Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime

Yu Cai[1], Gulay Yalcin[2], Onur Mutlu[1], Erich F. Haratsch[3], Adrian Cristal[2], Osman S. Unsal[2] and Ken Mai[1]
[1]DSSC, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
[2]Barcelona Supercomputing Center, C/Jordi Girona 29, Barcelona, Spain
[3]LSI Corporation, 1110 American Parkway NE, Allentown, PA
[1]{yucai, omutlu, kenmai}@ece.cmu.edu, [2]{gulay.yalcin, adrian.cristal, osman.unsal}@bsc.es, [3]erich.haratsch@lsi.com

***Abstract*— With the continued scaling of NAND flash and multi-level cell technology, flash-based storage has gained widespread use in systems ranging from mobile platforms to enterprise servers. However, the robustness of NAND flash cells is an increasing concern, especially at nanometer-regime process geometries. NAND flash memory bit error rate increases exponentially with the number of program/erase cycles. Stronger error correcting codes (ECC) can be used to tolerate higher error rates, but these have diminishing returns with increasing P/E cycles and can have prohibitively high power, area, and latency overheads. The goal of this paper is to develop new techniques that can tolerate high bit error rates without requiring prohibitively strong ECC. Our techniques, called Flash Correct-and-Refresh (FCR) exploit the observation that the dominant error source in NAND flash memory is retention errors, caused by flash cells losing charge over time. The key idea is to periodically read, correct, and reprogram (in-place) or remap the stored data before it accumulates more retention errors than can be corrected by simple ECC. Detailed simulations of a solid-state drive (SSD) storage system driven by measured experimental data from error characterization on real flash memory chips show that our techniques provide 46x average lifetime improvement on a variety of workloads at no additional hardware cost. We also find that our techniques achieve lifetime improvements that cannot feasibly be achieved with stronger ECC.**

Keywords-NAND Flash, reliability, error correction, multi-level cell (MLC)

I. INTRODUCTION

In the past decade, NAND flash memory has evolved from being used in only specialized consumer electronics (i.e., cell phones, digital cameras) to widespread use in the primary data storage systems of general-purpose computers, due to its high performance, large storage capacity, and non-volatility. This trend is primarily enabled by the steady per-bit cost reduction from manufacturing process technology scaling and the use of multi-level cell (MLC) technology. Thus, solid-state drives (SSDs) are now economically viable and have supplanted or enhanced spinning magnetic media in a number of high performance computing applications.

However, NAND flash based storage suffers from low endurance as each flash memory cell can tolerate only a limited number of program/erase (P/E) cycles. A 3x-nm (i.e., 30-39nm) generation MLC (2-bit per cell) NAND flash cell can be programmed only ~3k times [1]. Continued process scaling and storage of 3 or 4 bits per cell will likely further reduce the number of P/E cycles each cell can tolerate, resulting in even shorter lifetimes for NAND flash based storage systems, especially for write-intensive applications. Enterprise data storage systems typically require storage endurance capable of sustaining continuous 10 full disk writes per day for 3-5 years, which would require each flash cell to tolerate more than 50k P/E cycles, assuming ideal wear leveling algorithms and write amplification [2]. Thus, there is a significant gap between the available (~3k P/E cycles) and desired (>50k P/E cycles) endurance of flash cells.

One way to improve flash lifetime is to use stronger error correction codes (ECC) [3][4]. Stronger ECC detects and corrects *raw bit errors* that happen over the lifetime of a flash cell, thereby increasing the number of P/E cycles each cell can tolerate without exposing the raw bit errors to the user. Unfortunately, stronger ECC has two major shortcomings: (1) high implementation overhead and (2) diminishing returns on flash lifetime improvement. The latter is because the raw bit error rate increases exponentially with P/E cycles while ECC error correction capability increases less than linearly, as detailed in later sections. As such, techniques that tolerate raw bit errors in flash cells without relying on stronger ECC are desirable. In this paper, we present new techniques that achieve this and thereby allow us to utilize unreliable flash media in a reliable way at high numbers of P/E cycles.

To achieve the low bit error rate (BER) required by enterprise systems without strong ECC, the raw BER of flash memory must be greatly reduced. To this end, we propose methods to control and reduce the raw BER even when flash memory has already endured very high P/E cycles, which is far beyond current nominal endurance of flash. Our techniques are based on the following three empirical observations: (1) the total raw errors of flash memory consist of retention errors and erase/program/read errors [5][6]; (2) the most dominant errors in flash memory are retention errors [5][8]; and (3) retention errors increase with retention time and can be reduced by issuing periodic refresh operations [5][6]. Based on these three observations, we propose a suite of techniques called Flash Correct-and-Refresh (FCR): the key idea is to periodically read each page in flash memory, correct its errors using simple ECC, and either remap (copy/move) the page to a different location or re-program it in its original location by recharging the floating gates, before the page accumulates more errors than can be corrected with simple ECC.

We present three versions of FCR: (1) **remapping-based FCR**, which periodically and selectively corrects and remaps a page to control retention errors; (2) **hybrid reprogramming/remapping-based FCR**, which periodically and *selectively* corrects and re-programs data *in- place* and less frequently remaps the data to avoid errors that could accumulate due to continuous reprogramming; (3) **adaptive-rate FCR**, which adaptively adjusts the rate at which remapping/reprogramming is done based on the P/E cycles a page has already endured. These techniques can be implemented in device driver software or the firmware of the NAND flash controller (Figure 1), by having the controller issue flash data refresh requests. Since refresh requests can be initiated by software, leveraging the read, write, and remapping functionalities that already exist in flash-based SSDs, FCR does not require any additional hardware.

The major contributions of this paper are as follows:

1. We evaluate the relationship between raw bit error rate, strength of ECC, and lifetime of NAND flash memory. We show that increasing the strength of ECC provides diminishing returns on flash memory lifetime at significant additional cost.
2. Based on the observation that retention errors are the dominant errors in flash memory and can be corrected by periodically remapping each flash page to a different location, we propose our *remapping-based FCR* technique. We find that remapping-based FCR improves average flash memory lifetime by 9x, but performs considerable data movement and unnecessary refreshes, especially detrimental to read-intensive workloads.

3. To reduce the overhead of data movement caused by reprogramming, we propose *hybrid FCR*. The key idea is that a page can be refreshed by reprogramming it *in-place* instead of by remapping it to another location. This is based on the insight that retention errors are caused by charge loss and *cells with retention errors are reprogrammable without first erasing them*.
4. To further reduce unnecessary refreshes, we propose *adaptive-rate FCR*, which has a low refresh rate for a flash block when its retention error rate is low (i.e., early in its lifetime).
5. We evaluate all our techniques using real I/O workload traces and a high-fidelity simulation infrastructure that is driven by data obtained by performing error characterization on a real experimental flash platform. Our evaluations show that hybrid and adaptive-rate FCR respectively provide 31x and 46x lifetime improvement at only 5.5% and 1.5% energy overhead.

II. BACKGROUND

A modern flash-based solid-state drive (SSD) is organized as shown in Figure 1 [9]. It contains host interface logic to support a physical connection to the computer. A processing engine (SSD Controller) is required to process the requests from the host machine and schedule access to flash memory chips. Flash translation layer (FTL), including address mapping, wear leveling, and garbage collection, is implemented as firmware running on SSD controller to manage the SSD. A multiplexer emits commands and handles transport of data to the flash packages for each channel. A channel contains a set of flash chips. To overcome errors of noisy flash memory, each channel has its own error correction engine. The most widely used ECC algorithms for contemporary flash SSDs are binary BCH codes [11], which can correct multiple errors.

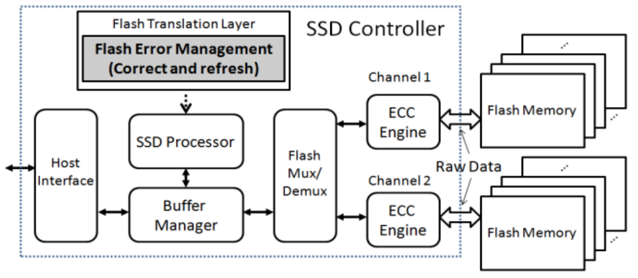


Figure 1. Flash-based SSD system architecture with proposed flash error management implemented in flash translation layer

III. ERRORS AND ECC ANALYSIS IN NAND FLASH

A. Errors in NAND Flash

NAND flash memory has three main operations: read, program (write), and erase. Various errors can occur during these operations; they are classified as read errors, program errors, and erase errors. In addition, while data is stored in flash memory, an already-programmed flash cell gradually loses charge from its floating gate, which can eventually alter the stored value, causing an error, called a retention error. Cai et al. used a real experimental test platform [10], to characterize errors that occur in 3x-nm MLC flash memory [5]. They observed the following main error characteristics: (1) All types of errors are highly correlated with the number of P/E cycles the flash cell endured. Raw BER increases exponentially as P/E cycles increase; (2) Retention errors are most dominant, and program errors are the second most dominant types of errors; (3) Retention error rates are highly correlated with retention time. These experimental findings are also supported by Tanakamaru et al. who report that retention error rate is >100x higher than program error rate [8].

In error analysis, *raw data* refers to the data read immediately out of NAND flash memory without any error correction. *Raw BER (RBER)* is the fraction of erroneous bits over all raw data bits read. After error correction, which happens in the SSD controller, the corrected data is sent to the host computer. However, error-corrected data might still contain some *uncorrectable* errors, as the ECC might not be strong enough to correct all errors in the raw data. The fraction of erroneous bits after error correction over all read data bits is called *uncorrectable BER (UBER)*.

B. ECC Power, Area, and BER Analysis

Generally, a given ECC can be described by the 3-tuple <n, k, t>, where n is the codeword length, k is the length of data that the code protects and t is the maximum number of errors that the ECC can correct. To select a suitable ECC for NAND flash memory, three factors need to be considered. First, the coding rate R = k/n, which determines the storage efficiency, should be as high as possible (e.g. >8/9 is typical for storage) to maintain high storage efficiency and low cost per bit. Second, the error correction capability t should be large enough to guarantee UBER to be below 10^{-15}, which is a common reliability requirement in data storage. Third, the code length n should be a factor of flash page length.

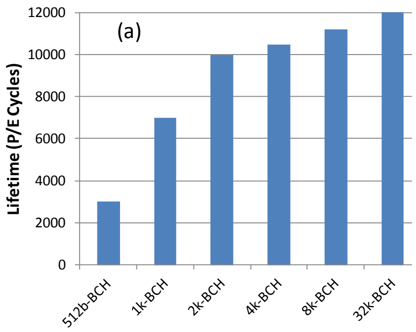
Table 1. Characterization of various error correction codes

Code length(n)	Correctable Errors (t)	Acceptable Raw BER	Norm. Power	Norm. Area
512	7	1.0×10^{-4} (1x)	1	1
1024	12	4.0×10^{-4} (4x)	2	2.1
2048	22	1.0×10^{-3} (10x)	4.1	3.9
4096	40	1.7×10^{-3} (17x)	8.6	10.3
8192	74	2.2×10^{-3} (22x)	17.8	21.3
32768	259	2.6×10^{-3} (26x)	71	85

Table 1 summarizes the characteristics of a series of applicable BCH codes, which are widely used for flash-based storage with various code lengths but all with the same coding rate. We keep the coding rate constant as it critically determines cost per bit. We list the acceptable raw BER each code provides to achieve an UBER of 10^{-15}, which would satisfy common storage industry reliability requirements, as well as the normalized power and area consumption to implement the required circuitry for each code, which all depend on the relative strength of each code. To evaluate power and area consumption, we synthesized the RTL design of each code with industrial 65nm bulk CMOS standard cell and memory libraries using the Synopsys Design Compiler. The power and area are normalized to the baseline 512-bit BCH code that consumes 1.76 mW and occupies 0.013 mm^2. Table 1 shows that as the code length of ECC increases given the same coding rate, the acceptable raw BER increases. However, the increase in power and area consumption is much larger than the increase in acceptable raw BER, especially for the strongest codes. This is because the implementation complexity is highly correlated with the code length (n) and the number of errors (t) that can be corrected. To achieve an endurance of a few thousand P/E cycles, up to 40 bit errors are expected to be corrected per 1k byte data for 2x-nm MLC flash [7]. Based on this analysis, we conclude that the power and area overheads of a strong ECC that can tolerate an acceptable raw bit error rate are likely to be prohibitive. As such, techniques that can tolerate high raw bit error rates without requiring strong ECC are very desirable.

C. P/E Cycle Lifetime Analysis

We define the lifetime of flash block as the maximum number of P/E cycles after which the ECC in the SSD controller can no longer guarantee the commonly required storage reliability (less than 10^{-15} UBER) within a certain guaranteed data storage time. The guaranteed storage time indicates how long the data is expected to be present in flash memory. We first analyze the effect of stronger ECC on the lifetime of flash memory for a fixed guaranteed storage time, and next provide insights into the results by characterizing raw bit error rate of flash memory at different lifetimes. To gather the data we present, we have used an FPGA-based experimental flash testing platform [10] to test a large number of 3x-nm NAND flash memory chips [5].



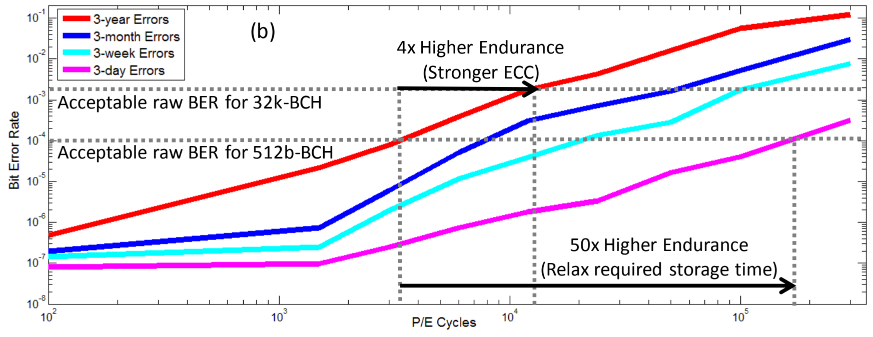


Figure 2. (a) The number of program/erase cycles that can guarantee 10^{-15} UBER given 3-year storage time with various error correction codes; (b) Raw BER for all errors at different number of P/E cycles; each line shows the raw BER observed after different required data storage times (3 days/weeks/months/years)

Figure 2(a) shows the maximum lifetime (in P/E cycles) different-strength BCH codes provide for the 3x-nm flash chips examined, assuming a guaranteed storage time of 3 years. Recall that lifetime indicates the maximum number of P/E cycles that are achievable to guarantee a UBER of $<10^{-15}$. We observe that increasing the strength (code length) of BCH codes results in diminishing returns in lifetime. To provide insight into why stronger ECC provides diminishing returns in number of P/E cycles, we show how the raw bit error rate changes with changes in P/E cycles in Figure 2(b). This figure plots the relationship between raw bit error rate and P/E cycles for different guaranteed/required data storage times. The raw BER data is obtained by writing data to the entire flash memory chip and observing the bit errors after the required data storage times [5]. The key observation is that as P/E cycles increase, the raw bit error rate observed in flash memory increases exponentially. In contrast, the error correction capability of BCH codes increases much more slowly (usually less than linearly) with increased code length, as shown in the *Acceptable Raw BER* column of Table I. Thus, increasing strength of BCH codes would provide diminishing returns.

As a concrete example, when we increase the code length from 512 bits to 32k bits, the acceptable raw BER increases by ~26x (see Table I), but the provided lifetime increases by only 4x (see Figure 2 (b)), under a 3-year data storage time requirement. The 4x increase in lifetime is much lower than the ~17x increase required by enterprise applications to achieve ~50k P/E cycles (not shown). Furthermore, increasing code length from 512 to 32k bits comes at 71x the power consumption and 85x the area overhead, which is likely prohibitive.

We also observe from Figure 2 (b) that: if the ECC strength, and thus the acceptable raw BER, is fixed, we can achieve a larger lifetime by relaxing the required storage time. For example, if the required storage time is 3 years, the lifetime provided by a 512-bit BCH code is only ~3k P/E cycles. On the other hand, if the required storage time is 3 days, the lifetime provided by the same code is ~150k P/E cycles. Hence, if we would like to achieve a lifetime of ~150k P/E cycles while still having the ability to store data for 3 years (or longer), we can relax the data storage time to 3 days, and *refresh* each flash cell every 3 days to retain the data integrity. Our techniques we described below are based on this basic observation, which we validate experimentally with extensive evaluations.

IV. PROPOSED TECHNIQUES

We propose a set of new techniques called Flash Correct-and-Refresh (FCR) that exploit the dominance and characteristics of retention errors to significantly increase NAND flash lifetime while incurring minimal overheads. The basic idea of the FCR schemes is to periodically read, correct, and refresh (i.e., reprogram or remap) the stored data before it accumulates more retention errors than can be handled by ECC. Thus, we can achieve a low UBER while still using a simple, low-overhead ECC. Two key questions central to designing a system that uses FCR techniques are: (1) *how* to refresh the data in flash memory? and (2) *when* to refresh the data? We address the first question with two techniques for how to refresh the data: remapping (Section IV.A) and reprogramming in-place (Section IV.B). We then tackle the second question with two techniques for when to refresh: periodically and adaptively based on the number of P/E cycles (Section IV.C).

A. Remapping Based FCR Mechanism

Unlike DRAM cells, which can be refreshed in-place, flash cells generally must first be erased before they can be programmed. To remove the slow erase operation from the critical path of write operations, current wear leveling algorithms remap the data to another physical location rather than erasing the data and then programming in-place. The flash controller maintains a list of free blocks that have been erased in background through garbage collection and are ready for programming. Whenever a write operation is requested, the controller's wear leveling algorithm selects a free block and programs it directly, remapping the logical block address to the new physical block.

The key idea of remapping based FCR is to leverage the existing wear-leveling mechanisms to periodically read, correct, and remap to a different physical location each valid flash block in order to prevent it from accumulating too many retention errors. Figure 3 shows the operational flow of remapping-based FCR: (1) During each refresh interval, a block with valid data that needs to be refreshed is selected; (2) The valid data in the selected block is read out page by page and moved to the SSD controller; (3) The ECC engine in the SSD controller corrects all the errors in the read data, including retention errors that have accumulated since the last refresh. After ECC, the data are error free; (4) A new free block is selected and the error free data are programmed to the new location, and the logical address is remapped. Note that the proposed address remapping techniques leverage existing hardware and software of contemporary wear leveling and garbage collection algorithms.

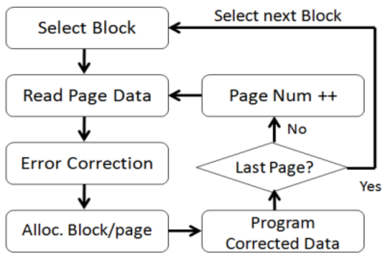


Figure 3. Operation of a remapping-based flash data refresh scheme.

Error Rate Model. We now provide an intuitive model for the benefits of remapping based FCR. As discussed in Section III.A, flash memory errors can be classified as retention, program, read, and erase errors. Given a remapping-based refresh period T and the number of refresh periods elapsed during the required storage time N, the total error count with remapping based FCR can be modeled as:

$$E_{total} = E_{retention}(T) + E_{program} + E_{read} + E_{erase}$$

Here the number of retention errors is determined by the refresh period T, as refresh operations can eliminate retention errors that accumulate since the last refresh. On the other hand, if the data are not periodically corrected and remapped, retention errors accumulate throughout the data storage time, and the total error count can be modeled as:

$$E_{total} = E_{retention}(N \times T) + E_{program} + E_{read} + E_{erase}$$

Here, the retention error count correlates with the total storage time. Since retention errors increase almost linearly with retention time and the retention errors are the most dominant errors, the overall error rate ratio with remapping based FCR as opposed to without is:

$$\frac{E_{refresh}}{E_{norefresh}} \approx \frac{E_{retention}(T)}{E_{retention}(N \times T)} \approx \frac{1}{N}$$

Thus, overall error rate can be almost linearly decreased by increasing the refresh rate (i.e., the number of refresh intervals, N). The benefits of remapping based FCR can be leveraged in two ways: (1) Given a fixed P/E cycle lifetime requirement, ECC on the flash controller can be greatly simplified. (2) Given a fixed ECC on the controller, FCR can increase the maximum number of P/E cycles that flash memory can tolerate to achieve a particular storage reliability requirement.

Unfortunately, periodic remapping of every block introduces additional erase cycles. This is because after the flash data are corrected and remapped to the new location, the original block is marked as outdated. Thus, the block will eventually be erased and reclaimed by garbage collection. The more frequent the remap operations, the more the additional erase operations, which wears out flash memory faster. As such, there might be an inflection point beyond which increasing the refresh rate in remapping-based FCR can lead to reduced lifetime. To avoid this potential problem, we next introduce enhanced FCR methods, which minimize unnecessary remap operations.

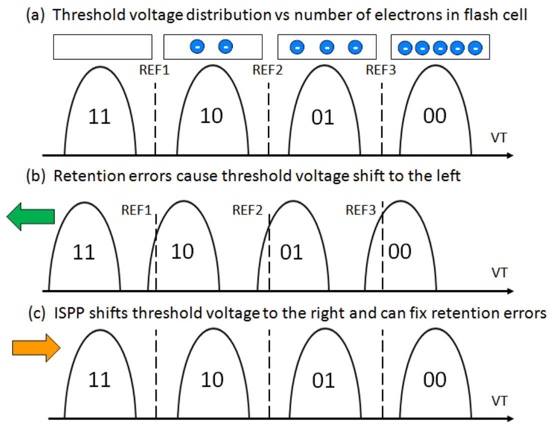
B. *In-Place Reprogramming Based FCR Mechanisms*

To reduce the overhead associated with periodic remapping, we propose a technique for periodic *in-place reprogramming* of the block most of the time, without a preceding erase operation, which can greatly reduce the overhead of periodic remapping. This in-place reprogramming takes advantage of the key observation that retention errors arise from the loss of electrons on the floating gate over time and *the flash cell with retention errors can be reprogrammed to its original correct value without an erase operation* using the incremental step pulse programming (ISPP) scheme used to program flash memory. We first provide background on ISPP.

ISPP. Before a flash cell can be programmed, the cell must be erased (i.e., all charge is removed from the floating gate, setting the threshold voltage to the lowest value). When a NAND flash memory cell is programmed, a high positive voltage applied to the control gate causes electrons to be injected into the floating gate. The threshold voltage of a NAND flash cell is programmed by injecting a precise amount of charge onto the floating gate through ISPP [21]. During ISPP, floating gates are programmed iteratively using a step-by-step program-and-verify approach. After each programming step, the flash cell threshold voltage is boosted up. Then, the threshold voltage of the programmed cells are sensed and compared to the target values. If the cell's threshold voltage level is higher than the target value, the program-and-verify iteration will stop. Otherwise the flash cells are programmed once again and more electrons are added to the floating gates to boost the threshold voltage. This program-and-verify cycle continues iteratively until all the cells' threshold voltages reach the target values. Using ISPP programming, flash memory cells can only be programmed from a state with fewer electrons to a state with more electrons and cannot be programmed in the opposite direction.

Retention Error Mechanisms. Retention errors are caused by the loss of electrons from the floating gate overtime. As such, a cell with retention errors moves from a state with more electrons to a state with fewer electrons. Figure 4(a) shows the relative relationship between the stored data value and its corresponding threshold voltage distribution for a typical MLC flash storing 2-bits per cell. The leftmost state is the erased state (i.e. state 11) with the smallest threshold voltage and there is no charge on the floating gate. The states located on the right in Figure 4(a) are programmed with more electrons and have higher threshold voltages than the states located relatively to the left. Over time, as the electrons on the floating gate leak away, the threshold voltage of a cell shifts to the left, as shown in Figure 4(b). If the threshold voltage of a cell shifts too far to the left (i.e., it loses too many electrons from the floating gate), it will cross the read reference voltage between adjacent states and can be misinterpreted during a read as the wrong value.

In-Place Reprogramming can Fix Retention Errors. A cell with a retention error can be reprogrammed to the value it had before the floating gate lost charge by re-charging additional electrons onto the floating gate through ISPP, as shown in Figure 4(c). Note that this does not require an erase operation because the only objective is to add more electrons (not to remove them), which can be accomplished by simple programming.



(a) Threshold voltage distribution vs number of electrons in flash cell
REF1
REF2
REF3
11
10
01
00
VT
(b) Retention errors cause threshold voltage shift to the left
REF1
REF2
REF3
11
10
01
00
VT
(c) ISPP shifts threshold voltage to the right and can fix retention errors
11
10
01
00
VT

Figure 4. Retention errors are caused by threshold voltage shift to the left and can be fixed by programming in-place using ISPP.

Basic In-Place Reprogramming Based FCR Mechanism. A basic FCR mechanism that uses in-place reprogramming works as follows. Periodically, a block is selected to be refreshed and read page by page into the flash controller. By selecting a suitable refresh interval, we can ensure that the total error number is below the correction capability of the ECC. Then we can re-program the flash cells in the same location with the error-corrected data, without erasing the whole block. If the new corrected value corresponds to a state with more charge than the old value, then the cell can be in-place reprogrammed to the correct value. If the corrected value is exactly the same as the original value, in-place reprogramming will not change the stored data value, as ISPP will stop programming the cell as soon as it detects that the target value has already been reached. Note that most of the cells are reprogrammed with exactly the same data value as error rates are generally significantly below 1%.

Problem: Accumulated Program Errors. While this basic mechanism can effectively fix retention errors, it introduces a problem because there is another error mechanism in flash cells that is caused by program operations, which are required to perform in-place reprogramming. When a flash cell is being programmed, additional electrons may be injected into the floating gates of its neighbor cells due to coupling capacitance [13]. The threshold voltage distribution of the neighbor cells will shift *right* as they gain more electrons, as shown in Figure 5(a). If the threshold voltage shifts right by too much, it will be misread as an error value that represents a state located to the right. This is called a program interference error (or simply a program error). Although it is a less common error mechanism than retention errors, periodic reprogramming can exacerbate the effects of program errors.

Two potential issues are: (1) As ISPP cannot remove electrons from the floating gate, program errors cannot be fixed by in-place reprogramming; (2) Reprogramming of a page can introduce additional program errors due to the additional program operations. Figure 5(b) illustrates both issues in the context of in-place programming. First, the original data is programmed into the page. This initial programming can cause some program errors (e.g., value 11 is programmed as 10 on the second cell from the left). After some time, retention errors start to appear in the stored data (e.g., first cell changes from state 00 to 01). Note that there are generally many more retention errors than the program errors. When the page is reprogrammed in-place, it is first read out and corrected using ECC. The error-corrected data (which is the same as the original data) is then written back (programmed) into the page. This corrects all the retention errors by recharging the cells that lost charge. However, this reprogramming does not correct the program error (in the second cell) because this correction requires the removal of charge from the second cell's floating gate, which is not possible without an erase operation. Furthermore, additional program errors can appear (e.g., in the sixth cell) because the in-place program operation can cause additional disturbance.

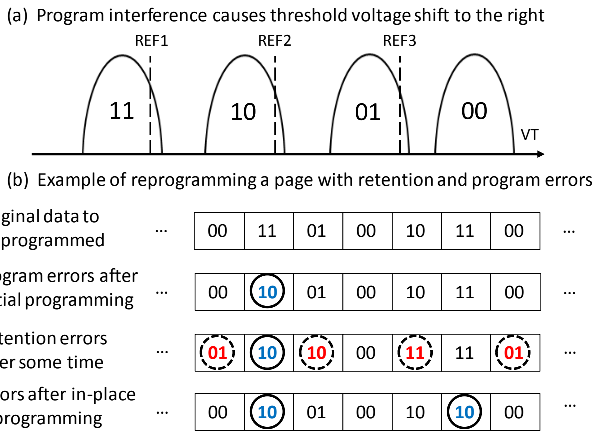

(a) Program interference causes threshold voltage shift to the right
REF1
REF2
REF3
11
10
01
00
VT
(b) Example of reprogramming a page with retention and program errors
Original data to be programmed
Program errors after initial programming
Retention errors after some time
Errors after in-place reprogramming
... 00 11 01 00 10 11 00 ...
... 00 10 01 00 10 11 00 ...
... 01 10 10 00 11 11 01 ...
... 00 10 01 00 10 10 00 ...

Figure 5. In-place reprograming can correct retention errors but not program errors because in-place programming can only add more electrons into the floating gate and cannot remove them. Note that red values with dotted circles are retention errors and blue ones with solid circles are program errors.

Error Rate Model. Given a reprogramming period T and the number of refresh periods during the required storage time N, the total error count with the basic in-place reprogramming can be modeled as:

$$E_{total}^{reprogram} = E_{retention}(T) + N \times E_{reprogram} + E_{program} + E_{read} + E_{erase}$$

As the number of refresh operations increases, the program errors due to reprogramming of the same block over and over accumulate. This is because program interference errors are mainly right shift errors (unlike retention errors), and cannot be corrected by in-place reprogramming. The program error rate increases linearly with the number of reprogram operations. For relatively small N, the accumulated error count introduced by reprogramming is still much smaller than retention error rate, and periodic reprogramming can still greatly reduce the raw BER. So, only in cases of where reprogramming frequency is high (large N), will the programming error rate become comparable to that of the retention errors. If the program error counts reach the error correction capability of ECC, it is highly probable that the data read in the next refresh interval can no longer be recovered as the sum of accumulated program errors and newly produced retention errors may exceed the error correction capability of ECC.

Hybrid FCR. To mitigate the errors accumulated due to periodic reprogramming, we propose a hybrid reprogramming/remapping based FCR technique to control the number of reprogram errors. The key idea is to monitor the right-shift error count present in each block. If this count is below a certain threshold (likely most of the time) then in-place reprogramming is used to correct retention errors. If the count exceeds the threshold, indicating that the block has too many accumulated program errors, then the block is remapped to another location, which corrects both retention and program errors. In our evaluation, we set the threshold to 30% of the maximum number of errors that could be corrected by ECC, which is conservative. Figure 6 provides a flowchart of this hybrid FCR mechanism. Note that this hybrid FCR mechanism greatly reduces the additional erase operations present in remapping based FCR because it remaps a block (i.e., requires an erase operation) only when the number of accumulated reprogram errors is high, which is rare due to the low program error rate.

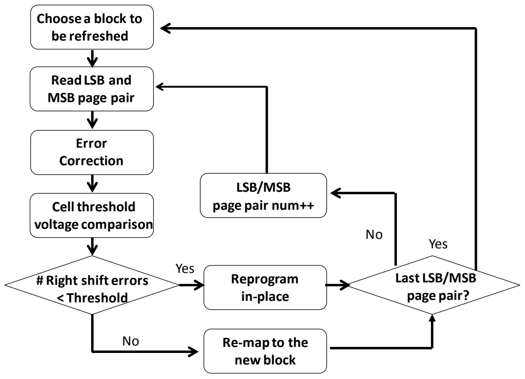

Choose a block to be refreshed
Read LSB and MSB page pair
Error Correction
Cell threshold voltage comparison
# Right shift errors < Threshold
Yes
No
Reprogram in-place
Re-map to the new block
Last LSB/MSB page pair?
Yes
No
LSB/MSB page pair num++

Figure 6. Hybrid FCR workflow: if re-program error count is less than a threshold, in-place reprogram the block; otherwise, remap to a new block

C. *Adaptive-Rate FCR*

So far we assumed that FCR mechanisms, be it based on in-place reprogramming or remapping, are invoked periodically. However, this need not be the case. In fact, we observe that the rate of (retention) errors is very low during the beginning of flash lifetime, as shown in Figure 2(b) and as also observed by others [5]. Until more than 1000 P/E cycles, the retention error rate is lower than the acceptable raw BER that can be corrected by the simplest BCH code (Figure 2 (b)). Hence, at the beginning of its lifetime, flash memory does not need to be refreshed. Retention error rate increases as the number of P/E cycles increases. We leverage this key observation to reduce the number of unnecessary refresh operations.

The main idea of adaptive-rate FCR is to adapt the refresh rate to the number of P/E cycles a block has incurred. Initially, refresh rate for a block starts out at zero (no refresh). Once ECC becomes incapable of correcting retention errors, the block's refresh rate increases to tolerate the increased retention error rate. Hence, refresh rate is gradually increased over each flash block's lifetime to adapt to the increased P/E cycles. The whole lifetime of a flash block can be divided into intervals with different refresh rates ranging, for example, from no refresh (initially), yearly refresh, monthly refresh, weekly refresh, to daily refresh. The frequency of refresh operations at a given P/E cycle count is determined by the acceptable raw BER provided by the used ECC and the BER that corresponds to the P/E cycle count, as shown in Figure 2(b). Note that this mechanism requires keeping track of P/E cycles incurred for each block, but this information is already maintained to implement current wear-leveling algorithms.

D. *Additional Considerations for FCR*

Implementation cost. The FCR mechanisms do not require hardware changes. They require changes in FTL software/firmware to implement the flowcharts shown in Figure 3 and Figure 6. FCR can leverage the per-block validity and P/E cycle information that is already maintained in existing flash systems to implement wear leveling.

Power supply continuity. To perform a refresh, the flash memory must be powered. As FCR is proposed for enterprise storage applications, these systems are typically continuously powered on. Our proposed techniques use daily, weekly or monthly refresh and it is rare for a server to be powered off for such long periods.

Response time impact. Refresh may interfere with normal flash operations and degrade the response time. To reduce this penalty, we can decrease the refresh priority making it run in background. SSD can issue refresh operations whenever it is idle, and refresh operations can be interrupted to avoid the impact on the response time of normal operations. Unlike DRAM, where refresh is triggered frequently (e.g., every 64ms), the refresh period of FCR is at least a

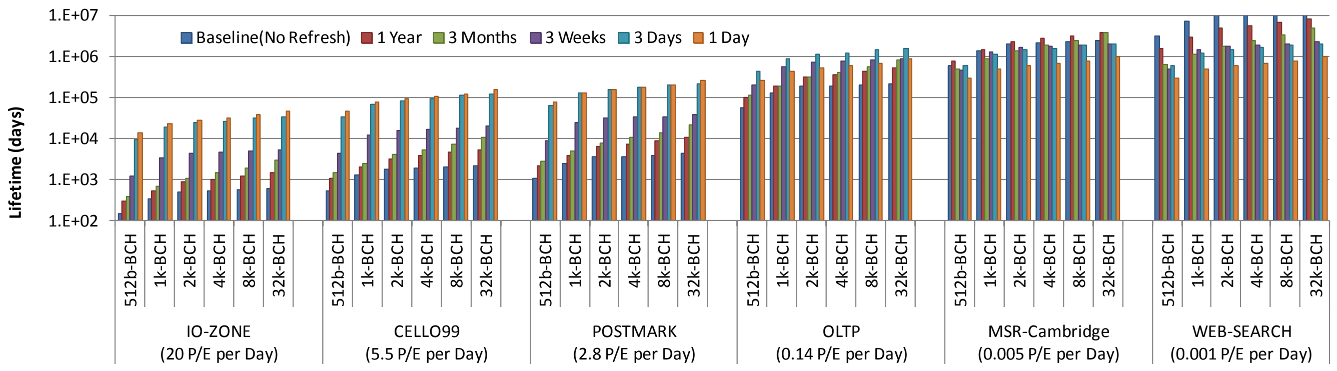


Figure 7. Flash lifetime (in days) provided by remapping based FCR. The baseline represents the lifetime without any refresh. Y-axis is in log scale.

day, and the SSD can finish refresh operations within the refresh period. Recent work has shown that the response time overhead is within a few percent for daily refresh [14]. Note that our hybrid and adaptive FCR techniques have much lower overhead for refresh operations than periodic remapping based FCR.

Additional erase cycles. FCR introduces additional erase operations. We will evaluate the impact of additional erase operations on effective lifetime and energy in Section VI.

Adapting to variations in retention error rate. Note that retention error rate is usually constant for a given refresh rate and P/E cycle combination. However, there are environmental factors, such as temperature, that can change this rate. For example, retention error rate would be dependent on temperature. To adapt to dynamic fluctuations in retention error rate, our hybrid FCR and adaptive-rate FCR mechanisms monitor the changes in the retention error rate at periodic intervals, and increase/decrease the refresh (i.e. FCR) rate if the error rate in the previous interval is greater/less than a threshold. These mechanisms are similar in principle to what is employed in DRAM to adapt refresh rate to temperature changes [22].

V. Evaluation Methodology

We use Disksim [15] with SSD extensions [9] to quantitatively evaluate the proposed techniques. All proposed techniques are simulated using various real workload traces: *iozone* [16], *cello99* [20], *oltp, postmark* [17], *MSR-Cambridge* [19] and a *web search engine* [18]. These workload traces are chosen to cover diverse read ratios out of all SSD operations, which are 0%, 38%, 52%, 83%, 80% and 99% respectively. *Iozone* and postmark are file system benchmarks. *oltp* is a database application running on a financial institution. *web search* contains the I/O traces from a popular engine. *Cello99* contains a set of traces taken from cello [20] over the period January 14 and December 31 in 1999. *MSR-Cambridge* contains 1-week-long block I/O traces of servers at Microsoft Research Cambridge.

We configure the simulated flash-based SSD with four channels. Each channel has 8 flash chips. Each flash chip has 8192 blocks containing 128 pages. The page size is 8KB. The total storage capacity is 256GB. We select a group of candidate ECCs for NAND flash memory controller as listed in Table 1. The energy of flash read, program, and erase operations are collected from an experimental flash memory platform [10], and are used in the simulation infrastructure to obtain the overall energy consumption. To evaluate flash lifetime provided by a given error correction and FCR mechanism, we first obtain the maximum number of P/E cycles per block based on the experimental data we obtain on the relationship between P/E cycles and acceptable raw BER of a given ECC mechanism, as shown in Figure 2(b). We then obtain the P/E cycles of each block after simulating a trace to completion. Dividing the simulated P/E cycles for the trace with the maximum possible P/E cycles provides us with the fraction of maximum possible flash lifetime (P/E cycles) that is exhausted by the trace. Since we know the real length of the trace (e.g., *cello99*'s length is 11.5 months), we know how long in real time the trace runs to exhaust that fraction of maximum lifetime. We use linear extrapolation to determine how long the trace would have been to exhaust the full maximum lifetime. Note that we use this methodology since it was impossible to obtain or simulate multiple-year-long traces that can actually exhaust the lifetime of flash memories we evaluate.

VI. Results and Analysis

A. Flash Lifetime with Remapping-Based FCR

Figure 7 compares the lifetime provided by remapping-based FCR (with refresh interval ranging from daily to yearly) to the baseline with no refresh. Flash lifetime is evaluated under various ECC configurations (ranging from weak 512b to strong 32k-bit BCH codes) with five refresh periods for all workloads. All P/E cycle overheads introduced by remapping are taken into account in these evaluations.

First, given the same workload and the same refresh interval (or no-refresh), stronger ECC always provides a longer lifetime than weaker ECC. For example, given a refresh interval of 3 days and the IO-zone trace, lifetime can be increased by 3.5 times if 32k-bit BCH codes are used instead of 512b codes. This is because strong ECC can tolerate high raw BER at high P/E cycles and thus improves lifetime.

Second, remapping based FCR provides significant lifetime improvements, especially for write-intensive applications, such as io-zone and cello99. Using FCR in conjunction with weak ECC provides higher lifetime than using strong ECC alone (with no refresh). For example, flash lifetime with 512b BCH codes and daily refresh is 24 times longer than with 32kb-BCH but no refresh, for the iozone workload. We conclude that using FCR can avoid the need for implementing high-cost strong ECC techniques while providing much higher lifetimes than such techniques.

Analysis. Refresh period impacts lifetime in different ways depending on the characteristics of the workloads. For write-intensive applications (i.e. iozone, cell099 and postmark), lifetime always increases as refresh period decreases. These workloads require a high number of P/E cycles due to frequent writes and a short refresh interval increases the maximum P/E cycles each flash block can tolerate. For read-write-balanced workloads, such as oltp, which has an approximately equal frequency of read and write operations from the host computer to SSD, lifetime first increases with reduced refresh period until a point (3-day refresh period). Lifetime decreases with reduced refresh period after that point, indicating that the additional P/E cycles introduced due to remapping outweighs the P/E cycle increase due to refresh. For example, for the oltp workload, the additional P/E operations increase from ~2x to ~7x when the refresh period changes from 3 days to daily, while the maximum P/E cycles a flash block can tolerate only increases by 50%. For heavily read-intensive applications (i.e., web search, MSR-Cambridge), remapping-based FCR reduces lifetime. This is because these applications do not require a high number of P/E cycles to begin with so there is little benefit to FCR. The additional erase cycles introduced by remapping leads to decreased lifetime even with a large refresh period of one year.

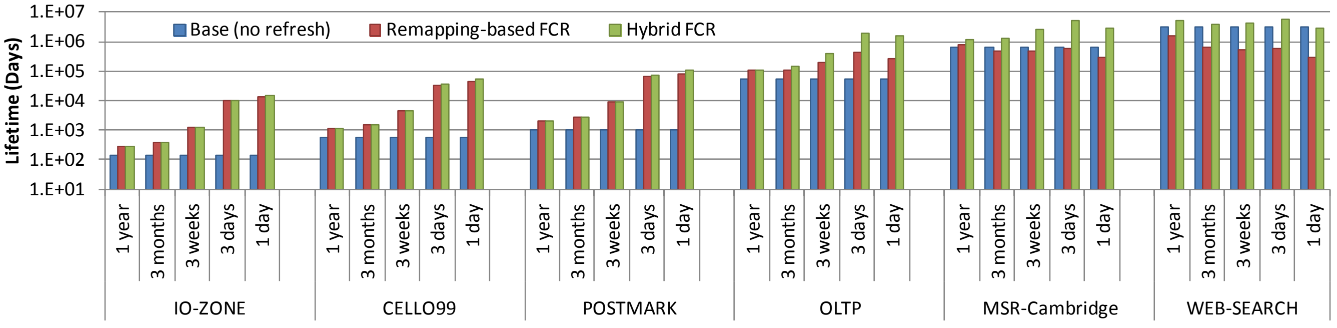


Figure 8. Flash lifetime (in days) provided by remapping based FCR versus hybrid FCR. Y-axis is in log scale.

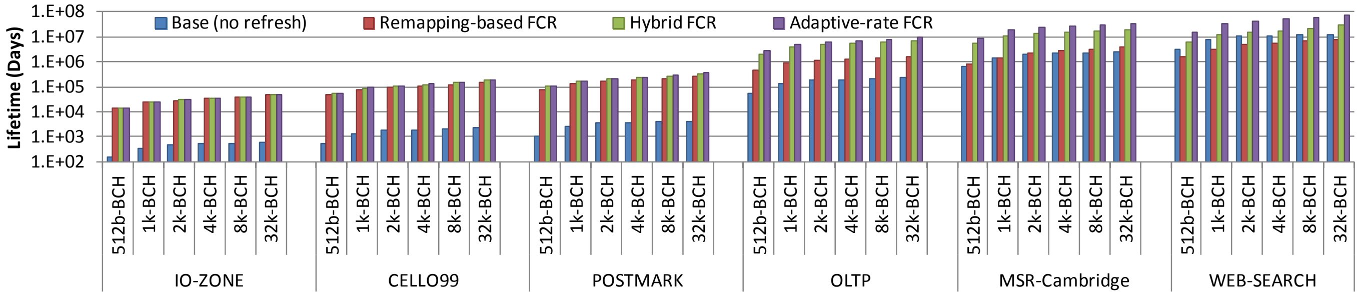


Figure 9. Lifetime comparison of no refresh, remapping-based FCR, hybrid FCR and adaptive-rate FCR. Remapping-based FCR and hybrid FCR use the refresh interval that provides the highest lifetime for each trace. Y-axis is in log scale.

B. *Flash Lifetime with Hybrid FCR*

Figure 8 compares the lifetime of hybrid FCR and remapping-based FCR at various refresh intervals, assuming both are implemented over a baseline with 512b BCH codes. Hybrid FCR remaps the block if the program errors affect more than 30% of the number of bits that ECC can correct. We make several major observations. First, hybrid FCR *always* improves lifetime compared to both remapping-based FCR and the baseline with no refresh for all workloads. This is because hybrid refresh greatly reduces the erase cycles by largely replacing the remap operations with in-place reprogramming operations. On average, hybrid FCR provides 3x higher lifetime than remapping based FCR and 31x higher lifetime than the baseline with no refresh. Second, hybrid FCR provides more lifetime benefit over remapping-based FCR for read-intensive applications than for write-intensive applications. Although hybrid FCR can greatly reduce additional P/E cycles, the relative P/E cycle overhead decrease in write-intensive workloads is much smaller, as these workloads have a very high number of P/E cycles to begin with. Third, hybrid FCR improves lifetime of read-intensive applications even when refreshes are infrequent (e.g., yearly) by greatly reducing the need to remap data. For balanced workloads, such as MSR-Cambridge, hybrid FCR increases lifetime over remapping-based refresh especially with short refresh periods. Note that remapping based FCR decreases lifetime for both web search and MSR-Cambridge workloads. Hybrid FCR fixes this problem and improves lifetime on all workloads. We conclude that hybrid FCR is a superior technique than remapping based FCR.

C. *Flash Lifetime with Adaptive-Rate FCR*

Figure 9 compares the lifetime improvement of adaptive-rate FCR (implemented over the hybrid FCR mechanism) to periodic remapping-based FCR and periodic hybrid FCR. The refresh period of each periodic mechanism is chosen on a per-workload basis such that the lifetime provided for a workload by the mechanism is maximized. Adaptive-rate FCR improves lifetime over both periodic FCR mechanisms for all workloads as it avoids unnecessary refreshes. The improvements are especially significant in read-intensive workloads since these workloads do not have high P/E cycles, causing the adaptive-rate FCR to keep the refresh rate very low. On average, adaptive-rate FCR provides 46.7x, 4.8x, and 1.5x higher flash lifetime compared to no-refresh, remapping-based FCR, and hybrid FCR, respectively. We conclude that adaptive-rate FCR implemented over the hybrid FCR mechanism is a promising mechanism for significant and consistent lifetime enhancement of flash memory.

D. *P/E and Energy Overheads*

FCR techniques can introduce two main overheads: (1) additional P/E cycles due to remapping, which might outweigh the increase in lifetime due to reduced retention errors; (2) additional energy consumed by refresh operations. Figure 10 shows the ratio of additional P/E cycles due to remapping-based and hybrid FCR over the number of P/E operations intrinsic to each workload. First, the P/E cycle overhead of hybrid FCR is lower than that of remapping-based FCR. Second, the P/E cycle overhead for write-intensive applications is low. On average, write intensive workloads (i.e. iozone, cell099, postmark) only have 20% and 2% P/E cycle overhead for remapping-based FCR and hybrid FCR respectively. Read-intensive applications have higher P/E cycle overhead since the workload itself has low P/E cycles to begin with. However, hybrid FCR's P/E cycle overhead is below 10x even for daily refresh. Since daily refresh improves P/E cycle endurance by nearly 100x, the P/E cycle benefit of hybrid refresh still outweighs the P/E cycle overhead, leading to the lifetime improvements shown in the previous sections. Note that all P/E cycle overheads have already been accounted for in the collection of the flash lifetime results.

Figure 11 shows the additional flash energy consumption of remapping- based FCR and hybrid FCR averaged over all workloads compared to a system with no FCR. The refresh energy is estimated under the worst-case scenario that all data are to be refreshed. Even if we assume we must refresh the entire SSD each day, the energy overhead is only 7.8% and 5.5% for remapping based FCR and hybrid FCR respectively. When the refresh interval is 3 weeks, the energy overhead is almost negligible (less than 0.4%). We also observe that hybrid FCR has less energy overhead than remapping based FCR. This is due to two reasons. First, hybrid FCR reduces the high-energy erase/remap operations by performing in-place reprogramming most of the time. Second, when the data are programmed in place, the programmed data are almost the same as the data already present in the block, as the error rate is usually less than 1%. ISPP programming stops programming as soon as it detects that the target threshold voltage has already been reached. Thus, charge is injected into only a small number of cells during in-place reprogramming and many cells' states do not change (which consumes little energy).

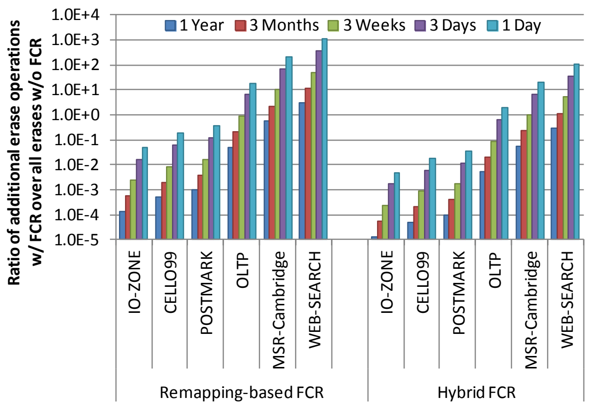


Figure 10. P/E cycle overhead of remapping-based and hybrid FCR.

We also evaluate the energy overhead of adaptive-rate FCR and find that it is only 1.5% (not shown in the figure). Recall that adaptive-rate FCR starts out with no refresh and gradually increases the refresh rate up to daily refresh as the P/E cycles accumulate. Yet its energy overhead is significantly lower than always daily refresh. We conclude that adaptive-rate FCR is the most superior of flash correct-and-refresh mechanisms in terms of both lifetime and energy consumption.

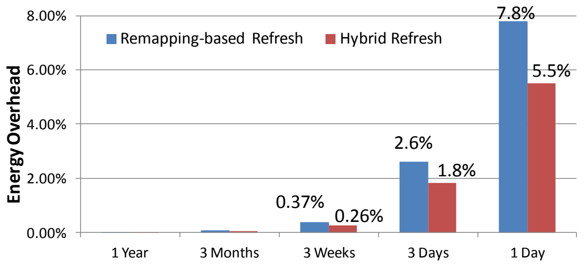


Figure 11. Energy increase of remapping-based and hybrid FCR v. no refresh.

VII. Related Work

To our knowledge, this is the first paper that exploits three major characteristics of retention errors in NAND flash memory to significantly improve flash lifetime without requiring any hardware changes. The three major characteristics are: (1) dominance of retention errors, (2) that retention errors can be fixed with in-place reprogramming, (3) correlation of retention error rate with P/E cycle lifetime. No previous paper developed mechanisms to: (1) reprogram in-place flash memory blocks to reduce retention errors; (2) adapt the reprogramming rate to the lifetime of the block based on the observation that retention errors increase with P/E cycle lifetime.

Flash controllers have turned to stronger ECCs to correct multiple errors. BCH codes [11] are the most widely used due to their powerful error-correction capability. Choi [3] and Lee [4] implemented 16-bit and 32-bit BCH codes for flash memory respectively. We extensively compare FCR to different-strength BCH codes in this work, showing that FCR provides higher lifetime at much lower complexity.

Pan et al. [14] propose the quasi-nonvolatile SSD technique, which relaxes non-volatility and uses refresh to improve P/E cycle endurance, similarly to remapping based FCR (which is concurrently developed). However, Pan et al. do not provide a lifetime evaluation using real experimental flash data and real workload traces. As shown in this work, additional P/E cycle overhead of remapping based mechanisms is very high and such mechanisms could even decrease the lifetime of NAND flash SSD for read-intensive applications.

Both FCR and [14] are similar to "memory scrubbing" techniques commonly applied to volatile memories [23], which periodically read each memory location, correct errors, and restore the corrected values in the memory location. The goal of such mechanisms to ensure errors are corrected before they accumulate beyond a point that ECC cannot correct, and thus improve reliability. In contrast, FCR's purpose is to improve P/E cycle lifetime of flash memory even if errors are correctable by ECC. As we show in this paper, simply applying scrubbing techniques to flash significantly increases erase operations, which degrades lifetime for especially read-intensive workloads. To overcome this, we introduce the new hybrid and adaptive-rate FCR mechanisms that exploit characteristics specific to flash memories.

Previous work [5],[6],[8] characterized the error patterns of NAND flash memory and showed that retention errors are caused by charge loss and are dominant failure mode. These observed error patterns build the foundation for the FCR techniques we propose in this paper.

Finally, Wilkerson et al. [24] observe that by increasing the strength of ECC, refresh rate in a volatile cache can be decreased, and thus the cache power can be decreased. Our proposal is the opposite and for a different purpose: we increase the refresh rate to reduce the need for strong ECC in order to improve P/E cycle lifetime in non-volatile flash memory.

VIII. Conclusion

We presented flash correct-and-refresh (FCR) techniques, which offer a low-overhead mechanism to significantly improve the lifetime of flash-based data storage systems, requiring only modifications to the SSD controller firmware or driver software. To our knowledge, this is the first work to improve flash reliability by leveraging the dominance of retention errors and using in-place reprogramming to correct retention errors. Our experimental evaluations using I/O traces from real workloads and error rates obtained from a real experimental flash platform show that FCR is effective in significantly improving flash storage system lifetime with only modest P/E and energy overheads. As flash continues to scale and the raw bit error rate and cell lifetime degrade, we hope our proposed FCR techniques are likely to serve as even more promising lifetime-enhancement techniques for future flash-based storage systems.

References

[1] Y. Koh, "NAND Flash Scaling Beyond 20nm", IMW 2009.
[2] S. Yasarapu, "Architectural Requirements for MLC based SSDs", FMS 2011.
[3] H. Choi et al., "VLSI Implementation of BCH Error Correction for Multilevel Cell NAND Flash Memory", IEEE Transactions on VLSI 2010.
[4] Y. Lee et al., "6.4Gb/s Multi-Threaded BCH Encoder and Decoder for Multi-Channel SSD Controllers", ISSCC 2012.
[5] Y. Cai et al., "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization and Analysis", DATE 2012.
[6] N. Mielke et al., "Bit Error Rate in NAND Flash Memories", IRPS 2008.
[7] M. Abraham et al., "NAND Flash Trends for SSD Enterprise", FMS 2010.
[8] S. Tanakamaru et.al,"95%-Lower-BER 43%-Lower-Power Intelligent Solid-State Drive (SSD) with Asymmetric Coding and Stripe Pattern Algorithm" , ISSCC 2011.
[9] N. Agrawal et al., "Design Tradeoffs for SSD Performance", USENIX 2008.
[10] Y. Cai et. al. "FPGA-Based Solid-State Drive Prototyping Platform", FCCM 2011.
[11] Shu Lin and D. J. Costello, Error control coding, Prentice Hall, 2004.
[12] C. Compagnoni et al., "First evidence for injection statistics accuracy limitations in NAND Flash constant-current Fowler-Nordheim programming," IEDM 2007.
[13] K. Park et al., "A Zeroing Cell-to-Cell Interference Architecture with Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories", JSSC 2008.
[14] Y. Pan et al., "Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications", HPCA 2012.
[15] J. Bucy et al., "DiskSim Simulation Environment Reference Manual", 2008.
[16] IOzone.org, "IOzone Filesystem Benchmark" , http://iozone.org.
[17] J. Katcher, "Postmark: a New File System Benchmark Technical Report", 1997.
[18] UMass Trace: http://traces.cs.umass.edu/index.php/Storage/Storage.
[19] SNIA: IOTTA Repository, http://iotta.snia.org/tracetypes/3.
[20] Open Source software at HP Labs, http://tesla.hpl.hp.com/opensource.
[21] K.-D. Suh et al., "A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme," JSSC 1995.
[22] J. Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh", ISCA 2012.
[23] D. Siewiorek et al., Reliable Computer Systems: Design and Evaluation, 2000.
[24] C. Wilkerson et al., "Reducing Cache Power with Low-Cost, Multi-bit Error-Correcting Codes", ISCA 2010.