Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems

JAEWOOK KWAK, SANGJIN LEE, KIBIN PARK, JINWOO JEONG, and YONG HO SONG,
Hanyang University, Korea

As semiconductor technology has advanced, many storage systems have begun to use non-volatile memories as storage media. The organization and architecture of storage controllers have become more complex to meet various design requirements in terms of performance, response time, quality of service (QoS), and so on. In addition, due to the evolution of memory technology and the emergence of new applications, storage controllers employ new firmware algorithms and hardware modules. When designing storage controllers, engineers often evaluate the performance impact of using new software and hardware components using software simulators. However, this technique often yields limited evaluation accuracy because of the difficulty of modeling complex operations of components and the interactions among them. In this article, we present a reconfigurable flash storage controller design that serves as a rapid prototype. This design can be synthesized into a field-programmable gate array device and used in a realistic performance evaluation environment. We show the usefulness of our design by demonstrating the performance impact of design parameters.

CCS Concepts: • **Information systems → Storage architectures**; • **Computer systems organization →** *Embedded systems*; • **Hardware** → *External storage*;

Additional Key Words and Phrases: Flash memory, storage system, solid state drive (SSD), flash translation layer (FTL)

ACM Reference format:
Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage* 16, 3, Article 15 (July 2020), 35 pages.
https://doi.org/10.1145/3385073

1 INTRODUCTION

Not AND flash storage systems are widely used in mobile applications and various computer systems due to the attractive aspects of NAND flash memory, including high performance, low power consumption, high shock resistance, and small form factor. They have been replacing traditional mechanical storage devices in many applications. The high cost overhead of flash memory has been offset by the increase of storage performance. Furthermore, the adoption of quad-level cell (QLC) and three-dimensional (3D) technologies has decreased overhead even more.

This work was supported by the R&D program of MOTIE/KEIT. [10077609, Developing Processor-Memory-Storage Integrated Architecture for Low Power, High-Performance Big Data Servers].
Authors' addresses: J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, Hanyang University, 222 Wangsimni-ro, Seoul, 04763, Korea; emails: {jwkwak, sjlee, kbpark, jwjeong, yhsong}@enc.hanyang.ac.kr.

1553-3077/2020/07-ART15 $15.00
https://doi.org/10.1145/3385073

Each flash storage device has a storage controller, to orchestrate internal operations related to media access, host interaction, status management, and the like. The storage controller is yet another embedded system consisting of hardware components and firmware modules: the former includes system components such as processors, buses, and memory controllers, while the latter implements complicated algorithms and policies related to storage operations. The operations of hardware and software components should be well tuned to meet different design requirements such as performance, reliability, and lifetime. Moreover, the emergence of new NAND flash technologies requires the controller design to evolve to deal with new challenges.

Once designed, a storage controller is evaluated to see whether it satisfies design requirements. One of the common tools used for this evaluation is a software simulator, which allows for easy implementation and fast analysis [10, 15, 19, 20, 32, 36]. However, simulations sometimes produce less accurate results because of the absence of accurate hardware models, the use of limited input sets, and inaccurate assumptions regarding the interactions between hardware models. Amber [10], which is a state-of-the-art simulator that can work in conjunction with gem5 [2] to improve the accuracy of the interactions between the storage system and the host system, has a maximum accuracy gap of 36% for real devices.

To overcome these limitations, some researchers have used a hardware storage model [4, 7, 11, 22, 29, 33, 35]. This model has its own hardware resources distinct from the host system to provide input/output (I/O) functions to the host system by using nonvolatile memory devices or hardware emulators. However, there are restrictions on each model. Lee et al. [22] and Wei et al. [35] employ outdated host interface protocols that provide low data bandwidth, which prevents sufficient workloads from being exposed to a storage system. Davis et al. [7] and Cai et al. [4] propose a software hierarchy, which differs from the traditional version and produces less accurate results. Seshadri et al. [29] emulate the phase change memory, which is not suitable for flash storage research. Gu et al. [11] and Torabzadehkashi et al. [33] do not allow for the modification of the controller hardware and software algorithms handling I/O requests. Furthermore, because all these groups do not currently provide their source code to the public, other research groups cannot utilize the models.

Recently, an open source platform called *Jasmine Open* solid state drive (SSD) [28] was introduced. Although it uses a commercial system-on-chip (SoC) as a storage controller, it allows its firmware to be freely modified. The SoC includes a serial ATA serial advanced technology attachment (SATA) 2.0 host interface, NAND flash controllers, and an error-correction code (ECC) engine. Because the controller is industry-proven, many people use the platform for firmware development and verification tasks. However, it leaves no room for modification of hardware functionality or parameters. Furthermore, it is impossible to support emerging flash devices, such as 3D triple-level cell (TLC) and quad level cell (QLC) memories.

In this article, we present a new reconfigurable storage controller design for rapid storage prototyping, called *Cosmos+ OpenSSD*. Our design includes the hardware and software components required to build a storage controller. Its design runs on a field-programmable gate array (FPGA), and it is very easy to implement and evaluate design deviations. Our design inherits features from its predecessor, *Cosmos OpenSSD* [31], and we have extended it to meet the following key design requirements:

— The prototype design is *reconfigurable* so that it can be used for architecture exploration and performance evaluation. The hardware components are easily expanded, modified, and replaced. To ensure this, the hardware components are designed to be modular, and their operations are managed by software-based scheduling. Details regarding the hardware components and scheduling are discussed in Section 3. Simultaneously, an FPGA platform is used to evaluate the performance and efficiency of firmware algorithms. However,

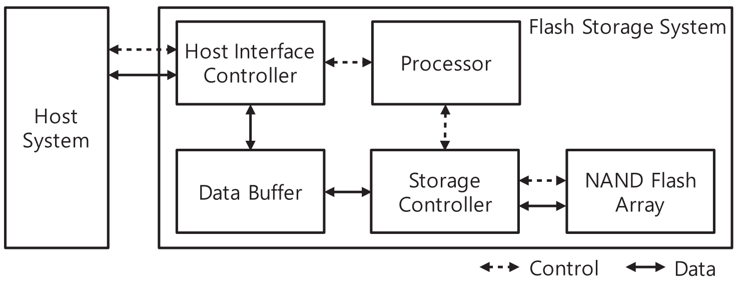

Fig. 1. Hardware architecture of a flash storage system.

considering the limited capacity of FPGA devices, we seek an efficient implementation of resource-hungry hardware components (e.g., a non-volatile memory express (NVMe) protocol controller and error-correction code modules). To demonstrate that the prototype design can be reconfigured for various studies, case studies are described in Section 7.

—The design runs with *real application workloads*. Therefore, it supports the high-bandwidth host interface, including the NVMe interface protocol [8]. Moreover, many operating systems include NVMe device drivers and support new attempts to host storage functions for them [3, 13, 26].

—The design is *scalable* in hardware configurations so that it can support multi-channel, multi-way organizations to provide data access parallelism. Moreover, the I/O performance of each NAND flash device is far less than the interface bandwidth. To provide high aggregated bandwidth, storage devices accommodate multiple flash devices and allow them to be accessed in parallel. There are many approaches to utilizing data access parallelism to boost performance [6, 9, 14, 24, 37]. The data path to the NAND array is also parallelizable. To facilitate channel-way scalability, the address translator uses location vectors. This is discussed in Section 3.3.

—The design provides an efficient way of *monitoring internal events and operations* and *collecting relevant statistics* (e.g., channel utilization and flash operation latency). The measured statistics can be temporarily stored until they are fetched by the host system. The data can be delivered on a sideband channel (e.g., universal asynchronous receiver/transmitter (UART) and universal serial bus (USB)) to avoid unnecessary interference of ordinary storage access from a host system.

—The full source codes of hardware and software design are *open to public* so that people use them as a baseline design and modify components and functions to build their own controllers.

2 BACKGROUND AND RELATED WORK

2.1 Flash Storage Organization

Many flash storage systems have similarities of organization despite having differences of performance and reliability. Figure 1 shows common hardware components of a flash storage system, including microprocessors, a host interface controller, a data buffer, flash array controllers, and NAND flash devices.

Microprocessors are core components for executing firmware responsible for storage operations and host interactions. The internal operation of storage systems is subject to a flash translation layer (FTL), which makes a set of NAND flash devices act as storage media. In fact, the FTL plays a crucial role in storage systems because it is used to hide many weaknesses of NAND flash devices,

Table 1. OpenSSD Feature Comparison

	Jasmine	Cosmos	Cosmos+
Host device driver	OS native	Special	OS native
Host interface	AHCI over SATA	AHCI over PCIe	NVMe over PCIe
Storage controller	SoC	FPGA	FPGA
NAND flash interface	Asynchronous	Synchronous	Toggle
Maximum configuration	4ch 8-way	4ch 4-way	8ch 8-way
Flash operation scheduling	Hardware	Hardware	Software

ensure data integrity, and boost storage performance. Because it performs several important tasks at the same time, such as address translation, garbage collection, wear-leveling, bad block management, sudden power-off recovery, and low-level interface, its tasks are sometimes multi-threaded on multiple microprocessors [17, 18].

The host interface controller logically and physically connects a flash storage system to a host system. Recently, NVMe has attracted more attention from storage designers because of its high bandwidth and performance scalability. The interface bandwidth easily increases beyond tens of Gb/s, which requires storage systems to be well-designed to support sufficient parallelism. In addition, many scale-out storage systems employ a fabric-based network of NVMe-based storages, called *NVMe-oF*, to implement high capacity and performance in storage service.

The data buffer is provisioned to fill the performance gap between a host interface and flash devices. It temporarily holds read and write data until they are delivered to their destination. In particular, it absorbs write data from a host to hide long flash operation latency and to sometimes escape from on-going housekeeping operations. It also serves as a space to collect small-sized individual writes to go into the same page. Another use of the buffer is as a workplace for internal housekeeping operations, such as garbage collection: whenever a storage system relocates data on the flash media, any valid data stay in the buffer until they are relocated.

The flash array controllers orchestrate the operation of flash memory devices based on the requests from the host and FTL. They generate control signals and data to flash memory devices, initiate designated operations, and verify the successful completion of the operations. The use of multi-channel, multi-way configurations in storage systems often requires the use of multiple flash array controllers, each responsible for a subset of flash devices.

2.2 Jasmine and Cosmos OpenSSD

Currently, there are many commercial SSDs on the market. They mostly implement the control functions in SoCs. The design details of controllers, including internal functions and algorithms, are invisible to others. Moreover, there have been few SSD platforms on which researchers could perform architecture exploration and performance evaluation.

The first open source SSD platform appeared in 2011 and is called *Jasmine OpenSSD*. It uses a commercial SoC as a controller hardware, but allows its firmware to be freely changed. Since its creation, it has been used by many research groups in academic and research institutes. However, it is impossible to modify the controller hardware, which significantly restricts the use of the SSD platform.

More recently, another open source SSD design, *Cosmos OpenSSD*, has emerged, whose hardware and software design is allowed to be modified. Unfortunately, it lacks the support for the NVMe host interface protocol. Some features of its design are summarized in Table 1 and compared with other designs including the one presented in this article.

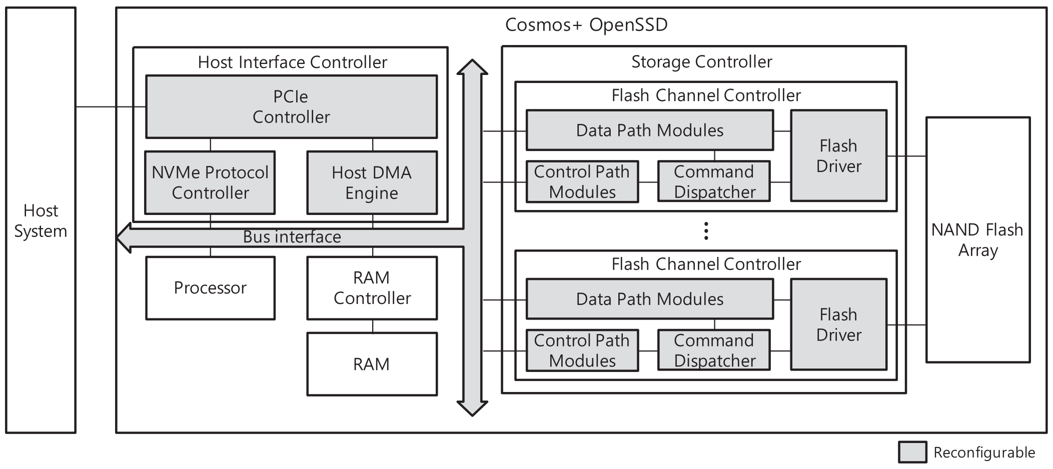

Fig. 2. Hardware architecture of Cosmos+ OpenSSD.

3 COSMOS+ OPENSSD ARCHITECTURE

In this section, we describe the Cosmos+ OpenSSD hardware design and firmware operations. Two important hardware function blocks in our design are a host interface controller and a storage controller. Figure 2 shows the blocks and their internal components. The firmware is responsible for controlling the function blocks.

3.1 Host Interface Controller

The host interface controller provides a communication channel between a host system and Cosmos+ OpenSSD. It runs the NVMe interface protocol over peripheral component interconnect express (PCIe) bus; the commands, data, and control information of the NVMe protocol are encapsulated into PCIe packets. This controller consists of three building blocks: a PCIe controller, an NVMe protocol controller, and a host direct memory access (DMA) engine.

The PCIe controller is responsible for exchanging control and data packets over serial communication lanes between a host system and the Cosmos+ OpenSSD. Essentially, it is responsible for lane initialization, flow control, end-to-end data transmission, and the other mandatory tasks defined in the PCIe bus protocol. Owing to the increasing use of PCIe bus in FPGA designs, many FPGA devices have already integrated the PCIe block as a built-in intellectual property (IP), which saves design effort on the low-level PCIe protocol layer. The actual implementation of the PCIe controller in the Cosmos+ OpenSSD is a simple protocol wrapper to the built-in PCIe block.

The NVMe protocol controller communicates with the host system through the PCIe controller. The NVMe interface standard allows a host system to use up to 64K I/O submission queues with a depth of 64K. The NVMe protocol controller in the device retrieves commands from the submission queues and puts them into a local host command queue, as shown in Figure 3. It maintains the data structures for communication, such as a door-bell register, for each submission queue. When it fetches the commands from submission queues, it may assign higher priorities to submission queues so that it gets more commands from specific queues. The default configuration of baseline design uses eight I/O submission queues and apply a round-robin policy to them. The number of submission queues can be increased up to 64K. Once fetched, a host command is translated into different commands and operations, as shown in Figure 4. The conversion process will be further discussed in Section 3.3.

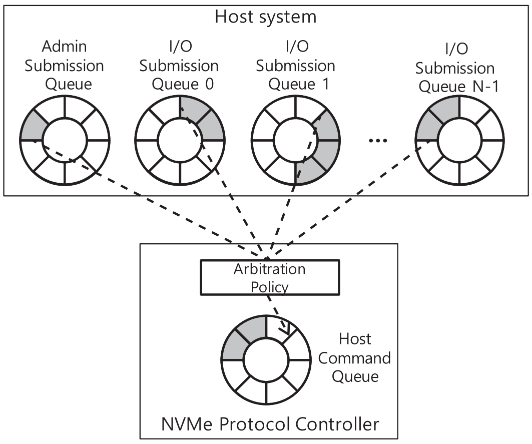

Fig. 3. Host command fetch process.

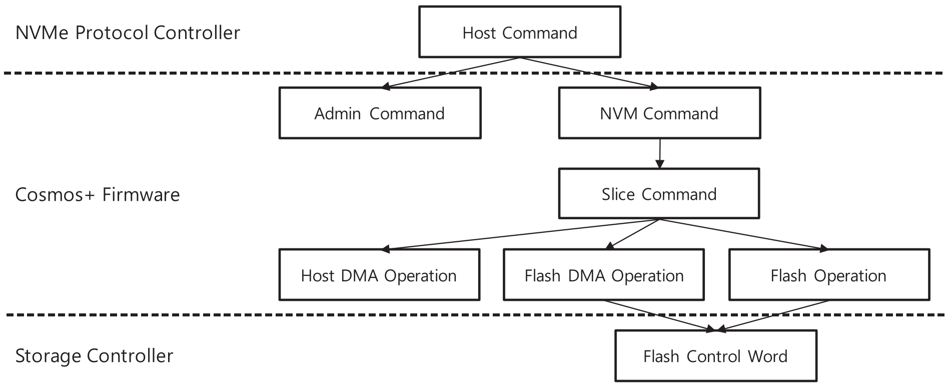

Fig. 4. Command transformation process.

The host DMA engine manages data transfer between the host memory and the data buffer in the storage. It directly accesses data in the physical memory of the host system using a physical region page (PRP) entry, using a fixed-size memory range as a unit of data transmission. The data transfer using the PRP has the advantage of simplifying the structure of the host DMA engine because the size of data to be transmitted is constant. The host DMA engine transfers data in units of 4 KB. When a large host data needs to be transferred, it is divided into 4 KB units and moved with as many DMA operations as the unit count.

The access requests to a flash storage are often reordered to boost overall storage performance. One of the common reordering policies is to assign a higher priority to reads from the host system. Such reordering can take place in the host interface controller and in the storage controller. Our design implements scheduling functions in software to easily reconfigure reordering schemes. For this, the firmware has command/operation queues that replace the host command queue of the host interface controller and the control word queue of the storage controller. The control word queue is described in Section 3.2. The reordering scheme can be applied by reordering the commands/operations in the command/operation queues. Thus, the host interface controller and the storage controller of our baseline design do not include the reordering function.

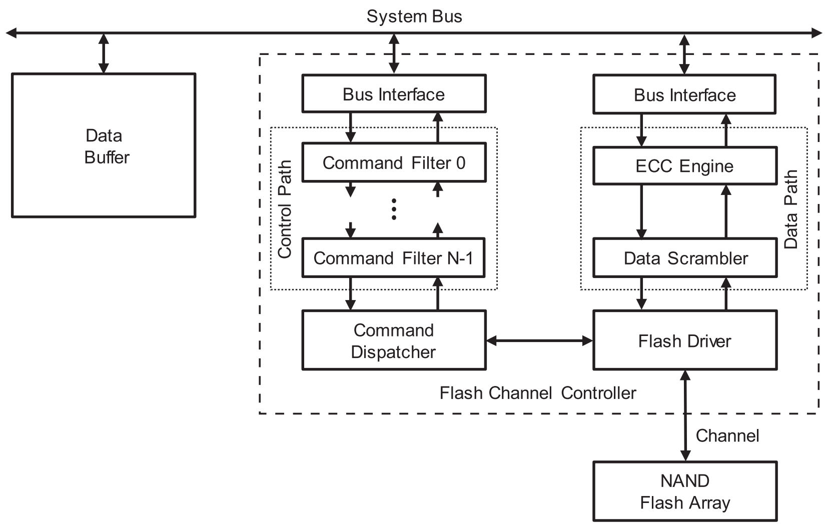

Fig. 5. Flash channel controller architecture.

Software-based scheduling may take longer for execution than hardware-based scheduling, which is also hidden by the long operation time of NAND flash devices.

3.2 Storage Controller

The storage controller provides the data access path between the host interface controller and flash devices. It has multiple flash channel controllers inside, each of which is responsible for orchestrating the operation of flash devices attached to the same channel. Each flash channel controller has its own flash driver to exchange physical signals with individual flash devices.

Figure 5 shows the organization of a flash-channel controller. Each controller has two internal information paths: control and data. When the firmware needs to perform a data access to a flash device or obtain operation status or statistics, it sends an appropriate Flash Control Word (FCW) to the control path. The FCW is the control information used for communication between the storage controller and the firmware. Upon arriving at the control path, it passes through command filters (if there are any) all the way up to the command dispatcher, which has a control word queue inside. Each command filter performs its own function on control words. One example is to gather access statistics and debugging information. The command dispatcher retrieves a FCW from the head of the control word queue, translates it to flash memory commands, and forwards the commands to the flash driver. In the case of administrative operations, it returns a response to the control word along the upstream in the control path without sending it to the flash driver.

The data path is used to transfer data between the host buffer and a flash memory device. One of the common tasks performed in the data path is to ensure data integrity. For a write operation, the ECC encoder first generates a code word for the write data, and then the data scrambler randomizes the data pattern. The ECC engine generates encoded data by adding parity bits to data to be stored in the NAND flash array and corrects data errors by using parity bits when reading data from the NAND flash array [12, 23]. The data scrambler is used to reduce bit error rate by ensuring that data written has an even bit distribution [5]. For a read operation, the page data pass through the data descrambler and the ECC decoder before being sent to the host buffer in the opposite direction.

In our design, the ECC engine implements the Bose–Chaudhuri–Hocquenghem (BCH) code, which efficiently corrects random errors and uses fewer hardware resources than low-density

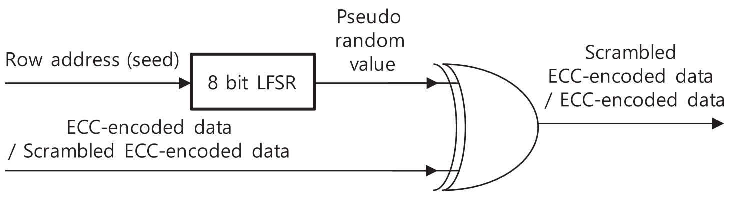
Row address (seed)
8 bit LFSR
Pseudo random value
ECC-encoded data / Scrambled ECC-encoded data
Scrambled ECC-encoded data / ECC-encoded data

Fig. 6. Data scrambling/descrambling process.

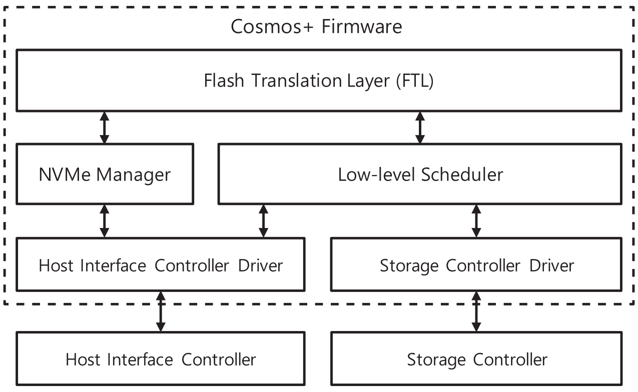
Cosmos+ Firmware
Flash Translation Layer (FTL)
NVMe Manager
Low-level Scheduler
Host Interface Controller Driver
Storage Controller Driver
Host Interface Controller
Storage Controller

Fig. 7. Cosmos+ firmware architecture.

parity-check (LDPC). To save hardware resources, an ECC engine is shared among multiple flash channels in a time-division manner (see Section 5 for more details).

Figure 6 shows the conversion process of the data scrambler. As shown in the figure, our scheme performs XOR operations on page data with the output of an 8-bit linear feedback shift register (LFSR), which uses the row address as its initial input: a seed for generating a pseudo-random value. The input data is transferred to an XOR gate in a byte unit with the pseudo-random value.

3.3 Cosmos+ Firmware Operations

Cosmos+ firmware interprets host commands to control the behavior of the host interface controller and the storage controller. It consists of a NVMe manager, an FTL, a low-level scheduler, a host interface controller driver, and a storage controller driver, as shown in Figure 7.

3.3.1 NVMe Manager. For the storage access, the host system prepares a set of queues—a submission queue and a completion queue. It uses host commands defined in the NVMe interface protocol. The commands belong to one of the two command sets—*Admin command set* and *NVM command set*. An *Admin command* performs an administrative operation such as setting up the NVMe interface parameters and checking the operation status, while an *NVM command* transmits user data between the host system and the storage device. The NVMe manager interprets the host commands. If the command is administrative, the manager handles it by itself. Otherwise, it forwards the command to the FTL.

3.3.2 Flash Translation Layer. The FTL is responsible for performing storage access operations specified by host commands. For each NVM command, it generates a set of subsequent operations—a host DMA operation to move data between the host memory and the data buffer, a flash DMA operation to move data between the data buffer and a flash device, and a flash operation such as page read and write. For each logical page included in an NVM command, the FTL allocates a

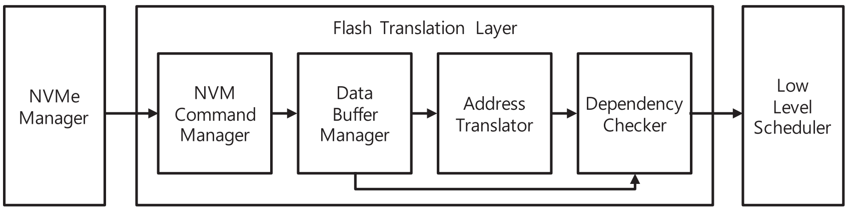

Fig. 8. Flash translation layer architecture.

data buffer entry. The operations performed by the FTL are grouped into four components: *NVM Command Manager*, *Data Buffer Manage*r, *Address Translator*, and *Dependency Checker*. Figure 8 shows the sequence of these components activated inside the FTL. In order to exploit data access parallelism, it should be able to run as many multiple flash operations as possible: it issues new DMA/flash operations even while the previous DMA/flash operations are still in progress. For this, it could process many *NVM commands* in a non-blocking manner.

Because the FTL manages data in a fixed size (the mapping unit size), the *NVM Command Manager* divides an *NVM command* requesting an arbitrary size of data into *slice commands* requesting fixed-sized data. It has a slice command queue inside and keeps *slice commands* until the *Data Buffer Manager* retrieves. The default policy of the *slice command* retrieval is first-in first-out.

The *Data Buffer Manager* has a pool of data buffer entries, which are a fixed size. It assigns an entry to each slice command. The data buffer provisioned in the storage dynamic random access memory (DRAM) is a temporary store for data transfer between the host system and the flash device. For the data transmission, the manager generates a host DMA operation based on the buffer entry number and the host memory location. Then, it delivers the DMA operation to the *Dependency Checker* and the slice command with the buffer entry number to the *Address Translator*. If there is no free space in the data buffer, the manager makes free space by recycling the buffer entries using the least recently used (LRU) policy.

The *Address Translator* is responsible for translating a logical address specified in a slice command to a *location vector* by looking up the mapping information. Another function of this module is to interpret the slice command and generate necessary flash operations and flash DMA operations. Once generated, these operations are passed to the *Dependency Checker*. For the easy variation of flash-array configuration and capacity, the address translator uses the location vector instead of physical location information. The vector comprises three address components—channel, way, and row. All the pages in the same channel and way are given a serialized page number across blocks and planes. The row address is the serialized page number. When used for accessing a specific page, it needs to be translated to page, block, and plane numbers later. By doing this, the FTL can vary the channel-way organization without redesigning the storage controller. By specifying the numbers of channel, way, page, block, and plane below the physically available maximum numbers, the FTL effectively reduces the storage capacity used for experiments. Figure 9 shows an example of the process of translating the logical address of an *NVM command* to the corresponding location vector. The use of location vector makes it possible to use only least significant bit (LSB) pages in the multi-level cell (MLC) flash memory by limiting the range of row addresses.

The *Dependency Checker* aims to keep the order of DMA operations and flash operations generated to execute a host command. For read and write commands, it should maintain the operation order as shown in Figure 10. When there are operations from multiple host commands in the queue, they may be sent to the low-level scheduler in an interleaved way to enhance the performance as long as they are in order from the perspective of a given command. For example, let's assume that a host command A is translated to a set of operations, A1, A2, and A3, which are queued in

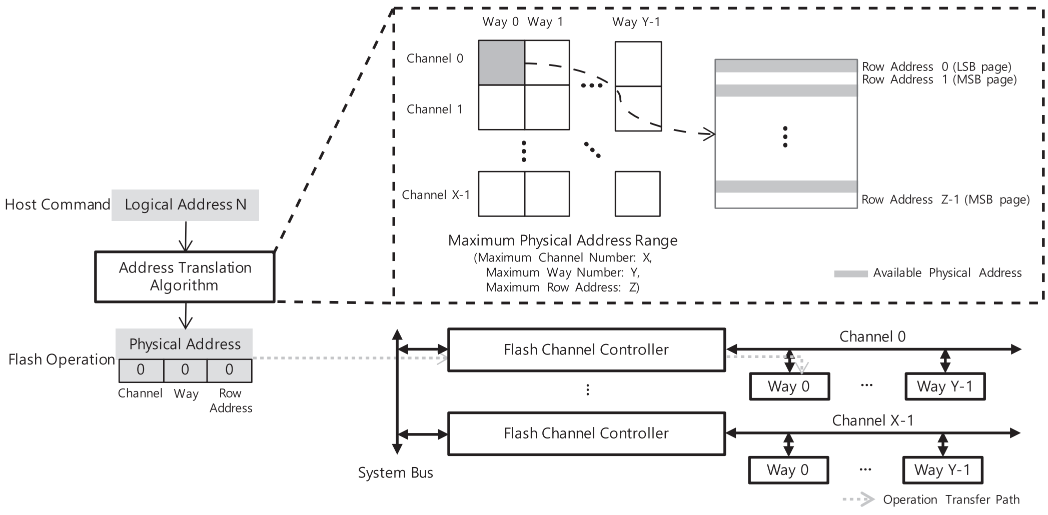

Fig. 9. Physical address restriction process of Cosmos+ firmware.

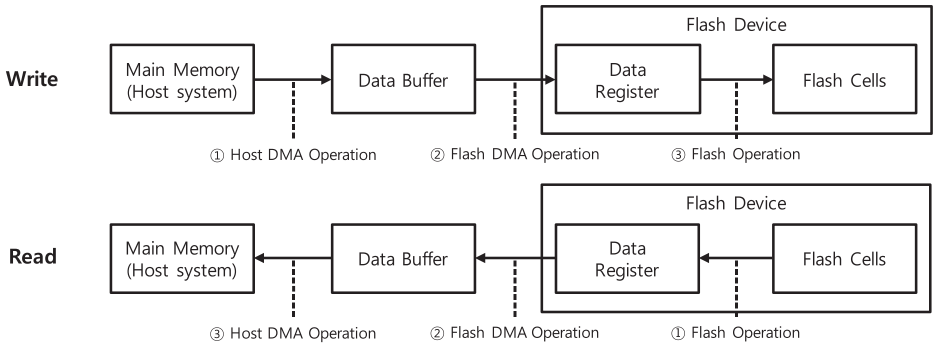

Fig. 10. Operation sequence in data flow.

sequence. The same assumption is made to a host command B. Now the Dependency Checker can send operations to the low-level scheduler in the order of A1, B1, A2, B2, A3, and B3. However, the transmission order for operations belonging to the same host command cannot be reversed (e.g., A2, B1, A1, B2, A3, and B3). Section 3.4 will describe the details on how the NVM command is processed by the FTL and how the operations generated by the FTL are performed.

3.3.3 Low Level Scheduler. The low-level scheduler determines the execution order of operations generated by the FTL. One of the objectives of using the scheduler is to improve the performance of the flash storage system by reordering flash and DMA operations [25]. For each channel, the scheduling algorithm performs two tasks. One sends operations to the host DMA engine and the storage controller. Flash operations and flash DMA operations issued by the scheduler are translated to FCWs by the storage controller driver. The other task reorders the operations based on priority before sending them. Moreover, the scheduler may assign different priorities according to operation type. The operations with higher priority can bypass ones with lower priority.

Figure 11 shows the structure of the low-level scheduler. The low-level scheduler consists of *Reservation Station*, *State Manager*, *Status Checker*, *Priority Manager*, and *Issuer*. The operation

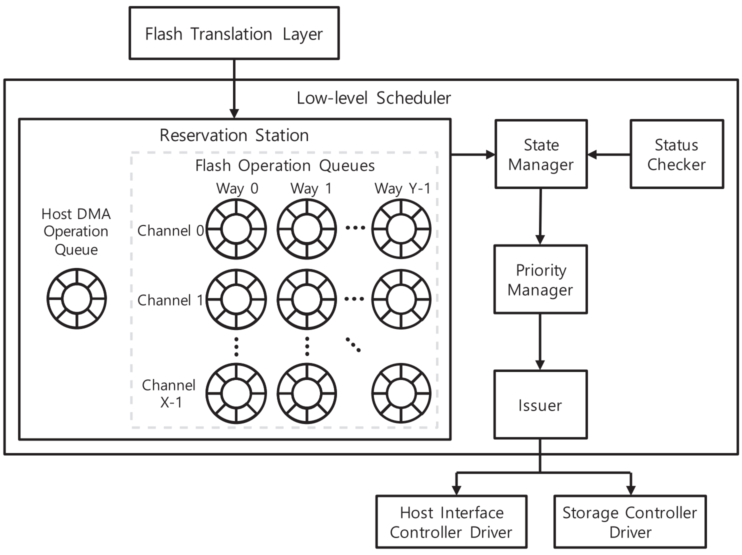

Fig. 11. Low-level scheduler architecture.

generated by the FTL is sent to the *Reservation Station*. The *Reservation Station* has a host DMA operation queue and flash operation queues one for each combination of channel and way. The flash operations and flash DMA operations are sent to the flash operation queue. The *State Manager* keeps track of the operation execution status of the storage controller and the host interface controller. If each controller is ready to receive a new operation, the *State Manager* forwards a queued operation to the *Priority Manager*, which then compares the priority of operations in the queue and determines new execution order. The *Issuer* passes the operation to the host interface controller or the flash channel controllers based on the execution order of operations. The *Status Checker* confirms the execution status of the operation issued by the *Issuer* and updates the status information of the *State Manager* and the dependency checker of the FTL.

The *Status Checker* collects the execution statuses of DMA/flash operations and informs the scheduler. The host DMA engine has special registers that periodically update the processing information of host DMA operations received by the *Issuer*. Each channel controller maintains a special register with a ready/busy bit for each way. By reading the registers, the *Status Checker* can identify the operation status. However, to obtain the precise flash operation result, it sends a status-check operation to each way and reads the result values. When the number of channels and ways increases, the overhead of the *Status Checker* increases accordingly. The scheduler calls the *Status Checker* to collect the status of outstanding operations and performs a scheduling operation based on the collected states.

The *State Manager* maintains states of the flash devices located on each way of the NAND flash array to check which device can perform a new operation. Figure 12 shows the state transition diagram. The flash devices of the *idle* state can perform a new operation. If a flash device that receives a new operation executes the operation successfully, the state of the device returns to *idle* via *busy* and *done*. If the operation fails, the transition process depends on the type of operation. In the case of the read operation, even if the operation of the flash device is successful, the operation can fail if the error correction process of the ECC engine fails because of many bit errors. The bit error rate is not a constant because of the inherent characteristics of the flash device. Therefore, if the read operation is performed again, there is a possibility that error correction process can

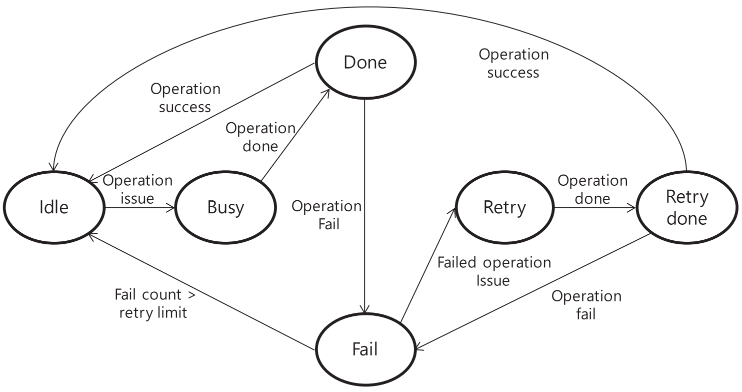

Fig. 12. State transition diagram for each way of the NAND flash array.

Table 2. Main Operation Set of Low-Level Scheduler

Operation	Type	Priority
Rx DMA	Host DMA	0
Tx DMA	Host DMA	0
Status-check	Flash	1
Read-trigger	Flash	2
Erase	Flash	3
Write	Flash + Flash DMA	4
Read-transfer	Flash DMA	5

be performed successfully. If the read operation fails, the low-level scheduler retries the operation within the retry limit. If the read operation consistently fails until the retry limit is reached, or if another operation fails, the low-level scheduler sends the information of the accessed block to the FTL so that information regarding the bad block can be updated, and the state changes from *fail* to *idle*.

The *Priority Manager* determines the execution order of operations based on the priorities listed in Table 2. Host DMA operations have the highest priority to reduce the storage access latency from host systems. Whenever there is room for a new DMA operation in the host interface controller, the scheduler fetches it from the host DMA operation queue and sends it to the host interface controller. Next, the scheduler attempts to handle flash operations. The flash DMA operations have lower priority than other flash operations because they occupy the flash channel for a longer time and interfere with other flash operations. Among the flash operations, an operation with lower latency is given a higher priority. The operations with the same priority are handled in a round-robin manner.

3.4 NVM Command Processing

This section details how the NVM command is handled by each component of Cosmos+ OpenSSD. An example shown in Figures 13 and 14 is the process of reading the data of logical page number (LPN) 0 and writing the data of LPNs 1 and 2. The data of LPN 0 is stored in the flash device in advance. The process is divided into two steps. In the first step, NVM commands are converted

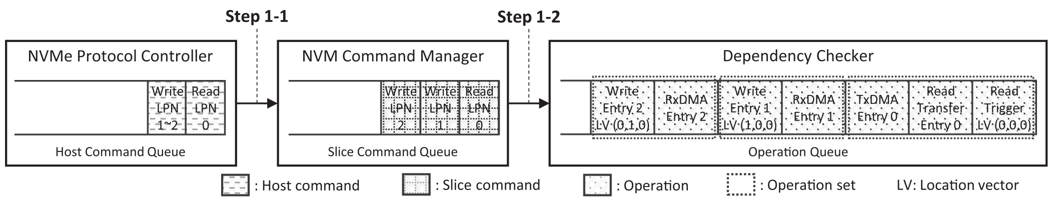

Fig. 13. First step of NVM command processing example.

to DMA/flash operations. In the second step, the data moves through the data path by DMA/flash operations.

Figure 13 shows the process in which the FTL ameliorates NVM commands. The NVM command contains the LPN and the main memory address of the host system. The address points to the space containing the data to be stored to the flash device or the space assigned for the data to be read from the flash device. For readability, Figure 13 does not contain information about the address. The information contained in the NVM command is insufficient for performing data I/O through the data flow shown in Figure 10. The FTL determines the data buffer entry to be used by the DMA operations and the physical location in the flash device to be accessed by the flash operation by using the data buffer manager and the address translator. Prior to this, because flash DMA operation and flash operation are performed page-by-page, it is necessary to convert the NVM command to slice commands as Step 1-1. While the slice command is converted into the operation shown in Table 2, Step 1-2 allocates the data buffer entry and the location vector. The operations generated by the FTL are entered into the operation queue according to the sequence of Figure 10.

Figure 14 shows the process that the low-level scheduler controls the host DMA engine and the storage controller. In Steps 2-1, 2-3, and 2-5, the dependency checker passes the operations to the reservation station of the low-level scheduler. In Steps 2-2, 2-4, and 2-6, the operations are scheduled and transferred to the host DMA engine and the storage controller.

Because the host DMA engine and the storage controller operate independently of each other, the low-level scheduler arbitrates the host DMA engine and the storage controller so that the sequence of Figure 10 is maintained in units of the operation set. An operation set is a set of operations generated from the same slice command. As shown in Steps 2-1, 2-3, and 2-5, the dependency checker delivers one or two operations per operation set to the reservation station. It delivers two operations only for the operation set having the read-trigger and the read-transfer operations, as shown in Step 2-1. Because these operations move to the same flash operation queue, the order of the two operations is maintained. When the host DMA engine and the storage controller complete the operations received from the low-level scheduler, the dependency checker transfers the operation of the next sequence to the reservation station, as shown in Steps 2-3 and 2-5.

The low-level scheduler compares the priority of the operations in the header position of each queue before issuing operations. As shown in Step 2-4, the write operation is issued prior to the read-transfer operation according to the priority of Table 2. Consequently, although the read-transfer operation is generated prior to the write operation, it can be executed after the write operation is done in this case. If the flash operation or the flash DMA operation fails, the scheduler retries the failed operation or informs the FTL of the bad-block information. Therefore, until the header operation of a flash operation queue is completed, other operations remain in the queue, as shown in Step 2-2.

The action of Step 2 is based on the status information updated by the status checker of the low-level scheduler. The status checker checks the registers of the host DMA engine and the storage controller to update the status information of the state manager and the dependency checker if

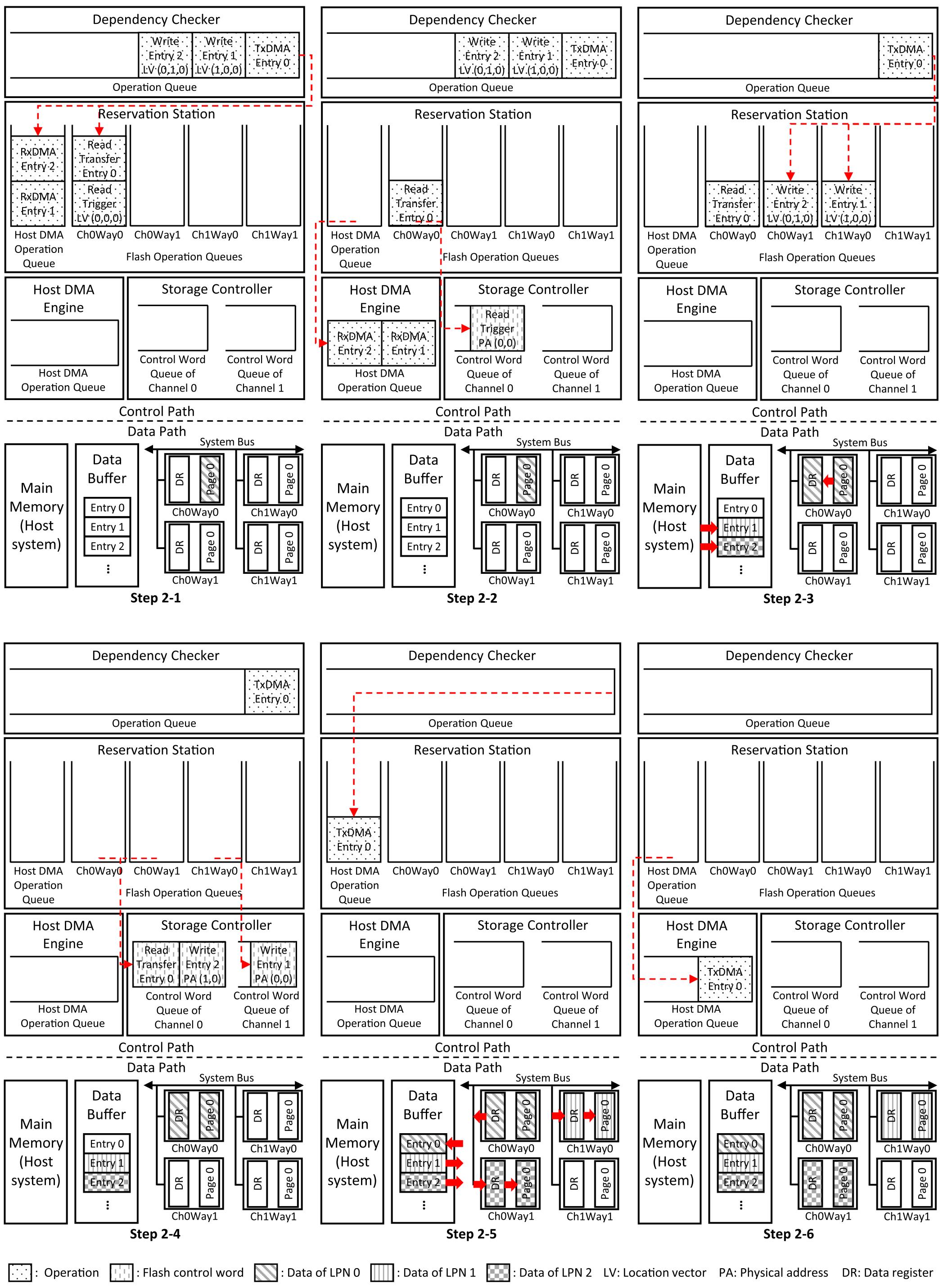

Fig. 14. Second step of NVM command processing example.

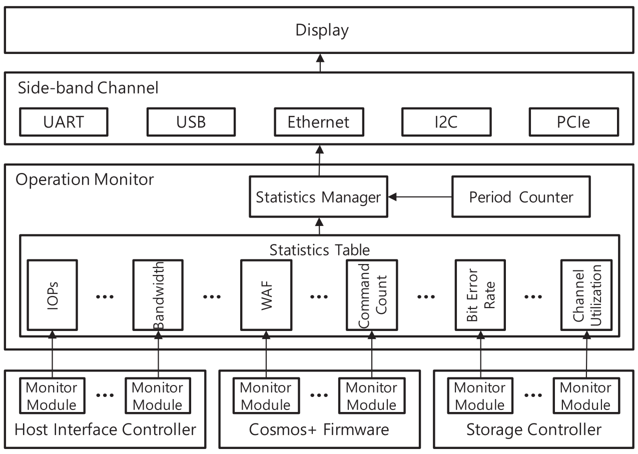

Fig. 15. Storage system operation monitoring process.

there are completed operations. The dependency checker passes the next sequence operation of the operation set related to the completed operation. The state manager changes the header position of the flash operation queue related to the completed operation to issue the operations waiting on the queue. Exceptionally, the status checker issues a status-checker operation to determine whether the flash operation has been completed. The status-checker operation does not affect the execution sequence of the operation sets because the flash device does not receive any other operation until the header position of the flash operation queue is changed.

4 STORAGE SYSTEM OPERATION MONITORING

Cosmos+ OpenSSD uses an FPGA to implement hardware components, which provides a way of monitoring the internal hardware operations. It actually collects various statistics on the internal operation of the host interface controller, the storage controller, and the Cosmos+ firmware as well. Then, it also provides a way to access such statistics via a sideband channel to avoid the unfruitful consumption of data bandwidth. Figure 15 shows the process of obtaining information about the internal operation by using an operation monitor.

The monitor module is a front-end agent that collects statistical data. It may sense hardware control signals or analyze the operation of software components. With this feature, the operation of the controller can be investigated in depth. The statistics that the monitor module can provide may vary depending on the component types. They include channel bandwidth, write amplification factor (WAF), and channel utilization. Figure 16 shows the structures of several monitor modules. Each monitor module updates the associated statistics table in the operation monitor with the sensed data.

The operation monitor gathers the analysis data from each monitor module and delivers it to a monitoring system through a side-band channel (e.g., UART). It has a statistics table and manager. The statistics table is a data structure used to store the data generated by each monitor module, and the statistics manager assigns the entry of the statistics table to each monitor module.

5 COSMOS+ OPENSSD SYSTEM IMPLEMENTATION

The prototype of Cosmos+ OpenSSD reuses the Cosmos OpenSSD platform hardware [31]. Figure 17(a) shows the main components of the platform. The prototype uses an application processor unit and programmable logics built into the FPGA. The application processor unit includes

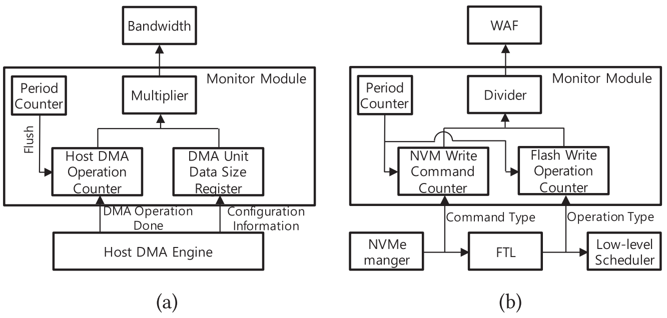

Fig. 16. Monitor module examples: (a) bandwidth monitor module; (b) WAF monitor module; (c) channel utilization monitor module.

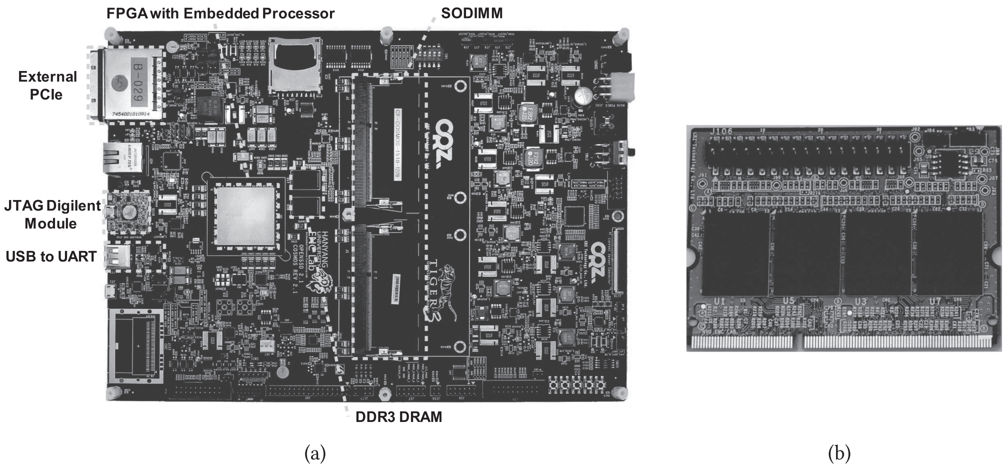

Fig. 17. Hardware modules of Cosmos+ OpenSSD prototype: (a) Cosmos OpenSSD platform hardware; (b) flash memory module.

two advanced RISC machine (ARM) Cortex-A9 cores. The external PCIe connector expands a PCIe slot in the host system to bridge the platform and the host system. The flash memory module is mounted to a small outline dual in-line memory module (SODIMM) slot. Our implementation uses MLC flash devices supporting a double data rate (DDR) toggle mode flash interface with 4-channel, 8-way organization. Figure 17(b) shows the flash memory module.

Figure 18 shows the hardware organization of the prototype. The host interface controller is connected to the host system via PCIe Gen2 8-lane. For the NVMe interface, the prototype uses the PCIe core built into the FPGA. The NVMe protocol controller and the host DMA engine use the transaction layer packets of the PCIe interface to communicate with the PCIe host. The storage controller has eight flash channel controllers to support up to an 8-channel, 8-way configuration. The host interface controller and storage controller are connected to the processor through the advanced extensible interface (AXI)4 interface. Table 3 shows the hardware resource usage of each component of Cosmos+ OpenSSD.

To reduce hardware resource usage, four flash-channel controllers share some components of the BCH ECC engine. Among the components of the BCH engine, the key equation solver (KES) uses the biggest share of hardware resources and runs only for a very short time. Several studies have been conducted on complexity reduction using KES in a time-multiplex method. [1, 21, 30, 34].

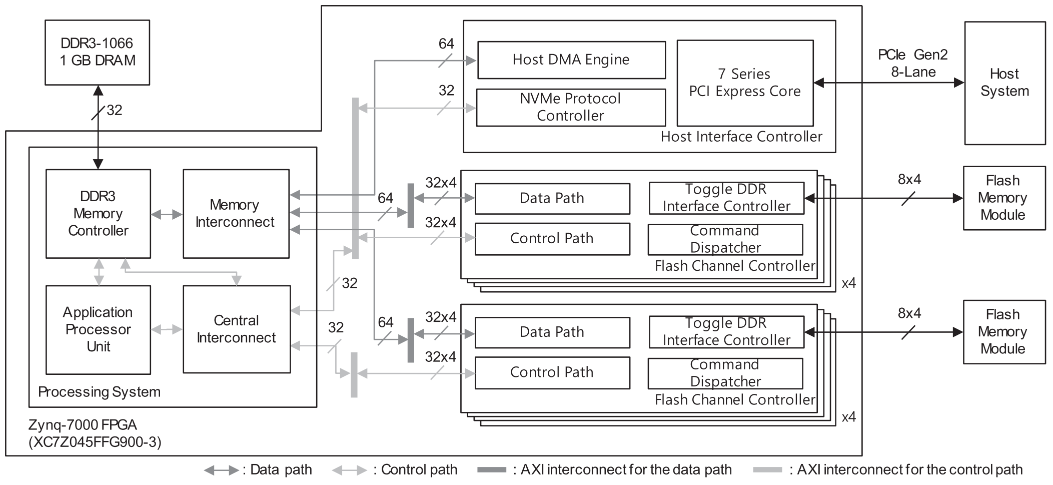

Fig. 18. Hardware architecture of the Cosmos+ OpenSSD prototype.

Table 3. Hardware Resource Usage

Component		LUTs	FFs	BRAMs	Implementation
Processing system		0	0	0	X (of-the-shelf)
AXI bus interface		17,568	25,732	21	X (customized IP)
Host interface controller	PCIe core	3,275	3,969	8	X (customized IP)
	Others	5,470	7,560	19.5	O
	Total	8,745	11,529	27.5	△
Storage controller	Flash channel controller	10,988	7,548	21	O (x8)
	Shared KES	6,548	8,203	0	O (x2)
	Total	101,000	76,790	168	O

Figure 19 shows the ECC decoding process with the shared KES to reduce the hardware resource usage. The pipelining scheme is applied to the ECC decoding process. It can hide the waiting time while other flash channel controllers use the shared KES. Therefore, the shared KES has little impact on the performance of the storage controller. Thus, the hardware resource usage can be reduced without a significant increase in congestion. The BCH ECC engine implemented in Cosmos+ OpenSSD can correct 13 error bits per 256 bytes. The NAND flash device in the flash memory module requires 100 bits of error-correction capability per 4 KB.

Cosmos+ OpenSSD runs its firmware as a single thread, including the NVMe manager, FTL, and low-level scheduler. Thus, the current implementation of firmware is to fetch a host command, call a command processor, and run the low-level scheduler in the same big loop. After the initialization, *FirmwareMain()* starts to execute the loop. When there is a command waiting in the host command queue, the firmware grabs one and calls either *FTL()* or *AdminCommandHandler()* based on the NVMe command group. When an *NVM command* arrives, the FTL prepares subsequent operations and deposits them to the reservation station. Otherwise, it must be *Admin command* and processed by the firmware without consulting the flash devices. Next, the firmware starts the scheduling operation by calling *LowLevelScheduler()* if there is any operation waiting in the reservation station. In this function, the firmware checks the operation status of flash channels and reorders operations based on its scheduling policy. This firmware implementation helps to

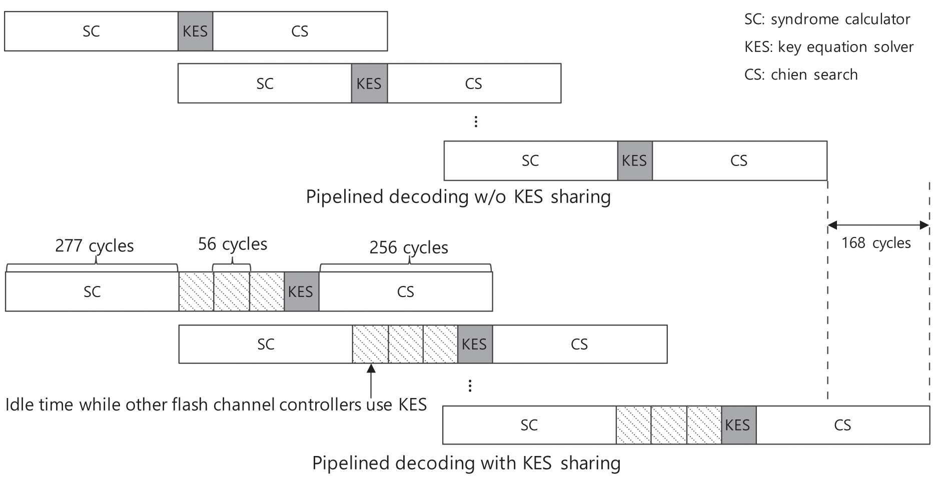
SC: syndrome calculator
KES: key equation solver
CS: chien search
SC
KES
CS
Pipelined decoding w/o KES sharing
277 cycles
56 cycles
256 cycles
168 cycles
Idle time while other flash channel controllers use KES
Pipelined decoding with KES sharing

Fig. 19. ECC decoding process with the shared KES.

ALGORITHM 1: Cosmos+ OpenSSD Firmware Operation

```
1:  Initialize host interface controller and storage controller
2:  Initialize firmware metadata
3:  Call FirmwareMain()
4:
5:  function FirmwareMain()
6:      while (1) do
7:          cmd = GetHostCommand()
8:          if (cmd == NVM command) then
9:              Call FTL()
10:         else
11:             Call AdminCommandHandler()
12:         end if
13:         if (reservation station is not empty) then
14:             Call LowLevelScheduler()
15:         end if
16:     end while
17: end function
```

minimize the number of cores to run to one, which reserves the other processor cores for extra tasks.

6 EXPERIMENTS

To identify the performance and operations behavior of Cosmos+ OpenSSD, we have performed experiments by attaching the prototype to a host PC and running a benchmark. When the prototype is modified in terms of hardware organization and the firmware algorithm, it is often necessary to measure the enhancement by comparing the performance of a baseline system and an enhanced one.

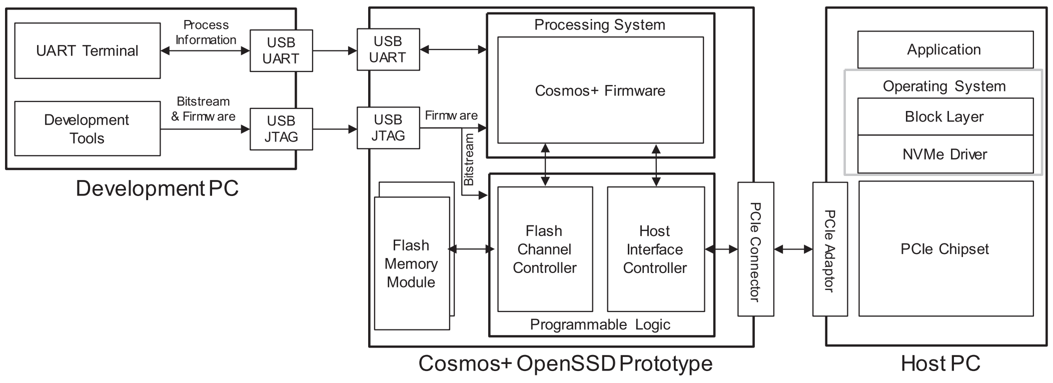

Fig. 20. Experimental environment architecture.

Table 4. Specification of Host PC

Component	Specification
CPU	i5-4670
Mainboard chipset	B85
Main memory	8 GB
Operation system	Window 8.1
NVMe driver	Native driver of OS

6.1 Experimental Setup

Figure 20 shows our experimental environment. The development PC is used to design the hardware and firmware of the prototype and to collect the operation statistics via an additional UART channel. The host PC uses the prototype as an NVMe storage device and runs applications to allow access to it. Table 4 is a brief summary of the host PC. The workload used to measure performance was generated by the Iometer benchmark tool [27], which ran as an application on the host PC. The Iometer accessed the prototype as a physical drive to avoid any impact of the file system operation on performance measurement. The default configuration of the number of submission queues to deliver I/O commands was set to four.

Table 5 describes the flash memory module used in the experiment. A flash device has 256 pages in a block and 8,192 blocks in a way. Each page consists of a 16 KB data area and a 1,664 byte spare area. The MLC flash device included in the flash memory module has LSB and most significant bit (MSB) pages (MLC mode). However, only LSB pages are accessed when the range of row address is halved (SLC mode). In the 8-channel, 8-way configuration, the storage capacity used in the single level cell (SLC) and MLC modes was 1 TB. If the MLC mode uses all blocks, the storage capacity increases to 2 TB. However, because the capacity of the DRAM was limited, the MLC mode used half of the total blocks to reduce the size of the mapping table used by the FTL. The capacity of the DRAM embedded in the platform board was 1 GB, and the mapping table occupied 512 MB. If the number of channels or ways used decreases, the size of the mapping table decreases.

Excluding the space used for the mapping table, the remaining space in the DRAM was used for the metadata managed by the firmware, data buffers, and the reserved space for expansion. For the experiments, the size of the data buffer was 16 MB. The workloads used in Sections 7 and 8 did not make the data buffer hit because they access sequential logical addresses or completely random

Table 5. Specification of NAND Flash Memory

Organization	Byte/Page			16,384 + 1,664
	Page/Block			256
	Block/Way			8,192
Latency	Read	SLC mode		99 μs
		MLC mode	LSB page	58 μs
			MSB page	90 μs
			Average	74 μs
	Write	SLC mode		486 μs
		MLC mode	LSB page	481 μs
			MSB page	2,295 μs
			Average	1,388 μs
	Erase			5 ms

Table 6. Specification of Buses of the Prototype

Bus ID	Location	Frequency	Bit width	Bandwidth
0	DRAM ↔ Memory controller	533 MHz DDR	32	4 GB/s
1	Memory controller ↔ Memory interconnect	356 MHz	128	5.7 GB/s
2	Memory interconnect ↔ Host interface controller	250 MHz	64	2 GB/s
3	Memory interconnect ↔ AXI interconnect[1]	250 MHz	64	2 GB/s
4	AXI interconnect[1] ↔ Flash channel controller	100 MHz	32	400 MB/s
5	Flash channel controller ↔ Flash memory module	100 MHz DDR	8	200 MB/s
6	Processor Unit ↔ Memory controller	1 GHz	64	8 GB/s
7	Processor Unit ↔ Central interconnect	333 MHz	64	2.7 GB/s
8	Central interconnect ↔ Memory controller	333 MHz	64	2.7 GB/s
9	Central interconnect ↔ AXI interconnect[2]	100 MHz	32	400 MB/s
10	AXI interconnect[2] ↔ Host interface controller	100 MHz	32	400 MB/s
11	AXI interconnect[2] ↔ Flash channel controller	100 MHz	32	400 MB/s

[1]AXI interconnect for the data path.
[2]AXI interconnect for the control path.

logical addresses. In addition, because the throughput of the prototype was large enough to fill the data buffer in a very short time, the size of the data buffer did not affect the experimental results.

As shown in Figure 18, the prototype uses several buses. Table 6 shows the information regarding each bus. Buses 0 to 5 are used for the data path, and Buses 0 and 6 to 11 are used for the control path. The bottleneck point of the data path is Bus 5. Because eight channels were used, the bandwidth of the data path was 1.6 GB/s. Because the data buffer exists in the DRAM, 3.2 GB/s of the DRAM bandwidth could be occupied by the data path. Even though buses of the control path have a large bandwidth, only a small portion of the bandwidth is actually used because the amount of information transmitted through the control path is very small compared to that transferred via the data path.

The control path is used by the processor to control each controller, to check the status information of each controller, or to access metadata required for the firmware execution. To input/output data through the data path, the control path bandwidth required for controlling each controller and checking the status information is approximately 1% of the data path bandwidth. Because of the cache memory, the amount of access to metadata mainly is affected by the large size of the

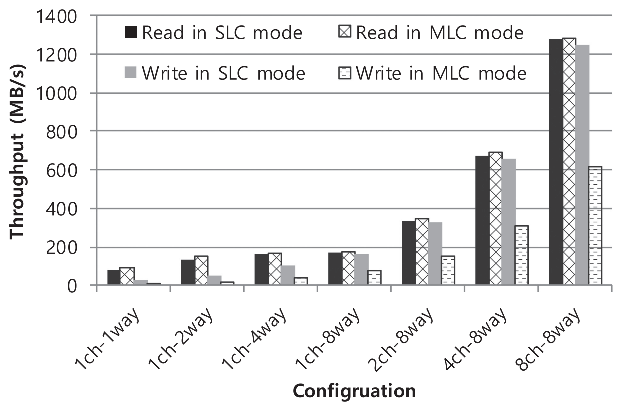

Fig. 21. Maximum throughput of the prototype.

metadata. The largest metadata of the firmware is the mapping table. If the prototype utilizes the full bandwidth of the data path, which is 1.6 GB/s, the processor uses less than 1 MB/s of the DRAM bandwidth for accessing the mapping table because 8 bytes are required per page. Even when other metadata for firmware execution were considered, the DRAM bandwidth used by the control path was small enough such that it did not affect the DRAM bandwidth of the data path.

6.2 Maximum Throughput

The two workloads used in the experiment read/wrote 128 KB of data with consecutive logical addresses. There were 16 host commands outstanding simultaneously, such that all flash devices were fully active. That is, sixteen 128 KB read/write commands were simultaneously issued to the prototype.

Figure 21 shows the maximum throughput when the prototype configuration varies in the number of channels and ways. As the number of ways increased, the throughput improved because the way accesses were interleaved. Because the storage controller operated channel buses in parallel, the maximum throughput increased in proportion to the number of channels. As explained in Section 6.1, the bottleneck point of the data path was Bus 5 of Table 6, and the bandwidth of the control path was sufficient to not degrade performance. In addition, because the bandwidth of PCIe Gen2 was larger than that of the data path, the throughput of the prototype was determined by the throughput of the flash channel controllers. Although the maximum throughput per channel was 200 MB/s, because the data and spare area of a page were accessed together, the effective maximum data throughput per channel (EMDTPc) was approximately 180 MB/s. Considering the EMDTPc, the maximum throughput of the 8-channel, 8-way configuration was 1,440 MB/s. The experimental results showed that the read throughput was 1,275 MB/s, which was approximately 88.5% of the maximum, while the write throughput in MLC mode was almost halved to 625 MB/s.

Because of the high latency of write operations, the maximum write throughput of the MLC mode was less than that of the SLC mode. Referring to Table 5, the average write latency in MLC mode was 1,388 μs. Considering the operation speed of the data bus, the time to transfer a page through the channel was approximately 90 μs. Because the write latency was much longer than the transfer time, the data bus was not fully utilized even though eight ways were used.

6.3 Maximum IOPs for Small Random Access

Figure 22 shows the maximum input/output operations per second (IOPs) of the prototype. The two workloads used in the experiment read/wrote 4 KB of data with random logical addresses.

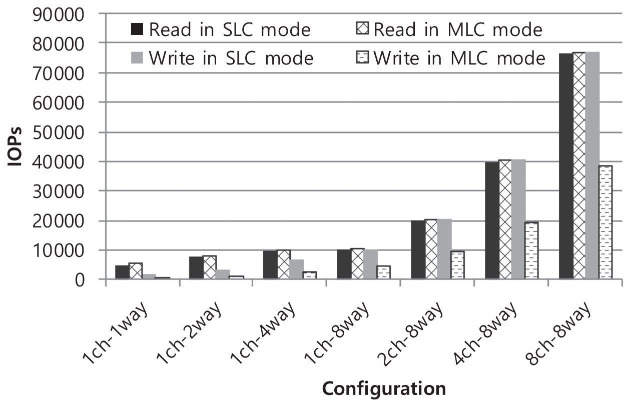

Fig. 22. Maximum IOPs of the prototype when host PC requests 4 KB data with random address.

There were 128 host commands outstanding simultaneously, such that all flash devices were fully active. The growth rate of the IOPs was similar to that of the throughput shown in Figure 21. However, the measured IOPs were approximately a quarter of the expectation, which was the throughput divided by the data size. This is due to the discrepancy between the page size (16 KB) and the request size (4 KB). Because the FTL used a page-level mapping, a host command for 4 KB data was converted to the operations for 16 KB data. Because the storage controller processed more data than the host system requested, the measured IOPs was lower than expected.

6.4 System Latency

In order to analyze data access parallelism of the prototype, we measured the system latency by changing the number of host commands outstanding simultaneously. The system latency is the sum of latencies in the host interface controller, firmware, and storage controller. The latency of each component was measured based on the state information collected by the low-level scheduler. For example, the latency in the storage controller was measured as an elapsed time between the operation issue done by the scheduler and the completion sensed by the scheduler. The flash operation latency is included in the storage controller latency. For this experiment, the Iometer generated host commands, each of which read/wrote 16 KB of data to the prototype.

Table 7 shows the system latency and the latency of each component in the 8-channel, 8-way configuration. The storage controller latency constituted most of the system latency due to the long operation time of flash devices. However, as the number of outstanding I/Os increased, the share of the storage controller in the total system latency decreased due to the effect of interleaving.

When the number of outstanding I/Os was eight or less, the storage controller latency remained the same because the channels were well interleaved. The FTL made the accesses to consecutive logical addresses interleave on the same way in different channels, as shown in Figure 23. However, when the number increased beyond eight, the operations were again interleaved on ways in the same channel. Then, it was inevitable for the operations to compete on the shared resource of the channel bus. Even though the operations in the same channel used the resource in a time-multiplexed way, such resource contention contributed to the increase of latency in the storage controller.

Channel/way interleaving impacted throughput variation according to the number of outstanding I/Os, as shown in Figure 24. When the number of the outstanding I/Os was eight or less, the throughput increased proportionally to the number of outstanding I/Os because of channel interleaving. When the number of the outstanding I/Os was greater than eight, the throughput growth

Table 7. Latency Breakdown of Prototype Operation

	IOs[1] =1		IOs=2		IOs=4		IOs=8		IOs=16		IOs=32		IOs=64	
	μs	%	μs	%	μs	%	μs	%	μs	%	μs	%	μs	%
R-Sys[2] (SLC)	239.9	100.0	246.9	100.0	254.4	100.0	269.8	100.0	352.0	100.0	512.3	100.0	972.2	100.0
HIC[3]	22.7	9.5	25.2	10.2	27.0	10.6	30.7	11.4	39.3	11.2	61.9	12.1	68.0	7.0
CF[4]	7.3	3.0	7.9	3.2	9.8	3.9	11.8	4.4	24.9	7.1	60.8	11.9	312.0	32.1
SC[5]	209.9	87.5	213.8	86.6	217.6	85.5	227.3	84.2	287.8	81.8	389.6	76.0	592.2	60.9
R-Sys (MLC)	214.8	100.0	220.6	100.0	227.8	100.0	251.5	100.0	341.1	100.0	497.4	100.0	978.0	100.0
HIC	22.8	10.6	26.0	11.8	26.8	11.8	33.7	13.4	38.9	11.4	68.6	13.8	69.1	7.1
CF	7.3	3.4	7.8	3.5	9.4	4.1	13.9	5.5	24.7	7.2	69.7	14.0	310.0	31.7
SC	184.7	86.0	186.8	84.7	191.6	84.1	203.9	81.1	277.5	81.4	359.1	72.2	598.9	61.2
W-Sys[6] (SLC)	611.8	100.0	620.4	100.0	624.7	100.0	633.1	100.0	641.5	100.0	678.9	100.0	871.1	100.0
HIC	24.2	4.0	27.7	4.5	27.2	4.4	27.9	4.4	29.6	4.6	34.6	5.1	44.9	5.2
CF	6.4	1.0	7.3	1.2	8.0	1.3	8.6	1.4	10.2	1.6	15.6	2.3	71.6	8.2
SC	581.2	95.0	585.4	94.4	589.5	94.4	596.6	94.2	601.7	93.8	628.7	92.6	754.6	86.6
W-Sys (MLC)	1,531.2	100.0	1,538.0	100.0	1,544.4	100.0	1,556.7	100.0	1,566.8	100.0	1,607.1	100.0	1,823.2	100.0
HIC	25.0	1.6	27.9	1.8	27.3	1.8	28.8	1.9	30.4	1.9	34.9	2.2	42.5	2.3
CF	6.4	0.4	7.2	0.5	7.8	0.5	8.5	0.5	10.3	0.7	16.4	1.0	131.2	7.2
SC	1,499.8	97.9	1,502.9	97.7	1,509.3	97.7	1,519.4	97.6	1,526.1	97.4	1,555.8	96.8	1,649.5	90.5

[1]Number of I/Os outstanding at the same time.
[2]System latency for a read operation.
[3]Host interface controller latency.
[4]Cosmos+ firmware latency.
[5]Storage controller latency.
[6]System latency for a write operation.

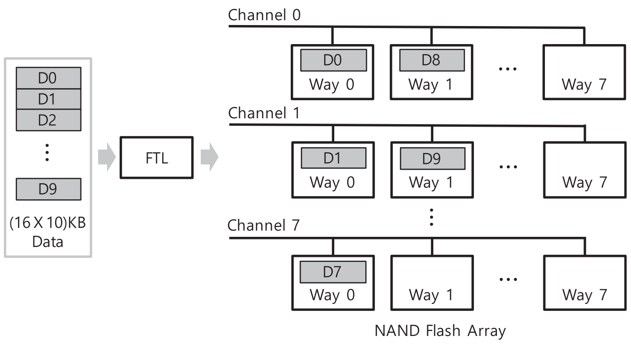

Fig. 23. Logical-to-physical address translation algorithm for the multi-channel, multi-way organization.

rate was less than the rate of the increasing number of outstanding I/Os caused by resource contention. In the read case, the resource contention was more intense than the write case because of the short flash operation latency. Therefore, when the number of outstanding I/Os was greater than eight, the throughput growth rate of the read case was less than that of the write case.

It is also notable that the firmware latency is not negligible in the system latency as the number of outstanding I/Os increased. As more I/Os are at work, the scheduler spends more time to check the operation status of flash devices and the existence of further operations in the reservation station. Because the firmware is single-threaded, the overhead increase due to the long execution

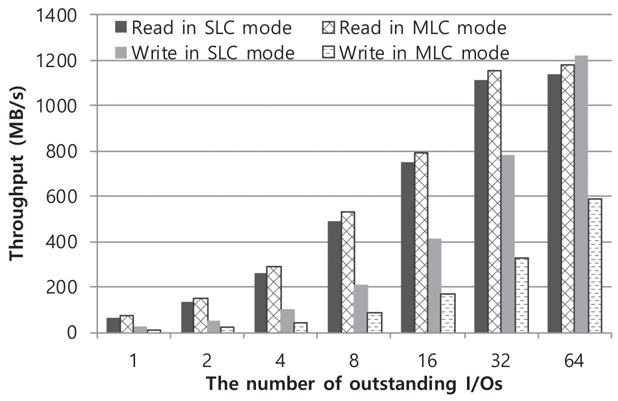

Fig. 24. Throughput of the prototype with various numbers of outstanding I/Os.

of firmware, especially the low-level scheduler, contributes to the increase in operation latency of previous host commands.

When the number of outstanding I/Os was 64, the firmware latency increased sharply because of the inconsistent flash operation latency. The flash operation latency is not constant even when accessing pages in the same block. Additionally, the average flash operation latency of each flash device could have been different, even if all devices were produced by the same process. We measured the flash operation latency using the monitor module and found a deviation up to tens of μs. The deviation prevented each flash device from executing the flash operation evenly. The FTL assigned the flash operation to each flash device evenly by using the regular order, as shown in Figure 23. However, because of the deviation, the order of completion of the flash operations differed from the order of the FTL. Such a non-ideal case created a situation in which some operations stayed in the reservation station until the flash devices completed the preceding operation, even when some flash devices were in an idle state.

6.5 Performance Variation Depending on Data Distribution

This section analyzes the effect of data distribution on data access parallelism. Depending on how the data requested by the host system is distributed to flash devices, the number of flash devices that can be activated in parallel and the competition situation of the channel bus could be different. To modify data distribution, the address translator uses three location vector assignment schemes: *ChPrev*, *WayPrev*, and *Random*. Each scheme updates the components of the location vector differently to allocate a free location vector to the new data. *ChPrev* updates a channel value prior to a way value, as shown Figure 23. In contrast to *ChPrev*, *WayPrev* updates a way value prior to a channel value. *Random* updates a way value and a channel by using a random value.

Figure 25 shows the system latency for processing various sizes of data. *ChPrev* had the lowest latency in all cases. As the size of data increased, the latency difference between *ChPrev* and *WayPrev* decreased, whereas the latency difference between *Random* and *ChPrev* increased. This is related to the data distribution.

Figure 26 shows the degree of the data distribution. Two indicators are defined to express the degree. *devC* is the standard deviation of the size of the data distributed to each channel. *devW* is the standard deviation of the size of the data distributed to each way. *ChPrev* kept *devC* at zero in all cases. This meant that the storage controller efficiently performed channel interleaving because the data was distributed evenly across all channels. While *WayPrev* kept *devW* at zero, it did not

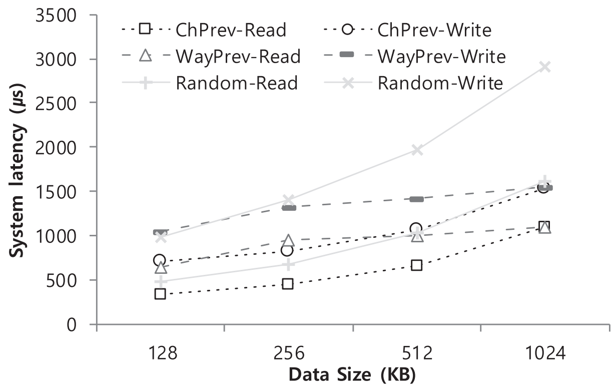

Fig. 25. System latency of the 8-channel, 8-way configuration in SLC mode.

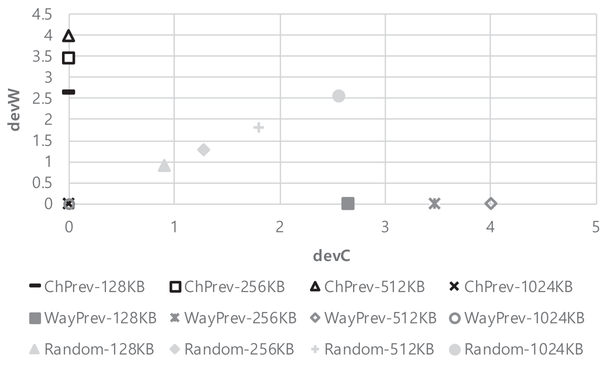

Fig. 26. Degree of the data distribution.

distribute the data evenly to each channel if the data were not large enough. Therefore, when the data size was smaller than 1,024 KB, the latency of *WayPrev* was larger than that of *ChPrev*.

Because *Random* does not guarantee even data distribution for all flash devices, data can be intensively distributed on specific flash devices. Unlike other schemes, *devC* and *devW* remained above zero, even when processing 1,024 KB data, which could be distributed evenly across all flash devices. When the data size was small, the latency of *Random* was smaller than that of *WayPrev* because the data was not likely to be intensively stored in some flash devices and *Random*'s *devC* was smaller than *WayPrev*'s *devC*. However, as the size of the data increased, the latency of *Random* increased drastically. Although the data distribution was the same in the case of processing the read workload and write workload, the read latency of *Random* was smaller than that of *WayPrev*, whereas the write latency of *Random* was larger than that of *WayPrev* in the case of processing 256 KB data. Because the read latency of a flash device is not much different from the transfer time of the data of one page through the channel bus, there is no significant difference between the time when read operations are performed by several ways in parallel and the time when few read operations are performed by a single way. Therefore, even with a non-uniform data distribution, the read latency of *Random* was small on average because the gain from spreading the data over the channels was large. However, because the write latency was very large compared to the transfer time of the data, the write latency of *Random* was large in the same situation.

Table 8. Reconfigured Components for Case Studies

Component	Section 7.1	Section 7.2	Section 7.3	Section 7.4	Section 7.5
Processor	O	X	X	X	X
Host Interface Controller	X	X	X	X	O
Cosmos+ Firmware	X	X	X	O	O
Storage Controller	X	O	O	O	X

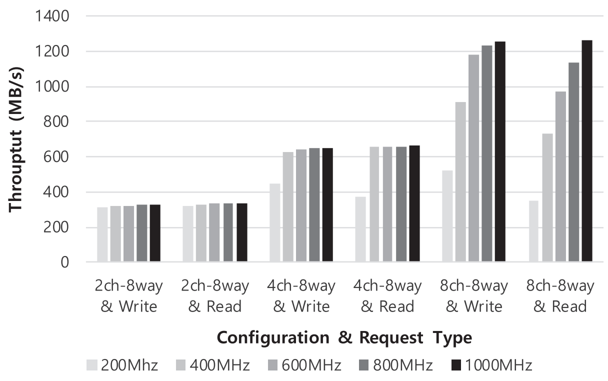

Fig. 27. Throughput of the prototype with various processor-clock frequencies.

7 CASE STUDY

Cosmos+ OpenSSD implements a flash-storage device using reconfigurable hardware and software components. Therefore, it can be used to explore new structures with flash storage devices. This section presents five case studies that show the possibility of using Cosmos+ OpenSSD for architecture exploration. The reconfigured components for each case are listed in Table 8. Each case shows the effect of the particular component on the overall system or illustrates the scope of research that the platform can support.

7.1 Impact on Performance of Processor-Clock Frequency

The processor-clock frequency is a key ingredient of storage-system performance. To decrease the cost and power consumption of a storage system, a processor often runs at a low frequency. However, it should be able to provide enough computation capability to run the function of the firmware, including NVMe manager, FTL, and low-level scheduler in a given amount of time. The embedded processor of the platform board has special registers to scale the processor-clock frequency, which can be easily changed by adjusting the register value. Because the FPGA of the platform board has multiple clocks, the processor-clock does not affect all of the components. In Figure 18, only the application processor unit and the central interconnect are affected by the processor-clock. Even if the processor-clock frequency changes, the bandwidths of Buses 0 to 5, and 10 and 11 in Table 6 remain the same. Because only a small portion of the bandwidth of the control path was actually used, the changed bandwidths of Buses 6 to 9 had a negligible impact on the storage performance.

Figure 27 shows the impact on the storage throughput of the processor-clock frequency. In all cases, the operating frequency of the storage controller and the host interface controller was not changed. As shown in the figure, the storage performance was more affected by the clock frequency as the number of flash devices increased. When the frequency decreased from 1,000 Mhz

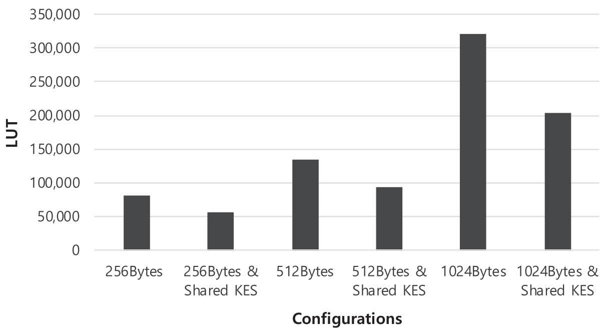

Fig. 28. Hardware resource usage of BCH ECC decoders having various chunk size.

to 200 Mhz, a storage with 2-channel, 2-way configuration experienced up to 6% performance degradation, and one with 8-channel, 8-way configuration had up to 72% degradation. In addition, the throughput became saturated at a lower clock frequency when there were fewer flash devices, and the read throughput was more sensitive to the frequency decrease because the read operations took less time in flash devices than the write ones. For an 8-channel, 8-way configuration, the read throughput decreased by up to 72%, while the write throughput decreased by up to 58%.

Because software-based scheduling was applied, the number of flash devices had a large impact on firmware overhead. As mentioned, we used software-based scheduling for easy reconfiguration. The number of flash devices had a direct impact on the low-level scheduler behavior. The low-level scheduler accessed the register of the storage controller for issuing the new operation or checking the status of the operation being performed by the flash device. The register access time was hundreds of ns, which was very large compared to the processor's operating time. As the number of flash devices increased, the low-level scheduler was forced to access the registers more often, because the number of operations that could be performed in parallel increased. In 8-channel, 8-way configurations, a significant portion of the firmware operation time was used by the issuer and status checker of the low-level scheduler. The issuer took 17.31%, and the status checker took 19.01%. The Cosmos + OpenSSD system uses 1 GHz as the default clock frequency so that the processor can handle the firmware overhead, even if all flash devices are operated.

7.2 Hardware Resource Usage of Error Correction Code Engine

To implement the storage controller with an FPGA that has hardware resource limits, it is important to design the ECC engine, which uses a large number of hardware resources, efficiently. There are many factors determining the hardware resource usage of an ECC engine. For example, the hardware resource usage can vary significantly depending on the type of ECC, the encoding/decoding units, and the error correction capabilities. The encoding/decoding unit is referred to as the chunk in this section. Because the storage controller is modular, it is possible to apply various ECC engines to it. In this section, we compare the hardware resource usage of ECC engines and explain the background for the design of the ECC engines included in the proposed design.

Because an ECC decoder uses most of the hardware resources of an ECC engine, we compared the resource usage of the ECC decoders. The hardware resource usage of the ECC decoders for each flash channel controller was summed. BCH-based ECC engines were used, and each ECC engine was designed to have a similar code rate despite having a different chunk size.

Figure 28 shows the hardware resource usage of the ECC decoders with various chunk sizes. Because an FPGA mounted on the platform board provides 218,600 LUTs, and the hardware components use approximately 70,000 LUTs excluding the ECC decoder, the ECC decoder should use

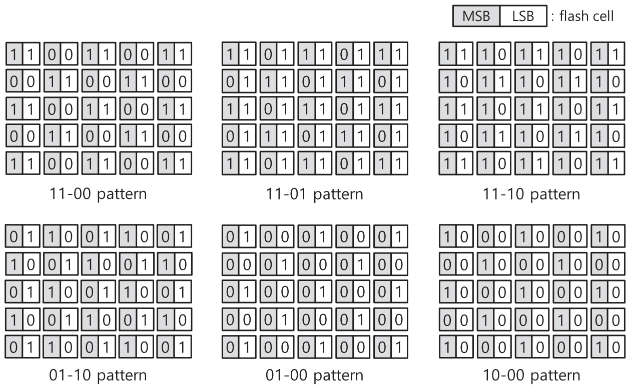

Fig. 29. Data patterns.

less than 148,600 LUTs. The ECC decoder with a chunk size larger than 1 KB is too large, even with a shared KES. That is, the ECC engine should use a chunk size smaller than 1 KB, which implies that LDPC is not suitable for use with the platform board. The LDPC-based ECC decoder uses more hardware resources than the BCH-based ECC decoder for a chunk size smaller than 1 KB [16]. To conserve hardware resources for future expansion, the ECC engine with a chunk size of 256 bytes was applied in the proposed design.

7.3 Impact on Data Integrity of Data Scrambler

To measure the reliability impact of a data scrambler, we prepared two storage controllers: one with a data scrambler, *Scr*, and another without a scrambler, *NoScr*. Because the storage controller is modular, the data scrambler can be easily removed. The flash cell of the MLC NAND device has four voltage levels to distinguish the two bits of data. The data pattern with which the cells having the highest voltage level and the cells with the third-highest voltage level surround one another were most susceptible to interference [5]. Because there was a limitation in confirming the relation between the data and voltage levels, various data patterns shown in Figure 29 were used for measuring the bit errors. *Scr* writes a randomized version of the given data pattern. The bit error was measured by the internal monitor module implemented inside the controller. The monitor module counted the bit-error information extracted from the ECC engine during the ECC decoding process and sent the bit-error count to the operation monitor.

Figure 30 shows an average error bit count as the program/erase (P/E) cycle increases, where the average error bit count is the average number of bit errors observed in a page. For low P/E cycles, there was no significant difference in the error bit counts of two controllers. However, for high P/E cycles, the average count of *Scr* was lower than that of *NoScr* except for the 11-00 data pattern. When the P/E cycle was 5,000, the average count of *Scr* became up to 13.7% lower than that of *NoScr*. The data scrambler was almost ineffective for the 11-00 pattern.

7.4 One-Shot Program for 3D TLC NAND

By replacing the flash memory module, various flash devices can be applied to the Cosmos+ OpenSSD system. In some cases, reconfiguration of the storage controller and the Cosmos+ firmware is required to support the characteristics of the flash memory module being attached. We reconfigured the prototype to use a flash memory module comprising 3D TLC NAND flash devices. The special feature of this device allows programming of three pages sharing the same word line at once. This program method is called the one-shot program. The operation packer was

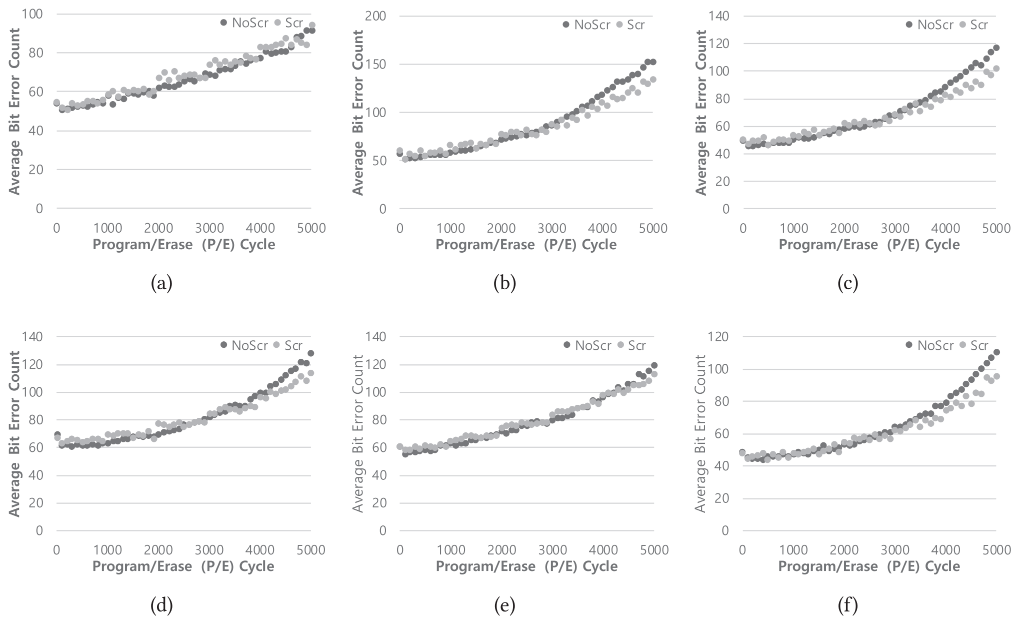

Fig. 30. Average bit-error count per page: (a) 11-00 pattern; (b) 11-01 pattern; (c) 11-10 pattern; (d) 01-10 pattern; (e) 01-00 pattern; (f) 10-00 pattern.

added to the FTL to support the one-shot program. It packs the three write operations that access the same word line among the flash operations generated by the address translator and sends them to the dependency checker. The dependency checker handles the packed operations as the single flash operation. The command dispatcher and flash driver of the storage controller were reconfigured to support the command set and timing condition of the 3D TLC NAND flash device. In addition, the storage controller driver of the firmware was changed to support the modified command set.

To analyze the effect of the one-shot program on the performance of the storage system, channel utilization was measured when the prototype stored a large amount of data. Channel utilization is the percentage of the activation time of the channel data bus per unit time. Because the flash device does not perform the flash DMA operation while performing the flash operation to protect data in the data register, the amount of data stored is the same as the amount of data transmitted through the channel data bus. Therefore, the higher the channel utilization, the greater the write throughput.

Figure 32 compares the channel utilization of the legacy prototypes with that of the reconfigured prototype using 3D TLC NAND devices when the host system writes large amounts of data to the prototypes. The latency of the one-shot program operation was 1,600 μs, which was larger than the write flash operation latency of the MLC mode legacy prototype (1,388 μs). However, the channel utilization of the reconfigured prototype was higher than that of the MLC mode legacy prototype because of the difference in the unit of flash operations. To store three pages, the reconfigured prototype performs one flash operation, but the MLC mode performs three flash operations. Figure 31 shows an example of the same. Because the execution time of a one-shot program is shorter than that of three normal write flash operations, more data can be stored to the reconfigured prototype for a given time.

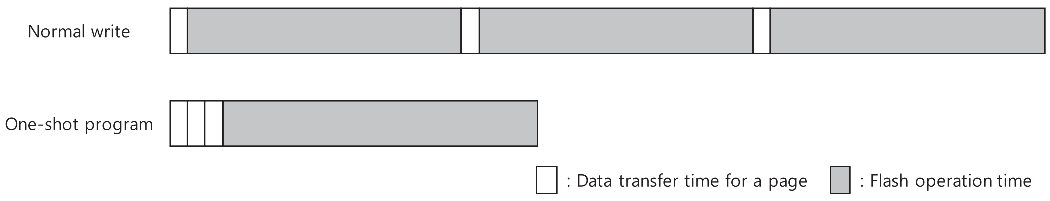

Fig. 31. Comparison of the one-shot program with the normal write flash operation for the process of storing three pages.

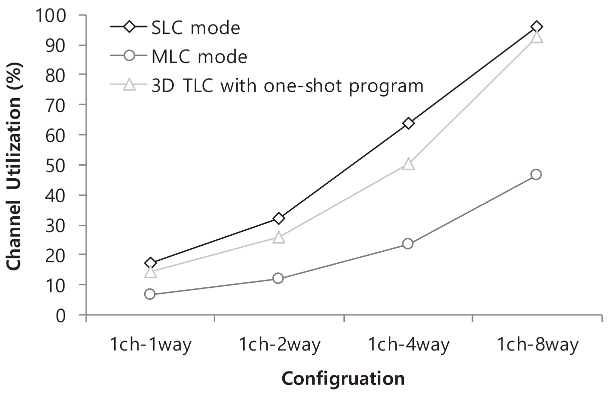

Fig. 32. Channel utilization comparison of the legacy prototypes and the reconfigured prototype.

7.5 Open-Channel SSD Implementation

The Cosmos+ OpenSSD system can be used as the Open-Channel SSD [3] system by reconfiguration. The Open-Channel SSD uses the NVMe standard interface with an extended command set, including custom commands. Several command handlers have been added to the NVMe manager to handle the extended command set. Custom commands can contain multiple physical addresses, in which case the physical address information cannot be included in the payload. To handle these commands, the added command handler can independently generate the host DMA command so that the physical address information can be retrieved from the main memory of the host system. The address translator converts the physical address information into a location vector using a simple conversion formula. Because the FTL no longer needs to maintain a mapping table, the amount of memory used by the Cosmos + firmware decreases. Furthermore, for the device driver of the host system to recognize the reconfigured design as the Open-Channel SSD, the configuration space of the PCIe controller was modified to match the vendor ID and device ID of the Open-Channel SSD registered in the peripheral component interconnect special interest group (PCI-SIG).

Figure 33 shows the throughput of the reconfigured prototype in the same experiment in Section 6.4. SLC mode 8-channel, 8-way configuration was used, and the operating system of the host PC was changed to Linux. Unlike that of the legacy prototype, the write throughput of the reconfigured prototype was steady regardless of the number of outstanding I/Os because *pblk*, which performs the main function of the FTL in the host system, acted as a write buffer. To analyze the effect of *pblk*'s buffering on the write bandwidth, the write latency was measured at the application and the prototype.

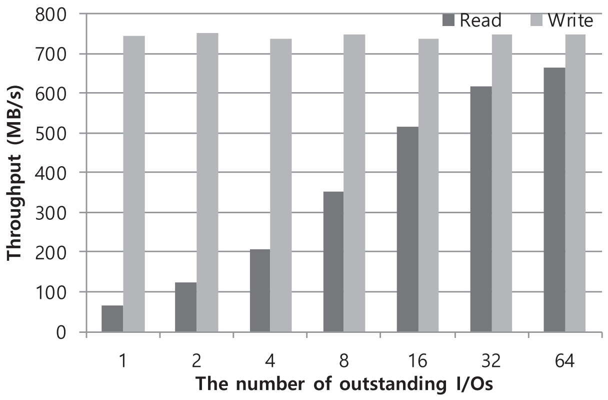

Fig. 33. Throughput of the prototype reconfigured for Open-Channel SSD.

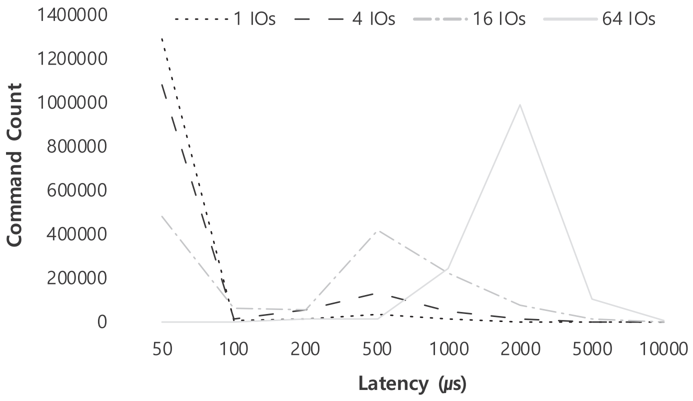

Fig. 34. Write latency distribution measured by the Iometer.

Figures 34 and 35 show the write latency distribution measured by the Iometer and the monitor module of the prototype. Figure 34 shows that there are many commands having a latency less than the flash operation latency in Table 5 because of *pblk*'s buffering. As the number of outstanding I/Os increased, the latency distribution measured by the Iometer moved to the right, because the write buffer of *pblk* filled up faster. However, the latency distribution measured by the monitor module was constant, despite the number of outstanding I/Os, as shown in Figure 35. This is because *pblk* always evicts a certain amount of data to the prototype when the write buffer is full.

As shown in Figure 33, except for the write throughput when the number of outstanding I/Os was small, the measured throughputs of the reconfigured prototype were lower than those of the legacy prototype. To analyze the reason for the degradation, using the monitor module, we measured the number of outstanding host commands being executed by the reconfigured prototype and the number of active channels and ways when the number of outstanding I/Os was 64. The number of outstanding host commands could be varied based on the number of outstanding I/Os because of the existence of *pblk* between the application and the prototype. The number of outstanding I/Os was measured in the application, and the number of outstanding host commands was measured in the prototype. Herein, the number of active ways denotes the average number of active ways of each channel.

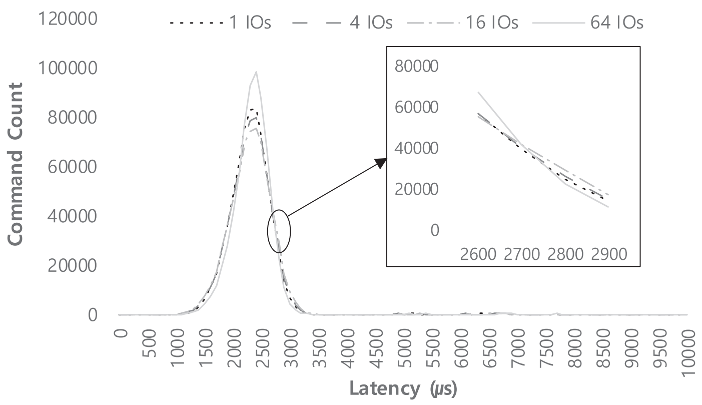

Fig. 35. Write latency distribution measured by the monitor module.

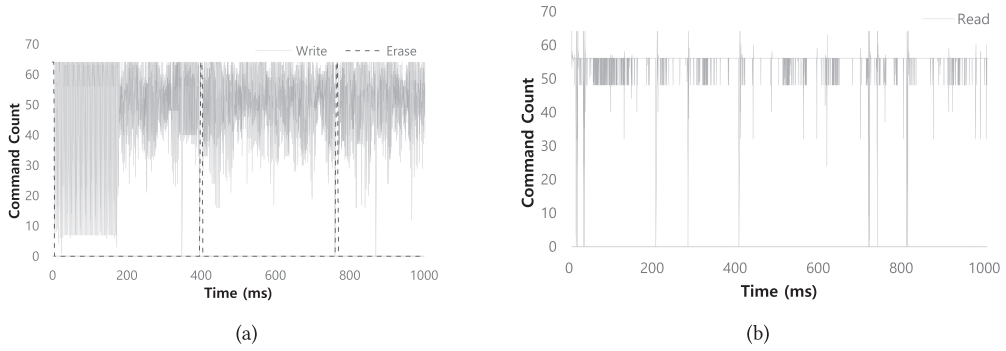

Fig. 36. Number of outstanding host commands: (a) when processing the write workload; (b) when processing the read workload.

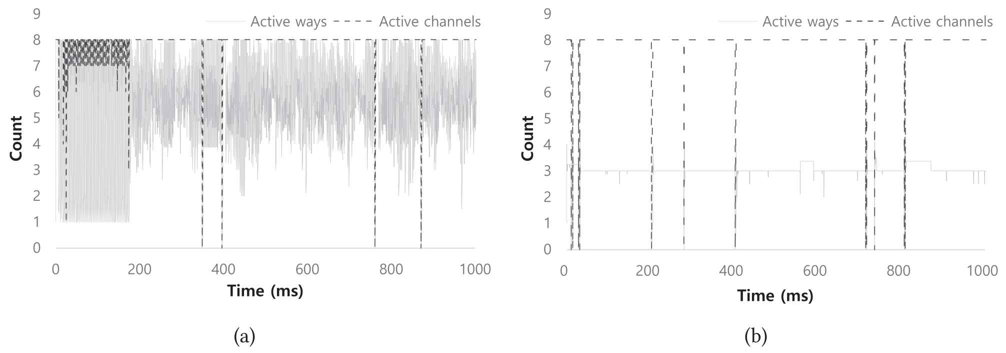

Fig. 37. Number of active channels and ways: (a) when processing the write workload; (b) when processing the read workload.

Figures 36 and 37 show the measured results. To take full advantage of data access parallelism for evicting data from the write buffer, *pblk* maintains the number of write commands required to activate all flash devices of the NAND flash array. In the ideal case, the number of outstanding write host commands is maintained at 64. When a specific command is completed, a new command is entered within a short time in the ideal case. However, as shown in Figure 36(a), the

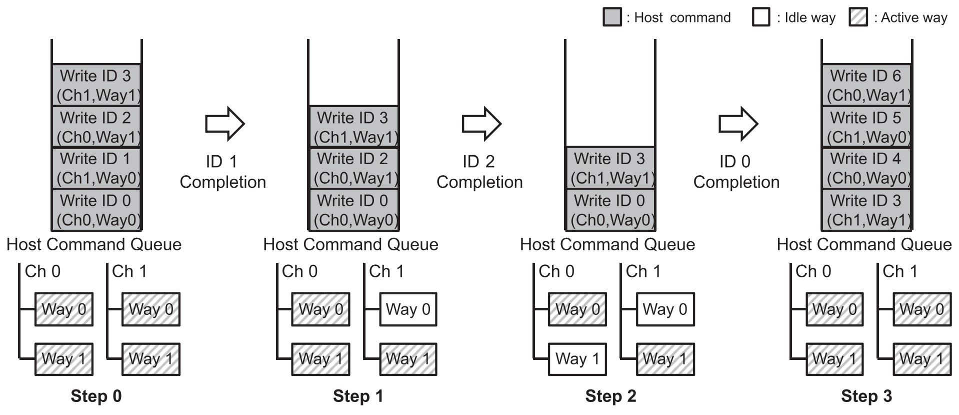

Fig. 38. Host command processing example for the Open-Channel SSD prototype.

number of outstanding host commands remained below 64 for a significant amount of time. This degraded the throughput because the prototype cannot activate all flash devices when the number of outstanding host commands is insufficient, as shown in Figure 37(a).

Although processing the read workload was not as severe as processing the write workload, the number of outstanding host commands was not maintained at 64 for the read workload, as shown in Figure 36(b). There were two reasons that the read throughput was lower than the write throughput. One was a mismatch between *pblk* and the storage controller. The flash device attached in the flash memory module has two planes; however, the storage controller does not support multi-plane operations. Because the address translation algorithm of *pblk* assumes that the storage device supports multi-plane operations, *pblk* sequentially allocates two pages per way. As shown in Figure 37(b), the number of active ways was four or less because each way was accessed by two host commands. The number of active ways of the read workload was different from that of the write workload because each write host command wrote two pages. The second reason was an implementation issue with regard to checking the physical address information. The process for checking the information was not suitable to the legacy firmware structure. The host DMA operation for checking the information regarding a single host command took 1.4 μs on average. Because the FTL cannot generate new operations without this information, the NVMe manager should wait until the completion of the host DMA operation. The longer execution time of the NVMe manger delayed the executions of the FTL and the low-level scheduler. Because of the difference in the flash operation latency, the read throughput was affected more by the second reason than was the write throughput.

To analyze the reason for the number of outstanding host commands being less than that in the ideal situation, the monitor module extracted information regarding the processing of the host commands of the write workload. The order of the accessing of the flash devices of the host commands had a certain sequence similar to that in Figure 23, and there was no host command to simultaneously access the same flash device. Figure 38 shows an example of the same. In Step 0, four host commands are entered to the prototype. The next sequence of accessing the flash devices is for the device attached to way 0 of channel 0. Because there is outstanding host command (ID 0) that is accessing the flash device of the next sequence, the next host commands (ID 4-6) do not enter to the prototype in Steps 1 and 2. Because of the absence of new host commands, the NAND flash array is not fully activated in Steps 1 and 2. This seems to be related to the characteristics of *pblk*. However, the analysis of *pblk* is beyond the scope of this article. We will conduct a detailed analysis of *pblk* as we proceed to optimize the prototype for the Open-Channel SSD in a future work.

8 CONCLUSION

This article introduced a fast flash storage prototype called Cosmos+ OpenSSD. The prototype has the following four features. First, the major components of the flash storage system can be reconfigured. Second, it supports a high-speed host interface to work in a realistic environment. Third, a multi-channel, multi-way organization can be reconfigured easily. Finally, the internal operation status of the flash storage system can be analyzed in real time. Based on these features, Cosmos+ OpenSSD can be used to explore hardware architecture and evaluate firmware efficiency of a flash storage system. The hardware and firmware components whose design is open to the public at source code level can be freely reconfigured on an FPGA platform. The prototype can also be used for the research of other storage architectures and applications, such as Open-Channel SSD and in-storage processing.

REFERENCES

[1] Seungyong An, Hoyoung Tang, and Jongsun Park. 2015. A inversion-less Peterson algorithm based shared KES architecture for concatenated BCH decoder. In *2015 International SoC Design Conference (ISOCC)*. IEEE, 281–282.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[3] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *FAST*. 359–374.

[4] Yu Cai, Erich F. Haratsch, Mark McCartney, and Ken Mai. 2011. FPGA-based solid-state drive prototyping platform. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 101–104.

[5] Jaewon Cha and Sungho Kang. 2013. Data randomization scheme for endurance enhancement and interference mitigation of multilevel flash memory devices. *ETRI Journal* 35, 1 (2013), 166–169.

[6] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 266–277.

[7] John D. Davis and Lintao Zhang. 2009. FRP: A nonvolatile memory research platform targeting NAND flash. In *Proceedings of 1st Workshop on Integrating Solid-State Memory in the Storage Hierarchy*.

[8] NVMe Express. 2017. NVM Express Specification 1.3. Retrieved January 21, 2019 from http://nvmexpress.org/resources/specifications/.

[9] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. 2014. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–11.

[10] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. 2018. Amber*: Enabling precise full-system simulation with detailed modeling of all SSD resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 469–481.

[11] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 153–165.

[12] Kin-Chu Ho, Po-Chao Fang, Hsiang-Pang Li, Cheng-Yuan Michael Wang, and Hsie-Chia Chang. 2013. A 45nm 6b/cell charge-trapping flash memory using LDPC-based ECC and drift-immune soft-sensing engine. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 222–223.

[13] Jhuyeong Jhin, Hyukjoong Kim, and Dongkun Shin. 2018. Optimizing host-level flash translation layer with considering storage stack of host systems. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*. ACM, 75.

[14] Myoungsoo Jung, Ellis H. Wilson III, and Mahmut Kandemir. 2012. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 404–415.

[15] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. 2017. SimpleSSD: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters* 17, 1 (2017), 37–41.

[16] Jonghong Kim, Junho Cho, and Wonyong Sung. 2010. Error performance and decoder hardware comparison between EG-LDPC and BCH codes. In *2010 IEEE Workshop On Signal Processing Systems*. IEEE, 392–397.

[17] J. Kim, Y. A. Winata, and I. Shin. 2016. Multi-thread flash translation layer for multi-core solid state drives. *International Journal of Applied Engineering Research* 11, 2 (2016), 1187–1191.

[18] Jaehyun Kim, Yuli Aria Winata, and Ilhoon Shin. 2015. Flash translation layer using multi-thread. *Advanced Science and Technology Letters (ASTL)* 117 (2015), 43–46.

[19] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. Flashsim: A simulator for nand flash-based solid-state drives. In *2009 1st International Conference on Advances in System Simulation (SIMUL'09)*. IEEE, 125–131.

[20] Jongmin Lee, Eujoon Byun, Hanmook Park, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2009. CPS-SIM: Configurable and accurate clock precision solid state drive simulator. In *2009 ACM Symposium on Applied Computing*. ACM, 318–325.

[21] Kihoon Lee, Han-Gil Kang, Jeong-In Park, and Hanho Lee. 2012. A high-speed low-complexity concatenated BCH decoder architecture for 100 Gb/s optical communications. *Journal of Signal Processing Systems* 66, 1 (2012), 43–55.

[22] Sungjin Lee, Kermin Fleming, Jihoon Park, Keonsoo Ha, Adrian Caulfield, Steven Swanson, Jihong Kim, et al. 2010. BlueSSD: An open platform for cross-layer experiments for NAND flash-based SSDs. In *WARP-5th Annual Workshop on Architectural Research Prototyping*.

[23] Shu Li and Tong Zhang. 2010. Improving multi-level NAND flash memory storage reliability using concatenated BCH-TCM coding. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 10 (2010), 1412–1420.

[24] Bo Mao and Suzhen Wu. 2015. Exploiting request characteristics and internal parallelism to improve SSD performance. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 447–450.

[25] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. 2011. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers* 60, 5 (2011), 653–666.

[26] Ivan Luiz Picoli, Carla Villegas Pasco, Björn Þór Jónsson, Luc Bouganim, and Philippe Bonnet. 2017. uFLIP-OC: Understanding flash I/O patterns on open-channel solid-state drives. In *8th Asia-Pacific Workshop on Systems*. ACM, 20.

[27] Iometer Project. 2014. Iometer 1.1.0. Retrieved January 21, 2019 from http://www.iometer.org/.

[28] OpenSSD Project. 2011. Jasmine OpenSSD. Retrieved January 21, 2019 from http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[29] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *OSDI*. 67–80.

[30] Leilei Song, Meng-Lin Yu, and Michael S. Shaffer. 2002. 10-and 40-Gb/s forward error correction devices for optical communications. *IEEE Journal of Solid-state Circuits* 37, 11 (2002), 1565–1573.

[31] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. 2014. Cosmos openSSD: A PCIe-based open source SSD platform. *Flash Memory Summit* (2014).

[32] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*. 49–66.

[33] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1260–1267.

[34] Hung-Yuan Tsai, Chi-Heng Yang, and Hsie-Chia Chang. 2012. An efficient BCH decoder with 124-bit correctability for multi-channel SSD applications. In *2012 IEEE Asian Solid State Circuits Conference (A-SSCC)*. IEEE, 61–64.

[35] Debao Wei, Youhua Gong, Liyan Qiao, and Libao Deng. 2014. A hardware-software co-design experiments platform for NAND flash based on Zynq. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1–7.

[36] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. 2013. Vssim: Virtual machine based SSD simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[37] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *USENIX Annual Technical Conference*. 87–100.

Received January 2019; revised December 2019; accepted February 2020