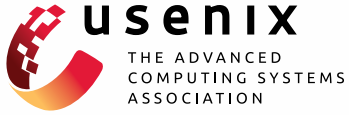


FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance

Ping Huang, ***Virginia Commonwealth University and Huazhong University of Science and Technology;*** **Pradeep Subedi,** ***Virginia Commonwealth University;*** **Xubin He,** ***Virginia Commonwealth University;*** **Shuang He and Ke Zhou,** ***Huazhong University of Science and Technology***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/huang

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance

Ping Huang[‡§], Pradeep Subedi[‡], Xubin He[‡✉], Shuang He[§], Ke Zhou[§]
[‡]*Virginia Commonwealth University, USA*
[§] *Wuhan National Laboratory for Optoelectronics*
Huazhong University of Science and Technology, China
{*phuang,subedip,xhe2*}*@vcu.edu, hshopeful@gmail.com,k.zhou@hust.edu.cn*

Abstract

The ever-growing capacity and continuously-dropping price have enabled flash-based MLC SSDs to be widely deployed as large non-volatile cache for storage systems. As MLC SSDs become increasingly denser and larger-capacity, more complex and complicated Error Correction Code (ECC) schemes are required to fight against the decreasing raw reliability associated with shrinking cells. However, sophisticated ECCs could impose excessive overhead on page decoding latency and thus hurt performance. In fact, we could avoid employing expensive ECC schemes inside SSDs which are utilized at the cache layer. We propose *FlexECC*, a specifically designed MLC SSD architecture for the purpose of better cache performance without compromising system reliability and consistency. With the help of an upper-layer cache manager classifying and passing down block access hints, *FlexECC* chooses to apply either regular ECC or lightweight Error Detection Code (EDC) for blocks. To reduce performance penalty caused by retrieving backend copies for corrupted blocks from the next-level store, *FlexECC* periodically schedules a scrubbing process to verify the integrity of blocks protected by EDC and replenish corrupted ones into the cache in advance. Experimental results of a proof-of-concept *FlexECC* implementation show that compared to SSDs armed with regular ECC schemes, *FlexECC* improves cache performance by up to 30.8% for representative workloads and 63.5% for read-intensive workloads due to reduced read latency and garbage collection overhead. In addition, *FlexECC* also retains its performance advantages even under various faulty conditions without sacrificing system resiliency.

1 Introduction

As system architects have always been pursuing to build and/or optimize storage systems in both high-performance and cost-effective ways, NAND flash-based Solid State Drives (SSD) have been intensively researched to be efficiently utilized in various storage systems during the past decade due to their highly desirable characteristics (e.g., high performance and low power consumption) [5, 35, 34, 39]. Compared to rotating hard disk drives (HDD), SSDs provide one order of magnitude higher performance while consuming much less power to finish running the same workloads[13, 20].

However, because of the relatively high cost per GB [30, 12] and limited lifetime concerns [2, 26, 19], NAND flash-based SSDs are nowadays particularly widely utilized as a cache in front of storage systems comprised of HDDs, aiming to exploit their complementary advantages [36, 4, 18]. For instance, SSDs have already been utilized as front-end caches in storage products, including EMC's *VFCache* [1], Apple's *Fusion Drive* and Fusion's *ioControl*^{TM} *Hybrid Storage* and deployed in various scenarios including networked environment [22], cloud infrastructures [27, 4].

The enabling factors of SSD's wide deployment as a large non-volatile cache are primarily attributed to their steadily-expanding capacity and the resultant affordable cost, which in turn are essentially driven by technology scaling and the employment of Multi Level Cell (MLC), i.e, scientists have pushed two or even more bits into each diminishingly-sized flash memory cell. Researchers have suggested a variety of ways to improve SSD cache performance according to flash peculiarities, including dividing the cache space into read and write caches [21] and designing effective cache algorithms [42]. However, technology scaling has also caused many concerns [15]. For example, recent research findings have revealed that increasing an additional bit in a storing cell would reduce chip's lifetime by 5-10%, and shrink throughput and increase latency by 55% and 2.3x on average, respectively [16]. These problems could become an impediment to further improving their performance as a cache which is supposed to provide high performance.

Each NAND flash memory cell is a floating gate transistor that is able to preserve electrons. Bits information stored in each flash memory cell are represented and differentiated by the different voltage levels of the trapped charges. The reliability-impacting factors such as programming inaccuracy, electron detrapping, cell-to-cell interference [33, 32] are becoming increasingly severe when cells are pushed to store more bits, causing flash memory to exhibit an climbing high raw bit error rate (RBER) [15, 16], which renders them not suitable for practical usage. For example, the RBER of MLC flash memory is around 10^{-6}, while manufacturers usually rate the uncorrectable bit error rate (UBER) in their data sheets to be 10^{-11} [29, 10]. To bridge the reliability gap, sector-level or page-level Error Correcting Codes are synergistically implemented in flash memory controllers to achieve a practically acceptable UBER. However, as flash storage gets denser, we have witnessed the deployed ECCs are becoming more and more advanced and complicated [7], from Hamming Code to Bose-Chaudhuri-Hocquenghem (BCH) and Reed-Solomon codes [10] to Low Density Parity-check (LDPC) [49]. Therefore, the ECC implementation complexity has correspondingly increased significantly, causing prolonged encoding and decoding latencies. For instance, the decoding latency of a BCH tolerating 12 bit errors for an 8KB page are around $180\mu s$ and $17\mu s$ in contrast to $90\mu s$ and $10\mu s$ in a BCH tolerating 6 bit errors, with software [21] and hardware [41] implementation respectively, which accounts for a significant percentage of the page access latency.

Fortunately, in cache-oriented MLC flash-based SSDs, the need for expensive error correcting codes could be obviated. The reason is that in occurrences of errors, accesses to corrupted blocks can be serviced by their backup copies in the next layer and most of the time accesses can be completed faster because of the absence of excessive decoding overhead. Based on this observation, in this paper, we advocate a novel cache-oriented SSD architecture called *FlexECC*, a cross-layer design targeting specifically for MLC SSDs exhibiting high RBER and employing expensive ECC schemes. *FlexECC* achieves better cache performance by flexibly and selectively applying either regular ECC or light EDC [21] to flash pages according to their consistency and reliability requirements. Specifically, taking advantage of the information conveyed by the frontend cache manager via proposed interfaces (Section 3.3), *FlexECC* can easily identify the storage requirements of different pages and accordingly apply the appropriate protection or correction schemes via programmable flash memory controller. When writing fresh data which have no backup copies in the next storage layer, *FlexECC* adopts normal error correction code (specifically, BCH in our design), otherwise it applies simple and fast error detection code (EDC) (specifically, cyclic redundancy code or CRC in our design). Due to the differences in decoding latencies between BCH and CRC (Section 2.2), read accesses to CRC-protected pages would be significantly speeded up, enhancing the cache performance. The more CRC-protected pages in the cache device, the greater cache performance *FlexECC* provides. Furthermore, in order to mitigate the performance impacts of fetching data from the underlying layer for corrupted pages in the critical path, *FlexECC* schedules a scrubbing process to verify the integrity of CRC-protected pages and populates corrupted pages in advance. Evaluation results with both representative and synthetic workloads have shown that *FlexECC* is able to improve cache performance by an impressive degree without sacrificing consistency and reliability relative to normal ECC-armed MLC flash-based SSD.

Our main contributions in this work are two-fold. First, to the best of our knowledge, the proposed *FlexECC* is the first work to selectively replace ECC with EDC to improve SSD cache performance without compromising cache consistency and reliability. This bears important implications for future-generation MLC SSDs which require more advanced error correction codes and incur high decoding latencies to read operations. We are not making trade-offs between performance and reliability or consistency [25, 32], instead, we aim to improve performance while maintaining the same level of resilience by leveraging the characteristics of cache systems. Second, we have implemented a proof-of-concept prototype of *FlexECC* and conducted extensive evaluations to show that *FlexECC* is able to improve performance over conventional SSDs for a variety of workloads, even under various faulty conditions.

The remainder of this paper proceeds as follows. We discuss the background and our motivation in Section 2. Following that, we elaborate on the details of *FlexECC* in Section 3. We conduct experiments to evaluate *FlexECC* in Section 4, followed by a discussion of related work in Section 5. Conclusions of this work are given in Section 6.

2 Background and Motivation

2.1 Flash Memory Reliability

Flash memory cells are floating gate transistors which hold electrons to represent information. Each flash memory cell can be designated to represent one bit information (SLC), two bits (MLC) or three bit (TLC). The represented storage state is differentiated by the voltage level of trapped charges. Programming or writing flash memory cell is the process of injecting electrons

into the cell to the level corresponding to the desired state, and reading is the process of sensing out the represented voltage level and comparing it with preset reference levels to determine its value. By its very nature, the trapped charges are constantly in a moving state and can shift to their neighboring cells (i.e., current leakage) over time, causing voltage shifting [32, 25] and therefore data corruption. Moreover, as flash memory cells experience more program and erase operations, their charge-trapping ability degrades and as a result are more prone to errors [33]. The occurring probability of these errors are called raw bit error rate (RBER). SLC flash memory typically exhibits two orders of magnitude better RBER than MLC [14, 10], because MLC flash memory has much shorter differential voltage window between adjacent voltage thresholds than SLC, which causes it more difficult for MLC to differentiate the statuses.

In design practice, flash-based SSDs typically implement ECCs in memory controllers [21] to meet reliability and endurance requirements, causing a performance-reliability trade-off in the design space. ECC is a kind of information encoding scheme which can tolerate a specified number of bit errors (called error correction capability t) by augmenting a certain amount of redundant information to the original message of length k, which typically equals to the page size in flash memory. Corrupted message can be reconstructed via decoding as long as the number of bit corruptions are within the ECC correction capability. Considering the wide adoption of BCH code in commercial SSDs, we base our discussions on BCH in the remaining sections. The bit error rate after applying ECC is called uncorrectable bit error rate (UBER). Assume an ECC scheme has an error corruption capability of t and the length of an encoded message is N, then the relationship between UBER (P_{UBER}) and RBER (P_{RBER}) is given by Equation 1.

$$P_{UBER} = \frac{\sum_{n=t+1}^{N} \binom{N}{n} * (P_{RBER})^n * (1 - P_{RBER})^{N-n}}{N} \quad (1)$$

Intuitively, to guarantee the same level of UBER (e.g., 10^{-11}), we can either increase ECC correction capability or decrease RBER. For example, more precise Incremental Step Pulse Programming control (i.e., using smaller $\triangle V_{pp}$ [44, 43]) produces smaller RBER and using more powerful ECC schemes also guarantees target reliability [6, 9]. However, as flash geometries become increasingly smaller (3x- and 2x-nm regimes) and denser [11], those techniques are no longer necessarily as effective as before, which is evidenced by the continuous increases in error correction requirements, program time and read time observed between different flash process generations [7]. When flash storage becomes denser, the noise margin narrows, necessitating very small $\triangle V_{pp}$ to program pages and thus causing prolonged programming process and imposing significant overhead on performance [33, 32]. On the other end, implementing more powerful ECC schemes on denser flash memory could be prohibitively expensive or even unrealistic for the following reasons. First, correction logic becomes complex, costly and occupies more silicon area and the resultant decoding latency increases correspondingly. Second, it increases power dissipation, whose side effects counteract ECC's efforts to improve reliability. Third, the page spare area may no longer have enough space for the expanding redundant information.

2.2 Replacing ECC with EDC for Cache

It has been observed in previous research [33, 25] that most of the occurred errors in flash memory are retention errors, i.e., errors caused by loss of charges over time, and flash memory exhibits reasonably high reliability at its early usage. In contrast to permanent storage, cached data are transient and live for a short lifespan, typically ranging from seconds or hours to days rather than months or years [48]. Therefore, the corruption probability of cached data is comparatively low. Moreover, even if corruptions do occur to cached data, corrupted data blocks can still be serviced by back-end storage as long as the blocks have been flushed down beforehand, at the cost of accessing disk or RAID storage systems.

Based on the above analysis, we are motivated to selectively relax ECC correction capability of certain cache blocks (i.e., the blocks which have consistent backup copies) to avoid decoding overhead for better performance. We only reserve error detection capability using lightweight EDC for ECC-relaxed blocks. Given the low corruption probability of short-living cached data and the significant discrepancy between ECC decoding latency and EDC verification overhead (to be discussed shortly), it is reasonable to expect performance improvement coming out of ECC-relaxed cache architectures while maintaining system reliability.

In the remainder of this section, we conduct theoretical performance analysis on ECC and EDC to demonstrate the potential performance gains that could be obtained by replacing ECC with EDC. Due to their popularity, we use primitive binary BCH [40, 41] for default ECC and CRC for default EDC. CRC, short for cyclic redundancy code, is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. The detection capability of CRC is characterized by how many concurrent bit errors it is able to detect. Binary BCH code has a form of (n,k,t), where n is the codeword length equal to $2^m - 1$ for some positive integer m, t is the correction capability indicating the maximum bit errors BCH is able to tolerate, and k is

the length of the original message. The BCH arithmetic operations are based on Galois field GF(2^m) [24]. For fairness, we configure BCH to be able to tolerate t bit errors and CRC to detect t bit errors for each flash page[1].

Encoding: Encoding operation is associated with every write operation in flash to calculate redundant bits which are then written to the flash page's spare area together with page data. CRC and BCH share the same simple encoding procedure. To encode a page message M(x), both CRC and BCH divide the original message by a polynomial generator G(x) whose degree is dependent on t. The resultant remainder R(x) is the redundant information. Equation 2 gives the encoding calculation. Therefore, the encoding latencies of both CRC and BCH are approximately the same and equal to the time taken by a polynomial multiplication [8]. In other words, CRC and BCH incur the same additional latency to write operations.

$$\frac{M(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \tag{2}$$

Decoding/Detecting: A decoding/detecting process accompanies every flash page read operation. After reading out each page content, the flash memory controller verifies the integrity of the page content according to the adopted protection scheme. In contrast to the similar encoding procedure, BCH decoding is far different from CRC detection. CRC detection process is quite straightforward. Suppose the read page content is M(x)$'$. CRC performs the same arithmetic operation as in Equation 2 and checks whether the new remainder is identical to the previous one or not. Essentially, it is equivalent to verifying Equation 3. If Equation 3 holds true, then it is assumed no error occurs, otherwise the message is considered corrupted, so CRC detection process consumes the same time as CRC encoding.

$$\frac{M(x)' - R(x)}{G(x)} - Q(x) = 0 \tag{3}$$

BCH decoding is much more complicated and involves three steps, *syndrome computations*, *finding error-location polynomial* and *error correction*. The first step is to compute $2t$ syndrome components $S_1, S_2, \cdots, S_{2t}$, each of which is essentially a polynomial calculation. Then, based on the $2t$ syndromes, the second step uses *Berlekamp-Massey* algorithm to calculate the error-location polynomial $\sigma(x) = 1 + \sigma_1(x) + \sigma_2(x^2) + \cdots + \sigma_t(x^t)$. Finally, the third step solves the roots of $\sigma(x) = 0$ by using exhaustive *Chien Search* algorithm and outputs an error vector indicating the error positions. It should be noted that after obtaining the $2t$ syndromes, if all of them are evaluated to zeros, the message is considered error-free and the decoding process terminates immediately. Specific details about BCH code can be found in [24].

According to [24], a polynomial calculation takes $(n-1)$ additions and $(n-1)$ multiplications, *syndrome computations* take $(n-1)t$ additions and nt multiplications, *finding error-location polynomial* takes $2t^2$ additions and $2t^2$ multiplications, and *error correction* takes nt additions and nt multiplication. Suppose T_a and T_m are the time needed by per addition and per multiplication, respectively. Then the achievable speedups of replacing BCH with CRC for decoding a correct message ($S_{correct}$) and a corrupted message (S_{error}) are given by Equation 4 and Equation 5, respectively.

$$S_{correct} = \frac{(n-1) \times t \times T_a + n \times t \times T_m}{(n-1) \times (T_a + T_m)} \tag{4}$$

$$S_{error} = \frac{(2t^2 + 2nt - t) \times T_a + (2t^2 + 2nt) \times T_m}{(n-1) \times (T_a + T_m)} \tag{5}$$

Suppose $T_m = T_a$, i.e., the time taken to perform an addition is equal to that of a multiplication[2], typically one clock cycle, then $S_{correct}$ and S_{error} become $\frac{(2n-1)t}{2(n-1)}$ and $\frac{4nt+4t^2-t}{2(n-1)}$, which in turn approximately approach t and $2t$, respectively, when $n \gg t$.

In summary, we have demonstrated that by using CRC instead of BCH, we are able to reduce the latencies of decoding uncorrupted and corrupted message by t and $2t$ times, respectively. Given the increasing value of t and the associated decoding latency in MLC SSDs, the extent of decoding latency reduction will increase correspondingly and potentially translate to more significant performance improvement.

In real implementations, BCH can be either realized in software or hardware. In this paper, we assume hardware implementation, since typically the memory controller inside SSDs employs electrical circuit to perform BCH encoding and decoding for high performance purpose. BCH hardware implementation presents trade-offs among chip area, cost and latency [41, 40]. Different implementation configurations would cause different latencies. The more circuits are deployed, the less latency it incurs, but the more energy it consumes. In our evaluations, according to the hardware implementation in [41], we use $10\mu s$ as the decoding latency of a BCH tolerating 6 bit errors out of a flash page. Based on this latency, we use the above analysis to derive other parameters as shown in Table 2 in Section 4.1.

3 *FlexECC* Design and Implementation

In this section, we elaborate on the design and implementation details of *FlexECC*. We first give an overview

of *FlexECC*, followed by a basic description of the cache manager in which we collect and pass down the block access information. Then we present the proposed extended interfaces via which access information is passed down to facilitate the underlying device's internal management. Following that, we describe the scrubbing process which is a precautional technique to suppress the performance overhead associated with accesses to erroneous pages. Moving on, we briefly discuss the garbage collection process in *FlexECC* with a focus on the differences relative to conventional SSDs. Finally, we give a holistic discussion on how the incoming requests are handled by *FlexECC*.

3.1 System Overview

As discussed previously, the idea of *FlexECC* is quite simple. It essentially comes down to two critical problems. The first problem is how to characterize block access behaviors and relay the collected information to the underlying device. The second problem is how the cache device can take advantage of the collected information to improve its performance. For the first problem, *FlexECC* augments a cache monitor into an ordinary cache manager. The monitor observes the cache behaviors and infers the storage requirements of the corresponding blocks which are are supposed to be stored in the cache layer. For the second problem, *FlexECC* employs a *Programmable Memory Controller (PMC)* inside the cache device to dynamically allocate CRC-protected or BCH-encoded pages to accommodate the incoming page writes according to their storage requirements.

Figure 1 shows a holistically architectural view of *FlexECC*. As depicted in the picture, the upper part is a modified cache manager which is able to collect block access information and send the information down to the SSD cache device to facilitate its internal management. In the middle is the SSD cache device with two added components including a hardware PMC and a software scrubber. In the bottom is the underlying storage system comprised of HDDs. In addition to constructing a basic hybrid storage system, the cache manager is augmented with the functionality of tracking and tagging block accesses to SSD. The collected information can be passed down to SSD via extending cross-layer interfaces, which has been proposed and evidenced by the techniques employed in previous researches including *Shepherding I/O* [17], *DSS* [28] and *FlashTier* [37]. The SSD cache device internally employs a *Programmable Memory Controller (PMC)* [21] which is able to program pages[3]to be either BCH-encoded or CRC-protected and allocate different types of pages to accommodate incoming requests according to their respective requirements, which is in spirit similar to the fast and slow pages allocation policy described in [16]. The *PMC* divides the entire cache space into two different regions, namely, CRC-region and BCH-region. Moreover, *FlexECC* actively initiates a scrubber process to verify the integrity of CRC-protected pages and prepares to populate corrupted ones from underlying storage before they are accessed. In addition, the SSD FTL is slightly modified, with each FTL entry having several added tags to provide auxiliary information, for example, in what code scheme (BCH or CRC) the page is protected, etc.

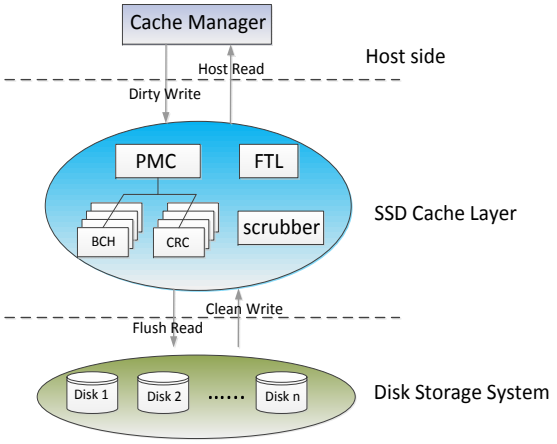


Figure 1: System Architecture.

3.2 Cache Manager

In our context, a *cache manager* interposes above the disk device driver in the operating system to send requests to either the flash device or the disk system directly, as it is with *FlashTier* [37]. It transparently constructs and manages an SSD-HDD hybrid storage system in which SSD acts as an inclusive cache layer above the disk system. The *cache manager* dictates which data blocks are written to the SSD cache through the adopted cache replacement policy, like Least-Recently-Used (LRU) or First-In-First-Out (FIFO). It supports two cache modes: *Write-through* and *Write-back*. In *Write-through* mode, on every write to the SSD cache, the cache manager also persists the data to the disk system before it reports write completion, which guarantees consistence at any time. In *Write-back* mode, the cache manager may write to the SSD cache without updating the disk system, causing dirty data in the cache. Cached blocks are flushed to the disk system for persistence at a configurable rate, e.g., every 5 minutes. For a write request, when there is no enough space, the cache manager evicts a victim block according to the replacement policy to make room for the incoming write. For a read request, the cache manager first consults the SSD. If it is not present in the cache, the cache manager fetches the data

from the underlying disk system and populates it into the cache. By default, we assume *Write-back* mode in the discussion of *FlexECC*, because *Write-through* is an extreme scenario in which the whole cache space could be safely CRC-protected. We implement the *cache manager* based on *FlashCache* [37]. Specifically, we monitor every triggering event that causes read or write operation to the SSD cache, and forward the information to SSD via extended access interfaces.

3.3 Extended Interfaces

Extending existing interfaces between neighboring layers to communicate useful information for various purposes has been proposed in previous literature [28, 38]. Such extensions can be conveniently realized via leveraging the reserved or unused bits in the communication protocols, e.g., SCSI protocol. In *FlexECC*, we use a similar approach to pass information about cache behaviors to SSD to help its internal management. We propose four extended access interfaces to capture different reasons that cause accesses to the SSD, namely *Dirty Write*, *Host Read*, *Clean Write* and *Flush Read*, which are indicated in Figure 1. These interfaces are defined from the perspective of the SSD cache device. We discuss each of them as follows:

Dirty Write: a request to write a fresh data block which has no backup copy in the disk system and thus requires high reliability guarantee. New content generated by upper-level applications are written into the cache device using this interface. In response to this operation, the programmable memory controller designates BCH-encoded pages to store the content.

Clean Write: a write request to write a clean data block which has consistent backup copy in the back-end storage. Block migrations originating from disk to cache, e.g., due to a miss or populating corrupted pages, are accomplished through this interface. In response to this operation, the programmable memory controller designates CRC-protected pages to store the content.

Host Read: This interface is used to satisfy data reads issued by applications. It corresponds to cache read hit. *Host Read* data can be either BCH-encoded or CRC-encoded, depending on its state when it is requested.

Flush Read: a read caused by flushing dirty data back to the disk system due to releasing cache space or periodical time-out flushing down. Internally, *FlexECC* monitors this operation and marks associated flash pages as eligible to be free from BCH-encoded. During garbage collection, the marked pages contained in victim blocks are relocated to clean blocks using *Clean Write* interface.

In *FlexECC*, a data block could be in four states, which are named *HOST*, *BCH*, *CRC*, and *HDD*, indicating when the block is in host memory, in a BCH-encoded page, in a CRC-protected page and in back-end storage, respectively. However, it should be noted that these four states are not necessarily exclusive to one another. For example, the same block can be in *HDD* and *CRC* states simultaneously. The data block changes its state in response to the operations applied to it. Figure 2 shows the page state transition diagram with respect to the interface operations. The *GC write* represented by dashed arrow denotes the state transition of marked data pages (by *Flush Read*) from *BCH* to *CRC* during garbage collection.

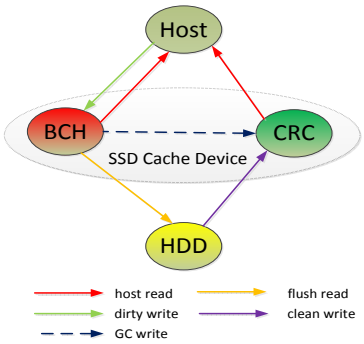


Figure 2: Page Content State Transition Diagram. The page content can be in the host memory (HOST), protected by BCH code (BCH), protected by CRC code (CRC) or in the back-end storage (HDD). Read and write are defined in respect to the cache device.

3.4 Disk Scrubber

While CRC can guarantee the minimum reliability, due to CRC's inability to correct errors, corruptions occurred to CRC-encoded pages may incur extraordinarily large overhead, since accesses to corrupted pages have to be satisfied by the back-end storage whose performance is typically one order of magnitude slower than that of the cache. To prevent such overhead severely impacting performance, *FlexECC* regularly schedules a *Scrubber* [31] process to verify the integrity of CRC-encoded pages. The *Scrubber* is a two-step process. The first step is a lightweight step which iteratively scans the CRC-protected pages to verify their checksums. If any inconsistency is detected, the corresponding page is marked as corrupted. The second step is to initiate a data migration process to replenish those corrupted pages found in the first step. Depending on when the corrupted pages are accessed, they could be forced to be fetched from back-end storage on the access path to them, which could cause significant performance overhead, or they could be prefetched by *Scrubber* into the cache before they are actually accessed. To minimize performance impacts,

we leverage the idleness in the workloads to launch the scrubber process. Specifically, when the observed inter-request interval T_{inter} is longer than a configured multiple m of the time T_{crc} taken by CRC verification, the *Scrubber* performs the first step; when T_{inter} is longer than the estimated time T_{disk} taken by fetching a block from the underlying disk system, the *Scrubber* performs the second step to prefetch $\lfloor T_{inter}/T_{disk} \rfloor$ blocks. Migrated blocks are written to the SSD cache via *Clean Write* interface. In our evaluation, we set m to be 10, T_{crc} equal to the CRC encoding latency and T_{disk} to be the average disk access latency 1.5*ms* [37].

3.5 Garbage Collection

In SSDs, garbage collection (GC) process is executed to reclaim flash space by erasing victim blocks and may bring about significant performance impacts because of its interference with normal activities [16, 37]. The GC overhead mainly comes from consolidating valid pages from victim blocks to clean blocks and erasing victim blocks. *FlexECC* employs a slightly modified greedy algorithm to perform GC. As it is with the normal greedy algorithm [2], *FlexECC* also selects the block which has the highest cleaning efficiency, i.e, the block contains the most invalid pages within it, as the victim block. When migrating the valid pages, CRC-protected pages are relocated to CRC-protected pages, while BCH-encoded pages which have been previously marked out by *Flush Read* are rewritten to elsewhere using *Clean Write* (the "GC write" operation in Figure 2) and the remaining BCH-encoded pages are again relocated to BCH-encoded pages. Due to the discrepancies in decoding latency between CRC-protected pages and BCH-encoded pages, the GC process in *FlexECC* consumes less time than that in conventional SSD. The more CRC-protected pages there are in the victim block, the shorter the GC process would be. The shortened GC process helps improving the overall performance.

3.6 Putting Them All Together

Summarizing the above discussions, we are able to arrive at the conclusion of how a request is handled by *FlexECC*. First, *FlexECC* determines the type (read or write) of the arriving request, and then determines which code scheme is applied to the target page. Reading uncorrupted CRC-protected page is straightforward, while reading corrupted CRC-protected page has to be satisfied by visiting the underlying disk if the page has not been brought in the cache beforehand by the *Scrubber*. A clean write is destinated to a CRC-protected page, and a dirty write is destinated to a BCH-encoded page. Equation 6 and Equation 7 give the estimated read latency T_r and write latency T_w, respectively. In the equations, R_{crc}, R_{bch} and R_{disk} denote the latencies of reading a CRC-protected page, reading a BCH-encoded page and reading a block from disk, respectively. Similarly, W_{crc} and W_{bch} denote the write latencies of writing a CRC-protected page and writing a BCH-encoded page, respectively. ξ is the corruption probability of a flash page which is relevant to the flash memory RBER, software errors, etc. η is the probability of reading a CRC-protected page and γ is the probability of writing a CRC-protected page. The values of η and γ are dependent on the cache manager configuration (e.g., replacement policy, flushing interval, etc.) and the features of workloads.

$$T_r = (1-\xi)(\eta R_{crc} + (1-\eta)R_{bch}) + \xi R_{disk} \quad (6)$$

$$T_w = \gamma W_{crc} + (1-\gamma)W_{bch} \quad (7)$$

4 Experimental Evaluation

4.1 Evaluation Methodology

To verify the effectiveness of *FlexECC*, we have implemented a prototype and conducted comprehensive evaluations. The evaluations consist of two steps. First, we add a monitor into the *Flashcache* [37] to track the cache block behaviors. The monitor outputs block access traces having the interface operations defined in Section 3.3. Then we use those traces to drive *FlexECC* which is implemented based on an SSD simulator [2]. We use Filebench [23] to generate four representative workloads, *Fileserver*, *Webserver*, *Mailserver*, *OLTP* and two micro-workloads, *R_75* and *R_90*, both of which are read-intensive workloads having 75% and 90% reads, respectively. Each of the workloads runs for 30 minutes on a *Flashcache*-created hybrid storage system comprising of a 1TB disk as the back-end storage and a 250GB PCI-e SSD as a cache. The cache space and working set sizes are set to 30GB and 360GB, respectively. The write-back cache mode and LRU replacement algorithm are used. Table 1 summarizes the traces characteristics.[4] We assume protection granularity is page-based and the ECC redundancy information is stored in the spare region of each page. Table 2 lists the main relevant operational latencies. We simulate an 8-chip 60GB SSD with 15% overprovisioning space and set ioscale to 100 to slow down replaying those too-intensive traces(they are collected on a PCI-E SSD). In addition, we enable the copy-back operation when performing garbage collection within the SSD.

Table 1: Trace Characteristics

	File	Web	Mail	OLTP	R_75	R_90
READ	15.2%	17.8%	21.9%	8.6%	75%	90%
WRITE	84.8%	82.2%	78.1%	91.4%	25%	10%

Table 2: Operational Latencies

Page Read	$25\mu s$	CRC Encoding	$0.8\mu s$
Page Write	$200\mu s$	CRC Decoding	$0.8\mu s$
Block Erase	1.5ms	BCH Encoding	$0.8\mu s$
BCH Correct Dec.	$5\mu s$	BCH Corrupted Dec.	$10\mu s$

4.2 Performance Comparison

In this section, we report the workloads average request response time to demonstrate the overall performance improvement of *FlexECC* over conventional SSD which is armed with regular BCH schemes. In the figures, these two kinds of devices are denoted as *FlexECC* and *BCH-SSD*, respectively. Figure 3 shows the comparison results. As it is clearly shown in Figure 3, *FlexECC* consistently improves the performance across all the tested workloads relative to the traditional SSD armed with regular BCH scheme. Specifically, *FlexECC* improves performance by 30.2%, 30.1%, 30.8%, 28.5%, 49% and 63.5% for *FileServer*, *MailServer*, *WebServer*, *OLTP*, *R_75* and *R_90*, respectively. Generally, read-intensive workloads benefit more from *FlexECC* than other workloads primarily due to the reduced page decoding latency associated with every CRC-encoded page reading, which is evidenced by the fact that the average read response time has been reduced by averagely around 35% and the shortened garbage collection process, which is further investigated in the next subsection. Unsurprisingly, the write response time exhibits only marginal improvement because the CRC encoding overhead is equal to that of BCH encoding. Figure 4 shows the response time CDF comparisons for workloads *FileServer*, *MailServer*, *WebServer*, *OLTP*. It is observed from the figures that each workload has a certain percentage of requests (covered by the visible red line) that have smaller response time in *FlexECC* than in *BCH-SSD*. Those percentages are nearly equal to the read percentages of the workloads (see Table 1), demonstrating the read requests serving have been speeded up.

It is worth noting that the overall performance improvement is a combined outcome, which means even though we are only able to achieve about 25% performance gain for individual page decoding by replacing BCH with CRC, we have seen more than 30% performance improvement for the workloads. The reason is that the shortened page reading and reduced garbage collection can also alleviate resource contention (e.g., reduce request queuing time) and thus further improves performance.

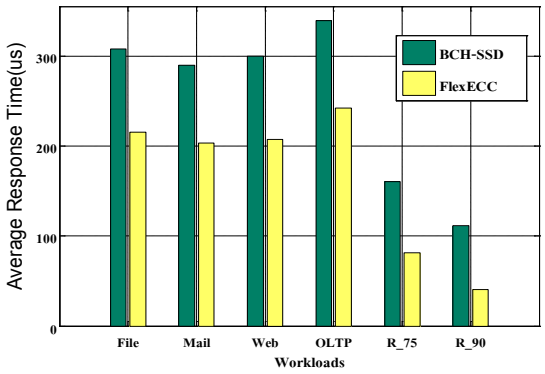
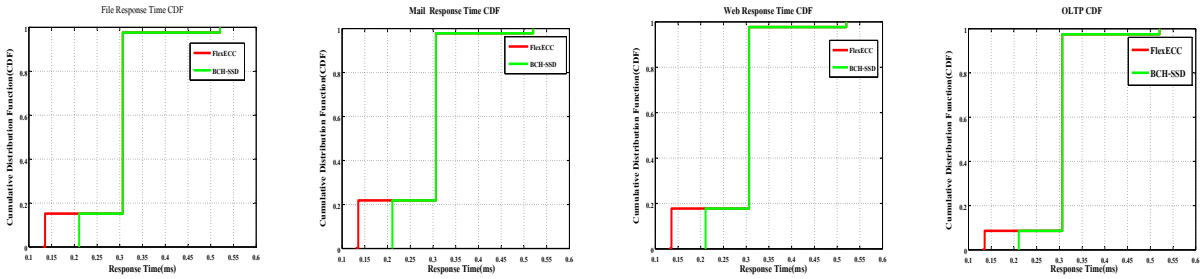


Figure 3: Workloads Average Response Time.

4.3 Garbage Collection

Garbage collection (GC) in SSDs affects performance because it may interfere with ongoing workloads. It spends time in reading out validate pages in victims blocks, writing them to clean blocks and erasing victim blocks. Generally, The shorter time the garbage collection process takes, the less negative impacts it imposes on performance. In our current *FlexECC* implementation, we use the greedy algorithm for selecting victim blocks. The victim blocks may contain CRC-protected pages and BCH-encoded pages. The more CRC-encoded blocks *FlexECC* garbage collects, the faster it migrates valid pages to clean blocks. Figure 5 shows the total cleaning time comparison. From the figure, we notice that compared to *BCH-SSD*, *FlexECC* reduces an impressive amount of total cleaning time, up to 21.8%, 21.8%, 21.7%, 21.7% and 17.8% for *FileServer*, *MailServer*, *WebServer*, *OLTP*, and *R_75*, respectively, even though they have recycled the same number of victim blocks, which are 208829, 215556, 218868, 197455 and 36189, respectively. It is worth noting that the workload *R_90* is not shown in that figure because its total cleaning time is 0. Unlike the average response time, read-intensive workloads do not exhibit the most cleaning time savings because there are fewer page migrations due to the lack of workload writes and associated erasures. Table 3 gives more explanations on the performance gains and garbage collection time reduction, by listing the number of reading BCH-encoded pages (BCH_READ), reading CRC-protected pages (CRC_READ) from the requests and moving CRC-protected pages (CRC_MOVED), transforming BCH-encoded to CRC-protected pages (BCH2CRC) during GC. From the table, we note that read-intensive workloads' performance gains are mainly attributed to the reduced decoding latency rather than saved cleaning time.



(a) File CDF (b) Mail CDF (c) Web CDF (d) OLTP CDF

Figure 4: Workloads Response Time Cumulative Distribution Function (CDF) Comparison.

Table 3: FlexECC Page Statistics

	File	Mail	Web	OLTP	R_75	R_90
BCH_READ	1,517,326	2,420,758	1,899,771	760,480	272,948	8,874,561
CRC_READ	5,191	12,303	6,435	1,265	7,247,563	146,982
CRC_MOVED	3,720,833	3,741,430	3,725,016	3,716,391	1,420,582	0
BCH2CRC	48,501	93,955	57,188	46,688	726	0

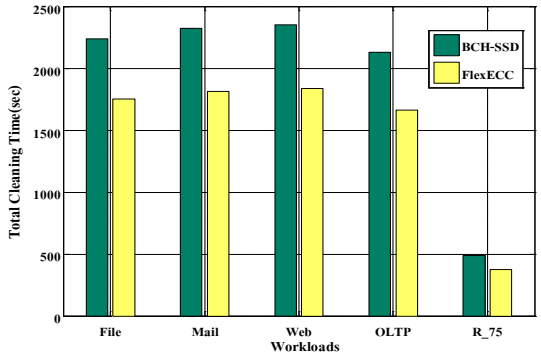


Figure 5: Total Cleaning Time Comparison.

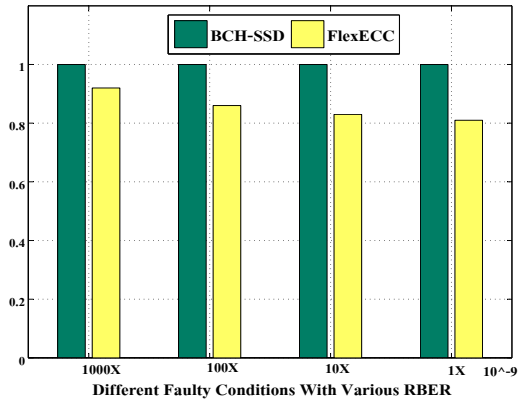


Figure 6: File Performance Under Faulty Conditions.

4.4 Performance Under Faulty Conditions

In this section, we compare the performance under faulty conditions. For simplicity but without loss of generality, we introduce errors to flash pages according to specific raw bit error rates (RBER). In more detail, given a specific RBER, we assume that the every $\frac{1}{RBER}$th bit is corrupted and the page contains this corrupted bit is considered erroneous. Corruptions could occur to both CRC-pages and BCH-pages. We also assume faulty pages only impact reads, because writes are inherently indirected at the FTL layer and thus bypass faulty pages. If the corrupted pages is CRC-protected, then reading it must be serviced by accessing the underlying disk, otherwise it is assumed to be corrected via BCH decoding.

Figure 6 shows the normalized average response time relative to *BCH-SSD* without errors for *Fileserver* workload. As demonstrated in the figure, even under faulty conditions, *FlexECC* still outperforms *BCH-SSD* in error-free conditions, achieving an average 20% improvement. This is because the performance gains brought by partially replacing BCH-encoded with CRC-protected pages and the *Scrubber* prefetching dwarf the overhead associated with handling accesses to corrupted pages. The statistics in Table 4 gives an in-depth explanation regarding the behind reasons. It lists the statistics for *FileServer*, *R_75* and *R_90* workloads. We can make the following observations. First, for write-intensive workloads (i.e., *Fileserver*) there is almost no disk accesses. That's because write requests postpone the visits to corrupted pages and thus increase their probability of being prefetched by *Scrubber*. For read-intensive

Table 4: Corruption Related Statistics When RBER=10^{-7}

Workloads	File	R_75	R_90	Notes
Corruptions	4107	4107	4107	# of corruptions introduced during running
Disk Access	0	8	7	# of disk accesses
Prefetched	2720	3917	4024	# of corrupted pages prefetched by *scrubber*
BCH Decoded	195	34	10	# of corrected pages via BCH decoding

workloads, there are disk accesses happening, because the corrupted pages would be visited with a high probability. Second, the number of disk access is rather small, because most corrupted pages have been prefetched in advance, which illustrates the *Scrubber* is efficient in leveraging workload idleness. Third, the high numbers of prefetched pages of *R_75* and *R_90* are attributed to the fact that they contain a high percentage of CRC-pages, which is also evidenced by the dominance of CRC_READ and CRC_MOVED in Table 3. It should be noted that the sum of *Disk Access*, *Prefetched* and *BCH Decoded* is not equal to *Corruptions*, since there could be corrupted pages that have not yet been prefetched or corrected.

5 Related Work

Flash-based SSDs have been extensively researched as a cache due to their widespread deployment in HDD-SSD hybrid storage systems. Kgil et al. [21] propose to partition the NAND flash cache space into read and write caches and employ a programmable flash memory controller to improve performance and reliability. They also utilize CRC within the cache device, but in a complementary way to reduce BCH's false positives, as opposed to our replacement of BCH for clean pages. Yang et al. [45] propose to improve SSD cache endurance via reducing media writes and erases. Koller et al. [22] present a study discussing write policies and consistency problems of SSD cache deployed in networked environment. More recently, Holland et al. [18] explore the design space of flash as a cache in the storage client side instead of server side and make several interesting findings. Albrecht et al. [4] present Janus, a cloud-scale distributed file system that is actively-used in Google Inc. In their paper, they formulate and solve an optimization problem to determine the flash cache allocation to workloads according to their respective *cacheability* and conclude that flash storage is a cost-effective complement to disks in data centers. These works all use flash-based SSD as a cache without taking into account synergistic optimizations. Our proposed *FlexECC* expands the design space from a new dimension and could be integrated into these systems to further improve the cache performance.

As flash technology scales, the reliability issue associated with increasing flash memory bit error rate and the required error correction code have specially received research interests. Mielke et al. [29] conduct a comprehensive study of bit error rate of MLC SSDs from different manufacturers. Grupp et al. [15] observe a trend of decreasing performance and reliability. Observing ECC is under-utilized most of the time, especially when SSDs are in their early usage stage, Pan et al. [33] propose to speed up writes and tolerate more defective cells by fully exploiting ECC's capability. Taking advantage of the retention time gap between specification and actual requirements, Liu el al. [25] propose to improve write performance and/or reduce ECC overhead by relaxing retention time. Wu et al. [44] propose to adaptively use different ECCs according to workloads to avoid consistently using strong ECC. Similarly, Cai et al. [6] suggest a technique called *Correct and Refresh* to avoid using strong ECC. Their idea is to periodically refresh charges in memory cells to reduce the dominant retention errors due to loss of charges. The prolonged ECC decoding latency problem associated with advanced ECC schemes in modern SSDs has recently been observed by Zhao [49]. In their work, they suggest effective methods to reduce the decoding latency of LDPC codes. While each of these works tries to make a preferential trade-off toward performance, reliability, or cost when designing ECC schemes for flash memory, our work differs from them in that *FlexECC* is cache-oriented and can safely get rid of ECC for clean flash pages.

Relaxing ECC for performance and/or energy purposes has also been explored in the memory systems. Yoon et al. [46, 47] suggest a two-tiered error protection mechanism for last-level cache. The tier-1 code is located with the protected cache line and only provides error detection, while the tier-2 code is stored on off-chip DRAM. In this scheme, the ECC consumes limited on-chip SRAM resource, but is able to provide arbitrarily strong tier-2 protection. Based on the observations that in low-power operating mode different cache lines exhibit different reliability characteristics, Alameldeen et al. [3] propose a variable-strength ECC scheme, in which cache lines having zero or single error are protected by fast and simple ECC, while cache lines having multiple errors are protected by stronger ECC.

The most relevant work to *FlexECC* is SSC [37] in

that SSC also proposes a cache-oriented SSD architecture and extends interfaces between applications and the device. However, the performance improvement of SSC mainly comes from the elimination of page migrations during garbage collection, while *FlexECC* benefits from reduced decoding latency. A potential shortcoming of SSC is that it might exhibit a high miss rate if the silently evicted pages are requested again in the future. By contrast, *FlexECC* only directs accesses to corrupted blocks to beneath storage system. Moreover, in *FlexECC*, every page read benefits from the reduced decoding latency. It should be interesting to quantitatively compare *FlexECC* with SSC, as planned in our future work.

6 Conclusions and Future Work

This paper presents *FlexECC*, a novel high-performance cache-oriented MLC SSD. It flexibly applies BCH or CRC to incoming page writes according to their storage requirement information which is conveyed down by an upper-level cache manager. We have given a theoretic analysis on the decoding latency of BCH and CRC and found the gap in their decoding latencies. Experimental results with a variety of workloads have shown that *FlexECC* is capable of improving the overall cache performance by an impressive extent and save the total amount of cleaning time, without compromising reliability and consistency. As part of the future work, we plan to further improve *FlexECC*'s cache performance by leveraging the space that is otherwise consumed by ECC redundant information as additional effective cache space. In addition, we also plan to investigate the energy savings by replacing off-the-shelf SSD caches with *FlexECC*, especially in a cloud environment. We believe in light of the trend of increasing flash memory RBER and widespread use of MLC SSD as caches, it could be significantly beneficial to deploy *FlexECC* in practical systems.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedbacks and constructive suggestions. This research is partially supported by the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1102605, CCF-1102624, and CNS-1218960, and the National Basic Research Program (973 Program) of China under Grant No.2011CB302305, the National Natural Science Foundation of China under Grant No. 61232004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Introduction to EMC VFCache. White paper. http://www.emc.com/collateral/hardware/white-papers/h10502-vfcache-intro-wp.pdf, February 2012.

[2] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., AND DAVIS, J. D. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX ATC* (2008).

[3] ALAMELDEEN, A. R., WAGNER, I., CHISHTI, Z., WU, W., WILKERSON, C., AND LU, S.-L. Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes. In *Proceedings of ISCA* (2011).

[4] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, C. E. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX ATC)* (2013).

[5] BADAM, A., AND PAI, V. S. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of NSDI* (2011).

[6] CAI, Y., YALCIN, G., MUTLU, O., HARATSCH, E. F., CRISTAL, A., UNSAL, O. S., AND MAI, K. Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime. In *Proceedings of the* 30^{th} *IEEE International Conference on Computer Design(ICCD)* (2012).

[7] CHIEN, A. A., AND KARAMCHETI, V. Moore's Law: The First Ending and a New Beginning. *IEEE Computer Magazine 46*, 12 (December 2013), 48–53.

[8] CHO, J., AND SUNG, W. Efficient Software-Based Encoding and Decoding of BCH Codes. *IEEE Transations on Computers 58*, 7 (July 2009), 878–889.

[9] CHOI, H., LIU, W., AND SUNG, W. VLSI Implementation of BCH Error Correction for Multilevel Cell NAND Flash Memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 18*, 5 (April 2010), 843–847.

[10] C.ZAMBELLI, M.INDACO, M.FABIANO, CARLO, S., P.PRINETTO, P.OLIVO, AND D.BERTOZZI. A Cross-Layer Approach for New Reliability-Performance Trade-Offs in MLC NAND Flash Memories. In *Proceedings of the Design Automation & Test in Europe Conference & Exhibition(DATE)* (2012).

[11] DEAL, E. Trends in NAND Flash Memory Error Correction. *Cyclic Design White Paper* (2009).

[12] DENG, Y., LU, L., ZOU, Q., HUANG, S., AND ZHOU, J. Modeling the Aging Process of Flash Storage by Leveraging Semantic I/O. *Future Generation Comp. Syst. 32* (March 2014), 338–344.

[13] DENG, Y., AND ZHOU, J. Architectures and Optimization Methods of Flash Memory Based Storage Systems. *Journal of Systems Architecture 57*, 2 (February 2011), 214–227.

[14] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., AND SWANSON, S. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of MICRO* (2009).

[15] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The Bleak Future of NAND Flash Memory. In *Proceedings of the* 10^{th} *USENIX Conference on File and Storage Technologies(FAST)* (2012).

[16] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX)* (2013).

[17] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving File System Reliability with I/O Shepherding. In *Proceedings of SOSP* (2007).

[18] HOLLAND, D. A., ANGELINO, E., WALD, G., AND SELTZER, M. I. Flash Caching on the Storage Client. In *Proceedings of the USENIX ATC* (2013).

[19] HUANG, P., WU, G., HE, X., AND XIAO, W. An Aggressive Worn-out Flash Block Management Scheme to Alleviate SSD Performance Degradation. In *Proceedings of Eurosys* (2014).

[20] HUANG, P., ZHOU, K., WANG, H., AND LI, C. BVSSD: Build Built-in Versioning Flash-based Solid State Drives. In *Proceedings of SYSTOR* (2012).

[21] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND Flash Based Disk Caches. In *Proceedings of the* 35^{th} *Annual International Symposium on Computer Architecture(ISCA)* (2008).

[22] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write Policies for Host-side Flash Caches. In *Proceedings of the* 11^{th} *USENIX Conference on File and Storage Technologies (FAST)* (2013).

[23] LEE, E., BAHN, H., , AND NOH, S. H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of FAST* (2013).

[24] LIN, S., AND COSTELLO, D. J. *Error Control Coding(2nd Edition)*. Prentice Hall,Inc., 2004.

[25] LIU, R.-S., YANG, C.-L., AND WU, W. Optimizing NAND Flash-Based SSDs via Retention Relaxation. In *Proceedings of FAST* (2012).

[26] LU, Y., SHU, J., AND ZHENG, W. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of FAST* (2013).

[27] LUO, T., MA, S., LEE, R., ZHANG, X., LIU, D., AND ZHOU, L. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *Proceedings of the* 22^{nd} *International Conference on Parallel Architectures and Compilation Techniques(PACT)* (2013).

[28] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated Storage Services. In *Proceedings of the* 23^{rd} *ACM Symposium on Operating Systems Principles(SOSP)* (2011).

[29] MIELKE, N., MARQUART, T., WU, N., KESSENICH, J., , BELGAL, H., SCHARES, E., AND TRIVEDI, F. Bit Error Rate in NAND Flash Memories. In *Proceedings of IEEE International Reliability Physics Symposium(IRPS)* (2008).

[30] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the* 4^{th} *ACM European conference on Computer systems(Eurosys)* (2009).

[31] OPREA, A., AND JUELS, A. A Clean-Slate Look at Disk Scrubbing. In *Proceedings of the* 8^{th} *USENIX conference on File and storage technologies(FAST)* (2010).

[32] PAN, Y., DONG, G., WU, Q., AND ZHANG, T. Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications. In *Proceedings of the* 18^{th} *IEEE International Symposium on High Performance Computer Architecture(HPCA)* (2012).

[33] PAN, Y., DONG, G., AND ZHANG, T. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *Proceedings of the* 9^{th} *USENIX Conference on File and Stroage Technologies(FAST)* (2011).

[34] PARK, S., AND SHEN, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of FAST* (2012).

[35] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In *Proceedings of ISCA* (2010).

[36] REN, J., AND YANG, Q. I-CASH: Intelligently Coupled Array of SSDs and HDDs. In *Proceedings of HPCA* (2011).

[37] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of Eurosys* (2012).

[38] SAXENA, M., ZHANG, Y., SWIFT, M. M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of FAST* (2013).

[39] SHEN, K., AND PARK, S. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX ATC)* (2013).

[40] STRUKOV, D. The Area and Latency Tradeoffs of Binary Bit-parallel BCH Decoders for Prospective Nanoelectronic Memories. In *Proceedings of Fortieth Asilomar Conference on Signals, Systems and Computers(ACSSC)* (2006).

[41] SUN, F., ROSE, K., AND ZHANG, T. On the Use of Strong BCH Codes for Improving Multilevel NAND Flash Memory Storage Capacity. In *Proceedings of the IEEE Workshop on Signal Processing Systems(SiPS): Design and Implementation* (2006).

[42] UNGUREANU, C., DEBNATH, B., RAGO, S., AND ARANYA, A. TBF: A Memory-Efficient Replacement Policy for Flash-based Caches. In *Proceedings of the IEEE* 29^{th} *International Conference on Data Engineering (ICDE)* (2013).

[43] WU, G., AND HE, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the* 10^{th} *USENIX conference on File and Storage Technologies(FAST)* (2012).

[44] WU, G., HE, X., XIE, N., AND ZHANG, T. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. In *Proceedings of the* 18^{th} *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems(MASCOTS)* (2010).

[45] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., SUNDARARAMAN, S., AND WOOD, R. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the* 6^{th} *Annual International Systems and Storage Conference(SYSTOR)* (2013).

[46] YOON, D. H., AND EREZ, M. Flexible Cache Error Protection using an ECC FIFO. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis(SC)* (2009).

[47] YOON, D. H., AND EREZ, M. Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture(ISCA)* (2009).

[48] ZHANG, Y., SOUNDARARAJAN, G., STORER, M. W., BAIRAVASUNDARAM, L. N., SUBBIAH, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Warming up Storage-Level Caches with Bonfire. In *Proceedings of FAST* (2013).

[49] ZHAO, K., ZHAO, W., SUN, H., ZHANG, T., ZHANG, X., AND ZHENG, N. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In *Proceedings of FAST* (2013).

Notes

[1] In context of cache, single bit detection capability of EDC can typically fulfill the purpose, so our analyzed speedup is conservative.

[2] We have investigated $T_m = \lambda T_a$ with varying λ ($\lambda > 1$) as well and have reached the similar conclusion.

[3] In reality, it is more common to use a block as the granularity. For simplicity, we assume page granularity.

[4] The characteristics are different from [23], because *Flashcache* has bypassed non-4KB and large (more than 128KB) sequential requests directly to HDD. All trace requests are 4KB in size.