Salvaging Runtime Bad Blocks by Skipping Bad Pages for Improving SSD Performance

Junoh Moon, Mincheol Kang, Wonyoung Lee, and Soontae Kim
School of Computing, KAIST, Republic of Korea
Email: {junoh.moon, mincheolkang, wy_lee, kims}@kaist.ac.kr

***Abstract*—Recent research has revealed that runtime bad blocks are found in the early lifespan of solid state drives. The reduction in overprovisioning space due to runtime bad blocks may well have a negative impact on performance as it weakens the chances of selecting a better victim block during garbage collection. Moreover, previous studies focused on reusing worn-out bad blocks exceeding a program/erase cycle threshold, leaving the problem of runtime bad blocks unaddressed. Based on this observation, we present a salvation scheme for runtime bad blocks. This paper reveals that these blocks can be identified when a page write fails at runtime. Furthermore, we introduce a method to salvage functioning pages from runtime bad blocks. Consequently, the loss in the overprovisioning space can be minimized even after the occurrence of runtime bad blocks. Experimental results show a 26.3% reduction in latency and a 25.6% increase in throughput compared to the baseline at a conservative bad block ratio of 0.45%. Additionally, our results confirm that almost no overhead was observed.**

***Index Terms*—bad block management, over-provisioning**

I. Introduction

Owing to its superior performance, a solid-state drive (SSD) has become the *de facto* standard storage devices for both consumers and industry. This superiority comes from the NAND flash, composing SSDs. However, in the NAND flash, only out-of-place updates are allowed, which is a major drawback [1]. This means that NAND flash cannot overwrite data at the same place, and that additional space is required. Hence, SSD vendors provide an overprovisioning (OP) space that is invisible to users. In addition to supporting out-of-place updates, an OP space is used to replace *bad blocks*, defective and uncorrectable blocks. This bad block is replaced by one of the functioning blocks in the OP space and then excluded from storing the data. Consequently, users are unaware of the capacity change, whereas the OP space is sacrificed.

However, a reduction in the OP space causes poor performance. A decrease in the OP space increases write amplification (WA), which is the amount of data written to an SSD divided by those written by the host, and results in severe performance degradation [2]. In other words, internal write operations steeply increase in the background as the OP space decreases. Therefore, maintaining the OP space is crucial, as detailed in Section III.

Previous research has focused on reusing worn-out bad blocks, exceeding maximum erase cycles, to keep the OP space; moreover, it was considered that bad blocks develop at the late stage of the lifetime, mainly because of the wearout. Wang *et al*. presented reusing worn-out bad blocks by creating a virtual block made up of functioning pages in worn-out bad blocks [3]; however, this method is impractical because it is incompatible with modern SSDs and has no information about differentiating reusable pages in worn-out bad blocks. Another study proposed the use of worn-out bad blocks as hot data storage [4]. Yang *et al*. attempted to reuse worn-out bad blocks as single-level cell (SLC) mode blocks. However, the capacity was significantly reduced as a side effect [5]. All these attempts have a fundamental drawback in that their performance improvements from re-use of worn-out bad blocks are negligible because worn-out bad blocks are considered to occur only at the late stage of the total lifetime. Furthermore, they did not consider *runtime bad blocks* which are different from worn-out bad blocks and occur at an early stage of the lifetime. Several studies in the industry found that *bad blocks develop at an early stage rather than the late stage of the lifetime*, contrary to popular belief [6], [7]. A runtime bad block not only occurs far earlier than a worn-out block, but also shows different characteristics from those of worn-out bad blocks. Accordingly, it is difficult to apply these works to the state-of-the-art SSDs.

To address this problem, we propose salvaging runtime bad blocks to mitigate performance degradation by detecting page errors because of write failures. We developed the *bad page skipping* procedure, which filters salvageable pages in a bad block, designed an efficient data structure for managing defective and unsalvageable bad pages, and examined its feasibility and overall complexity based on the fact that these blocks can be identified by detecting page errors. The experimental results show that the latency decreases by 26.3%, and the throughput improves by 25.6% in the conservative situation. To the best of our knowledge, this is the first study on salvaging runtime bad blocks that occur at an early stage to maintain performance.

II. Bad Block Background

SSDs are permanent storage devices; hence they should keep data for a long time, even in extreme environments. Unfortunately, some blocks do not satisfy vendor requirements and are filtered out beforehand. They are called *factory bad blocks* and identified using a predefined rule. Open-NAND-Flash-Interface (ONFI), a unified interface for an SSD to communicate with NAND flash, defines how to mark factory bad blocks [8], [9]; hence, it is possible to determine whether a block is a factory bad block by reading the mark of each block. Although not all vendors follow the requirements, they all specify identification methods in a similar manner, for example, using different numbers and locations [10], [11].

978-3-9819263-6-1/DATE22/©2022 EDAA

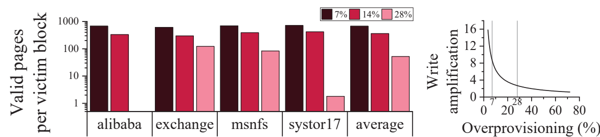

Fig. 1. Comparison of the number of valid pages between three different OPs, and relationship between OP and WA [2].

In addition to being filtered out in a fabrication facility, bad blocks may occur after excessive erase operations. The number of erase operations is called a program/erase (P/E) cycle, and after it exceeds a certain threshold, a block with exceeding P/E cycles is considered a *worn-out bad block* and excluded from storing data. Due to this exclusion, a reduction in capacity is unavoidable. SSDs, however, should provide the same capacity to users; hence, all types of bad blocks are replaced by blocks in the OP space.

III. Motivation

A. Performance Degradation by an Overprovisioning Space

An insufficient OP space not only makes the first garbage collection (GC) earlier as commonly known, but also degrades the quality of victim blocks and GC performance. This is because the GC performance is dominated by the number of valid pages in a victim block. In detail, our experimental results reveal that a reduced OP space does not allow a flash translation layer (FTL), a software stack in SSDs, to wait for a better victim block with fewer valid pages, but makes the FTL invoke GC prematurely; The left graph in Fig. 1 shows a 13-fold increase in the number of valid pages as the OP space decreases from 28% to 7%. This early invocation causes more page copies, resulting in higher WA, longer latency and lower throughput [12]. Consequently, the performance of the GC is heavily influenced by the OP space.

The OP space has greatly reduced in recent years. Early SSDs provided more than 28% for sufficient OP spaces; the decline in the OP space in these earlier SSDs was a minor problem because it was difficult to recognize the changes in WA at a ratio of approximately 28% (right graph in Fig. 1). Consequently, the decrease in the OP space was tolerable. However, as the capacity of SSDs increases, it has become challenging to maintain a sufficient OP space due to the cost, and thus it has been reduced to 7%. This results in a vulnerability where even a slight change in the OP space significantly impacts its performance. As demonstrated in the right graph in Fig. 1, there is a 12-fold increased change in WA between the 28% to 7% values. Therefore, the reduction in the OP space must be addressed.

B. Runtime Bad Blocks

Until recently, it was commonly believed that bad blocks developed because of defects in a process (factory bad blocks) or exceeded the P/E cycle threshold (worn-out bad blocks). However, research from the industry has revealed that bad blocks at runtime can occur even within 1–2 years, which is less than half of the intended lifetime [6], [7]. They are called *runtime bad blocks*. Although the reason is not clear without analyzing the NAND flash chips, there are two hypothesized conditions on why bad blocks occur at runtime. First, in modern SSDs with 3D NAND flash, more than a thousand pages are in a single block and even a single page error makes an entire block considered as defective. Hence, the changes of a page error are significantly raised as the number of pages in a block increases. Second, not all pages are evenly worn out; since each page raw-bit-error-rate (RBER) in a block differs from each other [13], pages with high RBER are heavily influenced by disturbance errors, resulting in page errors [14]; thus, they are able to be considered as defective earlier than the intended lifetime. This disparity among pages has become severe as 3D NAND flash has prevailed over planar NAND flash for larger capacity. In contrast to worn-out bad blocks, these runtime bad blocks develop in the midst of write operations and should be treated and managed differently from worn-out bad blocks. That is, all previous studies based on worn-out bad blocks cannot notice and reuse these blocks. As of now, runtime bad blocks have been handled inefficiently. As aforementioned, a block can be regarded as defective because of a single page error. However, most pages may well remain functional because their RBERs are lower than that of the bad page [13]. It implies this conventional method has significant potential for refinement. Accordingly, it has become a critical issue to salvage runtime bad blocks.

IV. Runtime Bad Block Salvation

In this section, we present a salvation scheme for runtime bad blocks using a *bad page skipping* procedure. First, this paper briefly introduces the core components of our scheme and presents an overview of this scheme. Second, we describe how to manage the runtime bad blocks in detail. Finally, we validate the feasibility of this method.

A. Overview

Fig. 2 provides an overview of the proposed scheme. The key concept of our salvation scheme is to skip bad pages in a runtime bad block; the following two important objectives must be met to realize the idea: 1) identifying bad pages in runtime bad blocks, and 2) managing the salvaged pages. To achieve the first goal, this scheme must detect a page error on every write failure and identify the address of this bad page. To address the latter, we introduce a *bad page history table* (BPHT), enabling an FTL to skip bad pages.

With the aid of BPHT, a write request proceeds as follows (left diagram in Fig.2). On every write request, the write allocator selects a free page using a free page index, as in other schemes. The difference is that the write allocator has no information about the previous defect history. Thus, BPHT always looks up the history of this page and skips bad pages if the defect history exists; a page with no defect history is selected to store data. Data are stored on the page within the NAND flash. If this write operation succeeds, the page information is updated on the page-mapping table and the request ends. Otherwise, it is evident that a page error is raised in the midst of the write operation, and this write failure must

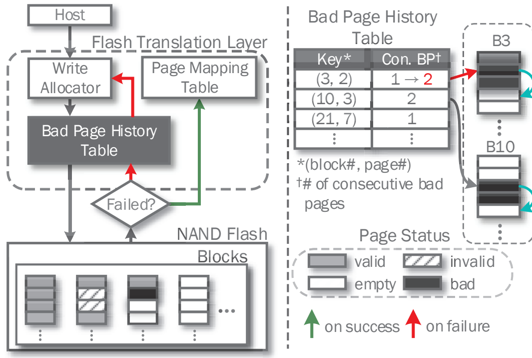

Fig. 2. System overview of runtime-bad-block salvation.

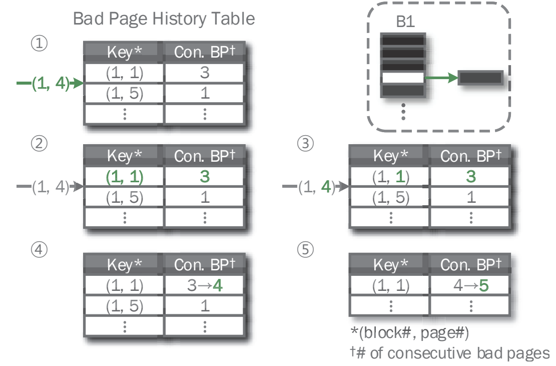

Fig. 3. Sequence of updating bad-page history table.

be handled; therefore, erroneous page information is updated on BPHT, and the request is repeated from the beginning until successful.

B. Skipping Procedure

The BPHT is a hash table in which a bad-page history is recorded. A pair of (b, i), a block index, and a page index, is used as a key, and its corresponding value v is the number of consecutive bad pages. Therefore, BPHT yields a constant time for skipping operations while keeping the size compact over the entire lifetime.

The right diagram in Fig. 2 shows a series of steps for skipping bad pages. Each entry in BPHT means that data should be stored after skipping v pages; for example, the entry at $(10, 3)$ and its value of two indicate that the third page in the block #10 is defective, and data must be stored on the fifth page after skipping two bad pages from the third page.

C. Update Procedure

As mentioned earlier, bad pages are recorded in BPHT. For the sake of space efficiency, the table stores a range of consecutive bad pages rather than a single bad page. Therefore, the update procedure not only creates a new entry but also merges adjacent entries if required. The preceding or succeeding entry is merged as follows: Fig. 3 shows an example when a new bad page at $(1, 4)$ is updated in the table (①). The BPHT then linearly searches for the preceding entry and finds the entry with the key $(1, 1)$ (②). The preceding entry of $(1, 1)$ points to the fourth page after skipping three bad pages, and the fourth page is now confirmed as bad (③); hence, it is merged to $(1, 1)$ with the corresponding value of four (④). Similarly, the merged entry points to the fifth page after skipping four pages, and the fifth page is also defective; thus, all these entries are merged to the key of $(1, 1)$ with the value of five (⑤).

Because this scheme linearly searches for a preceding entry, one may be concerned that it will reduce the overall performance. However, BPHT is updated only if a page error is raised, when barely occurs. In the experiment, we confirmed that the number of entries in the table would not exceed hundreds, even in the worst case. Consequently, it is proven that the update procedure has almost no impact on the overall performance.

D. Feasibility

To skip only bad pages in a block, the issue to be considered is twofold: programming order constraints in multi-level cells (≥ 2 bits/cell) and bad page identification.

1) Sequential Programming Constraint: Unlike SLC NAND flash, multi-level cell NAND flash cannot write at arbitrary pages, and this is called a *sequential programming constraint* (SPC). This constraint originated from the following problem: whereas SLC has only two states, 8–16 levels must be distinguished in modern multi-level cell SSDs, resulting in vulnerability to disturbance errors. As a result, pages in a block must be written in a predefined order, not randomly.

However, this constraint does not imply that every page in a block should be written. Rather, pages can be skipped in a block as long as it is written in the forward direction [15]. Consequently, it is feasible to skip bad pages.

2) Detecting Page Errors: There is nonetheless one remaining issue about detecting page errors due to write failure. It is not feasible for runtime bad blocks to use the same approach of worn-out bad blocks (monitoring P/E cycles) because runtime errors are different from the wearout and these errors develop much earlier than the wearout. The only way to determine the error is to check a write operation each time. ONFI can resolve this issue; the operation results are written in the zeroth bit of a status register, and then page errors can be readily identified [8]–[10]. Although some vendors have not yet involved in ONFI group, they offer similar validation methods [11]. Through this mechanism, an FTL can be aware of the command failure, and eventually can detect runtime bad blocks when a block continuously fails to handle a command.

V. Evaluation

To evaluate achieved performance by salvaged runtime bad blocks, we used the SimpleSSD simulator [16] with SSD configurations as shown in Table I. Four trace workloads are used [17]–[19], and each timestamp of Exchange and MSNFS workloads is ten-fold accelerated because their workloads are measured using conventional devices and insufficient to represent modern SSD usage.

The most important aspect of establishing the evaluation environment is modeling bad blocks and pages. There is as

TABLE I
SSD CONFIGURATIONS

Bits per cell	3
# of (channels, packages, dies)	(4, 2, 2)
# of (planes, blocks, pages)	(2, 1366, 768)
Page size	16384 B
tR, tPROG, tBERS	(45, 700, 3500) μs

of now insufficient information regarding this modeling, and hence we adopted the error distribution of the page-RBER; RBER follows a *lognormal*(μ, σ^2) distribution [13], [20]. We chose σ and μ to evaluate it based on realistic bad block ratios from 0.01% to 1.5% to show various situations. The σ value of 0.5 is taken from the real device [13], and six RBERs $(1 \cdot 10^{-12}, 5 \cdot 10^{-12}, 1 \cdot 10^{-11}, 3 \cdot 10^{-11}, 5 \cdot 10^{-11}$, and $1 \cdot 10^{-10})$ are chosen as input of μ after conversion as follows:

$$\mu(RBER, \sigma) = \ln(1 - (1 - RBER)^p) + \sigma^2,$$

where p is the page size in bits to make RBER the mode of the distribution.

Fig. 4 shows the normalized performance over the baseline, which does not salvage bad blocks as conventional SSDs. It shows that the proposed scheme substantially outperforms the baseline in both latency and throughput as errors accumulate. In particular, the latency decreases by 26.3% and the throughput increases by 25.6% at a conservative bad block ratio of 0.45% [6]. This is because our scheme retains the available blocks, whereas the baseline does not. These salvaged blocks not only delayed the first GC invocation but also made an FTL select a better victim block. The third graph in Fig. 4 shows the number of valid pages in a victim block during GC. As the chances of selecting a better block increase, the number of valid pages decreases, resulting in a reduction of page copies.

The impact was negligible in terms of overhead. Even at the RBER of 10^{-12}, with few bad blocks, it still performed at least the same as the baseline, as shown in the first two graphs. Thus, our scheme is orthogonal to the other components in an FTL with no overhead.

VI. CONCLUSION

In this paper we firstly investigated runtime bad blocks, which have a significant impact on performance of SSDs, and must be addressed to salvage them in order to mitigate performance degradation. Based on this observation, we presented a novel salvation scheme that mitigates performance degradation caused by runtime bad blocks. Finally, we confirmed that our scheme memory-efficiently manages bad pages in constant time regardless of the age of the SSD and the number of bad blocks. The experimental results in realistic environments show that the latency reduces by 26.3% and the throughput improves by 25.6% compared to the baseline. We expect this scheme to be widely applied to state-of-the-art FTLs.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation (NRF) grants funded by Korean Government (2021R1F1A106273011).

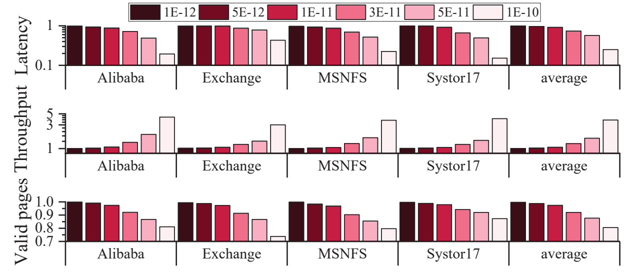

Fig. 4. Performance gained by salvaged blocks.

REFERENCES

[1] M. Kang, W. Lee, and S. Kim, "Subpage-aware solid state drive for improving lifetime and performance," *IEEE Transactions on Computers*, 2018.

[2] P. Desnoyers, "Analytic models of ssd write performance," *ACM Trans. Storage*, 2014.

[3] C. Wang and W.-F. Wong, "Extending the lifetime of nand flash memory by salvaging bad blocks," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012.

[4] P. Huang, G. Wu, X. He, and W. Xiao, "An aggressive worn-out flash block management scheme to alleviate ssd performance degradation," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[5] T. Yang, H. Wu, and W. Sun, "Gd-ftl: Improving the performance and lifetime of tlc ssd by downgrading worn-out blocks," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, 2018.

[6] S. Maneas, K. Mahdaviani, T. Emami, and B. Schroeder, "A study of SSD reliability in large scale enterprise storage deployments," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[7] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[8] *Open NAND Flash Interface Specification*, Open NAND Flash Interface Working Group, 2 2020, revision 4.2.

[9] *8Gb Asynchronous/Synchronous NAND Features*, Micron Technology, 4 2014, rev. F.

[10] *32Gbit(4096M x 8bit) Legacy NAND Flash Memory*, SK Hynix, 1 2011, rev. 0.7.

[11] *16Gb E-die NAND Flash Multi-Level-Cell (2bit/cell)*, Samsung Electronics, 3 2010, rev. 0.9.1.

[12] I. Fareed, M. Kang, W. Lee, and S. Kim, "Leveraging intra-page update diversity for mitigating write amplification in ssds," in *ICS*, 2020.

[13] N. Papandreou, H. Pozidis, T. Parnell, N. Ioannou, R. Pletka, S. Tomic, P. Breen, G. Tressler, A. Fry, and T. Fisher, "Characterization and analysis of bit errors in 3d tlc nand flash memory," in *2019 IEEE International Reliability Physics Symposium (IRPS)*, 2019.

[14] W. Lee, M. Kang, S. Hong, and S. Kim, "Interpage-based endurance-enhancing lower state encoding for mlc and tlc flash memory storages," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[15] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Comput. Surv.*, 2014.

[16] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *MICRO*, 2018.

[17] (2020) Alibaba cloud block storage. [Online]. Available: https://github.com/alibaba/block-traces

[18] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *2008 IEEE International Symposium on Workload Characterization*, 2008.

[19] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.

[20] I.-C. R. Motwani, "Exploitation of rber diversity across dies to improve ecc performance in nand flash drive," *Flash Memory Summit*, 2012.