Building Flexible, Fault-Tolerant Flash-based Storage Systems

Kevin M. Greenan † Darrell D.E. Long † Ethan L. Miller †
Thomas J. E. Schwarz, S.J. ‡ Avani Wildani †
Univ. of California, Santa Cruz † *Santa Clara University* ‡

Abstract

Adding flash memory to the storage hierarchy has recently gained a great deal of attention in both industry and academia. Decreasing cost, low power utilization and improved performance has sparked this interest. Flash reliability is a weakness that must be overcome in order for the storage industry to fully adopt flash for persistent storage in mission-critical systems such as high-end storage controllers and low-power storage systems.

We consider the unique reliability properties of NAND flash and present a high-level architecture for a reliable NAND flash memory storage system. The architecture manages erasure-coded stripes to increase reliability and operational lifetime of a flash memory-based storage system, while providing excellent write performance. Our analysis details the tradeoffs such a system can make, enabling the construction of highly reliable flash-based storage systems.

1 Introduction

Recent advances in the performance and density of NAND flash memory devices offer the promise of great performance and low energy consumption. NAND flash is being used to replace DRAM-based disk caches [7], store metadata [5], replace disks in laptops, and even replace high-performance disk arrays [8]. Unfortunately, NAND flash memory has limitations. First, write performance depends on the underlying architecture and does not rival its excellent read performance. For good write performance, each device must have multiple banks of NAND flash to be accessed in parallel [1]. Second, the known limited endurance of NAND flash generally requires wear-leveling algorithms and page-level ECC [2, 6, 7]. Finally, the raw bit error rate (RBER) of NAND flash increases with use [4]. Page-level ECC—which is typically used to provide reliability in NAND flash systems— does not provide protection from whole-page failures, chip failures, or whole-device failures. These types of failures are typically handled using inter-device redundancy, similar to RAID. With the exception of the RAMSAN-500 [8], we are unaware of any RAID-based solutions for fault tolerance across flash devices.

The contributions of this work are as follows. We briefly cover the important reliability and performance characteristics of NAND flash memory. Next, we provide an architecture for flash-based storage systems that uses erasure codes to tolerate failures at various levels from bit-flip to device failure. The level and scope of protection may be changed on-the-fly to adjust gracefully to the degenerative effects of aging. We conduct a preliminary reliability and performance analysis of this architecture. We find that our techniques provide a higher level of protection than existing approaches while maintaining good write performance.

2 Flash Background

Figure 1 shows an example NAND flash chip with 4 banks of flash that may be accessed in parallel. Each bank contains a single page register and 64 blocks. All accesses to a bank are *serialized* at the page register: data to be read or written to/from a bank must be first loaded into the page register. A block contains a total of 64 pages and each page has a 4KB data section and a 128B spare area. The spare area is used to store checksums, ECC bits and page-specific metadata. Individual data bits are stored in cells within the data section of a flash page. There are two primary technologies for storing bits in flash: single-level cell (SLC) and multi-level cell (MLC). SLC technologies store exactly one bit per cell, while MLC technologies store two or more bits per cell.

NAND flash memory has a unique access model. An entire block must be erased (all bits changed to 1) before any data can be programmed (written) into the pages

---

[†] Supported in part by the Petascale Data Storage Institute under Dept. of Energy award FC02-06ER25768 and by the industrial sponsors of the Storage Systems Research Center.

[‡] Supported in part by a research grant from the Stiles Family Foundation.

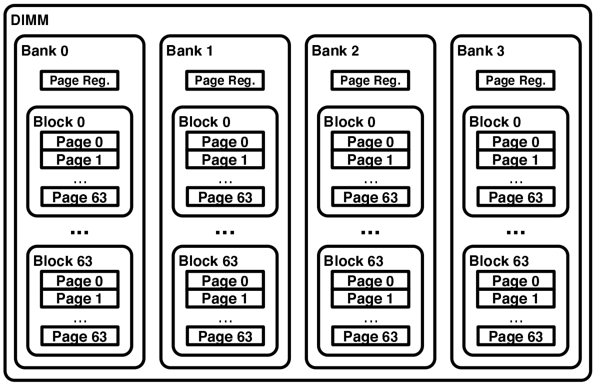

Figure 1: *A* 4 *bank NAND flash chip.*

within the block; thus, the act of writing to a block is generally called the *program-erase cycle*. Manufacturers rate the *endurance* of a block based on the underlying NAND architecture. Single-level cell (SLC) NAND flash blocks are generally rated at 10^5 program-erase cycles, while multi-level cell (MLC) blocks have a much lower endurance. Current two-level cell technologies are typically rated at 10^4 program-erase cycles. In addition to decreasing reliability, block erasures are slow; each block erase takes roughly 1.2 to 2 ms, which is an order of magnitude longer than a program operation.

Flash devices have unique reliability concerns. The most important of these concerns are *read disturb* and *program disturb* [4]. Read disturb errors occur as a side-effect of reading a page, which may induce multiple, permanent bit-flips in the corresponding block. Program disturb causes a program operation on one page to change the value of a bit on another page in the same block. These errors contribute to the raw bit-error rate (RBER), which is typically between 10^{-6} and 10^{-9}, depending on the number of bits per cell. A recent study has shown that program-erase cycles have a significant effect on RBER [4]. In some cases, the RBER increases by more than an order of magnitude as a block's program-erase count approaches its rated endurance. Given these observations, we expect *page errors* to be the primary failure pattern in flash-based systems.

We focus on three notable properties of NAND flash devices when building a flash-based storage system: *program-erase cycling*, *request serialization* at each bank and the *raw-bit error rate*.

3 System Design

Reflecting the needs of an industrial sponsor, we concentrate on a very large flash based design for storage servers where a single host or controller is connected to multiple PCI cards containing flash. We write to the flash devices in a log-structured manner, so that we do not need explicit wear leveling. Our design easily extends to a SSD (or USB) flash device as long as the host / controller has direct access to the pages on the flash device. While other reliable flash-based systems focus on bit-errors and extending flash lifetime using page-based ECCs, our design considers the proper application of erasure codes (or RAID) to a set of flash devices. In particular, we organize the pages into reliability stripes with additional parity pages, similar to what is being done in disk arrays. Because we write in a log-structured manner, the parity pages do not need to be maintained using the read-modify-write operation that is very expensive when used in flash.

There is a subtle difference between our approach and simply applying RAID algorithms to a set of flash devices. An out-of-the-box RAID approach would lead to hotspots on the flash devices and increased contention at each bank due to the traditional read-modify-write operation. We control the way data/parity is written to a set of flash devices to achieve the following goals:

(1) **Tolerate more than bit errors.** Most solutions rely on page-level ECC for reliability. In reality, the system must also tolerate burst, device, and chip-level errors. Our design achieves this goal by adding parity pages.

(2) **Uniform writes across all devices.** The reliability and performance consequences of program-erase cycles necessitates distributing the life-time number of writes uniformly. We manage writes in a log-structured manner that automatically stripes writes across all devices (or chips) and avoids read-modify-write.

(3) **Graceful degradation.** As a system ages, the probability of burst, device, and chip-level error as well as the bit error rate increases [4]. Our design has the ability to increase the fault tolerance for older systems at the cost of slightly lower capacity by changing the parameters of the page-level reliability stripes.

In achieving these goals, we essentially move the flash translation layer (FTL) off of the individual devices and onto the host. In an array of SSDs, this would be equivalent to moving the FTL into the array controller. By moving the FTL out of the individual devices, the host (or RAID controller) can optimize writes in a way that maximizes write performance, while writing uniformly across all of the devices.

3.1 System Architecture

Figure 2 illustrates the basic architecture of the flash-based storage system we are analyzing. The system is composed of PCI cards that contain flash memory chips. We assume a single host will contain multiple PCI buses, allowing the host to address flash on multiple cards. Communication between the host and the card uses a DMA channel. The host has a set of DMA rings, each of which

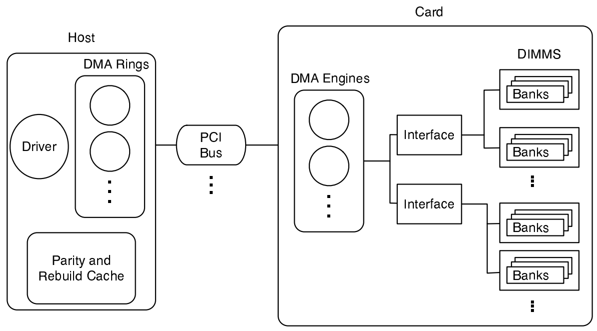

Figure 2: *NAND flash architecture.*

is associated with at most one DMA engine on a card—the pair forms a channel. All commands are issued by a driver at the host, which are queued on one of the host DMA rings. Each DMA ring can only service a single request at a time. Each DMA engine has the ability to send requests to any of the interfaces on the card. Each interface in turn is connected to one or more DIMMs. Each DIMM is connected to one or more banks of flash. The flash banks store the data and are the destination of the request stream. While requests may be parallelized at each interface, the page register requires requests to be serialized at each bank. Each flash bank contains multiple blocks, which can be accessed at page granularity.

The purpose of the driver is to allow a higher level process, such as a file system, to access data pages on the flash devices. The driver maintains the logical to physical page mappings (similar to the flash translation layer), block status and the parity relationship between pages on the devices. The driver exports two functions: read and write. The read function takes the logical page addresses as arguments and returns the requested data. The write function takes a page aligned buffer and length as arguments and returns the logical page addresses.

Each host also contains a cache that is used by the driver to stage parity updates and page rebuild requests. To prevent data loss during power or host failure, we assume the cache may contain non-volatile RAM, such as battery-backed DRAM, MRAM, PRAM or FeRAM. We currently store this cache in DRAM.

Throughout our design, we assume that the system contains C identical cards, each of which contain B banks. Each bank will contain b blocks that have 64 pages of 4K each.

3.2 Erasure Coding Across Devices

An erasure code is made up of codewords that have n symbols. Storage systems typically use systematic erasure codes, where each codeword contains k data symbols and $m = n - k$ parity (redundant) symbols. A code symbol generally refers to a sector or set of sectors on a single storage device. Storage systems often use a linear, Maximum Distance Separable (MDS) code, denoted (k,m)-MDS, which has m parity symbols, k data symbols, and can correct m erasures, the maximum possible with this many parity symbols. In this sense, an (k,m)-MDS provides optimal storage efficiency. A $(k,1)$-MDS is usually an XOR parity code, where the one parity symbol is the bitwise XOR of the contents of the data symbols. The main advantage of linear codes in disk-based systems is the ability to calculate new parity symbols when only one data symbol has changed without accessing the other data symbols. Such an update reflects the read-modify-write property of parity-based RAID systems, which is not well suited for similar flash-based systems.

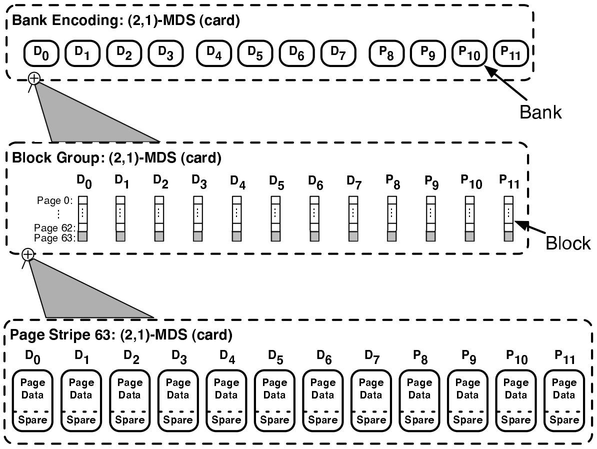
Bank Encoding: (2,1)-MDS (card)
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 P_8 P_9 P_{10} P_{11}
Bank
Block Group: (2,1)-MDS (card)
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 P_8 P_9 P_{10} P_{11}
Page 0:
Page 62:
Page 63:
Block
Page Stripe 63: (2,1)-MDS (card)
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 P_8 P_9 P_{10} P_{11}
Page Data
Spare

Figure 3: *Example block group.*

For erasure coding across flash devices, we form a *reliability stripe* out of n pages located in different chips (and *a fortiori* on different blocks). Since write requests to and from a bank are serialized, writing to all pages in a reliability stripe maximizes parallelism. The number of chips, banks per chip, and pages per chips imposes restrictions on the parameters of the erasure correcting code. In general, we want all $C \cdot B \cdot P$ pages be in exactly one reliability group, (unless we contemplate the use of "spare pages" to replace failed pages.) This placement becomes very simple when C divides $k + m$. Defining reliability stripes in our setting is a similar problem to that of laying out a declustered disk array. For example, we want to have an equal proportion of parity and user data on each chip and block in order to balance read load. However, because of the different performance characteristics of disks and flash blocks, these conditions do not need to be stringently enforced.

We show a very simple example encoding in Figure 3. In this example, we have three cards, each with four banks of flash for a total of 12 flash blocks. We use an (2,1)-MDS (similar to 3 disk RAID Level 4), which has reliability stripes consisting of blocks on three different cards. We use the first two cards to store user data and the third to store parity data. In this example, we define reliability stripes for all pages in a block simultaneously. We compute the parities as $P_8 = D_0 \oplus D_4$, $P_9 = D_1 \oplus D_5$, $P_{10} = D_2 \oplus D_6$ and $P_{11} = D_3 \oplus D_7$. This code protects against a single card failure. It involves writing a single parity page with each data page written. As we will describe in more detail below, we use caching in stable memory to minimize actual writes to parity pages.

A bad disk sector in a RAID Level 5 disk array might be discovered during recovery from a failed disk. Since the sector is necessary for reconstructing the contents of the failed disk, the RAID has lost data. In our system, we can protect against the analogous problem by either using an erasure correcting code with two parity symbols corresponding to RAID Level 6, or we can protect against individual page failure by using an erasure code within each block. For instance, we can organize the 64 pages in a block as a $63+1$ reliability stripe. The analogous solution for disks (intra-disk parity) is much less attractive since the disk capacity is much larger than the write cache (so that we have to update parity sectors immediately) and because rotational latency will add to the cost of writing the parity.

In what follows, we restrict ourselves to considering only inter-card codes that place complete blocks into reliability stripes and we consider only the $63+1$ intra-bank code.

3.3 Organizing Pages and Blocks

We use an abstraction for inter-chip reliability stripes, called a *block group*. Figure 3 gives an example block group based on the (2,1)-MDS (card) encoding. If the system contains $C \cdot B$ banks and a bank contains b blocks, then the system has b block groups of size $C \cdot B$. If each bank relatively addresses its blocks as $[0, b-1]$, then the i-th block group is defined as all of the blocks with relative address i from each of the $C \cdot B$ banks.

Each block group has an associated *erasure code instance*. The instance includes a *parity map*, which describes how data is encoded within the block group. Upon initialization, all of the block groups are placed into a queue, where the block group at the head of the queue is called the *current block group*. Similar to a log-structured file system, all writes go to the current block group. Once all of the pages within the current write group have been programmed, the block group is dequeued. While we are writing user data to the current block group, we maintain the parity data in cache. We only write parity data when we dequeue the block. After we dequeue the current block group, the head of the queue becomes the new current block group. We erase the blocks in it before the first write is applied to the group. Thus, the system will only incur the cost of an erase every $\frac{k}{n} \cdot (C \cdot B \cdot 64)$ page writes. Obviously, as the block group queue empties, a cleaner must be invoked to free active block groups. We plan to draw on existing work in log cleaners and leave the design of a block group cleaner to future work.

Block groups are further organized into *page stripes*. Since there are 64 pages in a block, there are 64 page stripes in a block group. As with block groups, the i-th page stripe is defined as all of the pages with a relative address of i in the corresponding block. Writes are applied to page stripes in order from 0 to 63.

Stripe policies determine the order at which data is written to the data pages in an individual page stripe. Currently, the policy is set using a single parameter: *stride*. The i-th write to a page stripe goes to page $i \cdot \text{stride} \% (D-1)$, where D is the number of data pages in the page stripe. A stride of 1 will simply write to the pages in order.

3.4 Changing Encoding On-the-Fly

This organization allows the system to effectively change the level of fault tolerance on a block group basis. For example, if an administrator wishes to perform a system-wide increase in fault tolerance, a new encoding may be chosen for all future block group allocations. Every block group with the old encoding may then be given priority for cleaning, resulting in a slow change in system-wide fault tolerance. Policies for choosing which block groups to clean is an open problem. In practice, changes in the level of fault tolerance would be triggered autonomously based on failure statistics collected similarly to SMART data in current disk drives.

3.5 Updating Parity

All parity updates for a page stripe are staged in the parity cache until all of the dependent data has been written in flash. As an example, assume a page stride of 1 is chosen for the page stripe shown in Figure 3. Each data page within a page stripe is written in order (e.g. $D_0, D_1, \ldots, D_7$) and each page write has an associated parity update (e.g. $P_8 = P_8 \oplus D_0$, $P_9 = P_9 \oplus D_1$, etc.). In this case, P_8 will be staged in cache until the 5-th write to the page stripe has been applied, since P_8 depends on D_0 and D_4. To minimize the time a parity page is staged in cache, the stride may be set to 4. In this case, data page writes are applied in the following order: $D_0, D_4, D_1, D_5, D_2, D_6, D_3, D_7$, allowing the associated parity pages to be hastily flushed from the cache.

Inter-block parity pages are also maintained in the parity cache until all dependent data pages have been added into the parity and written. If a (63,1)-MDS (block) encoding is chosen for each block in Figure 3, then the single parity page in a given block will be flushed from the cache after at most $(62 \cdot 8) + 1 = 435$ data page writes.

3.6 Limitations: Bad Blocks and Scaling

While the block group and page stripe abstractions simplify some of the issues that arise when applying erasure codes to flash, we find these data structures make bad block management and scaling a bit more complicated. One simple solution to the bad block problem is to treat all bad blocks as all-zero data blocks and adjust parity relationships within its parent block group accordingly. Once too many blocks in a single block group have gone bad, the entire block group will be marked as bad. A similar problem arises when incorporating more devices into an existing array. The solution to both of these problems remains open.

4 Evaluation

We evaluate the efficacy of our mechanisms by running microbenchmarks on a flash simulator and estimating the page-error rate under a variety of configurations. Very little is known about the failure rates of whole flash devices or chips, so we leave a device-level reliability estimate to future work. The goal here is to compare the page-error reliability, performance, and space tradeoffs when using our mechanisms in an array of flash devices. While, the results are not intended to entirely prove the effectiveness of these mechanisms in a real system, we believe that these initial results justify further analysis into the reliability mechanisms proposed for arrays of flash devices.

4.1 Setup

All of our measurements were taken on a host computer that is connected to 8 simulated flash cards. Each flash card is simulated using NetApp's Libra card simulator. Each card contains 4 DMA channels and 8 banks of flash. Each bank contains 64 blocks. The access latencies are as follows: 0.2 ms to program, 0.025 ms to read and 1.2 ms to erase. The host is equipped with two 2.74 GHz Pentium 4 processors and 2 GB of RAM. We run microbenchmarks to estimate write and rebuild latency with and without the use of erasure codes. Table 1 lists the erasure code configurations used in our evaluation. The stride in every configuration is set to the number of banks per card so each successive page write is sent to a different card, maximizing parallelism on writes.

4.2 Reliability

Code Config	Space Eff.
(7,1)-MDS (card)	87.5%
(6,2)-MDS (card)	75%
(7,1)-MDS (card) + (63,1)-MDS (block)	85.9%

Table 1: *Erasure codes used in our analysis.*

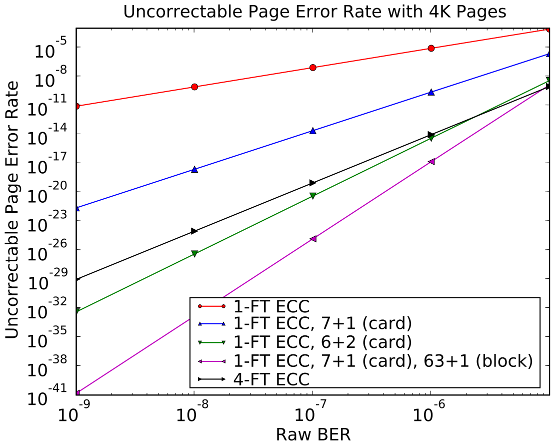

Figure 4: *Estimated Uncorrectable Page Error Rate*

We derive the estimated uncorrectable page error rate (UPER) using the cumulative Binomial distribution: $F(k;n,p) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$. The UPER of a page that is encoded with a T bit tolerant ECC is $\mathrm{UPER}_{\mathsf{ECC}}(T) = 1 - F(T; P, \mathrm{RBER})$, where P is the page size in bits and RBER is the raw bit-error rate. Furthermore, given a (k,m)-MDS code, the UPER is computed as $1 - F(m; k+m, \mathrm{UPER}_{\mathsf{ECC}}(T))/(k+m)$. Finally, as long as the corresponding erasure codes only compute a single parity element, the UPER of a two-level encoding with M symbols in the first encoding and N symbols in the second encoding is [†]

$$\frac{\sum_{i=4}^{MN} \beta(M,N,i) \cdot (\mathrm{UPER}_{\mathsf{ECC}}(T))^i \cdot (1 - \mathrm{UPER}_{\mathsf{ECC}}(T))^{MN-i}}{MN}$$

Figure 4 illustrates the reliability tradeoffs when using 1 and 4-bit fault tolerant, page-level ECC and 1-bit fault tolerant ECC with the erasure code configurations given in Figure 1. It is important to note that the erasure code configurations not only have the ability to tolerate high level failures (e.g. cards, chips, etc.), they also drastically decrease the UPER. We find that using a 1-bit fault tolerant ECC per page with the (7,1)-MDS (card) + (63,1)-MDS (block) configuration leads to the lowest UPER across the RBER spectrum, because a block group can tolerate the loss of at least 3 pages. The (6,2)-MDS (card) configuration can tolerate many but not all 3 page errors in a block group. Note that if we considered device failures, (6,2)-MDS (card) would have the best reliability.

[†] We calculate $\beta(m,n,i)$ as $\binom{mn}{i} - V_i$, where V_i is an estimate of the number of recoverable erasure patterns of size i [3].

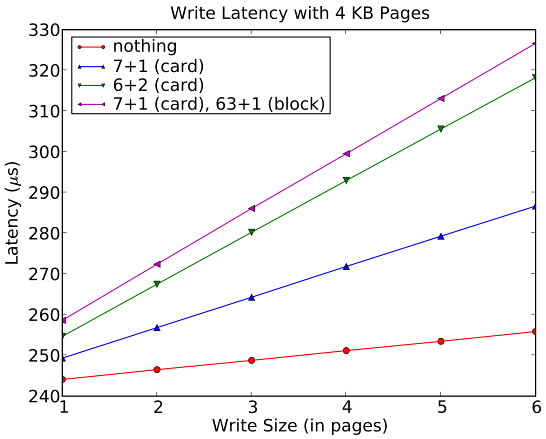

Figure 5: *Write Performance*

4.3 Performance

Figure 5 compares the write latency of no reliability mechanisms and the configurations given in Table 1. To show the effect of parallelism when writing across cards, we varied the size of writes from 1 to 6 pages. We chose the cutoff of 6 because of the contention introduced under this write policy when writing to a new page stripe under the (6,2)-MDS (card) configuration. As expected, the amount of parity written per page determines the performance of each scheme: (7,1)-MDS (card) write 1 parity per page, (6,2)-MDS (card) writes 2 parity per data page and (7,1)-MDS (card) + (63,1)-MDS (block) writes 4 parity per data page.

Another important performance measurement is the page rebuild performance. The time to read a single page is roughly 66.72 μs. Recovering a page under (7,1)-MDS (card) and (6,2)-MDS (card) takes, on average, 100.90 μs and 96.01 μs, respectively. If we try to recover a page from the block parity in the (7,1)-MDS (card) + (63,1)-MDS (block) configuration, a recovery operation takes approximately 4511.12 μs. This type of recovery performs poorly because each page involved in intra-block page recovery must be loaded by the same page register, resulting in 63 serialized reads.

5 Discussion and Open Issues

In this paper, we argue for specific mechanisms that may be used when building highly reliable flash-based storage systems. We also briefly analyzed a few tradeoffs a system designer can make when building such a system: reliability, performance and space efficiency. We found that each choice of erasure code configuration affects all three axes in the tradeoff space. Additionally, as the RBER increases, a reasonable level of reliability can be maintained by adjusting block group encodings on-the-fly.

This work only scratches the surface of flash-based storage system design. In addition to fleshing out how our mechanisms for erasure coding will fit into flash-based systems, there exist many open problems in this area. We must decide how to clean block groups and determine proper cleaning policies when changing erasure code on-the-fly. Additionally, we must figure out how to restructure block groups as new devices are added.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC), Kaladhar Voruganti and Garth Goodson, who provided valuable feedback on the ideas in this paper. This research was supported by the Petascale Data Storage Institute, UCSC/LANL Institute for Scalable Scientific Data Management and by SSRC sponsors including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Digisense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, NetApp, Seagate, Symantec, and Yahoo.

References

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

[2] K. R. Fei Sun and T. Zhang. On the use of strong bch codes for improving multilevel nand flash memory storage capacity. Technical report, Rensselaer Polytechnic Institute, 2006.

[3] M. A. Kousa. A novel approach for evlauating the performance of spc product codes under erasure decoding. In *IEEE Transactions on Communications*, 2002.

[4] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, and F. Trivedi. Bit error rate in nand flash memories. In *IEEE International Reliability Physics Symposium*, 2008.

[5] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

[6] F. Sun, S. Devarajan, K. Rose, and T. Zhang. Design of on-chip error correction systems for multilevel nor and nand flash memories. *Circuits, Devices & Systems, IET*, 1(3):241–249, June 2007.

[7] T. M. Taeho Kgil, David Roberts. Improving nand flash based disk caches. In *International Symposium on Computer Architecture*, 2008.

[8] Texas Memory Systems. *An In-depth Look at the RamSan-500 Cached Flash Solid State Disk.*