IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 15, 2020, accepted October 23, 2020, date of publication October 26, 2020, date of current version November 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033886

Improving Write Performance Through Reliable Asynchronous Operation in Physically-Addressable SSD

DAEYONG LEE[1], JAEWOOK KWAK[1], GYEONGYONG LEE[1], MOONSEOK JANG[1], JOONYONG JEONG[1], KEXIN WANG[1], (Student Member, IEEE), JUNGWOOK CHOI[1], (Member, IEEE), AND YONG HO SONG[1,2], (Member, IEEE)

[1]Department of Electronics and Computer Engineering, Hanyang University, Seoul 04763, South Korea
[2]Samsung Electronics Company Ltd., Hwaseong 18448, South Korea

Corresponding author: Yong Ho Song (yhsong@hanyang.ac.kr)

This work was supported by the Samsung Electronics' University Research and Development Program (Research on Wear-Leveling Algorithm for Open-Channel SSD 2.0.).

ABSTRACT Physically-addressable solid-state drives (PASSDs) are secondary storage devices that provide a physical address-based interface for a host system to directly control NAND flash memory. PASSDs overcome the shortcomings such as latency variability, resource under-utilization, and *log-on-log* that are associated with legacy SSDs. However, in some operating environments, the write response time significantly increases because the PASSD reports the completion of a host write command synchronously (i.e., *write-through*) owing to reliability problems. It contrasts asynchronous processing (i.e., *write-back*), which reports a completion immediately after data are received in a high-performance volatile memory subsequently used as a write buffer to conceal the operation time of NAND flash memory. Herein, we propose a new scheme that guarantees write reliability to enable a reliable asynchronous write operation in PASSD. It is designed to use a large-granularity mapping table for minimizing the memory requirements and performing internal operations at an idle time to avoid response delays. Results demonstrate that the proposed PASSD reduces the average write response time by up to 88% and guarantees reliability without performance degradation.

INDEX TERMS Flash translation layer, NAND flash memory, open-channel SSD, physically-addressable SSD, solid-state drive.

I. INTRODUCTION

NAND flash-based solid-state drives (SSDs) are secondary storage devices used in various computing environments, from mobile devices to server systems. It is rapidly replacing the magnetic-based hard disk drives owing to its advantages, such as high random access performance, low power requirement, small form factor, and cost-per-bit reduction that has continued for decades [1], [2]. However, SSDs possess some disadvantages, such as latency variability [3]–[7], suboptimal resource utilization [8], [9], *log-on-log* [10], [11], and long-tail latency [12].

Flash translation layer (FTL) is a software layer embedded in a SSD that abstracts NAND flash memory to a block input/output (I/O) device and ensures reliability. It has been investigated for a long time, and several policies and architectures have been proposed to improve the performance and lifespan of SSDs [13]–[18]. However, its high complexity and unpredictability primarily contribute to the aforementioned problems in legacy SSDs.

Recently, a new class of SSDs comprising a host-based FTL (hFTL) and physically-addressable SSD (PASSD) [19]–[22] has been proposed, i.e., open-channel SSDs (OCSSDs) [23] and *zoned namespace* SSDs [24], which operate with hFTLs having compatible interfaces, such as the *physical block device* (*pblk*) and *zoned namespace* file system [25], respectively. Unlike the traditional SSDs with FTLs built-in to provide an abstracted block I/O interface, PASSD exposes a physical interface of NAND flash memory, and a FTL is located on a host system. It offers various advantages, such as predictable response time at the host system level, physically independent I/O path that can be used

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

as a scheduling option (e.g., isolation) [26], and stack optimization with a file system. Therefore, the new class of SSD is expected to compensate for the shortcomings associated with legacy SSDs and provide better performances.

However, some researches have shown that the performance of a storage system, comprising a PASSD and hFTL, is below expectations or lower than that of legacy SSDs [27]–[31]. Most researches have reported that performance degradation was caused by the hFTL; however, the operating characteristics within the PASSD can also cause performance degradation. As we observed, PASSD is unable to utilize asynchronous write completion techniques that report the completion of the write command before the program operation of the NAND flash memory is completed; these techniques are used to conceal the slow write performance of NAND flash memory. For example, typical SSDs shorten the write response time by reporting the command as completed immediately after receiving data in a write buffer composed of high-performance nonvolatile memory [32]–[34].

In fact, a fundamental reason that PASSDs manage write operations synchronously is that they cannot ensure the reliability of write operations in case of a program failure [35]. As asynchronous write operations may report completion before data are stored in the NAND flash memory, reliability problems can occur when a command that has already been completed fails during the NAND flash program operation. In addition, in a latest NAND flash memory, a durability and data integrity of a cell are significantly degraded due to the scaling down of a semiconductor process and a reduction of a bit sensing distance due to multi-level-cell technologies such as TLC/QLC [36]–[39]. Therefore, to write asynchronously, PASSDs must be equipped with a function that can guarantee the reliability of a write command even if a program failure occurs.

Herein, we propose a *fault-tolerant PASSD* (FT-PASSD) with a built-in write reliability policy to process write operations asynchronously. In particular, the proposed policy minimizes memory requirements using a large-granularity (block-level) mapping table [40] and small-sized data structures. Furthermore, it is designed to eliminate response delays when writing a request, even with program failure.

Our proposed method was evaluated using a trace-driven simulator modeling OCSSD and *pblk*. The experimental results demonstrate that FT-PASSD reduces the write response time by up to 88% compared to OCSSD. In addition, the memory required by the FT-PASSD was only 24 kB based on a 192 GB NAND flash memory.

The remainder of this article is organized as follows. Chapter 2 presents the background information on NAND flash memory-based storage systems and address-mapping tables. Chapter 3 introduces the motivation for this study and the experimental observations and challenges faced. Chapter 4 describes the FT-PASSD architecture and a scheme to ensure the reliability of the write operation. Chapter 5 describes the evaluation environments and results. Finally, Chapter 6 provides concluding statements.

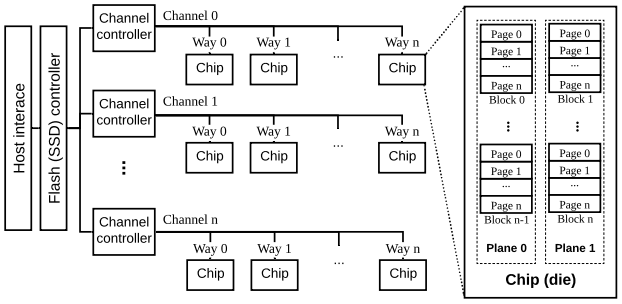


FIGURE 1. Layout of NAND flash memory-based SSD.

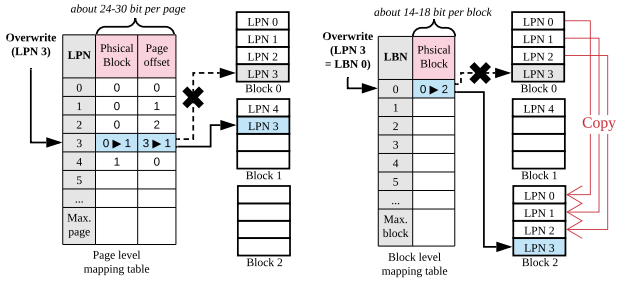


FIGURE 2. Comparisons of page and block level mapping tables; operation sequence of out-place update for overwrite.

II. BACKGROUND

A. NAND FLASH MEMORY

Fig. 1 depicts the layout of a typical NAND flash memory-based SSD. Most SSDs are composed of multiple NAND flash memory chips, and these chips operate in parallel by being tied to a channel and a way interface. Each flash chip comprises a number of blocks (e.g., 4–24 MB), which is the smallest erase unit, and each block comprises a number of consecutive pages (e.g., 2–32 kB), which are the minimum read/write units.

Because NAND flash memory has some inherent limitations in operation, SSDs provide a highly abstracted interface to the host system by embedding a high-performance controller. In particular, NAND flash memory blocks are required to be erased before writing; moreover, as only sequential writing is allowed for pages within a block, the overwriting and writing of arbitrary addresses are impossible. In addition, NAND flash memory can cause instruction failure due to various reliability problems [41]–[44].

B. MAPPING TABLE

Most SSDs utilize a mapping table to provide a logically abstracted address interface and high performance by allowing the overwrite to be processed as an *out-place update*.

Fig. 2 illustrates the processing of an *out-place update* of logical page number (LPN) 3 in the environment of a page-mapping table and a block-mapping table. In a page-mapping environment, not only blocks but also page addresses can be changed. However, as blocks can be physically written sequentially, the address that can be actually

TABLE 1. Example of memory requirement of the page-mapping table.

Volume	Page size	Page count	Address encoding	Total mapping table size
256 GB	16 KB	16.8 M	24 bit	48 MB
256 GB	8 KB	33.6 M	25 bit	100 MB
1 TB	16 KB	67.1 M	30 bit	240 MB

TABLE 2. Example of memory requirement of the block-mapping table.

Volume	Block size	Block count	Address encoding	Total mapping table size
256 GB	24 MB	10.9 K	14 bit	18 kB
256 GB	12 MB	21.8 K	15 bit	40 kB
1 TB	24 MB	43.7 K	16 bit	85 kB

written is a physical block number (PBN) 1 page offset 1 or a PBN 2 page offset 0. On the contrary, in the block-mapping environment, the page offset within the block cannot be changed; thus, even if the address of the block is changed, the page offset must be maintained. Therefore, to change the address of logical block number (LBN) 0 containing LPN 3 to PBN 2, all pages in the PBN 0 (LPN 0–3) must be copied.

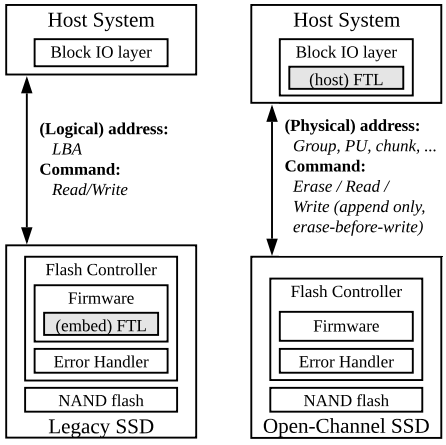
Tables 1 and 2 present the memory requirements of the page-mapping and block-mapping tables, respectively, through examples of NAND flash memory specifications. In general, since the mapping table is an array type data structure, the size of the mapping table can be obtained by multiplying the number of bits (address-encoding bits) for address expression for each mapping unit and the number of mapping entries. The address-encoding bits increase as the number of mapping units increases. Thus, the smaller the mapping unit and the larger the capacity, the more memory is required. For this reason, the page-mapping table requires very large memory compared to the block-mapping table. In the example, the page-mapping table requires 2.5–3k times more memory than the block-mapping table.

C. PHYSICALLY-ADDRESSABLE SSD

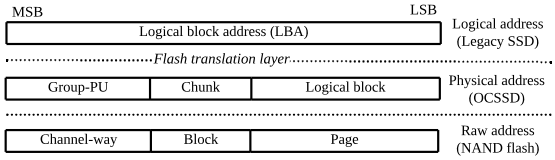
The OCSSD is the most representative PASSD, and it is well known in academia and the industry [45]–[48]. We used the architecture and interface of an OCSSD as an example of PASSD to compare it with legacy SSDs.

Fig. 3 shows a comparison of legacy SSDs and OCSSD. Both the legacy SSDs and OCSSD have built-in controllers. However, unlike legacy SSDs, where the FTL (and various reliability policies) is located in a built-in controller, the FTL of the OCSSD is located in a host system. Therefore, the complexity of the OCSSD built-in controller is reduced significantly, thereby solving the problem of unpredictable response delays. In addition, OCSSDs afford a relatively low manufacturing cost because the required hardware resources are significantly reduced by excluding the FTL.

Additionally, each environment operates under the different command sets and address types. Legacy SSDs operate



(a) Architecture and I/O interface



(b) Address encoding

FIGURE 3. Comparisons of legacy SSD and PASSD.

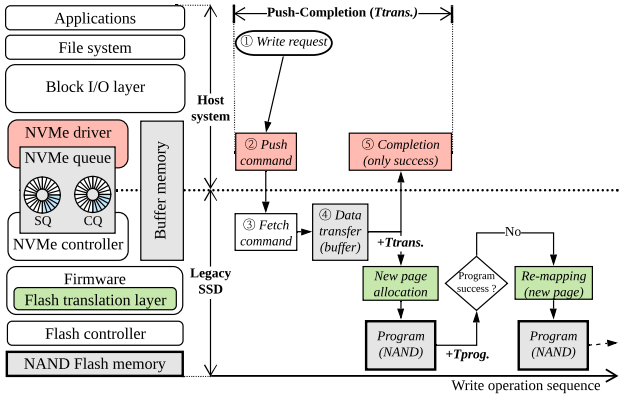
under read and write commands (allowing overwrite and random-access write) that comprise block I/O commands in logical block address units within the logical address space. On the contrary, PASSDs operate under the physical address and command set of the NAND flash memory. Therefore, the host system of a PASSD issues a write command only in sequential page units within a block and erases a block before overwriting.

III. MOTIVATION

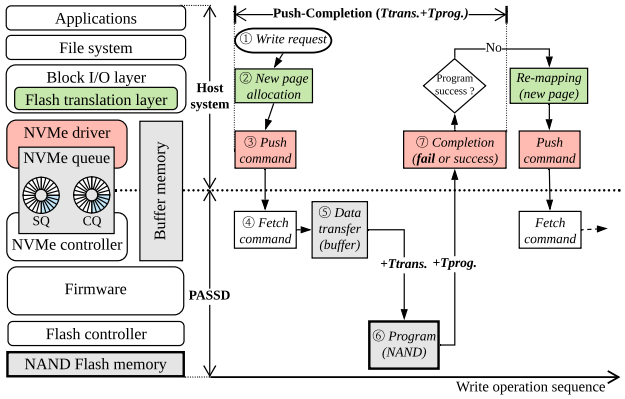
A. PERFORMANCE DEGRADATION IN PASSD

Fig. 4 shows a comparison between the write operations of a legacy SSD and PASSD. A write request is processed by exchanging submission and completion messages according to a protocol defined between the host system and the SSD [49]. As shown, the legacy SSD uses an asynchronous processing method and completes a request as soon as data are received in a volatile memory (write buffer); subsequently, the actual NAND write operation (buffer eviction) is scheduled in a built-in FTL. In this technique, a request is completed within the response time of the high-performance memory, which is faster than that of the NAND flash memory. However, data may be lost when a failure occurs in the program operation of the NAND flash memory. Hence, most built-in FTLs of the legacy SSD includes a policy that can guarantee the reliability (write success) of a failed write request.

Meanwhile, in the PASSD environment, the physical address of the NAND flash memory in which data is to be stored is determined by the hFTL. Since the address satisfies



(a) Legacy SSD; asynchronous write completion is possible as the built-in FTL always guarantees the success of the NAND flash program operation.

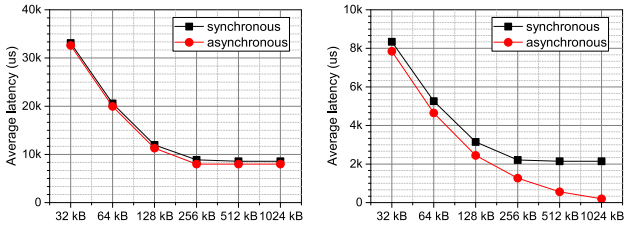


(b) PASSD; as the host FTL controls both success/failure of write command, only synchronous completion report is possible

FIGURE 4. Comparisons of architecture and write operation sequence.

all operational requirements of NAND flash memory such as erasure before writing and access to sequentially write pages in a block, there is no need to convert the address within the PASSD; However, due to the possibility that the determined physical address is a bad page (block), PASSD must report write completion synchronously. In other words, PASSD must report all write commands as success or failure after the operation of the NAND flash memory is finished. Owing to the differences in write operations, the PASSD may have a longer write response time than the legacy SSD.

Fig. 5 shows the average write response time for the synchronous/asynchronous write operation. The simple experimental results are interpreted as the response time converging to the performance of NAND flash memory because the buffer is full when the time interval between commands is very short (1 *us*). In contrast, the workload with sufficient time intervals (1 *s*) shows that asynchronous write responses are reduced compared with synchronous write responses. These results confirm that the synchronous write operation of a PASSD can significantly affect the response performance depending on the workload characteristics. One way to solve this problem is to enable asynchronous write completion by embedding a reliability policy in the PASSD.



(a) Sequential write workload (without idle time) (b) Partial write workload (with idle time)

FIGURE 5. Average latency in synchronous/asynchronous write completion and write buffer size (32–1024 Kb per way).

B. CONVENTIONAL WAY TO SECURING WRITE RELIABILITY

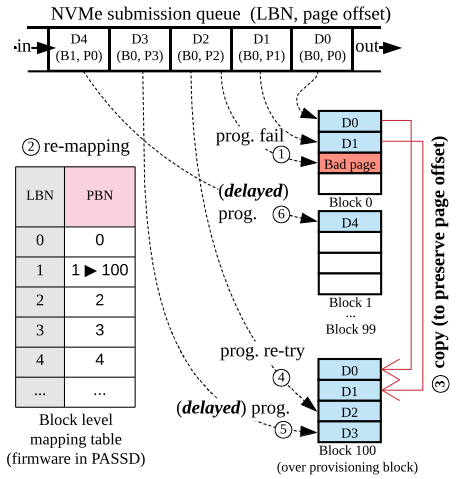
A typical approach to ensure the reliability of a failed write request is to assign a new address and retry the program operation [50], [51]. The address assignment is handled in the same manner as the overwrite described in the background section (Fig. 2). However, as the FTL layer, including the mapping table, has been removed from the PASSD, allocating a new address is impossible. If a mapping table is additionally embedded in the PASSD, having duplicate mapping tables of the same level is very inefficient because the host FTL already has a page-level (or sector-level) mapping table, and it is contrary to the original purpose of PASSD (e.g., *log-on-log* reduction). On the one hand, replacing the block-mapping table can reduce memory requirements; on the other hand, as the block-mapping table cannot change the address of a single page, additional data movement is required to maintain the page offset within the block.

Fig. 6 (a) shows an example of handling a write failure in a block-level mapping environment. The initial condition describes that the commands (D0–D4) in the submission queue have sequential page addresses because they are generated by hFTL, and a program failure occurred at page offset 3 of PBN 0. As the block-mapping table must maintain the page offset, the data stored in the block where the write failure occurred are moved, and consequently, the retrying of the write operation is delayed. Therefore, irregular and long delays occur because the number of stored pages varies based on the situation.

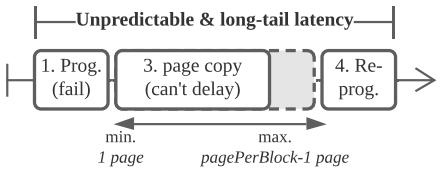
C. KEY IDEA: SHIFTED PAGE OFFSET TABLE

Fig. 7 illustrates our key concept to solve the aforementioned problems. A block-level mapping table is used in the environment; however, to reduce the response time, the failed write command is retried on the first page of a newly allocated block, as shown in step 3. The exceptionally changed address information is temporarily recorded in volatile memory. However, the changed address must be searched for each address translation, and if the number of exceptional addresses increases, the search load increases.

Hence, we introduce a new metric *shift* that records the initial offset of a page in a block. Using *shift*, the write response delay can be minimized because data movement

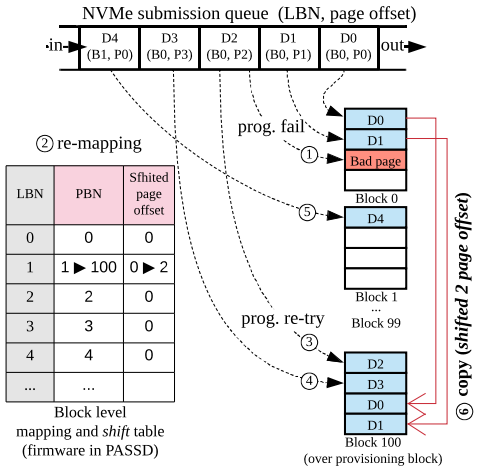


(a) Example of operation sequence

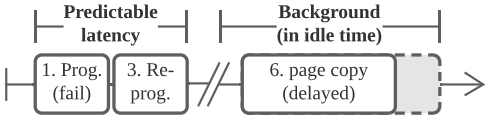


(b) Unpredictable and long-tail latency problem

FIGURE 6. Write reliability policy (re-program after failure) using the block-mapping table.



(a) Example of operation sequence



(b) Predictable latency

FIGURE 7. Write reliability policy using block mapping and shifted page offset table (our main idea).

can be delayed as shown in step 6. Note that, the *shift* can only change the initial offset of a page, so the order of pages in a block must be maintained. In general, since the hFTL issues all write commands with a sequential page address, the order of pages in a block cannot be complicated. However, if the data copying after a program failure proceeds arbitrarily, there is a possibility of complicating the page order. Therefore, copying is allowed only when the number of pages stored in several blocks after a program failure is enough to fill one block, as in step 4, 6. In addition, data copying can be delayed sufficiently, so there is no need to proceed while the host command is waiting. The host command is always processed first, and data is copied during idle time, as in step 5. More detailed implementation methods and operations are described in the next section.

IV. FAULT-TOLERANT PASSD FOR ASYNCHRONOUS WRITE OPERATION

In this section, we introduce a novel PASSD architecture, termed as FT-PASSD, that enables an asynchronous write operation by embedding a reliability policy.

Fig. 8 illustrates the overall architecture and write operation of the FT-PASSD. Unlike the conventional PASSD, its firmware has a built-in program failure manager. Therefore, the host FTL operates in the same manner as in a normal PASSD, but the device reports all write commands as successful immediately after receiving data in the write

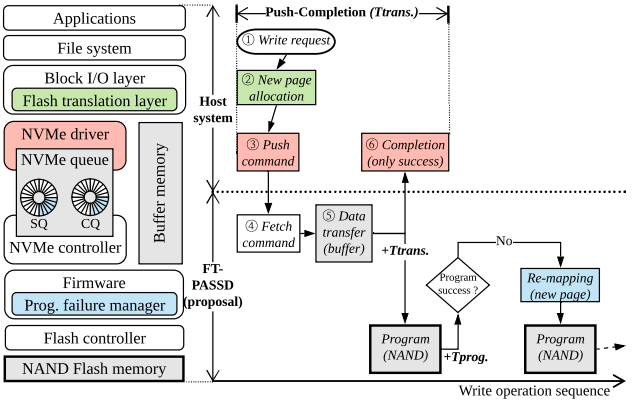


FIGURE 8. Architectural overview of FT-PASSD; our proposal enables reliable asynchronous write operation by allowing the firmware to handle program failures.

buffer. If a program failure occurs in the device, the reliability is guaranteed by internally reallocating the address via the embedded program failure manager.

A. DATA STRUCTURES OF PROGRAM FAILURE MANAGER

Fig. 9 shows the data structures of the proposed program failure manager including a block-level mapping table, *shift* table, and an *open block list*. The description of the data structures are as follows:

The block-mapping and *shift* tables are array-type data structures that record the PBN and *shift* for each LBN, respectively. Initially, most physical blocks are statically mapped 1:1 to each logical block, and the mapping table is changed

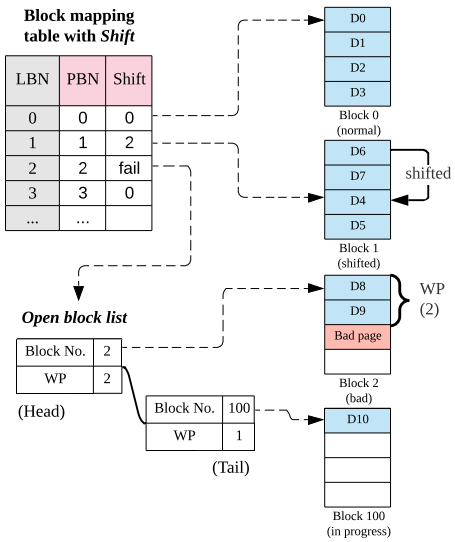


FIGURE 9. Data structures in the program failure manager; block-mapping table with *shift* and *open block list*.

only when a program failure occurs. Therefore, a small number of over-provisioned physical blocks must be ensured. The *shift* was introduced as the main concept of this proposal; when an initial page offset in a block is changed owing to program failures, the changed offset is recorded. Therefore, *shift* is used to convert a logical page offset (LPO) to a physical page offset (PPO).

In addition, since the address-encoding bits of the *shift* are smaller than that required for a block, the total memory requirement of the *shift* table is less than that of the block mapping table. This is because the number of pages per block of a typical NAND flash memory is smaller than the total number of blocks. Moreover, the total number of blocks increases as the capacity increases, but the number of pages in a block is fixed according to the specifications of the NAND flash memory.

The *open block list* is a list-type data structure. It is used to temporarily record pages to be stored in one logical block that is divided into multiple physical blocks owing to program failure; consequently, it is implemented as a list type because program failures may occur continuously. The information recorded in each element is the assigned PBN (*blockNo*) and the count of written pages (*write pointer; WP*).

An example of a snapshot is shown in Fig. 9, a program failure occurred in the third page of LBN 2 (mapped PBN 2) and an *open block list* is created. At the head element of the created list, the PBN 2 and the WP 2 are recorded to provide information on the block were the program failure. Subsequently, the write request that failed is retried to an over-provisioning block (PBN 100). If the re-program operation is successful, a new element is added to the tail of the list to record the changed mapping information that the data of the third page of LBN 2 stored in the first page of PBN 100.

Algorithm 1 Program Failure Handling

Input: *LBN* (logical block no.), *LPO* (logical page offset)

1: /*This function is called when a program failure occurs*/
2: /*"logical" means host access address */
3: /*"physical" means raw NAND flash address */
4: **if** is first program failure in *LBN* **then**
5: /* Create an empty *openblocklist* for LBN */
6: *openBlockList* ← *CreateOpenBlockList*(*LBN*)
7: *failedPBN* ← *blockMapTable*[*LBN*]
8: /* Add first element; add(blockNo, WP) */
9: *openBlockList*.*add*(*failedPBN*, *LPO*)
10: /* To record the occurrence of program failure */
11: *shiftTable*[*LBN*] ← *NULL*(−1)
12: **end if**
13: /* Allocate a new physical block */
14: *newPBN* ← *GetReservedBlockNo*()
15: *openBlockList*.*add*(*newPBN*, 0)

B. ALGORITHM DESCRIPTION

1) ADDRESS TRANSLATION FOR READ/WRITE REQUESTS

Algorithm 1 shows the process of creating an *open block list* and allocating a new block when a program fails. In the case of the first failure occurring in the LBN, an *open block list* is created. Subsequently, the first element containing the PBN and WP wherein the failure occurred is added to the created list. In addition, the *shift* corresponding to the LBN is set to a specific index (e.g., NULL, *page per block* + 1) to confirm that a program failure has occurred. Finally, an element that records a newly allocated PBN and WP is added to the tail of the created (or existing) list. The new PBN is an over-provisioned block number that has not been released to the hFTL, and the initial value of WP is zero.

Algorithm 2 describes the process of translating a logical address of a read/write command to a physical address. First, it is necessary to check whether the *shift* allocated to the LBN has a NULL value. When the *shift* is between 0 and the maximum number of pages in a block (*pagePerBlock*), a physical address for both read and write can be obtained through a block-mapping table and a *shift* table. The detailed address translation process is described in lines 4–6 of Algorithm 2.

If *shift* is NULL, it indicates that a program failure has occurred in the block mapped to the corresponding LBN, and the program failure manager then temporarily allocates a number of PBNs to the LBN by using the *open block list*. To obtain the physical address, the WP is compared with the LPO sequentially from the head of the list using an iterator. If the LPO is less than the WP, the LPO becomes the PPO and the block number of the iterator becomes PBN. Otherwise, WP is subtracted from LPO.

In fact, both read and write addresses can be translated in the above manner. However, because the host issues a write command to sequential page addresses in an LBN, the write address can be known without searching the *open block list*. In other words, the block number and WP obtained

Algorithm 2 Address Translation for Read/Write Request

Input: *LBN* (logical block no.), *LPO* (logical page offset)
Output: *PBN* (physical block no.), *PPO* (physical page offset)

```
/*This function is called for read/write command including retry after program failure*/
shift ← shiftTable[LBN]
if shift ≠ NULL then
    PBN ← blockMapTable[LBN]
    shift ← shiftTable[LBN]
    PPO ← (LPO + shift)%pagePerBlock
else
    openBlockList ← GetOpenBlockList(LBN)
    if is read then
        /* search read address in openBlockList */
        iter ← openBlockList.begin()
        while LPO > iter.WP do
            LPO ← (LPO − iter.WP)
            iter ++
        end while
        PBN ← iter.blockNo
        PPO ← LPO
    else
        /* allocate write address in openBlockList */
        PBN ← openBlockList.tail().blockNo
        PPO ← openBlockList.tail().WP ++
    end if
end if
return PBN, PPO
```

Algorithm 3 Background (in Idle time) Data Migration

Input: *LBN* (logical block no.)

```
/* This function is called only during idle time, and the sum of all WPs in the open block list allocated to LBN must equal the maximum number of pages in a block. */
/* TBN (target block no.), TPO (target page offset) */
/* SBN (source block no.), SPO (source page offset) */
openBlockList ← GetOpenBlockList(LBN)
TBN ← openBlockList.tail().blockNo
TPO ← openBlockList.tail().WP
iter ← openBlockList.begin()
while iter ≠ openBlockList.tail() do
    for i = 0 to iter.WP do
        SBN ← iter.blockNo
        SPO ← i
        Copy from SBN, SPO to TBN, TPO
        TPO ++
    end for
    iter ++
end while
blockMapTable[LBN] ← TBN
shiftTable[LBN] ← openBlockList.tail().WP
Delete openBlockList
```

from the tail of the list become the physical addresses to program. After executing the write command, the WP must be increased by 1.

2) DATA MIGRATION

Algorithm 3 describes the process of searching for an address for copying data. As data movement can be delayed sufficiently, it proceeds only at idle time, and all write commands to be written to one LBN must be completed. The copy starts from the source block at the head of the *open block list* to the target block at the tail of the list (the last allocated normal block), and when the copy of the source block is completed, the next element of the list is accessed using an iterator. When all copies are completed, the PBN in the mapping table corresponding to the LBN is changed to the target block number and the *shift* is updated. Thereafter, the allocated memory can be recovered by deleting the *open block list*.

In the next section, we describe the process of retrying write failures, moving data to a new block, and translating the read address through examples.

C. OPERATION SCENARIOS

1) FAILED PROGRAM RETRY AND ADDRESS UPDATE

Fig. 10 illustrates an example of an operation sequence when a program failure has occurred. Initially, the host system issued write commands (D0, D1, and D2) to LBN 0 (mapped PBN 0), and the write failure occurred in the third page (D2). In steps 2 and 3, the failed write command D2 is stored in the first page of the newly allocated physical block, i.e., PBN 100, and the *open block list* temporarily records the two PBNs and the *write pointer* of each block. The host system is unaware of write failures because all data have been successfully stored. Subsequently, when the next write command D3 is issued, D3 is the address for the LBN 0, where the write failure occurred; therefore, it must be stored in the second page in PBN 100 based on the information in the tail of the *open block list*. As shown in steps 4 and 5, D3 is stored in PBN 100, and the *write pointer* is increased by 1 in the tail of the *open block list*. This process (steps 4 and 5) is continued until the total number of pages stored in the *open block list* is sufficient to fill a single block or until a close command (to stop using an open block) is issued. In the illustrated environment, the block comprises four pages; therefore, this operation is terminated.

In the previous process, the maximum amount of data required to fill a single block was divided between PBN 0 and 100. Although all the data have been stored, D0 and D1 must move to PBN 100. This is because PBN 0, which has failed, has low reliability; additionally, the overhead of verifying the *open block list* for each address translation is high. Steps 6 and 7 show the data movement and mapping update. As data migration can be delayed, it was delayed to idle time and data were sequentially copied from the lowest page offset. Subsequently, the address mapped to LBN 0 was changed to PBN 100, and the *shift* was changed to 2

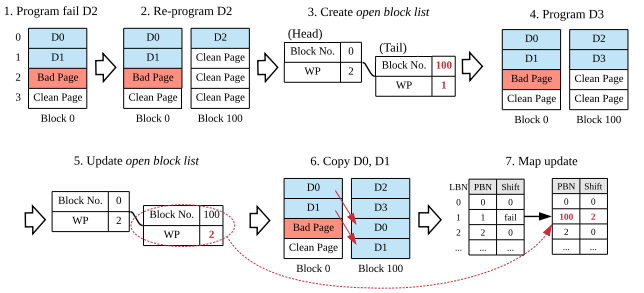


FIGURE 10. Proposed algorithm sequence: an example of program failure occurring in PBN 0.

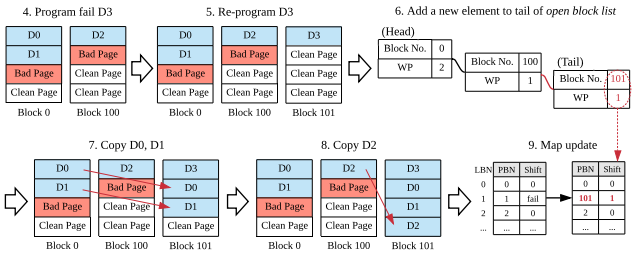


FIGURE 11. Proposed algorithm sequence: an example of program failure occurring in PBN 0 and PBN 100.

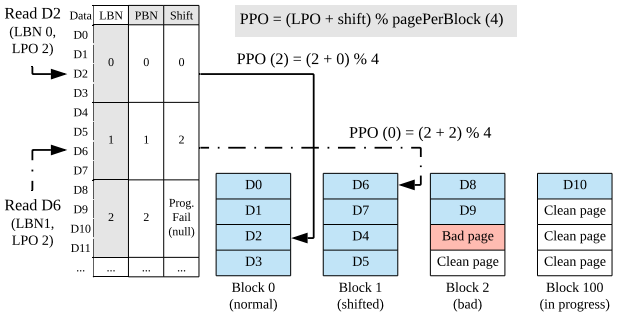
(the number of shifted page offsets). Finally, the *open block list* was deleted to recover the allocated memory, and address translation was enabled using the *shift* table without a search operation.

Fig. 11 presents an example of a situation where multi-program failures occur. It shows the situation where a program failure occurred once again in step 4 of Fig. 10. In this case, as the *open block list* already exists, the existing list is loaded and a new block is added to the tail. The main difference from the previous case is that there are multiple source blocks from which data are copied because the program has failed several times. Therefore, data movement proceeds sequentially from the head of the list, and all other processes proceed in the same manner as previously mentioned.

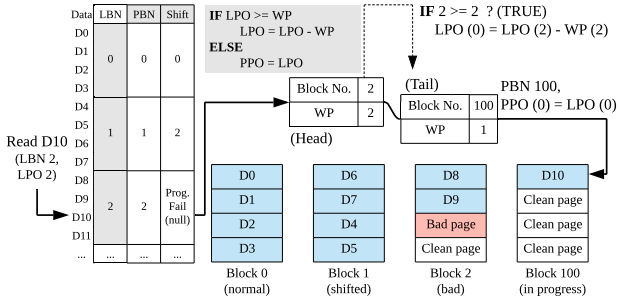
2) READ ADDRESS TRANSLATION

Fig. 12 shows examples of the address translation for the read commands (D2, D6, D10) in the proposed write failure retry technique. As the method depends on the presence or absence of an *open block list*, each process of address translation is explained separately.

Fig. 12 (a) shows address translation in a situation where there is no *open block list*. The PBN can be obtained by translating the LBN using the block mapping table, and the PPO must be converted using the *shift* and LPO. In the read command D2, the PPO is the same as the LPO because the *shift* is not changed (the initial value 0). On the other hand, in the read command D6, as the *shift* of the LBN 1 is 2, the page offset is shifted by two pages due to a previous write failure. In this case, the *shift* is added to the LPO, and the



(a) Not-in-progress program fail after retry algorithm



(b) In-progress program fail after retry algorithm

FIGURE 12. Proposed algorithm sequence: examples of read address translation.

TABLE 3. Evaluation parameters.

Category	Configuration	Value
hFTL (*pblk*)	Parallel unit size	24 GB
	Chunk size	24 MB
	Logical block size	16 kB
NAND Flash	Density (per chip)	24 GB
	Chip array	8 (2ch-4way)
	Channel data bus speed	500 MB/s
	Page size	16 kB
	Block size	24 MB
	Read latency	100 us
	Program latency	1.5 ms
	Erase latency	15 ms
Buffer mngt.	Write buffer size (per way)	256–2048 kB
	Flush (eviction) policy	First-in, First-out

sum divided by the *page per block*. Thereafter, the remainder of the division becomes the PPO.

Fig. 12 (b) depicts the address translation in the presence of an *open block list*. Data movement has not been completed as a write failure occurred, and *shift* is set to a specific index to verify the status (e.g., NULL, *page per block* + 1). In this case, address translation is required to search the *open block list*. The LPO of D10 is 2, indicating that the third page should be read. However, since the third page is not stored in PBN 2, the next element of the list is accessed. Before accessing the next element, the number of pages stored in PBN 2 (*write pointer*) is subtracted from the LPO to ensure that there is skipped pages. Consequently, one page is stored in PBN 100, and because the LPO is 0, the first page of PBN 100 is the address to be read.

TABLE 4. Configurations of PASSD and program failure management.

Evaluation env.	OCSSD	-	FT-PASSD (w/o *shift*)	FT-PASSD (with *shift*)
Write buffer policy (write completion)	Write-through (synchronous)	Write-back (asynchronous)	Write-back (asynchronous)	Write-back (asynchronous)
Program failure manager (additional metadata)	In host FTL	In host FTL	In device FTL (block map table; 13 kB)	In device FTL (block & *shift* map; 24 kB)
Process of program retry after failure	1) write (program) failure report to host FTL, 2) host reissue write request	Nothing *(because already reported completion)*	1) bad block re-mapping (with copy, in foreground), 2) failed program retry	1) failed program retry, 2) bad block re-mapping (with copy, in idle time)
Description	OCSSD 2.0 spec.	Unreliable write operation (excluding experiment)	Without proposed policy	Our proposal

V. METHODOLOGY AND EVALUATION

A. EXPERIMENTAL ENVIRONMENTS

To evaluate the proposed method, we used an in-house trace-based simulator modeling OCSSD and commercial TLC NAND flash memory, as shown in Table 3. The specifications of the PASSD and hFTL are based on OCSSD 2.0, whereas those of the NAND flash memory model are based on a document [52] provided by the manufacturer.

Table 4 presents the characteristics of the proposal and comparison environment. It briefly describes the buffer management and policies related to program failure that have a major influence on the experimental results. In the table, FT-PASSD refers to an environment wherein asynchronous write completion is enabled by embedding a program failure manager, and this is distinguished from OCSSD that uses the synchronous write completion. In addition, the proposed built-in program failure manager using the shifted page offset is denoted as *with shift*, and the program failure manager that operates only with simple block mapping is denoted as *without (w/o) shift*.

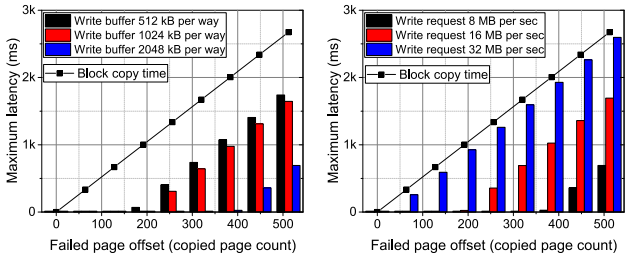
The memory requirement was calculated based on the experimental environment (8192 blocks, 1536 pages in a block). Block mapping for address encoding requires 13 bits per block, and the shift table requires 11 bits per block, i.e., the block-mapping table and the *shift* table require 13 kB and 11 kB of memory, respectively. In addition, assuming that page mapping is built into the environment, the number of pages is 12 M; thus, approximately 40 MB of memory is required.

The performance was measured is the time required by the PASSD to deliver a completion message to a command issued by the FTL located on the host system; it is noteworthy that the time recorded in the trace was not used for the performance measurement.

B. EVALUATION RESULTS

1) MICRO-BENCHMARK

We conducted performance analysis using micro-benchmarks to precisely analyze the performance of the assumed environments. All micro-benchmarks comprise only write commands, and each command includes an interval of 1 *s*. Further, each benchmark is divided into sequential and partial writes depending on the size of a single request.



(a) Effect of write buffer size (512–2048 kB per way), 8 MB per command

(b) Effect of write request size (8–32 MB per command, write buffer 1024 kB)

FIGURE 13. Maximum latency for failed program retry using the block-mapping table; FT-PASSD (w/o shift).

Fig. 13 shows the long latency problem in FT-PASSD (w/o shift) caused by multiple page copies when changing the page address of a failed program operation with only the block-mapping table. The page copying due to program failure occurs as many times as the number of pages stored in the block. Therefore, the page copy time displayed as a solid line is proportional to the failed page offset. However, in asynchronous write completion, even when the NAND flash memory is busy, the write command can be processed without delay using the write buffer; thus, the page copy time is not fully reflected in the maximum latency.

Fig. 13 (a) shows the effect of the write buffer size, and the larger the buffer, the smaller is the maximum response time; (b) shows the effect of the write command size, and the larger the command size, the larger is the maximum response time. That is, even for a program failure manager that operates only with block mapping, the response time can vary significantly depending on the characteristics of the workload and the size of the buffer. However, it is difficult to determine a sufficient buffer size to completely eliminate the response delay because program failure can occur continuously. Moreover, since the buffer is a volatile memory, increasing the buffer size increase the risk of data loss caused by sudden power failure.

Fig. 14 shows the effects of the experimental environment and workload patterns on average and maximum latency. Sequential writes are set such that the buffer is always full, and partial writes are set to flush the buffer during the time interval between requests. In addition, in all environments, program failures occur with an arbitrary low probability, that

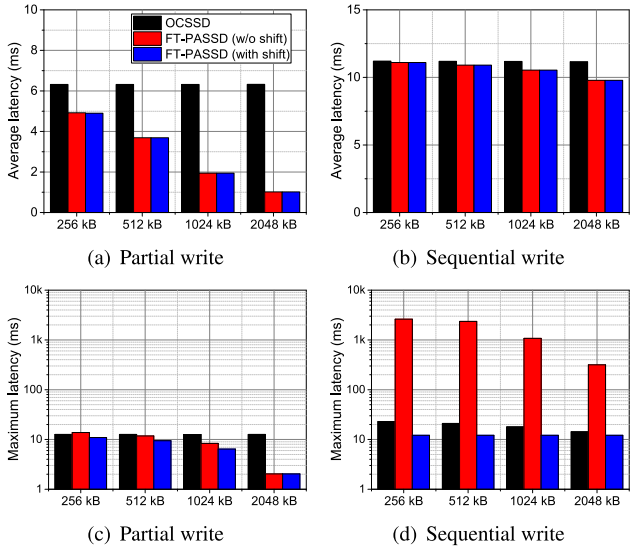


FIGURE 14. Experimental results of changing buffer size (256–2048 kB per way) and uniform workload (sequential write size 16 MB per request, partial write size 1 MB per request, time interval 1 s per request).

is, they occur very few times; thus, the average latency is hardly affected.

Fig. 14 (a) shows the average response time in the case of partial write workload. For OCSSD, the average response time is constant, whereas for FT-PASSD, the response time decreases as the size of the buffer increases. OCSSD provides a constant response time regardless of the buffer size because all write commands are completed after the NAND flash operation is completed. On the contrary, as the FT-PASSD reports completion immediately after receiving the write data in the buffer, the larger the buffer, the shorter is the response time. Particularly, a partial workload with a small write request size has sufficient time to empty the buffer, thereby further reducing the response time of asynchronous write completion. Fig. 14 (b) shows the average response time in the case of sequential workload. Both OCSSD and FT-PASSD provide similar response times. Unlike partial workloads, because sequential workloads do not have sufficient time to clear buffers, the latency reduction by asynchronous write completion rarely appears.

Fig. 14 (c) and (d) show the maximum latency for each workload. In particular, (d) shows that the maximum response time of FT-PASSD (w/o shift) is significantly higher than that of other environments. This increase in latency occurs because the device cannot process write commands during the data copy time in the program failure manager process with only the block-mapping table. However, the response time did not increase in (b) because the buffer was able to sufficiently accommodate the host write commands during the data copy process. In other words, the process of copying blocks does not always lead to an increase in response time, but there is a critical risk of a large response delay (freezing) of up to several seconds.

In summary, OCSSD has the disadvantage of low performance for a partial write workload, and FT-PASSD (w/o shift) involves the risk of a freezing for a sequential write workload. The proposed FT-PASSD (with shift) has the advantage of providing high response performance for all workload types and does not suffer from response delay due to program failure.

TABLE 5. Trace list.

Trace	Avg. size (kB)		N kB size distributions (%)				
	Read	Write	$\leqq 4$	$\leqq 8$	$\leqq 16$	$\leqq 32$	> 32
hm_0	8.1	9.3	77.1	7.8	5.5	1.6	8.0
prn_0	22.9	11.3	79.6	2.8	5.8	1.5	10.3
prn_1	22.5	11.7	71.4	7.7	11.4	2.6	6.8
proj_2	41.6	48.8	22.9	3.2	2.7	4.3	66.9
prxy_0	8.4	7.0	86.5	4.7	1.6	0.7	6.5
src1_1	35.8	14.7	66.3	7.5	5.2	5.1	15.9
usr_1	52.7	15.3	68.3	7.8	4.0	2.7	17.2
usr_2	50.8	13.9	67.3	10.4	4.3	3.9	14.1

2) BLOCK I/O TRACE

The workloads used in this experiment were block I/O traces of the Microsoft Research Center [53], which is widely used for secondary storage performance evaluation; detailed information is summarized in Table 5.

Fig. 15 shows the experimental results using the block I/O traces. As the probability of program failure of the actual NAND flash memory is extremely low, we performed the experiment under the assumption that only one program failure occurred at an arbitrary point in time while processing the entire trace. Fig. 15 (a) shows the average response time of each workload and environment. As the secondary storage I/O access in a real computing system involves considerable partial writes and idle time, the response time of FT-PASSD is significantly reduced (45.6% on average) compared to that of OCSSD for all workloads. In particular, workloads with a high proportion of partial write commands showed a significant effect of improving latency (88% of *src*1_1, 78% of *prn*_1), whereas workloads with a high proportion of sequential writes showed relatively little improvement (49% of *proj*_2).

Fig. 15 (b) shows the maximum latency cased by a program failure. As mentioned above, program failure was set to occur once in all environments; however, the only environment wherein a long response delay occurred is FT-PASSD (w/o shift) in some workloads. The reason that a long response delay did not occur in OCSSD is that the failed program is not retried. In contrast, the reason of FT-PASSD (with shift) is that the copying of additional data due to program failure was delayed by using the shifted page offset table.

Fig. 16 presents some information collected at the time of program failure to explain the cause of long latency occurring in the previous experimental results. (a) shows the number of pages copied by FT-PASSD after a program failure and the time it took to retry the failed program. In FT-PASSD (w/o shift), considerable time was wasted until the program was retried because the page was copied before retrying the program. Even in *prn*_0 where the fewest page copies occurred, it took approximately 250 *ms* or more to finally

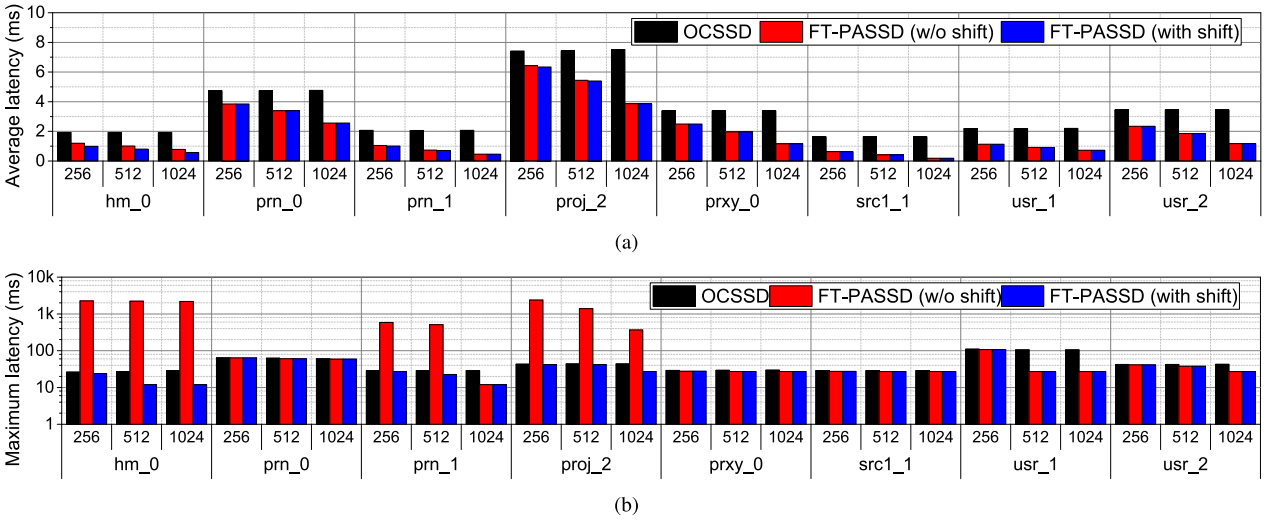


FIGURE 15. Experimental results of changing buffer size (256–1024 kB per way) and block I/O workload.

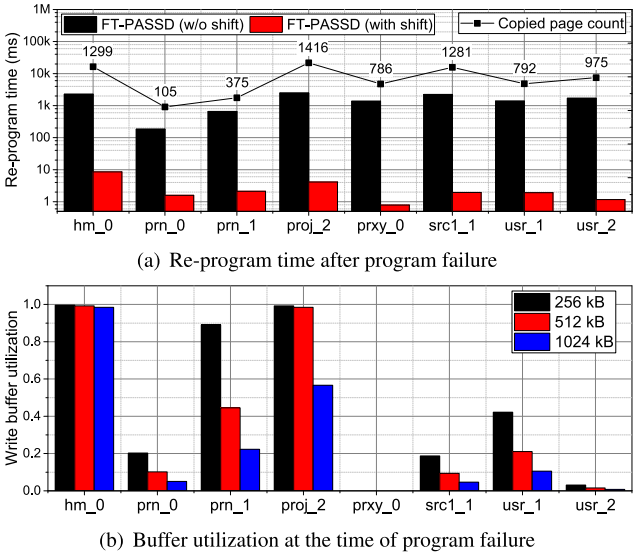


FIGURE 16. Analysis of long latency due to program failure.

complete the program. On the contrary, as FT-PASSD (with shift) immediately retries the program, all operations were completed within 10 *ms* regardless of the number of pages to be copied.

Figure 16 (b) shows the buffer utilization when a program failure occurs. In workloads with a particularly long response delay (*hm*_0, *prn*_1, *proj*_2), buffer utilization is high at the time of program failure. This explains why the long response delay occurred in only some workloads even though many page copies occurred in most workloads. In addition, since the utilization of the buffer varies from time to time in real workloads, the result of the experiment can be completely different depending on the time when a program failure occurs. In conclusion, the proposed write reliability policy solves the problem of unpredictable and long response delays that are caused by program failure during asynchronous write operations in FT-PASSD.

VI. CONCLUSION

In this study, it was experimentally verified that the synchronous write operation of the existing PASSD caused performance degradation; hence, a PASSD with a built-in reliability policy was proposed to enable an asynchronous write operation. Our proposed policy can ensure write reliability without a response delay, thereby solving the data loss problem that may be caused by asynchronous write operations. In addition, because the additional hardware resources required are only a small amount of volatile memory, the proposed program failure manager appears to be applicable to most existing PASSDs. The experimental results indicated an average reduction of 45.6% in write response time in certain workloads. Moreover, the proposed write reliability policy effectively eliminates the long response delay caused by program failure.

ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] H. Kim, S.-J. Ahn, Y. G. Shin, K. Lee, and E. Jung, "Evolution of NAND flash memory: From 2D to 3D as a storage market leader," in *Proc. IEEE Int. Memory Workshop (IMW)*, May 2017, pp. 1–4.
[2] P. Cappelletti, "Non volatile memory evolution and revolution," in *IEDM Tech. Dig.*, Dec. 2015, pp. 10.1.1–10.1.4.
[3] J. Kim, P. Park, J. Ahn, J. Kim, J. Kim, and J. Kim, "SSDcheck: Timely and accurate prediction of irregular behaviors in black-box SSDs," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 455–468.
[4] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and SSD deployments," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 263–276.
[5] Y. T. Jin, S. Ahn, and S. Lee, "Performance analysis of NVMe SSD-based all-flash array systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2018, pp. 12–21.
[6] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, Jul. 2018. Accessed: Oct. 27, 2020. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/koh

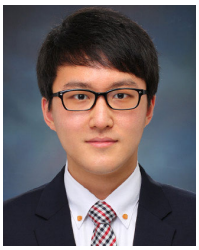
[7] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," *ACM Trans. Storage*, vol. 13, no. 3, pp. 1–26, 2017.
[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, vol. 8, 2008, pp. 57–70.
[9] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 524–535.
[10] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *Proc. 2nd Workshop Interact. NVM/Flash Operating Syst. Workloads (INFLOW)*, Oct. 2014. Accessed: Oct. 27, 2020. [Online]. Available: https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang
[11] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 257–270.
[12] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD," in *Proc. 55th Annu. Design Autom. Conf.*, Jun. 2018, pp. 1–6.
[13] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *J. Syst. Archit.*, vol. 55, nos. 5–6, pp. 332–343, May 2009.
[14] Q. Luo, R. C. C. Cheung, and Y. Sun, "Dynamic virtual page-based flash translation layer with novel hot data identification and adaptive parallelism management," *IEEE Access*, vol. 6, pp. 56200–56213, 2018.
[15] J. Li, X. Xu, B. Huang, J. Liao, and X. Peng, "Frequent pattern-based mapping at flash translation layer of solid-state drives," *IEEE Access*, vol. 7, pp. 95233–95239, 2019.
[16] C.-H. Wu, D.-Y. Wu, H.-M. Chou, and C.-A. Cheng, "Rethink the design of flash translation layers in a component-based view," *IEEE Access*, vol. 5, pp. 12895–12912, 2017.
[17] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai, "Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime," in *Proc. IEEE 30th Int. Conf. Comput. Design (ICCD)*, Sep. 2012, pp. 94–101.
[18] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Crista, O. S. Unsal, and K. Mai, "Error analysis and retention-aware error management for nand flash memory," *Intel Technol. J.*, vol. 17, no. 1, pp. 140–164, May 2013.
[19] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The Linux open-channel SSD subsystem," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 359–374.
[20] A. Batwara, "Leveraging host based flash translation layer for application acceleration," in *Proc. Flash Memory Summit*, Aug. 2012. Accessed: Oct. 27, 2020. [Online]. Available: http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TB11_Batwara.pdf
[21] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for web-scale Internet storage systems," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 471–484, 2014.
[22] M. Bjørling, "Zone append: A new way of writing to zoned storage," in *Proc. Linux Storage Filesyst. Conf. (Vault)*, Feb. 2020. Accessed: Oct. 27, 2020. [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/vault20_slides_bjorling.pdf
[23] M. Bjørling, "Open-channel solid state drives," *Vault*, vol. 12, p. 22, Mar. 2015.
[24] M. Bjørling, "From open-channel SSDs to zoned namespaces," in *Proc. Linux Storage Filesyst. Conf. (Vault)*, Feb. 2019. Accessed: Oct. 27, 2020. [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/nsdi19_slides_bjorling.pdf
[25] D. Le Moal and T. Yao, "Zonefs: Mapping the POSIX file system interface to zoned block device accesses," in *Proc. Linux Storage Filesyst. Conf. (Vault)*, Feb. 2020. Accessed: Oct. 27, 2020. [Online]. Available: https://www.usenix.org/system/files/vault20_slides_moal.pdf
[26] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving read performance by isolating multiple queues in NVMe SSDs," in *Proc. 11th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2017, pp. 1–6.
[27] H. Qin, D. Feng, W. Tong, J. Liu, and Y. Zhao, "QBLK: Towards fully exploiting the parallelism of open-channel SSDs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1064–1069.
[28] J. Jhin, H. Kim, and D. Shin, "Optimizing host-level flash translation layer with considering storage stack of host systems," in *Proc. 12th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2018, pp. 1–4.
[29] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: Introducing multi-queue SSD access on multi-core systems," in *Proc. 6th Int. Syst. Storage Conf.*, 2013, pp. 1–10.
[30] S. Kim, Y. Kang, and D. Shin, "Fsync-aware multi-buffer FTL for improving the fsync latency with open-channel SSDs," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Aug. 2019, pp. 1–2.
[31] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDACache: An integration of device and application for flash-based key-value caching," *ACM Trans. Storage*, vol. 14, no. 3, pp. 1–32, Nov. 2018.
[32] G. S. Choi and B.-W. On, "Study of the performance impact of a cache buffer in solid-state disks," *Microprocess. Microsyst.*, vol. 35, no. 3, pp. 359–369, May 2011.
[33] Y. Yao, X. Kong, J. Zhou, X. Xu, W. Feng, and Z. Liu, "An advanced adaptive least recently used buffer management algorithm for SSD," *IEEE Access*, vol. 7, pp. 33494–33505, 2019.
[34] J. Niu, J. Xu, and L. Xie, "Hybrid storage systems: A survey of architectures and algorithms," *IEEE Access*, vol. 6, pp. 13385–13406, 2018.
[35] B. Schroeder, A. Merchant, and R. Lagisetty, "Reliability of NAND-based SSDs: What field studies tell us," *Proc. IEEE*, vol. 105, no. 9, pp. 1751–1769, Sep. 2017.
[36] A. Goda and K. Parat, "Scaling directions for 2D and 3D NAND cells," in *IEDM Tech. Dig.*, Dec. 2012, pp. 1–2.
[37] P. Cappelletti, "Non volatile memory evolution and revolution," in *IEDM Tech. Dig.*, Dec. 2015, pp. 1–10.
[38] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proc. IEEE*, vol. 105, no. 9, pp. 1666–1704, Sep. 2017.
[39] R. Ma, F. Wu, M. Zhang, Z. Lu, J. Wan, and C. Xie, "RBER-aware lifetime prediction scheme for 3D-TLC NAND flash memory," *IEEE Access*, vol. 7, pp. 44696–44708, 2019.
[40] S. J. Kwon, "Address translation layer for byte-addressable non-volatile memory-based solid state drives," *IEEE Access*, vol. 7, pp. 73207–73214, 2019.
[41] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in NAND flash memories," in *Proc. IEEE Int. Rel. Phys. Symp.*, Apr. 2008, pp. 9–19.
[42] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error-correction codes in NAND flash memory," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 2, pp. 429–439, Feb. 2011.
[43] Y. Cai, Y. X. Luo, S. Ghose, E. F. Haratsch, K. Mai, and O. Mutlu, "Read disturb errors in MLC NAND flash memory," *IPSI BgD Trans. Internet Res.*, vol. 14, no. 2, pp. 82–93, Jul. 2018.
[44] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2012, pp. 521–526.
[45] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[46] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV performance with open-channel SSDS," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 1–19, 2017.
[47] Y. Lu, J. Zhang, Z. Yang, L. Pan, and J. Shu, "OCStore: Accelerating distributed object storage with open-channel SSDs," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 271–281.
[48] Y. Lu, J. Shu, and J. Zhang, "Mitigating synchronous I/O overhead in file systems on open-channel SSDs," *ACM Trans. Storage*, vol. 15, no. 3, pp. 1–25, Aug. 2019.
[49] K. Kim, E. Lee, and T. Kim, "HMB-SSD: Framework for efficient exploiting of the host memory buffer in the NVMe SSD," *IEEE Access*, vol. 7, pp. 150403–150411, 2019.
[50] C. N. Y. Avila, J. Hsu, A. K.-T. Mak, J. Chen, and G. S. Shah, "Program failure handling in nonvolatile memory," U.S. Patent 8 132 045, Mar. 6, 2012.
[51] S. Natarajan and W. Tran, "Internal copy to handle nand program fail," U.S. Patent 10 658 056, May 19, 2020.
[52] *3D NAND Flash Memory Specification*, Micron Technol., Boise, ID, USA, 2016.
[53] (2008). *SNIA Trace Repository, MS Server Block I/O Trace*. Accessed: Dec. 9, 2019. [Online]. Available: http://iotta.snia.org/tracetypes/

DAEYONG LEE received the B.S. degree from the School of Electronic Engineering, Soongsil University, Seoul, South Korea, in 2014, and the M.S. degree from the Department of Electronics and Computer Engineering, Hanyang University, Seoul, in 2017, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include embedded systems and NAND flash memories.

JAEWOOK KWAK received the B.S. and M.S. degrees in electronics and computer engineering from Hanyang University, Seoul, South Korea, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree in electronics and computer engineering.

His research interests include high-performance computing, computer architecture, and low-power systems.



GYEONGYONG LEE received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include embedded computing and NAND flash memories.

MOONSEOK JANG received the B.S. degree in electronic engineering from Hanyang University, Seoul, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include computer architecture, embedded systems, and flash memory storage.

JOONYONG JEONG received the B.S. degree from the Department of Information System, Hanyang University, Seoul, South Korea, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include NAND flash-based storage systems, databases, and key-value stores.

KEXIN WANG (Student Member, IEEE) received the B.S. degree from the School of Optoelectronic Engineering, Changchun University of Science and Technology, in 2017. She is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hanyang University, Seoul, South Korea.

Her research interest includes high-performance solid state drive architecture.

JUNGWOOK CHOI (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2008 and 2010, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign, USA, in 2015. He worked at the IBM T. J. Watson Research Center as a Research Staff Member, from 2015 to 2019. He is currently an Assistant Professor with Hanyang University, South Korea. His main research interest includes efficient implementation of deep learning algorithms. He has received several research awards, such as the DAC 2018 Best Paper Award and has actively contributed to academic activities, such as being a Technical Program Committee of DATE 2018–2020 (Co-Chair) and DAC 2018-2020 and a Technical Committee (DiSPS) of the IEEE Signal Processing Society.

YONG HO SONG (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1989 and 1991, respectively, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 2002.

He is currently a Professor with the Department of Electronic Engineering, Hanyang University, Seoul, and a Senior Vice President of Samsung Electronics Company Ltd. His current research interests include system architecture and software systems of mobile embedded systems, including SoC, NoC, multimedia on multicore parallel architecture, and NAND flash-based storage systems.

Prof. Song has served as a Program Committee Member in several prestigious conferences, including the IEEE International Parallel and Distributed Processing Symposium, the IEEE International Conference on Parallel and Distributed Systems, and the IEEE International Conference on Computing, Communication, and Networks.