cycles before wearing out. Therefore, mid-range and high-end SSDs implement cleaning and wear-leveling to uniformly spread the wear-down of blocks.

SSDs clean by retaining the most recent version of *all* the logical pages, including those that have been released by the file system, leading to a lot of useless activity. The effectiveness of cleaning and wear-leveling can be improved by using file-system-level semantic knowledge, specifically the block allocation status, which is not available to an SSD.

An SSD can use the block allocation status to implement *informed cleaning and wear leveling* that avoids retaining the free pages. We used our SSD simulator to analyze the benefits of informed cleaning by running block-level traces that contain read, write, and block-free operations. The traces were collected by running the

| Transactions | 5000 | 6000 | 7000 | 8000 |
|---|---|---|---|---|
| Relative pages moved | 0.31 | 0.25 | 0.35 | 0.50 |
| Relative cleaning time | 0.69 | 0.60 | 0.63 | 0.69 |

Table 5: **Improved Cleaning with Free-Page Information.** *The table shows decrease in pages moved and cleaning time with free-page information relative to the default SSD (i.e., without free-page information). The actual numbers of pages moved (in 1000s) for the default SSD are, 88159, 155465, 217130, and 284409, for 5K to 8K transactions. The actual cleaning times (in seconds) for the default SSD are, 49147, 71975, 93569, and 116185 for 5K to 8K transactions.*

Postmark benchmark [14] on a pseudo-device driver that uses Linux Ext3 knowledge to identify the free sectors. The SSD simulator was modified such that the cleaning and wear-leveling logic disregard the flash pages corresponding to the free logical pages. To the best of our knowledge, this is the first study to measure the effect of free-page knowledge in SSD cleaning.

Table 5 shows the improvements in cleaning in terms of the number of pages that need to be reclaimed and the cleaning time for an 8 GB SSD; both measures are shown relative to the default SSD that does not use the free-page information. We observe that informed cleaning reduces the number of reclaimable pages by at most about one-half. Informed cleaning reduces the cleaning time by 30-40%, which can improve the overall running time by about 3-4%. However, the improvements are workload-dependent.

## 3.6 Background Activity

Storage systems are assumed to be passive and to act only when a request is issued from a higher-level software layer. While this is true on a single HDD, SSDs perform a considerable amount of background activity due to cleaning and wear-leveling. Therefore, it becomes hard to predict the I/O latency; for example, it is hard to guarantee QoS on a system with SSDs because the host has no control over when the SSD engages in background activities. This is especially true if the SSD is full and the degree of internal fragmentation is high [4]. The background activity can be controlled by informing the SSD about I/O priorities or by marking certain objects as high-priority. For example, an SSD can provide preferential treatment to high-priority objects or I/Os by delaying its background activity.

We modified the cleaning logic of our SSD simulator to be aware of request priorities. If there are no outstanding priority requests, cleaning starts when the number of free pages falls below a *low* threshold. However, if there are priority requests, cleaning is postponed until the number of free pages falls below a *critical* threshold. We call this *priority-aware* and compare it with a *priority-*

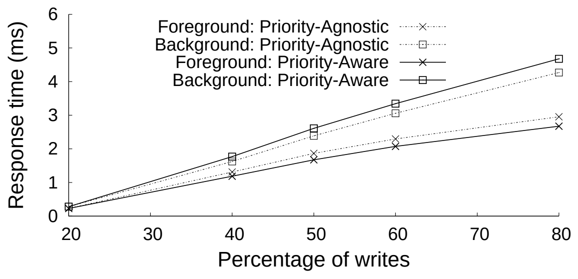

Figure 3: **Priority-Aware Cleaning.** *Priority-aware cleaning improves the foreground I/O response time by postponing cleaning; in contrast, priority-agnostic cleaning deteriorates the foreground I/Os.*

| Writes (%) | 20 | 40 | 50 | 60 | 80 |
|---|---|---|---|---|---|
| Improvement (%) | 0 | 9.56 | 10.27 | 9.61 | 9.47 |

Table 6: **Response Time Improvement From Priority-Aware Cleaning.** *When the percentage of writes is small (less than 40%), cleaning does not happen frequently and hence foreground requests are not affected (and therefore no improvement).*

*agnostic* scheme, which starts cleaning at the low threshold irrespective of the outstanding requests. Note that when there is no priority information available, priority-agnostic is the default technique to use.

We evaluated a 32 GB SSD using synthetic benchmarks with request inter-arrival times uniformly distributed between 0 and 0.1 ms. The fraction of priority requests was set to 10%; critical and low thresholds were fixed at 2% and 5% of free pages. In Figure 3, we plot the response time of priority requests (marked as foreground) and non-priority requests (marked as background). Table 6 shows the corresponding improvement in response time for priority requests. We observe that under the priority-aware scheme, the response time of foreground requests improve by about 10%. However, the cost of this improvement is reflected on the background I/Os, whose response time increased as well.

## 3.7 Object-based Storage and SSD

As storage devices grow more complex, assumptions made by higher layers fail. We believe that certain functionalities, specifically those related to block management, are more appropriately handled by the device controller with its intricate knowledge of the inner workings of the device. However, to perform the block management correctly and efficiently, devices must also understand the high-level intentions (semantics) behind the simple reads and writes.

Broadly, there are two ways by which the device can obtain more information: explicitly, through new or modified interfaces; or implicitly, by using reverse engineering techniques. Since reverse engineering techniques can add more complexity to the device firmware and do not minimize the functionalities at higher layers, we focus only on explicit approaches. Existing interfaces can be patched with additional commands to convey the operation semantics. For example, the TRIM command has been proposed to add file delete notifications to the ATA interface [8]. While this approach offers the least resistance in the device-interface evolution, it still operates on the block level, thereby letting the file system perform the block management. Moreover, existing interfaces may not provide sufficient extensions for new commands. In such cases, new interfaces such as NVMHCI have been proposed [13]. However, while NVMHCI conveys more information than traditional SCSI/ATA, it still lets the higher layers manage and operate at the block level. We believe that OSD interface provides a nice alternative by conveying more information and letting the device handle low-level operations.

For several of the aforementioned contract violations, an OSD provides a better alternative. For example, a file system should operate on objects and let the device handle the logical to physical mapping, sequential-random accesses to (parts of) objects, and stripe-aligned accesses. Additionally, an SSD can use the OSD interface and manage the space for objects (including the allocation and release of pages to objects) in order to implement informed cleaning. An additional benefit of using an OSD is that object attributes can be set to convey read-only data, which could be used for cold data placement during wear-leveling. Finally, I/Os to objects can be marked with a priority to schedule them appropriately with background activities.

## 4 Related Work

Several previous researchers have noted the need for more expressive storage interfaces for disks [1, 5, 6, 9, 16, 21], RAID arrays [7], and MEMS-based devices [12, 20]. Among these, the most closely related work is by Schlosser and Ganger, which examines the OS assumptions in the context of MEMS-based devices [20]. They list the first three terms of the unwritten contract and show how MEMS-based devices obey them, obviating the need for new interfaces or algorithms. In another related paper, Ajwani *et al.* characterize a variety of SSDs and argue for new algorithms for SSDs and hybrid devices [3]. However, they still use the block-level interface. We argue for a new, richer interface.

New interface specifications are being proposed for SSDs, like NVMHCI [13] and TRIM [8], but they still let the higher layers manage the blocks, resulting in most of the problems we discussed earlier. Using the OSD interface provides a clean separation between the file system and block management operations, enabling the SSD to handle them optimally.

## 5 Conclusion

Over the past 5 decades, OS and storage systems have evolved independently across a narrow and fixed storage interface. One of the side effects of this evolution is the accumulation of device-specific assumptions in the storage stack, specifically in the block management layer. Unless the block management is removed from the file system and delegated to the storage, such assumptions are likely to carry over and grow in the next generation of storage devices as well. SSDs are evolving and have the potential to become the ubiquitous storage media. As our initial results have shown, it is time we switch from the narrow, block-based interface to a richer object-based storage to improve the performance and longevity.

## 6 Acknowledgments

We thank our shepherd, Geoff Kuenning, the anonymous reviewers, Nathan Obr from Microsoft Device and Storage Technologies, and the members of Microsoft Research Silicon Valley for their detailed feedback and excellent suggestions on our paper.

## References

[1] 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*, October 1998.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[3] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design. In *Experimental Algorithms*, pages 208–219. Springer Berlin / Heidelberg, 2008.

[4] L. Bouganim, B. Jonsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR'09)*, 2009.

[5] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[6] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 15–28, December 1993.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 177–190, June 2002.

[8] Frank Shu. Notification of Deleted Data Proposal for ATA8-ACS2. `http://t13.org/Documents/UploadedDocuments/docs2007/e07154r0-Notification_for_Deleted_Data_Proposal_for_ATA-ACS2.doc`, 2007.

[9] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 92–103, October 1998.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.

[12] J. Griffin, S. Schlosser, G. Ganger, and D. Nagle. Operating Systems Management of MEMS-based Storage Devices. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[13] Intel Corporation. Non-Volatile Memory Host Controller Interface Specification. `http://www.intel.com/standards/nvmhci/index.htm`, 2008.

[14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[15] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.

[16] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases*, August 1998.

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

[18] Samsung Corporation. K9XXG08XXM Flash Memory Specification. `http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ds_k9f8g08x0m_rev10.pdf`, 2007.

[19] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 259–274, 2002.

[20] S. W. Schlosser and G. R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, pages 87–100, April 2004.

[21] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for active storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 264–276, October 2002.

[22] R. O. Weber. SCSI Object-Based Storage Device Commands (OSD). Technical report, T10 Technical Committee, July 2004.