Decoupled SSD: Rethinking SSD Architecture through Network-based Flash Controllers

Jiho Kim
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
jihokim@kaist.ac.kr

Myoungsoo Jung
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
mj@camelab.org

John Kim
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
jjk12@kaist.edu

ABSTRACT

Modern NAND Flash memory-based Solid State Drives (SSDs) are designed to provide high-bandwidth for I/O requests through high-speed NVMe interface and increased internal flash memory bandwidth. In addition to providing high performance for incoming I/O requests, the flash translation layer (FTL) also handles other flash memory management processes including garbage collection that can negatively impact I/O performance. In this work, we address how the sharing of system resources (e.g., system-bus and DRAM) for I/O requests and garbage collection can cause interference and performance degradation. In particular, we propose to rethink SSD architecture through a *Decoupled SSD* (dSSD) system that decouples the front-end (i.e. cores, system-bus, DRAM) with the back-end (i.e. flash memory). A flash-controller network-on-chip (fNoC) that interconnects the flash controllers together is introduced to enable decoupling of the I/O path and garbage collection path to improve performance and reliability. dSSD enables advanced commands such as copyback command to be exploited for efficient garbage collection and we propose to extend copyback command with global copyback through the fNoC. To improve reliability, we propose to recycle superblocks through superblock recycle table within the flash controller. Without any modification to the FTL, a hardware-based offloading mechanism within the flash controller of the dSSD is proposed to dynamically re-organize a superblock. Our evaluations show that decoupled SSD results in up to 42.7% I/O bandwidth improvement and 63.8% GC performance improvement, while achieving approximately 31.4× improvement in tail-latency on average. Dynamic superblock management through the dSSD results in approximately 23% improvement in lifetime with minimal impact on performance and cost.

CCS CONCEPTS

• **Computer systems organization → Interconnection architectures**; • **Information systems → Storage architectures**.

KEYWORDS

Solid-state drives, flash controller, garbage collection, on-chip network, superblocks

ACM Reference Format:
Jiho Kim, Myoungsoo Jung, and John Kim. 2023. Decoupled SSD: Rethinking SSD Architecture through Network-based Flash Controllers. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589096

1 INTRODUCTION

The bandwidth of NAND flash memory-based Solid State Drive (SSD) has increased with the improvement of flash memory technology and high-speed I/O host-interface [8]. To support high-bandwidth of read/write I/O requests, internal parallelism of an SSD is exploited (e.g., multi-channels, chips, dies, etc.) and multi-plane commands [35] enable additional bandwidth through the multiple (parallel) planes. As I/O input bandwidth continues to increase and result in more I/O requests, more *external* data movement occurs in the SSD — i.e., data within the SSD is moved to/from the host. In addition to external data movement, *internal* data movement can occur in SSD when garbage collection (GC) occurs and valid pages are copied to the destination free-blocks, and data movement is constrained to be within or internal to the SSD. However, GC uses the same system resources as I/O requests, including the core, DRAM, and the system bus and the conflicts between the two types of data movement can cause interference and impact overall SSD performance and throughput.

To address interference between GC and I/O, most prior work addressed the flash memory conflicts [17, 24, 35, 42] as the flash memory bandwidth was often the bottleneck. However, system resource (especially the system bus) is becoming a more critical bottleneck as the number of planes and/or the unit of data transfer per plane increases.[1] This not only increases flash memory parallelism (and bandwidth) but it also impacts system resource utilization, in particular the system-bus, as the I/O requests can be heavily impacted by garbage collection. In this work, we show this performance bottleneck is caused by a *tightly coupled* modern SSD system between the "front-end" or the system resources (i.e., system bus, DRAM, etc.) and the "back-end" or the flash memory chips. For example, when data transfer or copy is done for garbage collection, it consists of chip-to-chip (or die-to-die) data movement; however, the valid pages are sent to the system components (e.g., DRAM and ECC) through system-bus before arriving at their destination flash memory.

[1]Ultra-low latency (ULL) flash [6] have 2 or 4kB page size with 8 (or 16) planes while recent T/QLC [13] page sizes are 16 or 32KB and have 2 (or 4) planes.

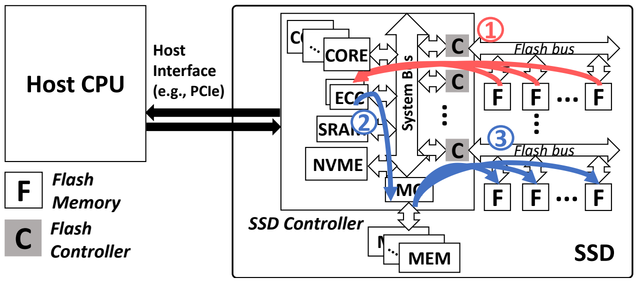


Figure 1: Data movement during the execution of garbage collection (GC) in modern SSD systems.

To overcome this bottleneck, we propose to rethink the design of SSD architecture through the ***decoupled SSD*** system where the *front-end* is decoupled from the *back-end* or the flash memory chips [21]. With the introduction of a decoupled flash controller, the decoupled SSD minimizes data movement during garbage collection as the front-end SSD resources are not utilized. In addition, we propose a dedicated flash controller network-on-chip (fNoC) that interconnects the flash controllers together. The decoupled SSD (dSSD) and the fNoC enable new opportunities in the design of the SSD controller as the back-end is "decoupled" and operates independently of the front-end. As a case study, we demonstrate how advanced commands such as copyback can be implemented with dSSD. Copyback commands [12, 31] usage is very limited in modern SSDs because of error propagation during the copy (or a read followed by a write). By decoupling the SSD, copyback commands are now handled by the flash controller and enable error check (and correction) for copybacks. In addition, existing copyback commands are effectively *local* copyback since copies occur within the same plane (die). However, the fNoC provides a mechanism to "route" data to not only enable *local* copyback but we propose *global* copyback where the destination of the data transfer can be located on the same flash bus channel or located across different channels to increase the flexibility of copyback.

We also propose how superblock reliability can be managed in a dSSD through the decoupled flash controller. In particular, we propose dynamic superblock management through *recycled* blocks – i.e., when bad blocks occur within a superblock, instead of discarding the entire superblock, the remaining good blocks within the superblock are reused or recycled to improve SSD reliability. This improvement in reliability does not require any support from the FTL but is achieved through the dSSD architecture through extra mapping tables within the decoupled flash controller of the dSSD. In summary, the main contributions of this work include the following.

- We propose Decoupled SSD (dSSD) that decouples the front-end of the SSD controller with the back-end that consists of the flash channel/chips in an SSD. In particular, a flash controller network-on-chip (fNoC) is introduced to enable flash-to-flash communication.
- By exploiting the flash-to-flash connectivity and the decoupled controller, we propose how advanced copyback command can be enabled to minimize data movement for garbage collection while reducing interference with I/O traffic.
- We propose *dynamic* superblock organization to improve reliability through remapping tables placed within the decoupled flash controllers by introducing *recycled* blocks.

2 BACKGROUND

In this section, we provide a background on modern SSD that consist of not only the SSD controller and flash memory but also the flash translation layer (FTL). In particular, we describe how modern SSDs are effectively "coupled" architecture between the front-end that consists of core/DRAM/system-bus and the back-end that consists of the flash memory chip. In addition, FTL is a critical component in modern SSD but as reliability challenges continue to increase in modern SSD, prior work often requires FTL support to improve reliability and further increase FTL complexity. In this work, we show how decoupled SSD can improve reliability without support from FTL by exploiting the decoupled flash controller.

2.1 SSD Architecture

A modern SSD controller consists of multiple subsystems including host interface, multi-cores, memory, error correction code (ECC), system-bus, and multiple flash controllers as shown in Fig 1. The host interface (e.g., NVMe) controller receives the arriving I/Os, and interprets the commands according to the protocol. Recent interface protocols, such as NVMe, have increased SSD I/O bandwidth and the number of concurrent I/O requests. SSD controllers often exploit multi-cores to provide high performance [16, 33, 41, 44]. A significant fraction of DRAM is used as a write-buffer cache [20] by the firmware (or FTL) to hide the relatively slow flash memory latency/bandwidth. The DRAM is also used to store mapping table [18, 25] and meta-data of flash block/pages. The ECC engines detect (and possibly correct) bit errors of pages that have been read for both I/O and garbage collection. Depending on the ECC algorithm and the implementation, a single ECC engine can be placed per flash controller or multiple flash controllers can share a single ECC engine. The different components in the SSD controller are interconnected by a system-bus (e.g., AXI [1]).

2.2 Flash Translation Layer (FTL)

FTL is the software layer within the SSD controller that consists of I/O request handling (including address translation), garbage collection, and wear-leveling. The role of FTL in an SSD architecture has become more significant as the amount of I/O requests (and bandwidth) increases and creates more *external* data movement. However, internal flash management such as page-allocation, garbage collection, wear-leveling, and bad block management also needs to be managed by the FTL. In particular, FTL is responsible for *internal* data movement during garbage collection.

I/O Request Handling: After the I/O requests arrive and are processed by the host interface, the FTL determines whether the requested data is in the DRAM buffer cache. The DRAM and system-bus are utilized if I/O requests result in a *hit*. If there is a *miss*, the FTL translates the logical page number (LPN) of the requests to a physical page number (PPN) based on mapping table [18, 25]. The FTL then issues the requests to the appropriate flash controller across the different channels and the system-bus is still used as the I/O requests move to/from flash memory.

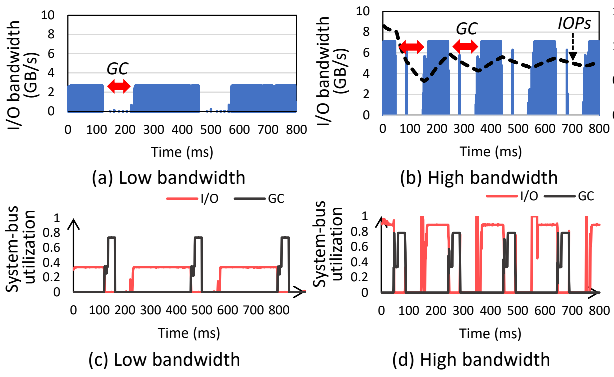


Figure 2: (a,b) I/O bandwidth and (c,d) system-bus utilization with low/high-bandwidth scenario on ultra-low latency device. The horizontal GC arrows show when GC occurs.

Garbage Collection: Garbage collection is a critical component in SSD to optimize storage space as writes are done at page granularity which is much smaller than the block granularity used for erase operations [17]. When the amount of remaining free blocks is less than a predetermined threshold value, FTL starts garbage collection by allocating a new physical page from the active free block, moving valid pages from the victim blocks to free or new blocks and FTL invalidates the pages in the victim block. Afterward, GC erases the victim blocks to optimize storage space. Different algorithms have been proposed to select the victim blocks and the free blocks, with trade-offs in performance [42] and reliability [36]. If the free block chosen is within the same plane as the victim block *local*, *advanced copyback command* [31] can be exploited to avoid transferring data outside of the flash memory chip. However, copyback commands cannot be used in modern SSDs because of error propagation since error correction capability is limited within the flash memory.

Error Handling: Multi-level cells (T/QLC) have effectively increased flash memory density by sacrificing performance [19]; however, it has also increased the *Raw Bit Error Rate (RBER)* by creating additional reliability challenges in read [3],write [5], and retention [4]. While advanced ECC algorithms (e.g., LDPC) are used in SSD [45], the impact from process variation [15, 28, 40] has a significant impact on flash endurance and alternative approaches to improve endurance have been proposed [15, 27, 28]. However, prior work often requires support from the FTL which increases the FTL complexity. In this work, we exploit the decoupled flash controller to improve superblock management without any FTL support.

3 MOTIVATION: CASE FOR DECOUPLED SSD

A high-level overview of data movement during garbage collection (GC) in modern SSD systems is shown in Fig 1. The flash controller sends read commands to the appropriate flash memory die and the valid pages are read out, go through error detection (and possibly correction) (Fig 1 ①), before being sent through the system bus [2] and written out to the DRAM memory (②). Afterward, FTL issues write operation to the flash controller and transfer the pages to the destination flash die by sending write commands to the flash memory chips (③). GC operation is essentially a "back-end" operation that requires the movement of data between different flash memory chips. However, modern SSD systems utilize the front-end, including the system bus and DRAM, during GC, and as a result, GC can interfere with I/O requests.

The performance impact of GC on the I/O requests is shown in Fig 2(a,b) where the I/O bandwidth, defined as the amount of bandwidth provided to the I/O requests is shown on the y-axis. The x-axis is time and we measure I/O bandwidth every 1 msec to observe the change in the I/O bandwidth. We simulate an SSD system based on a ULL device [6] (evaluation setup is described in Sec 6) and evaluate the impact of GC on I/O requests. We used greedy, global free block selections for our evaluation and all flash memory executes GC in parallel. The system-bus bandwidth is modeled as 8GB/s and is equal to the aggregate bandwidth of all flash bus channels. The WRITE bandwidth of an individual flash chip with 1 plane is 51.2 MB/s and 409.6 MB/s for 8 planes. The results are shown for a synthetic workload that consists of 4KB (as well as 32KB) sequential write I/O requests with an outstanding request queue depth of 64. The 4KB access is used to evaluate a "low bandwidth" scenario since only one out of the eight planes is utilized. In comparison, 32KB access results in all 8 planes being accessed using multi-plane commands to model "high bandwidth."

For low bandwidth (Fig 2(a)), approximately 3 GB/s of I/O bandwidth is sustained initially, equivalent to 51.2 MB/s $\times$ 8 channels $\times$ 8 ways; however, the I/O bandwidth drops after the GC is triggered. During GC, a limited amount of I/O requests are processed. Similar behavior can be observed for "high bandwidth" (Fig 2(b)) but the drop is more significant since higher bandwidth of the flash memory is used with multi-plane commands. The maximum bandwidth that can be achieved is approximately 8 GB/s which is equivalent to the system bus bandwidth. It is well-known that GC can interfere with I/O requests [17, 42]; however, as the flash memory bandwidth increases (e.g., exploiting multi-plane in modern SSD devices), the impact of system resources (such as the system-bus) becomes more critical. To understand the impact, we plot the corresponding system-bus utilization in Fig 2 (c,d). When GC is active, the system bus utilization for I/O requests drops; however, the drop in utilization is more significant for the high-bandwidth scenario and the system bus becomes a point of contention between I/O and GC within the SSD.

One potential approach to reduce the impact of interference is to increase the amount of system bus bandwidth. While this can improve performance, it does not fundamentally decouple the front-end (or the SSD controller-side) and the back-end (i.e., flash memory) as the data movement still occurs through the system-bus/DRAM (Fig 1). In this work, we propose a *decoupled SSD* architecture with the following objectives:

- **Minimize data movement:** Flash-to-flash data movement for GC involves both the system bus and DRAM in modern SSD. Decoupled SSD minimizes data movement by providing direct flash-to-flash communication.
- **FTL Complexity:** As the complexity of FTL continues to grow [33, 44], decoupled SSD enables some flash management to be offloaded to the decoupled controller to reduce the complexity of the FTL.

[2] In this work, we refer to the system bus as the bus within the SSD controller that interconnects the flash controllers, core, and other components within the SSD controller (Figure 1).

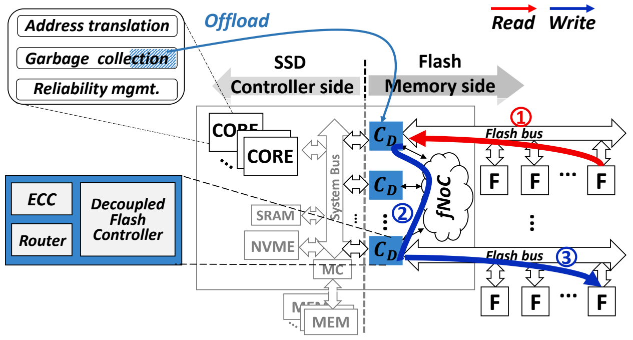


Figure 3: High-level block diagram of proposed decoupled SSD (dSSD). The main added components are the decoupled controller(C_D) and the flash controller network-on-chip (fNoC). (F: Flash Memory)

- **Offloading opportunities:** Advanced commands (such as advanced copyback command) can be implemented or "offloaded" to the decoupled controller. In addition, efficient super-page management can be handled by the decoupled controller, without support from the FTL.

To achieve the objectives outlined, the proposed decoupled SSD is based on the following two principles. First, minimize the impact on FTL while offloading functionality to the decoupled controller [3]. The second principle is to isolate or separate the I/O requests and internal data movement (e.g., GC) as much as possible through a separate dedicated flash controller network-on-chip.

4 DECOUPLED SSD (dSSD) ARCHITECTURE

In this section, we first describe how data movement is minimized by offloading data copy for GC to the flash controller by providing direct data movement between flash memory through the flash controllers. We also describe the dSSD microarchitecture, the flash controller network-on-chip (fNoC) to interconnect the controllers together, as well as the global copyback command that is enabled by dSSD.

4.1 Decoupled SSD Architecture

A high-level overview of the proposed decoupled SSD architecture is shown in Fig 3. The main hardware changes are the *decoupled flash controller* (C_D), and an on-chip hardware router to enable controller-to-controller communication. Since all controllers are located within the same chip [29], a flash-side network-on-chip (fNoC) is introduced that enables direct communication between flash memory chips across different flash channels. As a result, the flash memory-side (or the "back-end") is effectively *decoupled* from the SSD controller side (or the "front-end") through the decoupled flash controller (C_D) and fNoC. More importantly, it reduces contention for shared resources, including the system-bus and the memory, when I/O requests and page copies for GC are handled simultaneously.

An example of flash-side data movement (or copy) is shown in Fig 3. The flash controller reads out pages from a target flash die/blocks as shown in Fig 3①. However, instead of transferring the pages through the system-bus, the pages are stored temporarily in the decoupled flash controller, and error check/correction is performed by the integrated ECC engine. The pages are then sent to the destination across the fNoC that interconnects flash controllers as "packets" and arrives at the destination (Fig 3 ②). After arriving at the destination flash controller, the pages are written to the destination flash die/blocks through the flash bus channels (Fig 3 ③).

A detailed block diagram of a decoupled flash controller (C_D) is shown in Fig 4. The diagram includes components that exist in conventional flash controllers, including the channel command controller, command queue, page buffer, and bus/NAND interface. The I/O commands (i.e., read/write) are received through the bus interface and queued in the command queue while the data (for a write) is queued in the page buffer. The channel command controller interprets the commands and receives the data into one of the page buffers [37] [4]. The channel command controller then generates a sequence of low-level commands to the flash memory and transfers the commands and data (if necessary) to the flash memory according to the low-level flash interface protocol (e.g., ONFI [31]). In our proposed dSSD, the datapath used for the I/O commands is the same as the conventional SSD. Additional changes to the flash controller includes the addition of ECC logic that is incorporated into the controller, the decoupled buffer (dBUF) for flash-to-flash data movement, and the on-chip network interface as well as the fNoC router. With the support of these changes, the key difference compared to conventional SSD is that ***decoupled SSD provides support for internal data movement within the flash memory side without the support of the SSD front-end.***

4.2 Global Copyback Command

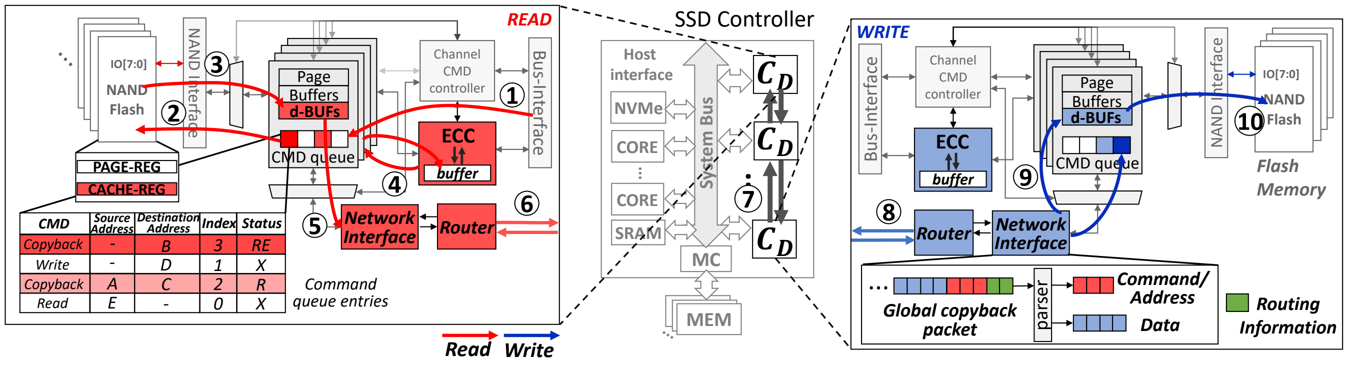
Copyback advanced commands [31] have been proposed to minimize the data transfer overhead of data copy (i.e., a read command followed by a write command). However, copyback commands are rarely used in modern flash memory [2] since error correction cannot be done within the die and result in error propagation. We refer to legacy copyback advanced commands as *local* copyback since the source/destination are restricted to the same flash memory die. In this work, we exploit the decoupled SSD to not only enable local copyback but also propose *global* copyback command. The key difference is that the global copyback command does not restrict the write address (or the destination) to the same flash memory die but the destination can be the same die, another die connected to the same flash channel, or a die/chip that is connected to a different flash bus channel. Thus, a page can be read and written to any location within the SSD using global copyback ***without using system-bus and without any support from the FTL.*** [5]

Since the source and destination of pages are decided by the FTL's garbage collection policy, the read (source) and write (destination) addresses of the page copy are provided to the decoupled flash controller as part of the copyback command. Fig 4 shows an example of how global copyback is executed. The global copyback command is

[3] Note that changes from dSSD are mostly transparent to FTL, aside from supporting copyback commands. However, dSSD does not restrict any flexibility of the FTL.

[4] The page buffer needs to be sized to be as large as one page (e.g., 4kB) per flash memory *way* that is defined as the unit of flash package that shares the flash interface. In our evaluation, we assume 8-ways per channel and size the page buffer to support 16 pages to enable multiplane operations across the multiple ways.

[5] The FTL does need to be aware that copyback command is available and the "destination" during garbage collection can be any other flash memory in the system.



CMD	Source Address	Destination Address	Index	Status
Copyback	-	B	3	RE
Write	-	D	1	X
Copyback	A	C	2	R
Read	E	-	0	X

Command queue entries

Figure 4: Detailed block diagram of a decoupled controller and global copyback data movement between different flash channels.

received via the system bus (①) at the source flash controller and updates the command queue with the copyback command that was just received. The global copyback is executed in multiple stages. The first step includes issuing a low-level read command/address such that the appropriate page is read out from the flash memory (②)[6]. The page that is read is stored into a *decoupled* page buffer (dBUF) (③). In the next step, the command queue moves the page from the dBUF into the ECC engine to detect (and potentially correct) errors (④). If the destination of the copyback is the same channel, a write low-level command is issued and write occurs. However, if the destination of the page is a different flash channel, "packetization" is required within the network interface (⑤) before accessing the router (⑥) and traversing the fNoC (flash-controller network-on-chip)(⑦). When transmitting a packet, the data (or page) is appended with the command information as well as the packet header.

When the packet arrives at the destination node router, the packet is parsed and divided into the command/address and data (page). Since the error check for the data (page) was already done at the source flash controller, no additional error check is necessary. The command/address is inserted into the destination controller's command queue (⑨) and the data is copied to decoupled buffer (dBUF) to avoid interference with the general I/Os. Finally, the write operation is conducted by the command controller (⑩) to finish the internal data movement or the global copyback. During the movement of data, the command queue keeps track of the commands; for the copyback commands, a "status" is also maintained to determine which stage of the command is currently being executed – e.g., `R` identifies that the read has been done, `RE` identifies that error detection/correction has been done after the read, etc. In addition, the command queue also maintains the source and/or destination and the pointer to location within the page buffer that contains the data.

5 DYNAMIC SUPERBLOCK MANAGEMENT

In this section, we describe how dSSD can be exploited to dynamically manage superblocks and improve SSD lifetime without FTL support. By introducing a mapping table within the decoupled controller and fNoC, superblock management can be improved. In particular, we propose *recycled block* that enables efficient usage of

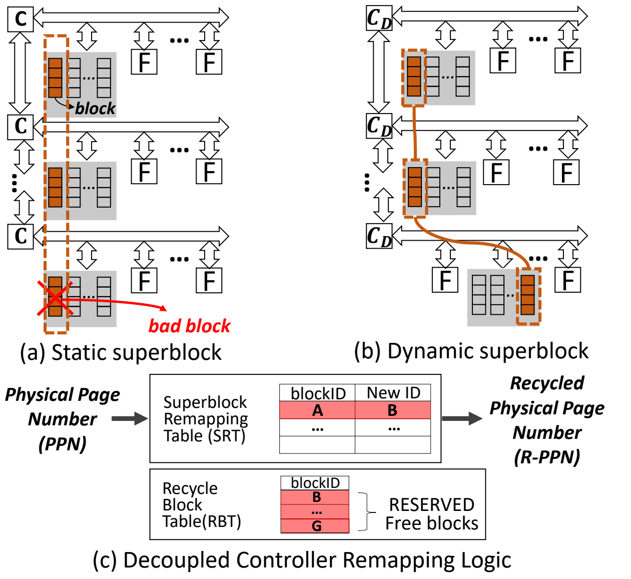

C
F
block
bad block
(a) Static superblock
C_D
(b) Dynamic superblock
Physical Page Number (PPN)
Superblock Remapping Table (SRT)
blockID
New ID
A
B
Recycled Physical Page Number (R-PPN)
Recycle Block Table(RBT)
blockID
B
G
RESERVED Free blocks
(c) Decoupled Controller Remapping Logic

Figure 5: Flash-memory side of an SSD controller for (a) static superblock and an example of a bad sub-block, (b) how dynamic superblock can be created, and (c) a high-level diagram of the remapping logic within the decoupled controller.

superblock through a recycle table – a hardware-based address translation for re-usable sub-blocks[7] of a superblock.

5.1 Recycled Blocks

A superblock [18] is often organized as multiple blocks across different flash channels as shown in Fig 5(a). [8] To simplify superblock organization while maximizing parallelism across the channels, the same block ID across multiple channels (or planes) is grouped together to create a superblock. The advantages of superblock include reduced mapping table size and higher performance from optimizing garbage collection overhead [18]. However, conventional superblock mapping can be inefficient since it does not consider process variation across different flash dies [15, 40]. For example, if an uncorrectable error (detected by the ECC engine) occurs in one of the

[6]Note that a global copyback command does not use (or local) copyback operation [31] even if the page is destined for the same die to avoid error propagation.

[7]We use sub-blocks to refer to blocks that make up a superblock.

[8]For simplicity, each flash memory is shown as a collection of blocks and other flash hierarchy (e.g., die, planes, etc.) are not shown.

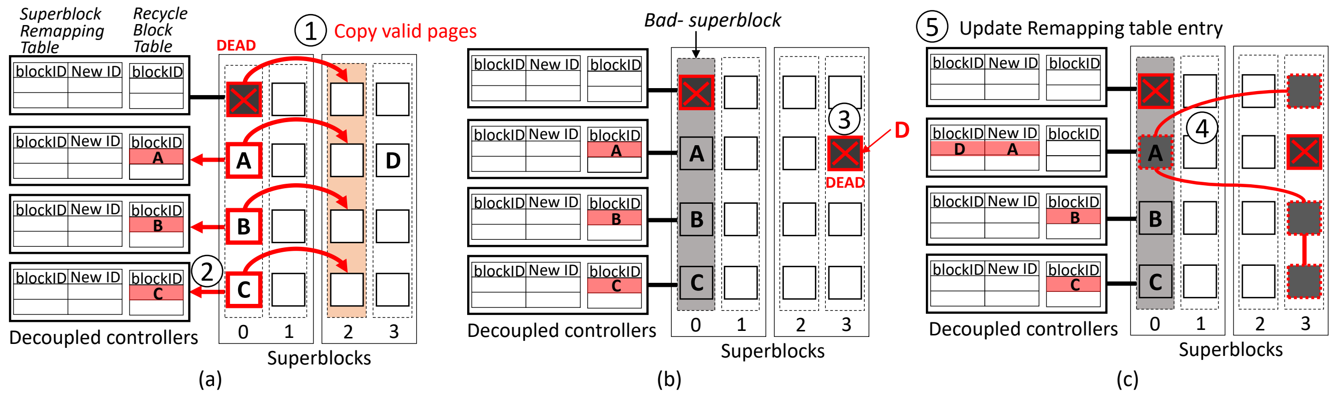


Figure 6: An example of dynamic superblock using superblock remapping table (SRT) and recycle blocks table (RBT). (a) When a bad superblock occurs, entire valid pages are copied to a new superblock by the FTL. At the same time, the re-usable sub-blocks from the superblock are added to the RBT. (b) When another bad superblock occurs, valid pages in the sub-block that are identified as "dead" are moved to a recycled block, and (c) an address remapping of ([D] to [A]) is inserted into the SRT for the new dynamic superblock.

pages in a sub-block of a superblock (Fig 5(a)), other sub-blocks of the superblock are marked as a "bad" block by the FTL to prevent further usage of the given superblock – even though there are sub-blocks within the superblock that can still be used. We refer to this superblock organization as *static* superblock since the same offset across the channels/dies is leveraged to form a superblock.

To overcome the limitation of the static superblock and provide the ability to reuse sub-blocks that are still good, we propose to exploit decoupled SSD by introducing a hardware-based mapping table within the decoupled controller (C_D) to *recycle* sub-blocks from "dead" superblocks. In particular, we propose *dynamic* superblock organization as shown in Fig 5(b) that changes the physical location of sub-blocks within a superblock. The recycled sub-block is *transparent* to the FTL by exploiting the decoupled SSD architecture – thus, there is no overhead to the FTL while the benefits of the superblock are still provided and improve SSD endurance. The key observation that we exploit is that a "dead" superblock does not necessarily mean all of the sub-blocks are dead.

In this work, we define *recycled blocks* as blocks or sub-blocks within a superblock that are still "good" or useful because none of the pages within the blocks have reached the uncorrectable status. [9] To enable dynamic superblock management, we propose a hardware-based remapping to *recycle* valid blocks that remain from superblocks through two tables in the decoupled flash controller – a superblock remapping table (SRT) and a recycle block table (RBT) (Fig. 5(c)). Each decoupled controller has both SRT and RBT and the tables are maintained individually by each controller. The RBT contains information about valid blocks that can be accessed by the controller's flash channel. Entry to the RBT is added when a superblock is "dead" but there are valid sub-blocks. The RBT is effectively a recycling bin of blocks that can be recycled and used as part of a *dynamic* superblock. The SRT is the hardware-based remapping table of the superblock and is updated when a recycled block from the RBT is utilized in a dynamic superblock. The FTL is unaware of the SRT as the dynamic superblock is created without FTL support and extends the lifetime of individual superblocks.

[9] A sub-block consists of multiple pages and each page has a different raw bit error rate (RBER) as program/erase (P/E) cycle increases, but the page with the highest RBER triggers uncorrectable error [40].

5.2 Dynamic Superblock Walk-through Example

An example of how the SRT and RBT tables are modified and utilized is shown in Fig 6. Initially, both tables (superblock remapping table (SRT) and recycle block table (RBT)) are empty. The SRT is not initially accessed by any commands from the command controller during normal operations. Once the first uncorrectable error is observed, the information is communicated to the FTL and the superblock is considered to be a "bad" superblock. FTL then moves all valid pages in the bad superblock to another free superblock (Fig 6(a)-①). These steps are identical to a conventional static superblock. However, the decoupled flash controller where the error was detected (i.e., the top flash channel in Fig 6(a)-②) notifies the other flash controllers that a sub-block in its channel has gone dead. This information is used to determine that their corresponding sub-blocks should be added to the RBT. From the FTL's perspective, the sub-blocks added to the RBT are not "available" but the hardware recycles them for dynamic superblock.

When an another uncorrectable error occurs to a different superblock, (Fig 6(b)-③ at `D` (2nd sub-block of superblock 3)), the second flash controller does not notify the FTL as before since an entry or a recycled block in the RBT is available (Fig 6(c)-④ `A`). The flash controller prolongs the life of the superblock by using the "spare" or the recycled block and updates the superblock remapping table (SRT) (i.e., sub-block 1 in superblock 0) by inserting the mapping `D` $\rightarrow$ `A` (Fig 6(c)-⑤). In addition, the decoupled flash controller performs an internal copy of the valid pages in block `D` to block `A` using global copyback described earlier in Section 4. After the new superblock mapping is dynamically created, FTL continues to access superblock 3 without knowledge of the dynamic superblock remapping. However, within the flash-memory side, any commands destined for 1st sub-block of superblock 3 are internally remapped to the 1st sub-block of superblock 0 (Fig 6(c)).

5.3 Reservation-based Dynamic Superblock

Dynamically re-organizing a superblock improves SSD endurance as the occurrence of bad-blocks (or bad "superblocks") is delayed by utilizing recycled blocks. As shown later in Section 6.4, dynamic superblock helps with the *rate* at which additional bad superblock occurs through better utilization of the sub-blocks (and recycled blocks). However, dynamic superblock does not delay the occurrence of the first bad superblock since a bad superblock is necessary to create an initial set of recycled blocks – i.e., a superblock needs to be "sacrificed." To delay the occurrence of initial bad superblocks, we extend recycled superblock with *reservation*-based recycled superblock where some number of physical blocks are initially reserved or provisioned as recycled blocks – i.e., the RBT is not initially empty but filled with reserved recycled blocks. A predetermined number of blocks (or superblocks) is leveraged by the decoupled flash controller, as recycled blocks, to delay the occurrence of the bad superblocks and improve SSD endurance. The only change required is that the reserved blocks are properly initialized within the RBT prior to any access to the flash memory.

6 EVALUATION

6.1 Methodology

We used Simple-SSD simulator [10] in standalone mode and integrated Booksim [14] interconnect simulator to implement decoupled SSD. A bus structure is modeled for system-bus in SimpleSSD while Booksim is used for flash controller interconnect (i.e, fNoC). Both synthetic input and trace workloads [23] are used in the evaluation. Synthetic input consists of both DRAM "miss" where all access go the flash memory as well as DRAM "hit" where all I/Os are serviced by the DRAM. Each decoupled flash controller in the decoupled SSD (dSSD) has the ECC engine, a router, and the decoupled buffer (dBUF). The topology of fNoC is a 1-D mesh and minimal (deterministic) routing is used. For performance evaluation, we assumed a flash chip with one flash die for simplicity and assumed ULL parameters [10, 44] as shown in Table 1.[10] Detailed simulation parameters are listed in Table 1. We assume SSD is fully utilized and some random fraction of the pages are invalidated such that garbage collection will be triggered in the simulator. We assume an I/O queue depth of 64 requests to fully utilize the SSD and use I/O requests of 4kB and 128kB to model both low and high I/O request demands. We assume the I/O requests are fully utilizing the SSD's internal bandwidth and we compare the performance when GC is performed. For data movement, packetization, and routing overheads are modeled.

The different SSD configurations that we compare are summarized in Table 2. We assume a baseline that supports high-throughput parallel GC (PaGC) [35], which is commonly used [9, 26]. All of the other architecture configurations compared have 1.25× extra on-chip bandwidth. `BW` is identical to the baseline but the additional bandwidth is used by the system bus. `dSSD` has the decoupled flash controller to implement dSSD but the same system bus configuration as `BW`. $\texttt{dSSD}_b$ is identical to `dSSD` but has a separate dedicated bus that interconnects the flash controllers while $\texttt{dSSD}_f$ has the flash-controller network-on-chip (fNoC). We also compare preemptiveGC [24] where GC is pre-empted to handle I/O traffic as well as Tiny-Tail [42] that tries to minimize tail-latency by avoiding GC interference with partially executing GC.

[10] For superblock evaluation, we used SSD that consists of 8 channels 4 ways 2 dies 2 planes with TLC-based flash memory parameters and we simplified pages/block to 32 for feasible simulation time.

Components	Parameters
Simple-SSD organization	PCI-E 3.0 x8 lanes, system-bus = 8GB/s (×1) DRAM = 8GB/s, a flash bus = 1GB/s (1000Mhz,8 bits) 8 channels 8 ways 1 die 8 planes 1384 blocks 384 pages gaussian dist., E=5578, σ=826.9, provision ratio 7%
Flash (ULL)	read=5us, write=50us, erase=1ms, 4KB page
Memory (TLC)	read=60-95us, write=200-500us, erase=2ms, 16KB page
fNoC	Topology = 1D mesh, k = 8, n = 1, routing = dim order

Table 1: Simulation Parameters.

Name	Description
`Baseline`	Conventional SSD with parallel GC
`BW`	`Baseline` with additional system-bus bandwidth
`dSSD`	decoupled SSD with the same bus bandwidth as `BW`
$\texttt{dSSD}_b$	`dSSD` with a separate, dedicate bus to interconnect flash controllers
$\texttt{dSSD}_f$	`dSSD` with a fNoC

Table 2: Summary of different architectural configurations compared.

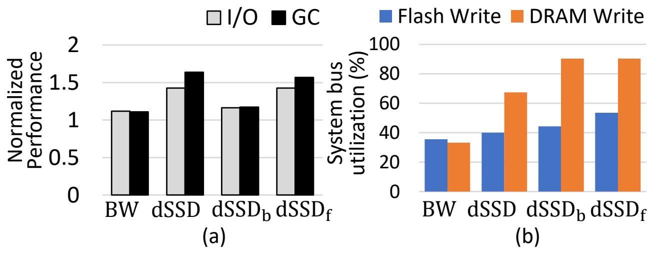


Figure 7: (a) Normalized I/O and GC performance and (b) system bus utilization comparison. All comparisons have the same amount of on-chip bandwidth for high-bandwidth flash memory.

6.2 Results

The results of I/O bandwidth and GC performance are shown in Fig 7(a) with the results normalized to `Baseline`. To ensure a fair comparison, the total on-chip bandwidth is held constant across the 4 architectures compared. For `Baseline` and `dSSD`, the total on-chip bandwidth is used by the system bus but `dSSD` provides the ability for the flash controllers to directly communicate with each other. $\texttt{dSSD}_b$ has a dedicated bus while $\texttt{dSSD}_f$ uses a 1D mesh (fNoC) for the internal data movement.

`BW` performance (both I/O and GC) benefits from the increase in system-bus bandwidth; however, the performance improvement from providing the same bandwidth to `dSSD` is much higher. For example, `dSSD` results in 42.7% (63.8%) improvement for I/O (GC) while the improvement for the `BW` is only 11.8% (10.9%). The main benefit of `dSSD` comes from the reduced amount of data movement between flash controllers by directly transferring pages; however, `dSSD` still suffers from long tail latency because of shared network resources (Fig 10(a)). The performance improvement from $\texttt{dSSD}_b$ is rather small (compared to `dSSD`) because of fixed, partitioned bandwidth, and GC performance is bottlenecked by the serialization across the relatively small bandwidth of the added bus. However, the

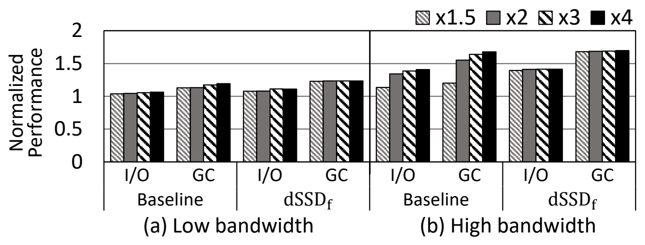


Figure 8: Performance improvement for I/O and GC on (a) low and (b) high bandwidth flash memory as the amount of on-chip bandwidth is increased.

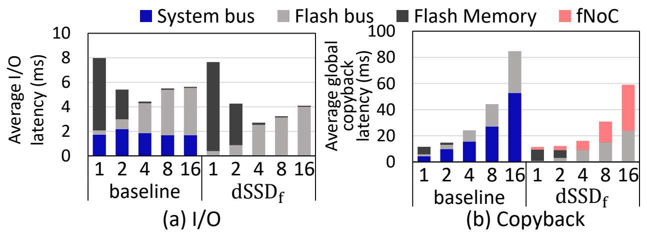


Figure 9: Latency breakdown for (a) I/O and (b) copyback as the number of planes is increased.

performance of dSSD$_f$ nearly matches that of dSSD – although the bisection bandwidth is similar to dSSD$_b$, multiple channels within the fNoC can be utilized in parallel – thus, improving the effective bandwidth and achieving higher performance (compared to dSSD$_b$). In Fig 7(b), we measure I/O system-bus utilization during GC for when all access hit in the DRAM (i.e., DRAM Write) and when all accesses "miss" in the DRAM and access the flash memory (i.e., Flash Write). dSSD$_f$ shows 18.1% and 66.9% increased system-bus utilization (compared to baseline) for DRAM Hit/Miss I/O cases as it successfully decouples contention from 'sharing.'

In Fig 8, an additional sensitivity study is performed where the system bus bandwidth is increased up to 4× and the results are normalized to Baseline. Results are shown for "low" and "high" bandwidth simulations similar to in Fig 2. We also compare against four different implementations of dSSD$_f$ where the total bandwidth is increased similarly as well (×1.5 - ×4). Since "low" bandwidth does not fully utilize the system-bus bandwidth even during I/O and GC contention, increasing system-bus bandwidth has minimal impact on overall performance – resulting in only 4.6% (for I/O) and 13.6% (for GC) improvement even when the system-bus is increased by 2×. The benefit with dSSD$_f$ is slightly higher. However, for "high" bandwidth (Fig 8(b)), increasing the system-bus bandwidth directly impacts both I/O bandwidth and GC performance. With baseline (×1.5), I/O and GC improves by 13.5% and 19.9%, respectively. With the *same* amount of on-chip bandwidth, dSSD$_f$ is able to provide further improvement in performance Compared to baseline, dSSD (×1.5) improves I/O and GC performance by 39.4% and 68%, respectively. (and 22.8% and 40% compared to baseline (×1.5)). ***The decoupled SSD effectively decouples the bandwidth used by the front-end (for I/O) with the back-end (copyback) – thus, when high on-chip bandwidth is required, it is more effective to "decouple" the bandwidth or separate them to improve overall performance.***

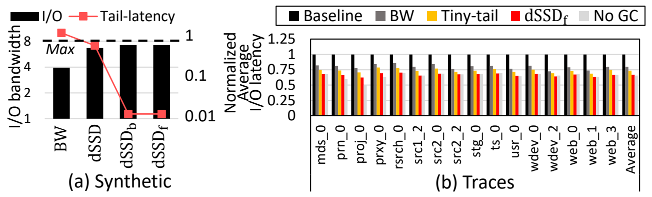


Figure 10: (a) I/O bandwidth and tail-latency with 100% DRAM cached I/O accesses when GC is triggered on different dSSDs and (b) workloads results of I/O latency

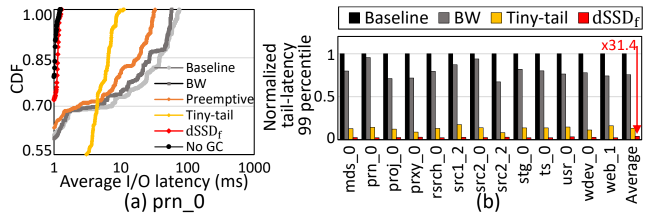


Figure 11: (a) Tail latency comparison for (prn_0), and (b) average tail-latency improvement for workload traces.

To understand the benefits of dSSD$_f$ compared to Baseline, the latency breakdown of I/O requests and copyback is shown in Fig 9. With 1 plane, the I/O latency is determined mostly by the flash memory contention; as the number of planes increases, the contention for flash memory (chip) is significantly reduced but the contention for the flash bus increases, for both the baseline and the dSSD$_f$. However, the key difference is that the contention for the system bus is removed for dSSD$_f$. The copyback latency breakdown during GC (Fig 9(b)) shows copyback operation is heavily dominated by the system-bus and flash bus utilization/contention. In the decoupled system, the fNoC latency dominates as the number of planes increases but since fNoC is only used by the copyback requests, the increase in latency is lower than the added latency from the system-bus contention in Baseline.

The benefit of dSSD$_f$ is maximized when I/O request accesses hit in the DRAM and does not need to access the flash memory since the interference between the I/O requests and GC can be avoided. The I/O bandwidth and tail latency when all I/O accesses are serviced from the DRAM is shown in Fig 10(a). dSSD$_f$ is able to achieve maximum I/O bandwidth since copyback commands are not utilizing the system bus or the DRAM. As a result, tail-latency is significantly reduced 77× (39×) compared to BW and dSSD. In comparison, even though the system-bus bandwidth is increased in BW and in dSSD, the I/O bandwidth is not maximized (54.6%) as the shared system bus results in long tail-latency. Fig 10(b) shows, on average, 31.9% reduced average I/O latency, compared to the Baseline and 16.1% improvement compared to BW. TinyTail [42] reduces system-bus contention by partially performing GC per flash channel but dSSD$_f$ still outperforms it by 7.5% since contention is minimized in dSSD$_f$.

Fig 11 shows the impact on tail-latency for different workloads. In Fig 11(a), the dSSD$_f$ shows that the 99% tail-latency from one workload (prn_0) improves by 43.7× and 31.2×, compared to Baseline and BW. Compared with PreemptiveGC, dSSD$_f$ shows 20.8× lower tail-latency since PreemptiveGC must copy pages when

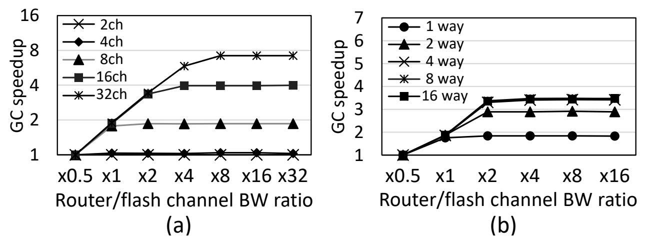


Figure 12: Impact on garbage collection as the router channel bandwidth is varied with (a) different number of channels and (b) different number of ways per channel. The baseline assumes 8 channels and 1 way per channel.

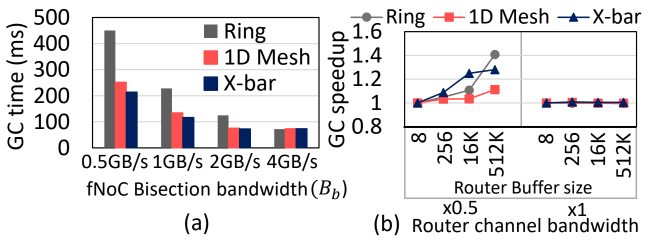


Figure 13: Performance impact of (a) channel bandwidth and (b) on-chip router buffer size on garbage collection for alternative fNoC topologies.

the FTL can no longer postpone GC. TinyTail [42] significantly reduces tail-latency compared to Baseline; however, I/Os latency still contends with GC operations due to system-bus conflict even with increased system-bus bandwidth. As a result, the dSSD_f results in 6.19× lower tail-latency than TinyTail. Across all workloads, on average, dSSD_f results in 31.4× (compared to Baseline), 5.17× (compared to TinyTail) reduction in tail-latency, respectively (Fig 11(b)).

6.3 Flash controller Network-on-Chip (fNoC)

One key component of NoC is the amount of channel bandwidth [7]. In Fig 12, we vary the amount of router channel bandwidth as the number of flash channels (Fig 12(a)) and the number of ways per flash channel (Fig 12(b)) is increased and measure GC performance. The flash channel bandwidth is held constant and the router channel bandwidth is varied and the x-axis shows the ratio between the router channel and flash channel bandwidth. The results of Fig 12(a) show that as the number of channels increases, more router channel bandwidth is necessary to maximize GC performance. However, the performance saturates when there is "sufficient" bandwidth – i.e., the *bisection* bandwidth of the NoC is sufficient to handle the random traffic from the flash channels. In comparison, if the number of ways is increased (Fig 12(b)) while the number of channels is held constant at 8, the benefit of dSSD_f saturates around ×2, regardless of the number of ways. Since $N = 8$ or 8 channels, the bisection bandwidth $B_b = N/2 \times B_f$ where B_f is the flash channel bandwidth. Since there are bidirectional channels, ×2 provides a sufficient amount of bandwidth. Although not shown, a similar analysis also holds if a different topology is used for the evaluation.

Alternative topologies for fNoC is compared in Fig 13 where bisection bandwidth (B_b) is held constant across the different topologies (1D mesh, ring, crossbar). The amount of bandwidth *per* channel of a ring is lower than a 1D mesh since ring has twice the number of channels crossing the bisection. As a result, the performance of 1D mesh is much higher than the ring when the amount of on-chip bandwidth is limited because serialization latency has a dominant impact on performance for large packets, compared to hop count. When there is sufficient bandwidth (i.e., 2GB/s of bisection bandwidth), the performance of 1D mesh matches the performance of the crossbar. The impact of on-chip buffer is shown in Fig 13(b). When there is an insufficient amount of bandwidth, the on-chip buffer in the routers has a significant impact on overall performance and the impact of buffers can be costly; however, when there is sufficient bandwidth, the impact of on-chip buffer is relatively small. In this work, we assumed a simple 1D mesh topology as it matched the floorplan of flash controllers [6]. However, as the number of flash controllers increases and/or the amount of on-chip bandwidth increases, it remains to be seen what the optimal topology for the fNoC will be.

6.4 Dynamic Superblocks

In this subsection, we compare the impact of the dynamic superblock and compare a baseline (BASELINE) with recycled superblock (RECYCLED) (Sec 5.1) and reservation-based recycled superblock (RESERV) (Sec 5.3). Fig 14(a) shows the result of lifetime improvement from RECYCLED and RESERV. We evaluated an SSD where a continuous stream of 128K write I/O requests is assumed until 90% of the superblocks are used. To model the RBER variation of blocks, we leveraged the P/E cycle distribution model for block-level variation [40] (i.e., $E(x) = 5578$, $\sigma(x) = 826.9$). To make the simulation time feasible, we simplified the size of the SSD – and for the reservation-based recycled superblock (RESERV), we provisioned 7% of the blocks as part of recycled blocks, which means 7% of blocks are not visible to the FTL.

Fig 14(a) plots the number of bad superblocks on the y-axis while the x-axis is the amount of data that is written to the SSD. As discussed earlier in Sec 5.3, the first occurrence of bad-block is the same in both the BASELINE and RECYCLED. However, there is a difference in all of the subsequent bad superblocks since the recycle blocks are re-used as a part of another superblock (i.e., dynamic superblock). For RESERV, the reserved blocks that are available for recycled blocks enable the first bad superblock occurrence to be significantly delayed by 65%. As a result, endurance at the point of the small number of bad blocks, measured in terms of additional amount of data written, improved by approximately 19% and 35%, compared to the BASELINE, for RECYCLED and RESERV, respectively, and the benefits of RESERV decreases as the number of bad superblock increases. The lifetime of an SSD can be defined as when a certain fraction of the blocks become bad-blocks [34]. As shown in Fig 14(a), when the fraction of blocks is lower, RECYCLED can significantly extend the lifetime of the SSD.

To further understand the impact of process variation, the standard deviation of the RBER distribution is varied and impacts the P/E cycle limit for the blocks within a superblock. The benefits of RECYCLED is higher as the variation increases (Fig 14(b)), compared to the BASELINE. The benefits of RESERV over RECYCLED

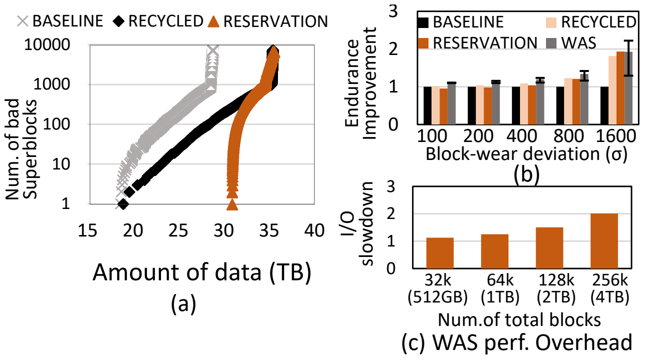


Figure 14: (a) Improvement in lifetime comparison, (b) endurance improvement for different block-wear variation, and (c) I/O performance overhead from Wear-aware scheduling (WAS).

is relatively small at low variation; however, as the variation increases, there is more benefit from `RESERV`. We also compare our approach with a *software*-based superblock management that has been recently proposed: Wear-aware scheduling (WAS) [40]. WAS is able to achieve higher endurance since the superblock is managed in the software (or the FTL) and thus, has variation information for all of the blocks. In addition, it performs wear-leveling to complement superblock management.

While our decoupled SSD architecture does introduce additional hardware overhead compared to WAS, the decoupled SSD architecture can be exploited to minimize any performance degradation from the superblock management. WAS requires endurance information for each block to make a decision which blocks have similar endurance by reading at least one page per block. In Fig 14(c), we evaluate the performance (average I/O latency) as the total number of blocks (the number of pages to be readout for RBER update) is varied. For synthetic write I/Os, there can be up to 2× degradation in average I/O latency when using the software-based superblock management (i.e., WAS), resulting from accesses to the shared system-bus and DRAM memory to collect all block's endurance status.

The hardware-based approach can result in performance degradation based on the remapping – for example, if the remapping occurs such that two blocks within a superblock are mapped to the same channel, channel-based parallelism cannot be exploited. Compared to `BASELINE` , `RECYCLED` can improve endurance but comes at the cost of performance degradation when all channels are not equally utilized. Fig 15 shows normalized performance for synthetic workloads and workload traces. In Fig 15(a), a worst-case synthetic workload is used to measure the potential performance degradation as the number of entries of SRT is varied for ULL and TLC-based SSD. The impact from READ synthetic workload is relatively small but becomes more noticeable for TLC and frequent WRITE random accesses – up to approximately 2× degradation on performance with 2k SRT entries. Thus, more SRT entries can improve endurance but at the same time, have a higher negative impact on performance as the remapping can potentially introduce more channel and flash conflicts. However, as discussed earlier, `RECYCLED` improvement in endurance does not necessarily come at the cost of performance degradation before the bad-block is created. As endurance is increased, however, the performance when SSD lifetime is extended, is degraded compared to `BASELINE`.

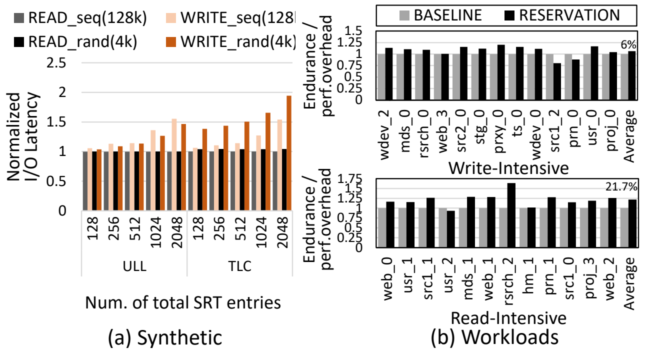


Figure 15: (a) Synthetic and (b) workload, trace evaluation of `RESERV` and `BASELINE`. (a) plots the performance impact as the number of SRT entries increases while (b) normalizes the endurance improvement by the performance overhead.

Trace-based evaluations are shown in Fig 15(b). We use a normalized metric (endurance/performance overhead) as performance overhead is used to approximate "cost" of dSSD – thus, higher is better as it means either higher endurance and/or lower cost. On average, dSSD results in a higher metric, by approximately 21.7% and 6% for read and write-intensive workloads, respectively. For read-intensive workloads, most workloads result in improved the metric compared to the baseline but some workloads (e.g, `usr_2`, `hm_1`) have smaller improvements since these workloads contain some fraction of write operations. For write-intensive workloads, some workloads (`src1_2`, and `prn_0`) show a performance drop since they have a higher fraction of write I/O with a large I/O size and lead to more flash channel/memory contention. However, most of the workloads show higher metric than baseline, even with performance degradation when the SRT size is 2048.

6.5 Overhead

The overhead of dSSD consists of three parts: 1) integrating ECC engine in the flash controller, 2) the routers, and 3) the decoupled buffers (dBUFs). An LPDC engine uses approximately 2.56 mm^2 in 90nm [11] (or equivalent to 0.122 mm^2 in 14nm [38]) and represents approximately 1.5% overhead of the entire SSD controller for the 8 channels (assuming an SSD controller area is approximately 64mm^2 [30].) The router cost is negligible due to the small input buffer size. We synthesized routers to estimate cost based on 45nm process based on FreePDK [39]. A router area occupies approximately 0.02mm^2, and the proposed fNoC introduces approximately 0.25% area overhead in the SSD controller. The major area overhead is in the decoupled buffers (dBUFs). However, dBUFs are shared among multiple flash dies (or ways) in the same flash channel to reduce the area cost – thus, two 32KB dBUFs (1/8th of the baseline page buffers (2×32KB×8 flash ways)) are required in a decoupled flash controller, which adds an additional 2.46% area overhead to the SSD controller.

The main overhead from the proposed dynamic superblocks is the tables added to the decoupled flash controller. The RBT has very little area overhead (approximately 32 bits for each decoupled controller) since new recycle blocks are created only when no recycled block remains. However, `RESERV` recycle blocks, the RBT table

	Description	Average I/O Performance	I/O Tail-latency	GC Performance	System-bus Interference	FTL Modification	Cost
Preemptive GC [24]	GC is preempted when I/O arrives	++	+	–	o	o	FTL modification
Tiny-tail [42]	Service I/Os with partial/non-blocking GC	+	++	–	+	-	FTL, parity pages for RAIN
PaGC [35]	perform GC in parallel across all flash memory	+	+	+	–	o	FTL modification
dSSD (This work)	Decouple I/O & GC datapath	+	+	+	++	++	fNoC

Table 3: Qualitative comparison with representative prior work.'++' is excellent, '+' is good, 'o' is fair, and '-' is poor.

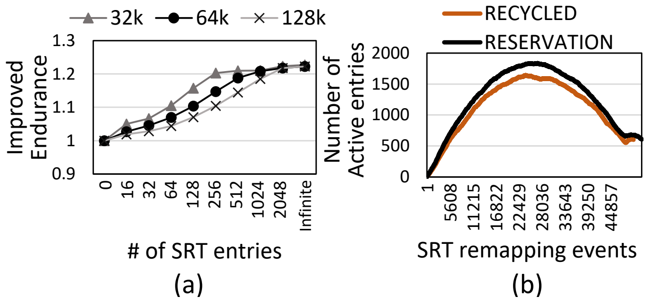


Figure 16: (a) Endurance improvement by increasing the size of superblock remapping table (SRT) for different SSD capacities and (b) the number of active SRT entries as amount of remapping increases. In (a), the number of superblock is increased to evaluate the impact of SRT.

requires more entries to prepare reserved recycle block information and is proportional to the reservation ratio (around 1KB per channel for 7%). In comparison, the remapping entries in the SRT has to be continuously stored and accessed for dynamic superblock management and the size of SRT can impact endurance/performance trade-off. In Fig 16(a), we evaluate the improvement in endurance as the number of entries in the SRT is increased. In general, more SRT entries result in higher improvement in endurance. For larger SSD capacity (i.e., 128k superblocks), more SRT entries are also needed to maximize endurance improvement. The improvement in endurance saturates around 1k SRT entries per decoupled flash controller in our evaluation. Assuming each SRT entry is 32 bits – 16 bits for the source and 16 bits for the destination (7 bits to specify the die and 9 bits used for block), the SRT table overhead is approximately 4kB.

Fig 16(b) illustrates why endurance improvement saturates. In the evaluation, we assume an infinite size SRT and measure the number of *active* entries in the SRT – i.e., the number of entries that contain valid remapping information and compare `RECYCLED` and `RESERV` . The plot shows the active SRT entries for one flash channel but similar trends were observed for other channels. The *x*-axis is the number of remapping events (or updates to the SRT) that occurred during evaluation. As more remapping (or bad superblock) occurs, the number of active SRT entries increases but when all of the original or "static" superblock are no longer available, the number of active SRT entries will no longer increase. `RESERV` has a higher number of active SRT entries because of the reserved recycle blocks but the trend is similar to `RECYCLED`.

7 RELATED WORK

Table 3 provides a qualitative comparison of this work with representative prior work. The preemptive-GC [24] avoids GC by postponing or preempting GC as long as possible; however, it does not eliminate GC contention and eventually, system bus interference occurs at some point [42]. Tiny-tail [42] greatly improves I/O tail-latency through partial execution of GC across the flash chips and services I/O requests at the same time. However, the system bus contention cannot be avoided because I/O and GC still need to utilize the system bus. PaGC [35] accelerates GC performance by exploiting plane-level parallelism; however, the page size that needs to be transferred is also significantly higher (i.e., more bursty traffic) that PaGC can instantaneously increase the system bus contention. In addition, dSSD is orthogonal to prior techniques as dSSD is not necessarily trying to minimize flash memory conflict, but is an approach to avoid on-chip system bus contention.

It is well known that the impact of process variation increases as the technology process shrinks, and it is becoming more significant as the density of the flash memory reaches its physical limitation. The block-level process variation [32, 43] needs to be considered, especially with superblockFTL [18, 40]. Recently, WAS [40] proposed intelligent superblock organization by dynamically understanding the wear-out of each blocks, and is carried out by the FTL. The dynamic superblock management in this work has similar goals; however, our work decouples the FTL and flash memory, and improves endurance with only hardware support. Understanding the block error similarity in 3D NAND [43] provides efficient bad-block management. However, this introduces significant FTL complexity in order to capture such error similarity of the blocks and comes at the cost of sacrificing the I/O performance. NetworkSSD [22] was recently proposed to provide direct flash-to-flash connectivity. While such SSD can minimize on-chip system bus utilization, it requires changes to not only the flash memory interface (from dedicated signals to packetized interface) but also requires changes to the external flash channel bus. In comparison, the decoupled SSD introduces an on-chip network between the flash controller that is co-located on the same chip.

8 CONCLUSION

In this work, we proposed *decoupled SSD* that reduces system resource conflict between front-side and flash controller-side. The decoupled SSD enables direct communication between flash controllers for efficient *internal* data movement. To exploit the communication between different flash memory, we proposed *global* copyback command that can be offloaded to the decoupled flash controller. We also show how SSD endurance can be improved without the support of the FTL through dynamic superblock management — in particular, reliability management is offloaded to the hardware (or the decoupled controller) by proposing to *recycle* superblocks to improve endurance.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by IITP grant funded by MSIT No. 2020-0-01303 and in part by NRF-2023R1A2C2004229. Dr. Jung is supported in part by NRF's 2021R1A2C4001773, IITP's 2021-0-00524, IITP's 2022-0-00117, Samsung (G01200447), KAIST IDEC, and Samsung HiPER (G01220296).

REFERENCES

[1] ARM. 2023. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification.

[2] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proc. IEEE* 105, 9 (2017), 1666–1704.

[3] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. 2015. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 438–449.

[4] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. 2015. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 551–563.

[5] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. 2013. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 123–130.

[6] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, et al. 2018. A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 338–340.

[7] W. J. Dally and B. Towles. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.

[8] NVM Express. 2019. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[9] Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang. 2020. Boosting the performance of SSDs via fully exploiting the plane level parallelism. *IEEE Transactions on Parallel and Distributed Systems* 31, 9 (2020), 2185–2200.

[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] Kin-Chu Ho, Chih-Lung Chen, and Hsie-Chia Chang. 2015. A 520k (18900, 17010) array dispersion LDPC decoder architectures for NAND flash memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 4 (2015), 1293–1304.

[12] Duwon Hong, Myungsuk Kim, Jisung Park, Myoungsoo Jung, and Jihong Kim. 2019. Improving SSD Performance Using Adaptive Restricted-Copyback Operations. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[13] Hwang Huh, Wanik Cho, Jinhaeng Lee, Yujong Noh, Yongsoon Park, Sunghwa Ok, Jongwoo Kim, Kayoung Cho, Hyunchul Lee, Geonu Kim, et al. 2020. 13.2 a 1tb 4b/cell 96-stacked-wl 3d nand flash memory with 30mb/s program throughput using peripheral circuit under memory cell array technique. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 220–221.

[14] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 86–96.

[15] Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear Unleveling: Improving *NAND* Flash Lifetime by Balancing Page Endurance. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 47–59.

[16] Myoungsoo Jung. 2020. OpenExpress: Fully hardware automated open research framework for future fast NVMe devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 649–656.

[17] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. 2014. HIOS: A host interface I/O scheduler for solid state disks. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 289–300.

[18] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. 161–170.

[19] Doo-Hyun Kim, Hyunggon Kim, Sungwon Yun, Youngsun Song, Jisu Kim, Sung-Min Joe, Kyung-Hwa Kang, Joonsuc Jang, Hyun-Jun Yoon, Kanabin Lee, et al. 2020. 13.1 A 1Tb 4b/cell NAND Flash Memory with t PROG= 2ms, t R= 110μs and 1.2 Gb/s High-Speed IO Rate. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 218–220.

[20] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage.. In *FAST*, Vol. 8. 1–14.

[21] Jiho Kim, Myoungsoo Jung, and John Kim. 2021. Decoupled ssd: Reducing data movement on nand-based flash ssd. *IEEE Computer Architecture Letters* 20, 2 (2021), 150–153.

[22] Jiho Kim, Seokwon Kang, Yongjun Park, and John Kim. 2022. Networked SSD: Flash Memory Interconnection Network for High-Bandwidth SSD. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 388–403.

[23] Miryeong Kwon, Jie Zhang, Gyuyoung Park, Wonil Choi, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. 2017. Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–96.

[24] Junghee Lee, Youngjae Kim, Galen M Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A semi-preemptive garbage collector for solid state drives. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 12–21.

[25] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 3 (2007), 18–es.

[26] Chun-Yi Liu, Yunju Lee, Wonil Choi, Myoungsoo Jung, Mahmut Taylan Kandemir, and Chita Das. 2021. GSSA: A resource allocation scheme customized for 3D NAND SSDs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 426–439.

[27] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. 2018. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 504–517.

[28] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. 2018. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 3 (2018), 1–48.

[29] Marvell. 2020. 88SS1098 SSD Controller. https://www.marvell.com/content/dam/marvell/en/public-collateral/storage/marvell-storage-88ss1098-product-brief-2018-03.pdf.

[30] MARVELL. 2021. Marvell Bravera SC5 SSD Controllers. https://www.marvell.com/content/dam/marvell/en/public-collateral/storage/marvell-ssd-mv-ss1331-1333-product-brief.pdf.

[31] ONFI. Feb 2020. Open NAND Flash Interface Specification rev 4.2". http://www.onfi.org/specifications.

[32] Yangyang Pan, Guiqiang Dong, and Tong Zhang. 2012. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 7 (2012), 1350–1354.

[33] Yunhui Qiu, Wenbo Yin, and Lingli Wang. 2021. A high-performance and scalable NVMe controller featuring hardware acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 5 (2021), 1344–1357.

[34] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 67–80. https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder

[35] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. 2016. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 561–572.

[36] Youngseop Shim, Myungsuk Kim, Myoungjun Chun, Jisung Park, Yoona Kim, and Jihong Kim. 2019. Exploiting process similarity of 3d flash memory for high performance ssds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 211–223.

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

[38] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[39] James E Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W Rhett Davis, Paul D Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. 2007.

FreePDK: An open-source variation-aware design kit. In *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*. IEEE, 173–174.

[40] Shunzhuo Wang, Fei Wu, Chengmo Yang, Jiaona Zhou, Changsheng Xie, and Jiguang Wan. 2019. WAS: Wear aware superblock management for prolonging SSD lifetime. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[41] Jin Xue, Tianyu Wang, and Zili Shao. 2022. MCMQ: Simulation Framework for Scalable Multi-Core Flash Firmware of Multi-Queue SSDs. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 502–507.

[42] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–26.

[43] Jui-Nan Yen, Yao-Ching Hsieh, Cheng-Yu Chen, Tseng-Yi Chen, Chia-Lin Yang, Hsiang-Yun Cheng, and Yixin Luo. 2022. Efficient Bad Block Management with Cluster Similarity. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 503–513.

[44] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 121–136.

[45] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 243–256.