Extending the Lifetime of NAND Flash Memory by Salvaging Bad Blocks

Chundong Wang and Weng-Fai Wong
School of Computing, National University of Singapore, Singapore
Email: {wangc, wongwf}@comp.nus.edu.sg

***Abstract*—Flash memory is widely utilized for secondary storage today. However, its further use is hindered by the lifetime issue, which is mainly impacted by wear leveling and bad block management (BBM). Besides initial bad blocks resulting from the manufacturing process, good blocks may eventually wear out due to the limited write endurance of flash cells, even with the best wear leveling strategy. Current BBM tracks both types of bad blocks, and keeps them away from regular use. However, when the amount of bad blocks exceeds a threshold, the entire chip is rendered non-functional. In this paper, we reconsider existing BBM, and propose a novel one that *reuses* worn-out blocks, utilizing them in wear leveling. Experimental results show that compared to a state-of-the-art wear leveling algorithm, our design can reduce worn-out blocks by 46.5% on average with at most 1.2% performance penalties.**

I. Introduction

Flash memory is a ubiquitous non-volatile memory technology nowadays. It is widely used for secondary storage in both embedded devices and solid state drives (SSDs).

Flash memory has two types, namely NOR and NAND flash, and currently the latter is the predominant form. NAND flash can either be of single-level cell (SLC) or multi-level cell (MLC) technology. MLC NAND flash can store more than 1 bit in a cell. There are, however, a number of technical challenges in deploying NAND flash. The first one is how to manage flash *blocks* and *pages*. A block is the unit for erase operation and a page is the unit for write and read operations. A block has a fixed number of pages and a page cannot be programmed unless the block it is in is erased first. Thus, data in a page have to be updated *out of place*. Pages in a block of SLC flash can often be randomly programmed while MLC flash supports sequential programming only [7].

Another challenge is NAND flash's limited *write endurance*. As mentioned, a page cannot be programmed unless its block is first erased. A program operation followed by an erase operation constitutes a *program/erase cycle*. Typically NAND flash can withstand 100,000 cycles for SLC type [12] or 10,000 for MLC type [7]. To maximize the lifespan, a technique called *wear leveling* is usually employed to distribute erasures as evenly as possible. It usually classifies data according to their update frequency as *hot* or *cold*, which will affect block allocation and data movements. However, unevenness is inevitable even with the best wear leveling design, and some blocks will still wear out eventually. Traditionally worn-out blocks are discarded together with blocks that were defective at the time of manufacture [8]. All bad blocks are marked and tracked by the *bad block management* (BBM) module of the *flash translation layer* (FTL), and made unavailable from regular use. When the number of bad blocks reaches a threshold defined by manufacturers, usually 2% [8], the entire chip will be rendered defective.

In this paper, we reconsider the way to manage bad blocks with wear leveling. The main contributions of this paper are:

- Many pages in a bad block may be functional. We propose that worn-out blocks could be reused. This paper presents the *Bad Block Salvaging* (BBS) scheme that recycles and reuses worn-out blocks efficiently.
- A wear leveling algorithm that utilizes *salvaged worn-out blocks* is proposed. In brief, our BBS creates a set of salvaged blocks wear leveling will use to store cold data.

II. Wear Leveling of Flash

Flash resources are managed by an embedded software called the flash translation layer. The FTL performs wear leveling to evenly distribute program/erase cycles over all blocks. A flash page can only be programmed (bits selectively set to 0's) and erased (all bits set to 1's) for a limited number of times. When a page is programmed, some bits will be flipped from 1's to 0's. During an erasure, these same bits will be flipped back to 1. If there is no wear leveling, one (or more) cell that has endured too many bit flips will permanently fail. This will cause the page and the block to fail. When too many blocks fail, the entire chip will fail.

At runtime, cold data are preferably stored in heavily erased blocks, i.e., *old* blocks, while hot data should be put in lightly erased blocks, i.e., *young* blocks. Wear leveling schemes usually have a block management module and a data swap strategy. Blocks are organized into separate *pools*. Free blocks can be allocated from the free block pool using FIFO or the youngest block first [1]. Access frequencies of blocks with valid data are tracked. A block with hot data will be swapped with a block with cold data to "cool down" the former.

Chang et al. [1] surveyed existing wear leveling schemes, and proposed a dual-pool algorithm, which partitions blocks with valid data into a hot pool and a cold pool that are prioritized by their erase counts. Blocks may exchange data and move to opposite pools on occasions. *Lazy wear leveling* [3] was a recently proposed scheme. In its design, when a block D with dirty data, is to be erased, its erase count will be checked. If it is higher than the average by a threshold, after erasing

978-3-9810801-8-6/DATE12/©2012 EDAA

D, the FTL will find a data block with cold data, say C, and transfer C's data to D. C will be free then.

Although wear leveling is used, pages in some blocks may still wear out. The BBM module of the FTL will handle such blocks. If the proportion of bad blocks exceeds a predefined limit, the flash chip will be rendered defective.

III. Bad Block Management

A bad block is one that has permanently faulty cells. *Initial bad blocks* are defective even at manufacture time. They are marked and recorded in a *Bad Block Table* (BBT) [8]. Over time, blocks that become defective due to excessive use, i.e., the *worn-out* bad blocks, will also be recorded in the BBT.

A. Traditional Management of Bad Blocks

Bad blocks first need to be identified. The checking and marking operations are performed by the BBM module in the FTL. A NAND flash page has two parts, namely the *data area* and the *spare area*. The former is used for data, and the latter can store useful information. Once a bad block is identified, a special marker will be placed in the spare area.

A startup scan upon bad block markers can be conducted at boot-up, and the BBT will be reconstructed. Besides marking, BBM also needs to perform necessary data movements. When a block is found to be worn out, all its valid data have to be moved to a free block. In addition, data intended to be written to this bad block must also be redirected.

Once created, the BBT may be saved to the flash itself [8] [2] so that on rebooting, it can be directly loaded into FTL. This will be faster than scanning spare areas across the chip. Note that worn-out blocks are treated in the same way as initial bad blocks, and will not be used any further.

B. Reusing Bad Blocks

When the proportion of bad blocks reaches a limit, typically 2% [8], the entire chip is rendered non-functional. This is a large number given today's flash chip capacity. Since a block may contain multiple pages [12], discarding an entire block due to the permanent failure[1] of a bit in a page is wasteful.

Let us revisit the issue of worn-out blocks due to excessive program/erase flips. The unit of write (a page) is different from that of erasure (a block). It is very likely that some pages may be reprogrammed much more than others in a block. Moreover, the failure of a page does not affect data in other pages in the same block [8]. This means that good pages are still functional even if a page in the block has permanently failed. If we could reuse such good pages, the utilization of blocks would be improved.

The challenge is how to reuse good pages effectively. Since initial bad blocks are marked at the block level by manufacturers, what pages are bad in a block cannot be exactly known. Hence, this paper will focus on worn-out blocks. Should their information be made available, however, initial bad blocks may also be used in our scheme. This is decided by the nature and extent of failures.

As far as we know, this paper is the first attempt to reuse bad blocks. Chang et al. [5] also claimed to "recycle" blocks. However, what they did is propose an algorithm to avoid erroneously identifying a good block to be bad. Bauer et al. [4] gave a method to isolate bad blocks at the circuit level. It aimed to localize the effect of a defective cell, and ensure other blocks would not be affected. In their schemes, bad blocks were not reused. Next we will present a new scheme called *Bad Block Salvaging* that manages and reuses blocks that actually contain defective cells.

IV. Salvaging Bad Blocks

The idea of salvaging bad cells in SRAM cache has been investigated in [9]. Essentially, if defects are localized, it is possible to reuse good cells in a block to "patch" defective ones in others. Note that flash differs significantly from SRAM. Other than the general idea, our BBS scheme shares nothing in common with the SRAM cache salvage scheme.

In terms of salvaging, bad blocks can be classified into three categories according to their degrees of wear. Blocks that are the least worn-out form the *backing blocks*. They have the most good pages to support other worn-out blocks. The second class is the *discarded blocks*, whose proportion of bad pages exceeds a certain *discard threshold*, δ. Discarded blocks are not worth salvaging. The third class is the *salvaged blocks*. These have a sufficient number of good pages to make it worthwhile for repair. In essence, good pages from the backing blocks will be used to stand in for bad pages in the salvaged blocks.

In BBS, all bad blocks are recorded in the *Bad Block List* (BBL). We also need another mapping table, the *Salvaging Mapping Table* (SMT), to track the mapping between backing and salvaged blocks. With BBS, if a write or read request refers to a faulty page, the FTL must redirect the request to its backing page using the SMT. The maximum size of SMT occurs when both the backing and salvaged blocks are just below δ, giving rise to a one-to-one mapping. If there are N blocks with M pages in a block, the maximum number of entries in SMT will be $\alpha/2{\cdot}N{\cdot}M/2$, where α is the percentage of bad blocks. A 4 Gb chip with 4096 blocks, each with 64 pages, each page being 2KB [12] will have a maximum of 2621 entries in the SMT if $\alpha = 4\%$. If an entry has 7 bytes (4 for two blocks, 2 for two pages and 1 for status), the SMT will merely take up about 18KB, i.e., 9 pages, at most. We can store the SMT in flash and cache parts of it on demand in RAM like DFTL scheme [6].

BBS is activated when worn-out blocks appear. By scanning the BBL, the least worn-out block will be selected as the initial backing block. Pages from the backing block will be used to repair other worn-out blocks. If good pages in the backing block are sufficient for salvaging, these pages will be used. The SMT will be updated accordingly. When good pages in the backing block are used up or not enough, the BBL will be scanned again for another backing block. Fig. 1 shows the scheme at work. $B1$ is picked to be a backing block. The SMT

[1]Transient failures can be corrected by error correcting codes (ECC) stored in the spare area.

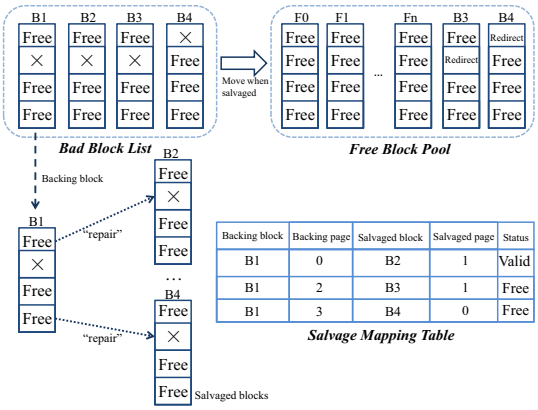

Fig. 1. The proposed BBS scheme at work

reflects the mappings between $B1$ and $B2, B3$ and $B4$. Note that the salvaged page of $B2$ is in use.

Salvaged blocks can be reclaimed like good blocks. If all salvaged blocks associated to a backing block are reclaimed, it will be reclaimed. All of them will be returned to the BBL for future salvaging if they have not reached δ to be discarded.

V. Wear Leveling With BBS

Salvaged blocks if used like normal ones risk being damaged soon. Note that blocks with cold data are less likely to be erased. Wear leveling usually swaps these cold data out, and utilizes the occupied young block to store hot data. Salvaged blocks are well suited for these swapped-out cold data. They ensure that salvaged blocks will not undergo too many erasures. On the other hand, good blocks still have better endurance than salvaged ones, and they are more suitable for holding hot data. We shall now describe how we propose to do wear leveling in the presence of salvaged blocks.

A. A Wear Leveling Algorithm with BBS

The key ideas of our wear leveling algorithm that is cognizant of salvaged blocks are as follows:

- Upon the allocation request for free block, the one that has the smallest erase count will be selected, namely the "youngest block first" policy [1].
- Data that have been in the valid block pool for a long time without updates will be considered to be cold, and they will be moved to a free salvaged block.
- If cold data are found but no salvaged block is available, a free normal block will be allocated instead. The free block should be an elder one in the free block pool.

For the first item, temporal locality indicates newly written data are likely to be updated soon, so the youngest block will be allocated for them. To conduct the latter two items, we can order block with valid data and free blocks according to last data-updating time and erase counts, respectively. The latter two move cold data to salvaged blocks, or good blocks that have been heavily erased. The second idea suggests moving cold data to a salvaged block first. This policy avoid filling good blocks with cold data where possible.

The procedure of wear leveling is shown in Algorithm 1. We use a relative measure for the age of blocks by comparison to their average erase count. This ensures that the evenness of wear is assessed over all blocks at any point in time. Particularly, the FTL checks a small portion (ω at line 4) of valid blocks for efficiency. The procedure of wear leveling is called upon the completion of each write request like dual-pool algorithm [1].

B. Alternative Approach with BBS

There is another way to utilize salvaged blocks. Instead of completely replacing the current wear leveling scheme, we can just add a module to it to identify cold data and move them to salvaged blocks on occasions. This is easier to implement, and its overhead will be smaller. Our experimental results show that this approach can also be quite effective.

Algorithm 1: Wear Leveling Procedure with BBS

```
1  begin
2      cnt := 0, flag := FALSE;
3      blk_pt := GetValidPoolHead (void);
4      while (cnt < valid_blk_quantity · ω) do
5          if (GetErsCnt(blk_pt) < (avg_erase_cnt)/2)) then
6              if (HasColdData(blk_pt) == TRUE) then
7                  flag := TRUE, cold_blk := blk_pt;
8                  break;
9          cnt++, blk_pt := GetNextVldBlk (blk_pt);
10     if (flag == FALSE) then
11         return;
12     logical_blk := GetLogicalBlkNum (cold_blk);
13     if (IsSalBlkSetEmpty (void) == TRUE) then
14         free_blk := AllocOldFreeBlk (void);
15         MoveValidData (cold_blk, free_blk, FREE);
16         UpdateMapTab (logical_blk, free_blk, FREE);
17     else
18         sal_blk := AllocSalvagedBlk (void);
19         MoveValidData (cold_blk, sal_blk, SAL);
20         UpdateMapTab (logical_blk, sal_blk, SAL);
21     Reclaim (cold_blk);
22     return;
```

VI. Experiments

To evaluate our proposal, we implemented the BBS-based FTL in FlashSim simulator [6] for SLC NAND flash. The mapping scheme is FAST [10]. Our compiler was GNU GCC 4.6 in Linux. All parameters of the simulated flash memory, like latencies of write and erasure, were obtained from [12]. For our experiments, we have used three families of disk traces from [13], [14] and [11]. We believe these traces are representative of various workloads.

To see blocks wearing out, we reduced the endurance limit to be from tens to dozens instead of using normal 100,000 cycles [12]. The simulated flash chip was configured to have no initial bad blocks but a number of worn-out blocks whose percentage is 0.5% of all. Moreover, when BBS is introduced to reuse bad blocks, the original upper bound of the proportion of bad blocks was increased from 2% [8] to 4%.

Four sets of results are presented here. `baseline` is a configuration that has no wear leveling or BBS. `lazy` employs lazy wear leveling [3]. `aug` also uses lazy wear leveling, but it

TABLE I
WORN-OUT BLOCKS AT RUNTIME

trace	baseline	lazy	aug	bbs
SPC1	0	70	64	0
SPC2	0	183	75	0
TPC-C	95	95	80	64
MSR-hm_0	1034	607	212	411
MSR-mds_0	16	8	3	3
MSR-prxy_0	761	300	273	149
MSR-rsrch_0	36	26	22	12
MSR-stg_0	0	14	9	0
MSR-ts_0	573	412	369	275
MSR-web_0	22	17	19	12
TOTAL	2537	1732	1126	926

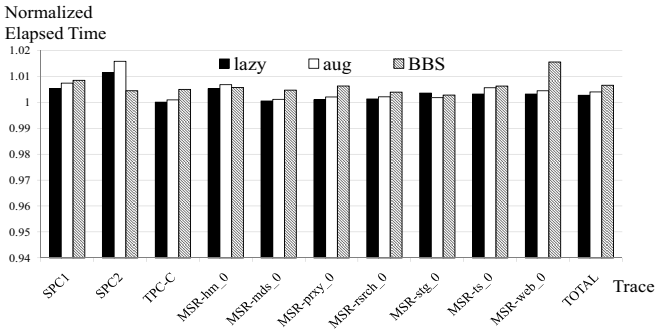

Fig. 2. Elapsed time of `lazy`, `aug` and `BBS`

includes the alternative approach to utilize salvaged blocks for cold data. `BBS` uses our wear leveling algorithm with BBS. For `aug` and `BBS`, δ and ω were set to be 50% and 0.1% by default, respectively; the impact of them would not be shown for space limitation.

Table I shows the number of worn-out blocks caused at runtime. For `aug` and `BBS`, all worn-out blocks that are in salvaging or discarded are counted. This measure for `BBS` is reduced by 46.5% compared to `lazy`. Fig. 2 is the elapsed time to finish each trace with four configurations, normalized against that of `baseline`. The introduction of wear leveling will definitely bring in performance overheads. From Fig. 2, we can see that `BBS` has comparable overheads to `lazy`, at most 1.2% more than the latter at MSR-web_0.

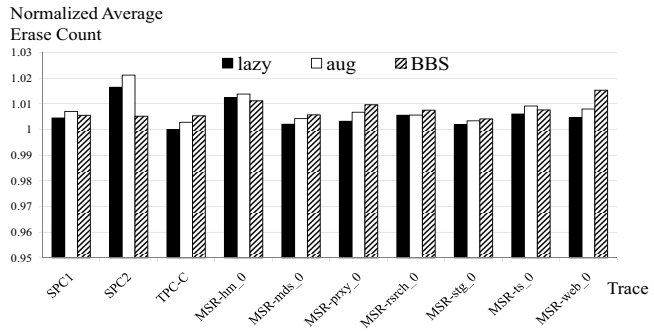

Fig. 3. Average erase count of `lazy`, `aug` and `BBS`

The effect of wear leveling is presented in Fig. 3 and Fig. 4. Fig. 3 shows the average erase count of all blocks with four algorithms, normalized against that of `baseline`. Fig. 4 shows standard deviations of erase counts. Wear leveling has to move data with more erasures to evenly distribute program/erase flips, as can be seen in Fig. 3. The wear

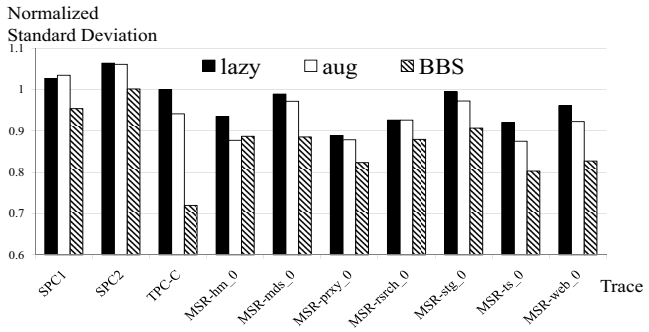

Fig. 4. Standard deviation of `lazy`, `aug` and `BBS`

evenness is evident from the standard deviation in Fig. 4. For similar average erase counts, the smaller the standard deviation is, the better the evenness is. Obviously `BBS` is better than `lazy` at every trace in Fig. 4 with smaller standard deviation.

The effect of `aug` is also evident. `aug` can outperform `lazy` significantly and is even better than `BBS` at MSR-hm_0. Hence, `aug` is a good alternative choice on wear leveling.

VII. CONCLUSION

In this paper, we revisited bad block management and wear leveling in NAND flash. We proposed a design that reuses worn-out blocks to prolong the lifespan of flash chips. Specifically, worn-out blocks are salvaged by using good pages in some blocks to stand in for bad pages in other blocks. The salvaged blocks are then used for cold data in wear leveling. Experiments showed that, compared to a state-of-the-art wear leveling scheme, our design can reduce the number of worn-out blocks by 46.5% on average with 1.2% performance penalties at most. As for future work, we plan to investigate how the algorithm can adapt to workloads at runtime.

REFERENCES

[1] L.-P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07*, 2007.

[2] K. P. Garvin et al. Method and system for managing bad areas in flash memory, 2001.

[3] L.P. Chang et. al. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *LCTES '11*.

[4] Mark E. Bauer et al. Method and circuitry for usage of partially functional nonvolatile memory, October 1998.

[5] R. C. Chang et al. Unusable block management within a non-volatile memory system, 2004.

[6] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09*.

[7] Y. Hu and D. Moore. MLC vs. SLC NAND flash in embedded systems. Technical report, September 2009.

[8] Micron Technology Inc. Bad block management in NAND flash memories. Technical report, July 2010.

[9] C.-K. Koh, W.-F. Wong, Y. Chen, and H. Li. The salvage cache: a fault-tolerant cache architecture for next-generation memory technologies. In *ICCD'09*.

[10] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[11] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4:10:1–10:23, November 2008.

[12] Numonyx. 4-Gbit, 8-Gbit, 2112-byte/1056-word page, multiplane architecture, 1.8 V or 3 V, SLC NAND flash memories. Technical report, February 2010.

[13] Storage Performance Council. SPC traces. http://traces.cs.umass.edu/, December 2009.

[14] BYU trace distribution center. TPC-C database benchmark traces. http://tds.cs.byu.edu/tds/, 2001.