IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 27, 2021, accepted June 17, 2021, date of publication June 30, 2021, date of current version July 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3093764

Two-Stage In-Storage Processing and Scheduling for Pattern Matching Applications

JOOHYEONG YOON[1,2], (Student Member, IEEE), YOONJIN LEE[2], WON SEOB JEONG[2], (Member, IEEE), AND WON WOO RO[1], (Senior Member, IEEE)
[1]School of Electrical and Electronic Engineering, Yonsei University, Seoul 03722, South Korea
[2]Memory Business, Samsung Electronics, Hwaseong-si 18448, South Korea

Corresponding author: Won Woo Ro (wro@yonsei.ac.kr)

This work was supported in part by the Ministry of Trade, Industry and Energy (MOTIE) under Grant 10080674 (Development of Reconfigurable Artificial Neural Network Accelerator and Instruction Set Architecture) and Korea Semiconductor Research Consortium (KSRC) Support Program for the Development of the Future Semiconductor Device, in part by Institute of Information and Communications Technology Planning and Evaluation (IITP) funded by the Korea Government (MSIT) under Grant 2021-0-00853 (Developing Software Platform for Programming of PIM, 50%) and Samsung Electronics Company Ltd., under Grant IO201210-07936-01.

ABSTRACT In-storage processing technology allows applications to run on embedded processors and accelerators inside solid-state drives (SSDs) for efficient computing distribution. Especially, in pattern matching applications, in-storage computing can be processed quickly due to low data access latency, and the number of I/Os can be reduced by returning only a small amount of results to the host system after processing. Previously proposed in-storage processing is separated into three phases: command decoding, data access, and data processing. In this case, data processing is strictly isolated from data access, and the isolation constraints the utilization of storage. Merging data access and data processing among the phases can enhance the utilization of storage. To efficiently merge them, we propose two-stage in-storage processing and scheduling, especially for the pattern matching application. The first stage processing during data access reduces the second stage processing latency. Also, leveraging the pattern matching results of the first stage processing, our scheduler prioritizes key requests that should return the results to the host system so that they are completed earlier than non-key requests. The proposed scheduling reduces the response time of in-storage processing requests by 52.6 % on average.

INDEX TERMS In-storage processing, solid-state drives (SSDs), scheduling, pattern matching.

I. INTRODUCTION

Solid-state drives (SSDs)-based in-storage processing has been applied to pattern matching applications such as search engines [18], [50] and key-value stores [5], [25], [55]. The in-storage processing technique distributes workloads by offloading all or part of the application functions to storage devices. In the case of a search engine, for example, a storage device internally completes data retrieval and returns only a small amount of results to the host. This mechanism reduces the number of I/Os and increases the overall throughput of query processing.

Unlike typical I/O operations, the request flow of in-storage processing is generally separated into three phases: command decoding, data access, and data processing. First, the in-storage processing command from the host is decoded to internal requests for flash memory operations. Second, the internal requests are handled by a flash translation layer (FTL) as in the typical I/O operations [7], [12], [20], [39], [42], [53]. At the FTL, an internal request is divided into several sub-requests to exploit the parallelism of SSD [11], [22], [23], [30]. These sub-requests read data from each flash memory chip into an internal buffer for in-storage processing. Third, after the data are ready in the buffer, the data are processed by the in-storage processing functions and only the results of in-storage processing are returned to the host.

However, the strict separation of each phase constrains the utilization of storage, especially when pattern matching applications are executed. First, data processing after data access increases the in-storage processing latency. In general, data access is the loading of data from flash memory into an internal buffer (i.e., DRAM). Also, data processing

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

is typically handled after all necessary data are collected. For that reason, the processing element needs to reload the collected data from the internal buffer for data processing. After loading, processing elements can start to compute. This redundant data movement increases the latency of query processing. Second, all data after accessing must be stored in the internal buffer, and these operations lower memory efficiency. It is because certain data do not need to be sent to the host system depending on the processing results in the pattern matching applications. If such data are not written to the buffer, it can increase memory efficiency. Third, data access is scheduled regardless of data processing status, which reduces data access efficiency. In the pattern matching application, the priority of the request varies depending on the data processing result. Traditional scheduling cannot leverage the results of data processing due to architectural constraints. This data access method can reduce the efficiency of in-storage processing.

Recent studies have proposed in-storage processing as alternative architecture to improve the performance of pattern matching application functions [18], [24], [45], [50]. They take advantage of in-storage processing by processing pattern matching in embedded processors or dedicated accelerators. However, data access and processing are still separated, making in-storage processing less efficient. On the other hand, there have also been various studies on in-storage scheduling [15], [21], [36], [49], [58]. Most of them improve the overall throughput by leveraging the parallel processing of flash memory. Some of them provide additional techniques to increase the quality of service (QoS) for queries. However, they are constrained to optimizing only data access, regardless of the characteristic of in-storage processing, with different priorities depending on the internal request processing.

In this paper, the data access and processing phases are merged to improve storage utilization for in-storage processing. To efficiently merge them, we propose a two-stage data processing and pattern matching detection-guided scheduling, named *2PM*. The first stage of processing is executed on the path for data access. During the first stage, it classifies each sub-request by a page-level pattern matching. Then, in the second stage, it processes a combinatorial analysis for the remaining data that are not classified clearly at the first stage. After that, the proposed scheduling for data access is triggered by the feedback of the first stage processing. Finally, our chip-level scheduler reorders with unfair priorities based on the pattern matching results.

The proposed two-stage data processing can effectively take the advantage of device parallelism by out-of-order sub-request handling. The data classification during pre-processing at the first stage can reduce post-processing time at the second stage, reducing overall in-storage processing latency. It also increases memory efficiency by filtering the data to be written to an internal memory according to the results of pre-processing.

Also, the proposed scheduling leverages the pre-processing results and employs anti-starvation techniques to prevent false reordering due to unfair priorities. Each chip-level scheduler changes the priority of the corresponding sub-request based on the pattern matching results of pre-processing and previous status. The scheduler reorders sub-requests that are issued to the chip-level queue based on the changed priority. In this way, the scheduler completes the processing of key requests, with high priority, earlier than the processing of other requests. Overall, the proposed scheduling can reduce the response delay to the host system.

Our experimental results show the proposed scheduling reduces the response time of in-storage processing requests by 52.6 % on average compared with the baseline scheduling. And the anti-starvation techniques included in the proposed scheduling improve the total delay by 79.8 % on average compared to the scheduling without the anti-starvation techniques. In addition, case studies under various conditions such as request queue depth, sub-request queue size, and a number of connected chips show sufficient effectiveness of the proposed scheduling.

Our main contributions can be summarized as follows:

- The proposed two-phase in-storage processing architecture is a novel architecture different from the conventional in-storage processing architecture, which is strictly separated into three phases. In this architecture, processing results that have been referenced only in the data processing phase can be leveraged for scheduling when accessing data.
- This paper proposes pattern matching detection-guided scheduling as a novel in-storage scheduling method. The proposed scheduling can process key requests earlier than non-key requests and reduce response time for the host. Despite the selective reordering method, the disadvantage is negligible due to the application of anti-starvation techniques.
- The proposed two-stage pattern matching algorithm is energy efficient for storage devices operated by page units. The pre-processing during the data access can process data in page units without buffering. In the next stage, post-processing can be computed simply based on the results of the pre-processing stage.

The rest of this paper is organized as follows. Section II introduces the concept of in-storage processing and describes the request flow of in-storage processing. Section III presents our two-phase in-storage processing architecture with a two-stage data processing algorithm. Section IV presents three basic priority rules, the pattern matching detection-guided reordering method, and the implementation overhead of the proposed scheduling. Section V analyzes the experimental results on our two-phase in-storage processing architecture and scheduling. Section VI discusses related work. Finally, the conclusion is given in Section VII.

II. BACKGROUND AND MOTIVATION

In this section, the concept of in-storage processing is introduced and the general request flow of in-storage processing is described. In addition, key challenges are defined and design

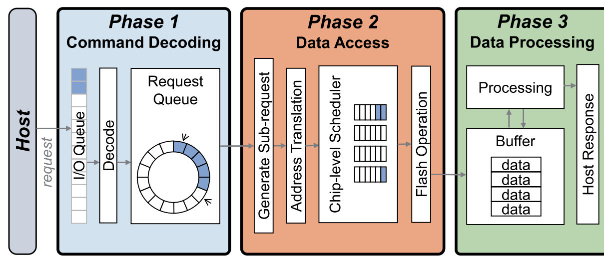


FIGURE 1. Request flow of in-storage processing. There are three phases: command decoding, data access, and data processing. This separation is the simplest way to handle in-storage processing in a legacy storage architecture, but it can degrade storage utilization.

goals are set based on the case analysis of the in-storage processing.

A. CONCEPT OF IN-STORAGE PROCESSING

The main idea of in-storage processing is to execute all or part of the functions in an application on the storage device. With the advance of in-storage processing, the data traffic between host machines and storage devices are reduced and that eventually contributes to saving data transfer delay as well as power consumption. Many researches on SSD-based in-storage processing have been introduced recently [14], [18], [24], [35], [40], [45], [46], [50], [59]. Historically, SSDs have been developed as a small computing device that integrates multiple embedded processors, DRAM, and flash memory to perform high-speed I/O operations internally [2], [8], [10], [11], [33]. Therefore, it inherently provides a certain level of computing power and is well equipped as an in-storage processing platform.

A host sends requests for data access to the SSD through the host interface. The most representative standard protocol for host interfaces is NVMe (NVM Express) [1], a new standard for high-speed SSDs that connect to host systems via PCIe. NVMe supports block I/O commands and other types of commands for in-storage processing. In the format of the vendor-specific command, the host transmits the command to the storage device. The storage device updates the results to host memory and responds to the host after processing the command.

Meanwhile, in-storage processing provides many advantages when it is used for pattern matching applications [18], [24], [45], [50]. First, the latency for data access is relatively low since the processing happens nearby data. Second, energy consumption is relatively low. Besides, pattern matching applications such as search engines can return only smaller search results instead of sending all the raw data to the main memory. Therefore, the host interface and memory utilization during in-storage processing are less than that of typical I/O processing methods.

B. DESCRIPTION ON REQUEST FLOW OF IN-STORAGE PROCESSING

This section describes the typical request flow of in-storage processing. Figure 1 shows the overall processing flow, which is separated into three working phases: command decoding, data access, and data processing. In the command decoding phase, the in-storage processing commands from a host are translated into multiple internal requests for specific flash memory operations. The data access phase is the same as the flow of the typical I/O operations. The FTL splits each request, varying in size from 512 bytes to several megabytes, into multiple sub-requests in logical page units. The FTL stripes the sub-requests to multiple flash memory chips for internal parallelism, and schedules sub-request processing at the chip level [15], [32], [42], [49]. In the data processing phase, the processing elements, such as embedded processors or accelerators, execute in-storage processing functions and then return the results to the host.

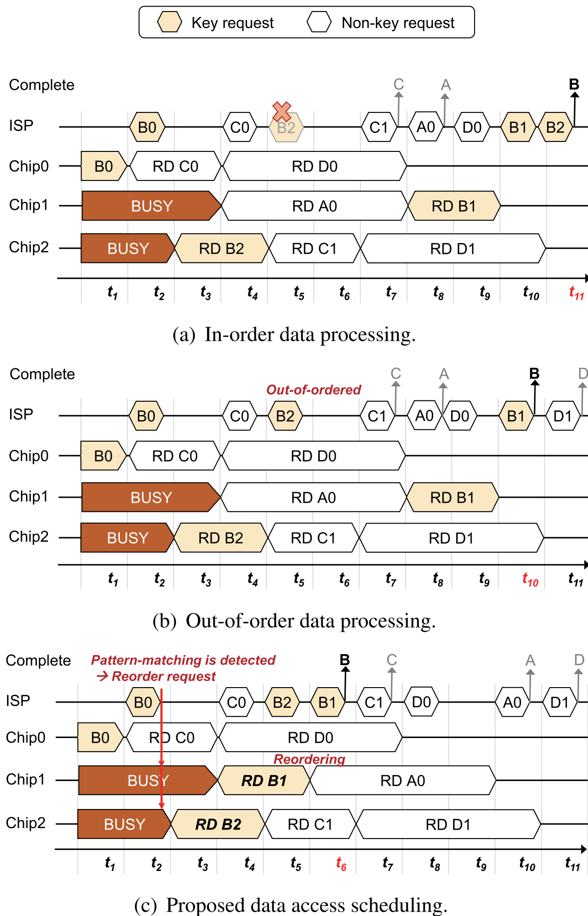
Most of the previously proposed in-storage processing studies are based on gathering all the necessary data and then processing the data in order. Pattern matching applications also have the same limitations. Because there is data dependency among sub-requests, data must be entered in order for data processing. However, data may be accessed out of order due to the nature of an SSD. Typically, data are distributed and written to other NAND flash memory chips for flash-level parallelism in an SSD. When reading the distributed data later, each data may be prepared out of order due to resource conflicts. Therefore, out-of-order data processing is required for high utilization of an SSD. Out-of-order data processing can reduce scheduling constraints by removing the dependency of data access. It also eliminates the strict separation of the data access phase and the data processing phase. For example, data processing-aware scheduling can be a possible solution for achieving high in-storage processing throughput.

C. CASE STUDY OF IN-STORAGE PROCESSING

In general, a host request is sequentially distributed multiple sub-requests on several flash memory chips and the request does not complete until all sub-requests have been processed. The end of processing for the last sub-request, which has a higher latency than other sub-requests, determines the response time of the request [15]. The last sub-request is defined as a critical sub-request. In the case of in-storage processing, the storage device typically responds to the host once the data processing is complete after accessing the critical sub-request.

Figure 2(a) shows an example of processing sub-requests for the in-storage processing, which consists of three chips: *Chip* 0, *Chip* 1, and *Chip* 2. In the figure, ISP stands for in-storage processing. Each request A, B, C, and D is divided into several sub-requests and the sub-requests are issued to the chip-level sub-request queue. Applying first come first served (FCFS) scheduling [32] as a baseline, sub-requests $B1$ and $B2$ start services at $t7$ and $t2$. Data access of $B2$ ends first at $t5$, but due to the in-order nature of in-storage processing, it needs to wait until $B1$ is processed. As a result, the request B completes at $t11$ after $B2$ has been processed.

On the other hand, if out-of-order data processing is available, $B2$ can be processed at the end of $B2$ data access.



(a) In-order data processing.

(b) Out-of-order data processing.

(c) Proposed data access scheduling.

FIGURE 2. Examples of data access and data processing. Out-of-order data processing improves the efficiency of in-storage processing comparing with in-order data processing. The proposed scheduling that leveraging pattern matching detection can significantly reduce the latency for the key request. A key request *B* can complete earlier, despite a non-key request *A* completes later.

So request B is completed at $t10$ after $B1$ is processed. In this way, request C, request A, and request B are completed and the results are sent to the host at $t7$, $t8$, and $t10$, respectively, as shown in Figure 2(b). Therefore, if out-of-order data processing is supported, processing can be completed in an early time.

Now, considering the pattern matching applications, the processing priority of each request can be determined depending on whether pattern matching is contained or not. The internal requests are classified into two types of request: key request and non-key request. The key request contains the key pattern and the non-key request does not contain any key pattern. Because only the processing results of the key request are returned for the in-storage processing request, so it is important to prioritize the key request such as B rather than non-key requests such as A or C. In the example, the response for the host starts at $t10$. Even if it supports out-of-order data processing, the improvement compared to in-order data processing is insufficient with typical scheduling.

This paper proposes scheduling that uses unfair priority to improve the response time of the in-storage processing request. The basic idea is that if a pattern matching is detected in a sub-request, sub-requests of the same request are processed before other requests. Figure 2(c) shows an example of the proposed scheduling. In the case of the baseline scheduling, the sub-request $B1$ becomes a critical sub-request, delayed by the sub-request $A0$, which was already requested on the *Chip* 1. The sub-request $B0$ completes early and the pattern matching result is verified at $t2$. If the scheduler knows that request B is a key request and swaps the processing order of $A0$ and $B1$ in *Chip* 1, it can reduce the response time for request B. The response time for the request A is delayed, but it is not a problem because the response to the request B determines the overall QoS in a pattern matching application.

III. 2PM ARCHITECTURE

This section proposes the two-stage data processing mechanism and the two-phase in-storage processing architecture for the proposed scheduling.

A. TWO-STAGE DATA PROCESSING

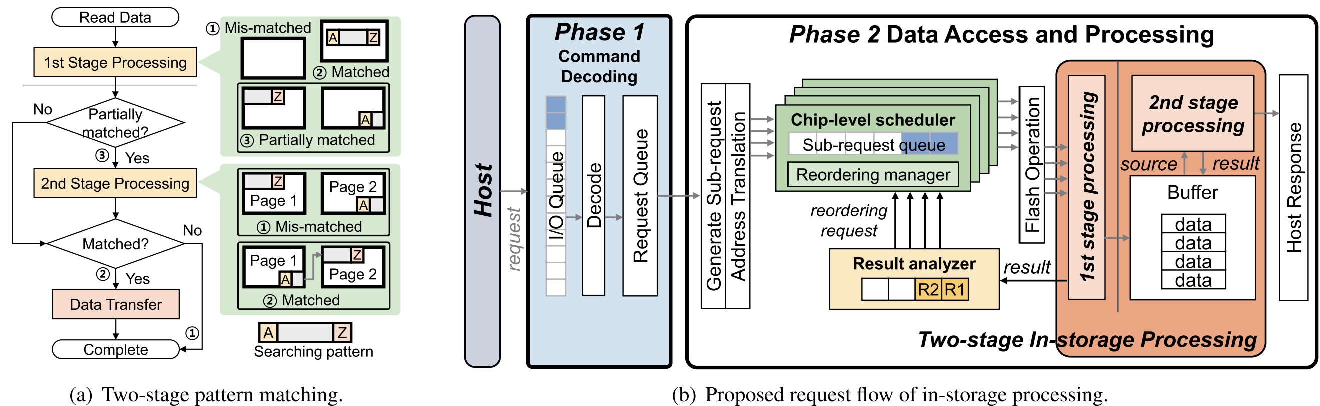
To apply the proposed scheduling, the out-of-order processing must be supported on a page-by-page basis. Therefore, this paper proposes the two-stage pattern matching method. Figure 3(a) shows the flowchart of our two-stage pattern matching.

In the first stage, a page-level pattern matching is used for the out-of-order processing of each sub-request. The processing results are classified into three types: mismatched, matched, and partially matched. For example, a pattern matching application finds a pattern that includes a start-key A and an end-key Z. If both key patterns are included and the order is also satisfied, it is called matched. If any key patterns are not included, it is called mismatched. However, if only one of the key patterns is included, or both key patterns are included but the order is reversed, then it is called partially matched, which requires additional processing in the second stage.

In the second stage, post-processing is based on the results of the first stage. If all sub-requests are mismatched in one request, or at least one sub-request is matched, the processing result can be transferred to the host without further processing. If two or more sub-requests are partially matched, further processing is required to determine matched or mismatched. In the post-processing, partially matched pages are combined, and if the pattern order is correct, the request becomes matched, otherwise, the request becomes mismatched.

B. TWO-PHASE IN-STORAGE PROCESSING ARCHITECTURE

The proposed architecture is the two-phase in-storage processing architecture with two-stage pattern matching. Figure 3(b) shows the request flow of in-storage processing in our architecture. In phase 1, the command decoding is the same as the typical flow, but data access and processing are integrated into phase 2. The two-stage processing



(a) Two-stage pattern matching.

(b) Proposed request flow of in-storage processing.

FIGURE 3. Two-phase in-storage processing architecture with two-stage pattern matching. Pattern matching consists of two stages: the first processing for each sub-request and the second processing for the final decision. The command decoding of phase 1 is the same as the typical flow, but data access and processing are integrated into phase 2. The two-stage processing units, a result analyzer, and reordering managers are added to the in-storage processing architecture.

elements and a result analyzer are added to apply the proposed in-storage processing mechanism on a modern SSD architecture with request queues and sub-request queues.

The initial flow of the proposed two-phase in-storage processing architecture, which decodes commands and inserts sub-requests into the chip-level scheduler, is similar to the request flow of conventional in-storage processing. In phase 1, the in-storage processing commands from a host are translated into multiple internal requests, and the internal requests are inserted into the request queue. The request queue supports multiple streams, and the arbitration policy to dispatch an element from the request queue follows a round-robin policy. In phase 2, the internal requests dispatched from the request queue are split into multiple sub-requests through address translation and physical mapping. And the created sub-requests are inserted into the sub-request queues for each flash memory chip. The module that manages this queue is called a chip-level scheduler. Each sub-request can be inserted at a different time, depending on the state of the sub-request queue.

There are many changes in the flow after sub-requests are issued to the chip-level scheduler. In general, the chip-level scheduler issues sequentially the queued sub-requests to flash memory. Data read from flash memory are packetized with key information and sent to the first stage processing unit before data are transferred to internal memory. The first stage processing unit located on the data path can be pipelined with other functional blocks, so data transfer performance is not degraded. In addition, the processing unit is close to data access path so that results can be quickly transferred to the scheduler.

The results of the first stage processing are sent to the result analyzer. Connecting processing units and schedulers directly in the form of a crossbar is very complicated and increases the size of the logic. Thus, the result analyzer receives the processing result of each completed sub-request and analyzes the results to reduce scheduling complexity. After analyzing the results, when the corresponding request is identified as a key request, the result analyzer sends a reordering request to chip-level schedulers. If multiple results are received at the same time, the result selected by the round-robin policy is analyzed, so one reordering request is sent in one cycle. When receiving a reordering request, the chip-level scheduler increases the priority of the key request, reorders the entries in the sub-request queue, and accesses flash memory in a changed order.

On the other hand, the first processing result and data are filtered and written to the internal buffer. The second stage processing unit waits for each sub-request to complete and then checks the completion of all sub-requests by referencing the bitmap of completion flags. For requests that require the second stage processing, the pre-processing results are analyzed and the final result is produced when all sub-requests are completed. The processing result is returned to the host when post-processing is completed or when the pre-processing result is already valid.

IV. 2PM SCHEDULING

This section presents three basic priority rules and a novel pattern matching detection-guided reordering method. The proposed scheduling works based on reordering using the pattern matching results of the first stage processing. In addition, the implementation overhead of the proposed scheduling is provided.

A. BASIC PRIORITY RULES

Our architecture uses chip-level scheduling that operates independently. The default policy for scheduling consists of three rules: read priority, oldest priority, and earliest prepared priority. First, the read priority scheduling increases the priority for the read requests over the write requests. Because the write latency of flash memory is much higher than the read latency, processing read requests first is beneficial for improving overall response time. Second, the flash

memory operation of the oldest sub-request begins first, and the later-requested sub-request begins later. Since the proposed reordering mechanism is based on the processing results, at least one sub-request must be completed. In the state without any priority information, incoming sub-requests will be processed in order. Third, the sub-request that is ready to be read first in the flash memory chips is transferred for data processing first.

B. PATTERN MATCHING DETECTION-GUIDED REORDERING

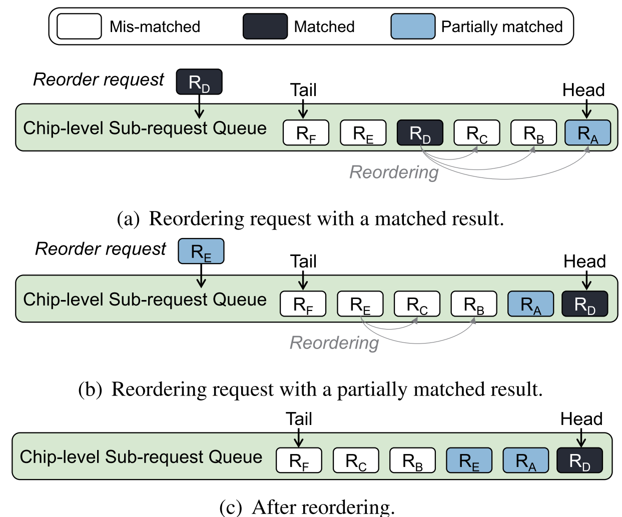
This paper proposes the pattern matching detection-guided reordering method that changes the order of data access according to the result of the first stage processing. If pattern matching is detected, the request becomes a key request, so rapid processing is required. Therefore, the result of the first stage processing is transferred to the chip level scheduler. At this point, the processing results can be sent to the chip-level scheduler through the result analyzer without the intervention of the embedded processor for fast reordering. The overhead of sending the results to the embedded processor and sending a reordering request back to the scheduler can be eliminated.

When the chip-level scheduler receives a reordering request with priority information from the result analyzer, the sub-requests in the chip-level queue are reordered. The priority information is the result of the first stage processing and takes high priority in the following order: matched, partially matched, and mismatched. This priority may change later depending on the matching results of other sub-requests.

Figure 4 shows an example of reordering. The sub-requests of R_A, R_B, R_C, R_D, R_E, and R_F are inserted in the sub-request queue in the order they will be processed. At this point, the scheduler receives a reordering request for request R_D whose processing result is matched. If there is no sub-request of R_D in the queue, the reordering request is ignored. But if sub-request of R_D exists, the priority of sub-requests is compared in order of sub-request R_C, R_B, and R_A which is located before sub-request R_D. In the example, the sub-request R_D has the highest priority, so it is placed in the head and the other requests are pushed back one by one. After that, the scheduler receives a reordering request for a request R_E whose processing result is partially matched. If the sub-request R_E exists, the priority of sub-requests is compared in order of sub-request R_C, R_B, and R_A before the sub-request R_E. Since the sub-request R_E has the same priority as the sub-request R_A, the sub-request R_E is located after the sub-request R_A and before the sub-request R_B.

C. REORDERING WITH SLACK-AWARE SUB-REQUEST INSERTION

The proposed reordering method does not determine to reorder until the processing of at least one sub-request is completed. When all the sub-requests of urgent requests are already waiting in a heavy queue, a large delay is inevitable because the reordering condition is satisfied late.



(a) Reordering request with a matched result.

(b) Reordering request with a partially matched result.

(c) After reordering.

FIGURE 4. Illustration of scheduling examples based on reordering requests. Each figure describes a snapshot of when each reordering request is processed. The matched request has a higher priority than the partially matched request.

To overcome the shortcoming, our scheduler accompanies the slack-aware sub-request insertion scheme, which is the strength of Slacker [15] among the last studies. The slack-aware sub-request insertion scheme selects the location to be inserted by calculating the slack when inserting a sub-request into the chip-level queue.

The scheme requires additional hardware resources to manage slack. Each sub-request queue has a set of registers indicating the estimated time to end processing for all inserted sub-requests. Each element in the queue contains its own slack information. The slack information of each sub-request is set when the sub-request is inserted. The slack is calculated as the difference from the worst time among the expected end times of each sub-request queue into which the sub-request will be inserted. At that time, the expected operation time of the sub-request is added to the expected end time. The expected end time decrements every cycle and is compensated slightly at the end of the queued sub-request. Each queue contains sub-requests running or pending flash operations, so these predictions and corrections are required.

When a sub-request is inserted, the location to be inserted should be determined considering the slack of sub-requests already pending in the queue and the estimated service time of the current sub-request. Figure 5 shows an example of slack-aware sub-request insertion. R_E has arrived in the sub-request queue is shown in the incoming stream. First, the received R_E is placed last in the linked list queue and is individually checked whether it can move ahead of other sub-requests pending in the queue. The check repeats the comparison between the slack of the pending sub-request and the expected service time of the current sub-request along with the previous pointer of the linked list. If the slack of the previous sub-request is larger than the expected service time of R_E, the order of the two sub-requests are swapped with each other, and the slack of the swapped

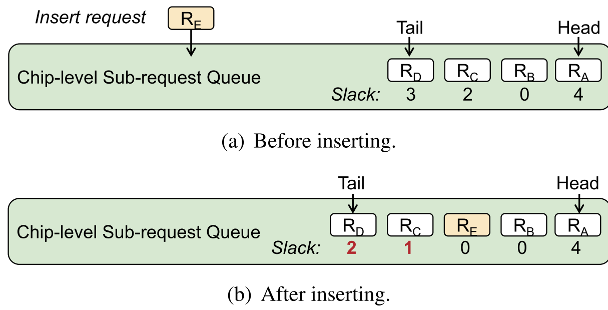


FIGURE 5. Example of slack-aware sub-request insertion.

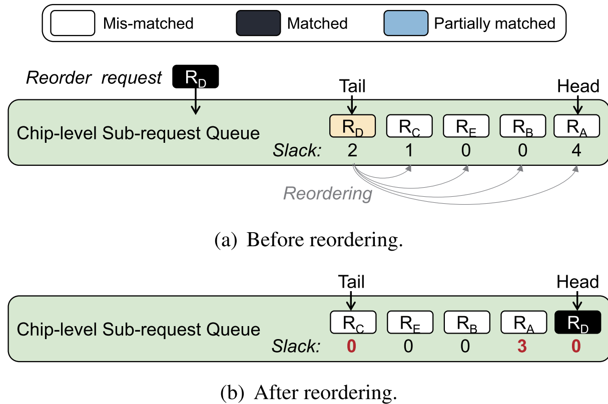


FIGURE 6. Example of slack calculation when a reordering request arrives.

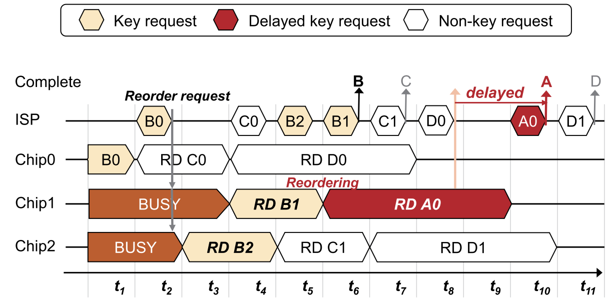


FIGURE 7. Example of delayed key request. Because key request *A* is reordered by another key request *B*, data access and processing for key request *A* are delayed.

sub-request is reduced by the amount of time to be delayed. The comparison and swap are repeated until they arrived sub-request can no longer move forward. In the example, the sub-request R_E is located after the sub-request R_B and before the sub-request R_C.

The slack-aware sub-request insertion scheme should be accompanied by a modification of the proposed reordering algorithm. Figure 6 shows an example of recalculating slack when a reordering request arrives. If the urgent sub-request has a large slack, the order of urgent sub-request may be pushed back by another request regardless of priority. To prevent this, the slack of the sub-request R_D is updated to 0 when the priority is increased by a reordering request. Also, the slack of other sub-requests whose order is changed should be recalculated. If the slack of the reordered sub-request is greater than the expected service time of the urgent sub-request, the slack changes to the current slack value minus the estimated service time, as like R_A. When the slack is smaller than the expected service time, the slack is changed to 0, as like R_C.

D. ANTI-STARVATION

The priority-based reordering approach can result in significant performance gains for some requests, but it can also reduce the performance of any other request. For example, there may be a key request that reads the mismatched page first and reads the matched page very late. This is called false negative, and it can lead to starvation for a specific request, in the worst case. Figure 7 shows an example in which one key request is delayed and processed due to wrong scheduling. In the example in Figure 2(c), it is assumed that request *A*, whose data access is delayed due to low priority, is a key request. If there is no reordering, request *A* will be completed at *t*8, but it will be completed at *t*10 due to false-negative.

To prevent starvation, the proposed scheduling prevents all sub-requests for a request from being pushed back and constraints the maximum number of swaps for low priority sub-requests. The former is intended to give all requests a chance to be prioritized. Each request is evaluated for a priority through data processing for at least one sub-request. Although it can be judged as a non-key request only due to the result of the selected sub-request, it may actually be a key request. However, our scheduling can reduce the possibility of false-negatives because it utilizes the data structure when selecting the sub-request. The latter is a common starvation prevention technique, adjusting the maximum number of swaps. These anti-starvation techniques cannot completely eliminate the overhead caused by false negatives, but they can reduce the amount of overhead caused by incorrect scheduling.

E. IMPLEMENTATION OVERHEAD

1) PATTERN MATCHING DETECTION ANALYSIS

The SSD requires sub-request queues as many as the number of chips, and a pattern matching unit on the data path for each channel. Here, let m be the number of the pattern matching units and n be the number of sub-request queues. To leverage the result from the pattern matching unit, reordering requests should be sent to all sub-request queues.

If the outputs of all pattern matching units are fully connected to all sub-request queues, it has a complexity of $m \times n$. However, the proposed structure accumulates the result of the pattern matching unit to the result analyzer and the result analyzer delivers the reordering requests sequentially to each sub-request queue every cycle. Therefore, the connection complexity is reduced to $m+n$, and the buffer size to accumulate the request is $(m-1)\times$ (*the size of reordering request*). In general, because m is significantly smaller than n, the area overhead of the result analyzer is not large, and connection complexity can be significantly reduced.

A high-capacity SSD that uses a lot of low-capacity chips may have a large n. In this case, it can increase area overhead due to connection complexity or increase computational overhead due to frequent reordering requests. Therefore, the proposed architecture constrains the maximum value of n and uses grouping to reduce the overheads. If the number of chips is 128 and the maximum value of n is limited to 32, the architecture can be organized into four groups.

2) SLACK-AWARE SUB-REQUEST INSERTION

Applying the slack-aware sub-request insertion requires additional logic to calculate slack. When the depth of the sub-request queue is d and the size of the slack counter is c, the total number of registers for the slack counter is $d \times n \times c$. Generally, d is 32 or less, and c is 32 to 64 bits depending on the resolution to calculate the slack. When N is the number of flash chips, the complexity of the required logic is $O(N)$.

In addition, each sub-request queue must estimate the total execution time of the sub-requests that are being processed or remain in the queue to calculate the slack for the next input. The number of registers for the total execution time is $n \times c$.

The request queue manager snoops the estimated total execution time of each sub-request queue, calculates each slack, and finds the worst when the new request is dispatched. These operations require additional operators, and the sub-request insertion time is delayed by the slack calculation, but it can be hidden by other operations.

3) REORDERING CALCULATION

The complexity of the reordering is low due to the nature of the sub-request queue, which is a structure of a doubly-linked list queue. When a reordering request arrives, comparing the priority is needed to find the target sub-request and then find the location to change. However, these comparisons are at most less than the depth of the sub-request queue. The reordering operation has no performance loss because the pointers of elements can be changed in one cycle.

Due to the addition of the slack-aware sub-request insertion, reordering is also required when the sub-requests are inserted. In this case, the overhead is increased because the priority comparison of 2-bit to 3-bit size is changed to the slack comparison of 32-bit to 64-bit size. If the module has only one comparator, reordering can be processed during maximum d cycles. Since sub-request queues increase by the number of chips, it is necessary to use fewer comparators to reduce area overhead. The reordering time increases by reducing the number of comparators but the latency can be hidden by pipelining.

V. PERFORMANCE EVALUATION

This section introduces an experimental setup to evaluate the proposed scheduling and analyzes the experimental results against other scheduling.

A. EXPERIMENTAL SETUP

To evaluate the proposed architecture and scheduling, we developed an in-house behavioral design that models an SSD controller. Our synthesizable design is based on a state-of-the-art commercial SSD architecture and implemented using Verilog HDL. In addition to the SSD controller, the testbench environment consists of a host system model, in-storage memory models, and flash memory models, and they are implemented using System Verilog. The in-house simulator supports various configurations and can obtain realistic results by applying the parameters of the flash memory. The basic configuration of the simulation is shown in Table 1.

TABLE 1. Experimental configuration of SSD controller and flash array.

Layer	Category	Parameter
SSD controller	Host interface	PCIe Gen4 4-lane
	Protocol	NVMe 1.3
	Logical page size	4 KB
	Device queue depth	256
	Queue depth per group	4 - 32
	Sub-request queue size	4 - 32
	Pattern matching throughput	1 GB/s
Flash array	Channel bandwidth	1 GB/s
	Number of chips per group	2 - 64
	tR (TLC read time)	50 μs
	tProg (TLC program time)	700 μs
	tBER (Block erase time)	3500 μs

For in-storage processing, the simulator uses features of the protocol for storage devices called NVMe. The host transmits the in-storage processing commands to the storage device using the packet of the vendor-specific command. The packet is already in the NVMe specification and processed by sending and receiving it stored in the host memory. Then, the storage device updates the results to the host memory and responds to the host while processing the command. The results and data are sequentially transmitted to the address of the allocated host memory. The result includes pattern matching detection information such as the number of matched patterns for each range of logical blocks, and the results and data are transmitted in units of 512-byte logical blocks. The programming interface for the vendor-specific command is as simple as setting or decoding a packet.

As a benchmark for evaluation, we generate 16 workloads that are referenced from the traces provided by MSR [43]. Each workload has different read/write ratios, data size, the ratio between key request and non-key request, and interference rate of background operations. In Figure 8, each workload is expressed as the ratio of internal data migration, write request, and read request. Additionally, read requests are divided into key requests and non-key requests. To briefly express the experimental results, we categorize these diverse workloads into three I/O characteristics: write-intensive, read-write balancing, and read-intensive.

In this experimental environment, we evaluated the performance of six different scheduling methods: baseline (PAQ), Slacker, CARS, FLIN, 2PM, and 2PM+. The first is PAQ [31], which is a baseline out-of-order scheduler. PAQ groups requests without resource contention based on physical location. The second is Slacker [15], which is a slack-aware scheduler. When a new sub-request is inserted into the

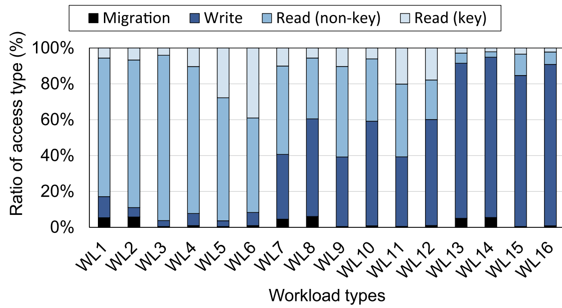


FIGURE 8. Access characteristics per workload.

chip-level scheduler, pending sub-requests can be reordered based on the pre-calculated slack. The third is CARS [58] that uses scheduling that filters requests in case of resource conflict and provide fairness for multi-queue. The fourth is FLIN [49], a scheduler that also includes reordering for requests to provide fairness for multi-queue. The fifth is 2PM, proposed scheduling that includes a reordering method based on pattern matching. The sixth is 2PM+, which adds anti-starvation and slack-aware queue insertion functions to 2PM.

B. IN-STORAGE PROCESSING LATENCY

The performance of the proposed scheduling is evaluated by comparing the latency of query processing. Figure 9 shows the average latency during in-storage processing of each scheduler according to various workloads. The request types are separated into key requests and non-key requests. Figure 9(a) shows the average latency for a non-key request and Figure 9(b) shows the average latency for a key request. Generally, prior studies have a balanced improvement for both requests. Among them, Slacker reduces the average latency for non-key requests and key request by 5.4 % and 4.5 %, respectively. CARS and FLIN, which are the fairness-seeking schedulers, only improve from 1 to 3 %. On the other hand, the proposed scheduling increases the latency for non-key requests by 1.5 % on average. However, the latency for key requests is reduced by 52.6 % on average. The factor that determines the QoS in in-storage processing is the completion of key requests, not non-key requests. Therefore, the proposed scheduling to process key requests quickly shows a better QoS in in-storage processing.

We further evaluated the effectiveness of the proposed architecture in cases where it supports out-of-order data processing. Our two-stage pattern matching method removes the constraints of processing sub-requests in order and supports out-of-order data processing. Figure 10 shows the pending delay for data processing of a sub-request that has completed data access. In the case of the in-order data processing, the sub-request delay is much higher than that in the case of the out-of-order data processing. Experimental results show that when the out-of-order data processing for sub-requests is supported, performance can be improved compared to the in-order data processing.

C. PENALTY ANALYSIS

The 2PM scheduler uses anti-starvation techniques to reduce the penalty of reordering. We analyzed the penalty to evaluate the anti-starvation techniques. Figure 11 shows the average delay and the ratio of delayed key requests. The delay means the difference between the latency of applying each scheduling and the latency of applying baseline scheduling. The inefficiency caused by delayed key requests has been reduced by about 79.8 % compared to before adding the anti-starvation techniques on average. The ratio of delayed requests is about 3 % of all requests, and the tail latency of 99.99 percentile is also lower than 1 ms. These delays are negligible losses compared to the overall performance gain.

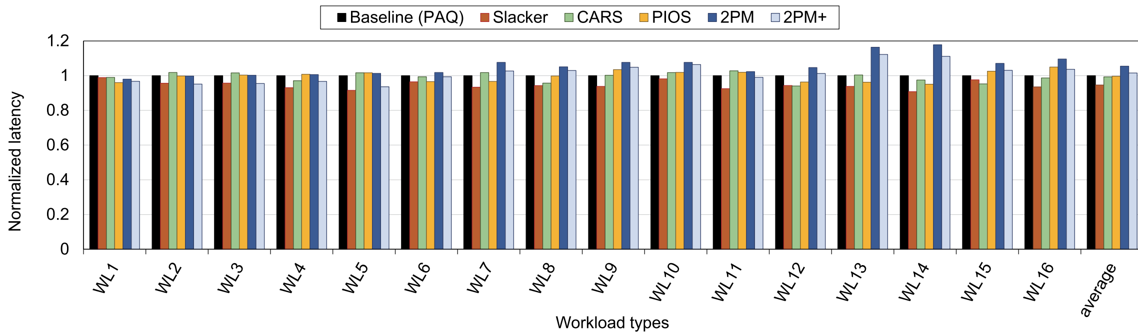
D. SENSITIVITY STUDY

There are additional experiments for sensitivity studies to analyze how the effects of the proposed scheduling depend on the different conditions: the request queue depth, the sub-request queue size, and the number of chips connected to the result analyzer.

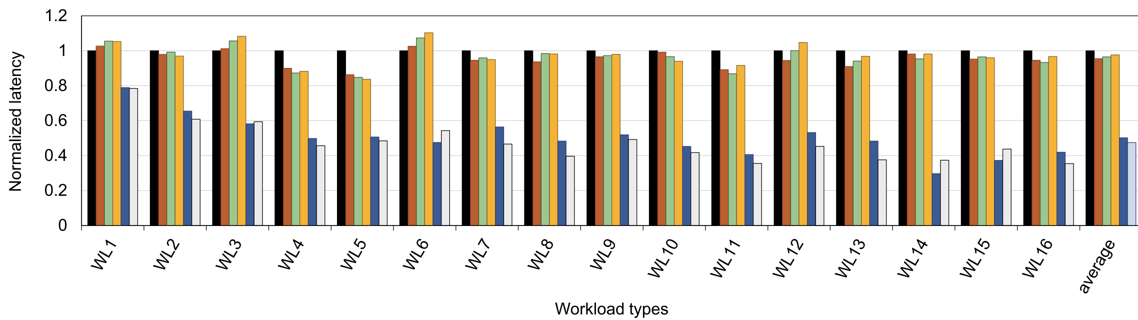
Figure 12(a) shows a comparison of the response time according to the request queue depth. The larger the queue depth, the greater the number of processing simultaneously in the storage device. In this case, more sub-requests are allocated to each chip, and more items in the sub-request queue are reorderable. As the number of reorderable items increases at the end of sub-request processing, the gain from reordering increases. Therefore, the proposed scheduling further reduces the response time for host requests as the request queue depth increases.

The second case study is about the sub-request queue size in terms of resources. Figure 12(b) shows a comparison of the response time according to the sub-request queue size. The larger the sub-request queue size, such as the request queue depth, the greater the effect of reordering. The minimum condition for reordering so that sub-requests of a key request are processed first is that sub-requests are issued to the sub-request queue. If the sub-request queue size is small, the probability increases that sub-requests for key requests cannot be en-queued due to the constraint. Therefore, the proposed scheduling further reduces response time for host requests as the sub-request queue size increases.

The third case study is how the proposed scheduling has the effect according to the number of chips connected to the request logic. As the number of flash memory chips in a group increases, the number of sub-request queues also increases. As the number of flash memory chips increases, sub-requests are distributed evenly, but too many sub-request queues can increase overhead. Figure 12(c) shows a comparison of the response time according to the number of chips connected to the request logic. According to experimental results, the proposed scheduling improves the latency of query processing until the number of chips increases up to 16. However, from 32, the effect decreases slightly. In general, the performance of an SSD increases as the number of chips increases until the SSD can fully utilize internal parallelism but the inflection



(a) Non-key requests.



(b) Key requests.

FIGURE 9. The average latency during the in-storage processing according to various workloads. The proposed scheduling reduces the latency for key requests by 52.6 % on average and increases the latency for non-key requests by 1.5 % on average.

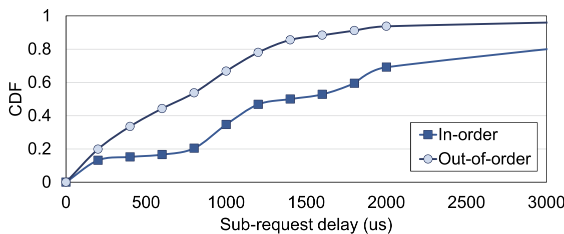
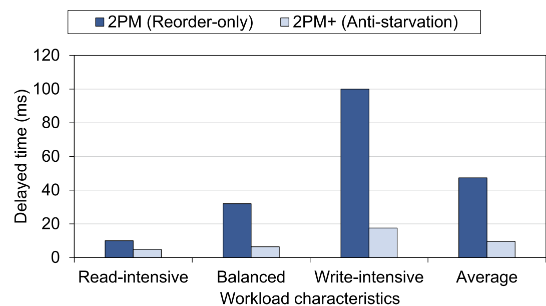
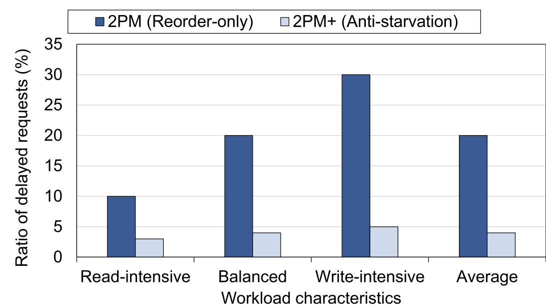


FIGURE 10. Difference in sub-request delay between in-order data processing and out-of-order data processing. Sub-request delay can be improved by out-of-order data processing.



(a) Total delay.



(b) Ratio of delayed requests.

FIGURE 11. Penalty analysis. The penalty by delayed key requests is improved by applying anti-starvation techniques.

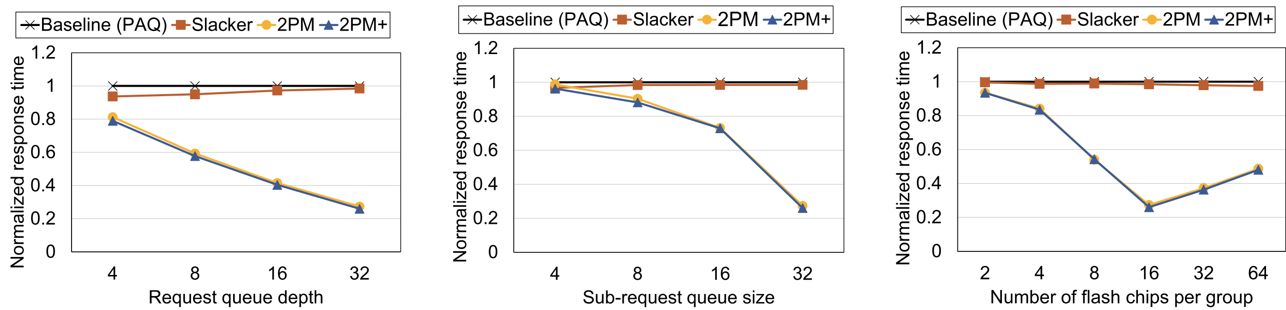
point of performance can be changed depending on the characteristic of requests.

VI. RELATED WORK

This section presents related work for in-storage processing architecture and in-storage scheduling, compared with the proposed architecture and scheduling.

A. IN-STORAGE PROCESSING ARCHITECTURE

In-storage processing is applied in various fields such as compression, encryption, search, and key-value stores. In particular, in-storage processing is widely used in database systems that require large data processing, and the field is expanding to neural networks. Some studies provide the interface for running various applications on the storage device [3], [35], while others offload only specific functions suitable for the storage device [5], [25], [55], [59]. For in-storage processing, computing resources to process typical I/O commands on the storage device are generally used, but most of them process data by adding separate built-in accelerators. [24], [46] [45].



(a) Normalized response time according to the request queue depth.

(b) Normalized response time according to the sub-request queue size.

(c) Normalized response time according to the number of chips per group.

FIGURE 12. Sensitivity studies depend on three conditions: request queue depth, sub-request queue size, and the number of chips per group. The proposed scheduling is more efficient than other mechanisms in various configurations, and it can be more effective in high performance.

Recent literature provides an in-storage processing method that offloads pattern matching application functions such as regular expressions for query processing. Several of these studies have suggested multi-stage data processing methods [18], [24], [45], [50]. Biscuit [18] proposes an in-storage processing architecture and framework. Biscuit has an accelerator that filters data through pattern matching after data access, and processes only filtered data in an embedded processor. REGISTOR [45] also accelerates regular expression searches in storage devices where large data sets are stored to solve the I/O bottleneck problem. Unlike Biscuit, all the data are processed in an accelerator with a deep pipeline structure. These two studies provide data processing on the data path closely related to data access but do not utilize data processing results for data access. REACT [24] proposes a regular expression accelerator embedded in an SSD and a data access scheduling algorithm for sub-requests. REACT increases throughput by buffering the input data of the accelerator into internal SRAM rather than DRAM and providing context switching in the accelerator. However, the inherent delay due to the strict separation of data access and processing cannot be completely overcome, and the constraint by the in-order processing hinders on-the-fly processing.

In addition to pattern matching applications, the proposed architecture is also useful for other applications that can skip computational tasks based on partial results. For example, the proposed architecture is applicable to conditional computation in neural networks [4], [6].

However, there are still challenges to be addressed before adopting in-storage processing. For example, compatibility for various systems and changes to the conventional programming model are required. The proposed method uses the open protocol specification and provides a simple programming interface, but minor changes in the host system are required due to the nature of in-storage processing.

B. IN-STORAGE SCHEDULING

As SSDs have become a representative high-performance storage solution replacing hard disk drives, I/O scheduling technologies from various perspectives suitable for the characteristics of NAND flash memory have been proposed. There have been studies on system level scheduling techniques such as HIOS [29], DLBQ [26], Amphibian [41], and D2FQ [51]. HIOS [29] showed that it can improve the QoS of devices by estimating garbage collection (GC) cost and redistributing I/O requests. DLBQ [26] proposed a dynamic load balancing queue by modeling the virtual time for the read and write ratio of flash memory. Amphibian [41] proposed size-based ordering and GC-aware dispatching. D2FQ [51] proposed a fair queue I/O scheduler that utilizes weighted round-robin arbitration of NVMe. However, system level scheduling has limitations to fully utilize parallelism at the flash level.

Meanwhile, to reduce the limitations of system level scheduling and distribute workloads, device level scheduling has also been studied from various perspectives. Among them, Several studies suggest cache management and scheduling methods based on the access pattern in the host interface layer [19], [36], [47], [48]. The scheduling methods based on logical addresses at the host interface layer are similar to system level scheduling. PGIS [19] increases channel level internal parallelism by identifying hot data based on trace characteristics and then collecting hot data. AAS [48] proposes a method to schedule and mitigate redundant writes based on the analysis of the address pattern. PBWS [36] identifies frequently written data with patterns and dispatches them to the same flash blocks having a small erase count. Co-Active [47] manages internal write-back cache by separating hot and cold data, reducing the number of writes and improving performance. These techniques have the advantage of reducing unnecessary flash memory access. However, there is a limitation based on logical address, and it is not suitable for flash level parallelism when translated to physical address due to data migration in storage.

In storage devices, flash level schedulers have also been studied steadily to increase the utilization of flash memory. Early research on flash level scheduling technology included striping schemes for flash level parallelism and avoidance

schemes for channel and chip conflict. Due to the nature of flash memory, flash level scheduling is generally used for out-of-order execution, separating read and write requests and prioritizing read requests [9], [28], [42]. [44] PIQ+ [16] is an internal scheduler evolved from PIQ [17] that considers access conflicts when making decisions to dispatch I/O requests and improves performance in heavy access conflict workloads by grouping non-conflicting requests. PAQ [31] proposed the scheduling method that reduces the collision of the flash channels and chips by detecting the physical flash address. Sprinkler [27] improves flash level parallelism by scheduling I/O requests based on the status of internal resources. CARS [58] increases the utilization of flash memory and provides fairness by filtering I/O requests according to conflict status. Slacker [15] proposed a slack-aware scheduler that improves response times for hosts by focusing on critical sub-requests that delay I/O responses. The slack-aware scheduler reorders to match the end of critical sub-requests when sub-requests have different slack due to resource contention on each target flash memory chip. PIOS [56] allocates requests to different batches without conflict and prioritizes smaller requests in each batch. These flash level schedulers show excellent performance in general I/O requests and are widely used commercially. However, they do not consider the nature of in-storage processing, where the priority is applied differently for each request.

As the characteristics of NAND flash memory get worse, in-SSD scheduling technology is also being studied in various forms. They consider the GC processing or the wear level of the flash memory [34] and change striping or physical allocation methods when writing [37], [38]. [52], [61]. Recently, several techniques have been proposed to reduce the GC and thereby reduce the burden on scheduling [13], [57]. WODSA [34] is one of these I/O scheduling techniques, and unlike other studies, it proposes dynamic write allocation according to wear conditions of the flash memory. Furthermore, scheduling that improves fairness in terms of QoS has also been studied [21], [49]. FLIN [49] focuses on fairness issues caused by highly concurrent I/O flows on modern NVMe SSDs. It reorders the sub-requests for flash operations, thus balancing the fairness of the requested I/O. In addition, scheduling techniques in multi-layers rather than just one layer are also studied [60].

Now, there are needs for in-storage scheduling that supports multiple levels and is suitable for in-storage processing while improving performance, QoS, and lifetime of flash memory. In this paper, we propose multi-level scheduling that is suitable for in-storage processing and improves performance and QoS. In this paper, the proposed scheduling and advanced flash-aware scheduling technologies such as PAQ, Slacker, CARS, and FLIN are compared through experimental results.

VII. CONCLUSION

This paper proposes the two-stage data processing and the scheduling for pattern matching applications. In the proposed mechanism, data processing is divided into two stages: pre-processing for sub-requests and post-processing after merging data. It is because, in an SSD, a request is divided into several sub-requests, distributed, and stored in several flash memory chips. The pre-processing for sub-requests is executed during data access, and the pre-processing reduces the execution time of the post-processing stage. In addition, this paper proposes a reordering method based on pattern matching detection. The scheduler prioritizes key requests to ensure they are completed earlier than non-key requests by leveraging the pattern matching results from the pre-processing stage. The proposed architecture eliminates the disadvantage of a strict separation between data access and data processing in a typical in-storage processing architecture. In addition, the proposed scheduling improves overall performance by reducing response time selectively for key requests that need to be sent to the host. The pattern matching detection-guided reordering method may suffer some penalties due to incorrect predictions. However, experimental results show that the proposed anti-starvation technique can reduce overhead.

REFERENCES

[1] *NVM Express*. Accessed: Feb. 28, 2021. [Online]. Available: http://www.nvmexpress.org/

[2] N. Agrawal, V. Prabhakaran, T. Wobber, D. John Davis, M. Manasse, and A. R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. Annu. Tech. Conf.*, 2008, pp. 1–14.

[3] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia, "BlockNDP: Block-storage near data processing," in *Middleware*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 8–15.

[4] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup, "Conditional computation in neural networks for faster models," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–12.

[5] T. Bisson, K. Chen, C. Choi, V. Balakrishnan, and Y.-S. Kee, "Crail-KV: A high-performance distributed key-value store leveraging native KV-SSDs over NVMe-oF," in *Proc. IEEE 37th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Nov. 2018, pp. 1–8.

[6] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017, pp. 527–536.

[7] L. Bouganim, B. Å. Jónsson, and P. Bonnet, "UFLIP: Understanding flash IO patterns," in *Proc. Conf. Innov. Data Syst. Res.*, 2009, pp. 1–12.

[8] M. Adrian Caulfield, J. Coburn, I. Todor Mollov, A. De, and A. Akel, "Understanding the impact of emerging non-volatile memories on high-performance in io-intensive computing," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–11.

[9] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Trans. Storage*, vol. 12, no. 3, pp. 1–39, Jun. 2016.

[10] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, Jun. 2009.

[11] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 266–277.

[12] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 77–90.

[13] W. Choi, M. Jung, M. Kandemir, and C. Das, "Parallelizing garbage collection with I/O to improve flash resource utilization," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, New York, NY, USA, Jun. 2018, pp. 243–254.

[14] J. Do, Y.-S. Kee, M. Jignesh Patel, C. Park, K. Park, and J. D. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1221–1230.

[15] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting intra-request slack to improve SSD performance," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2017, pp. 375–388.

[16] C. Gao, L. Shi, C. Ji, Y. Di, K. Wu, C. J. Xue, and E. H.-M. Sha, "Exploiting parallelism for access conflict minimization in flash-based solid state drives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 168–181, Jan. 2018.

[17] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha, "Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives," in *Proc. 30th Symp. Mass Storage Syst. Technol. (MSST)*, Jun. 2014, pp. 1–11.

[18] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 153–165.

[19] J. Guo, Y. Hu, B. Mao, and S. Wu, "Parallelism and garbage collection aware I/O scheduler with improved SSD performance," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 1184–1193.

[20] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. oper. Syst. (ASPLOS XIV)*, 2009, pp. 229–240.

[21] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-queue fair queuing," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 301–314.

[22] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.

[23] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 96–107.

[24] W. S. Jeong, C. Lee, K. Kim, M. K. Yoon, W. Jeon, M. Jung, and W. W. Ro, "REACT: Scalable and high-performance regular expression pattern matching accelerator for in-storage processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 5, pp. 1137–1151, May 2020.

[25] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 373–384.

[26] M. H. Jo and W. W. Ro, "Dynamic load balancing of dispatch scheduling for solid state disks," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 1034–1047, Jun. 2017.

[27] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 524–535.

[28] M. Jung, "Exploring parallel data access methods in emerging non-volatile memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 746–759, Mar. 2017.

[29] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "HIOS: A host interface I/O scheduler for solid state disks," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 289–300.

[30] M. Jung and M. T. Kandemir, "An evaluation of different page allocation strategies on high-speed ssds," in *Proc. HotStorage*, 2012, pp. 1–4.

[31] M. Jung, E. H. W. III, and M. Kandemir, "Physically addressed queueing (PAQ): Improving parallelism in solid state disks," *ACM SIGARCH Comput. Archit. News*, vol. 40, pp. 404–415, Apr. 2012.

[32] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 295–304.

[33] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 636–649, May 2012.

[34] X. Kong, Y. Yao, N. Gu, W. Feng, and X. Xu, "Wear-aware Out-of-Order dynamic scheduling for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 67, no. 1, pp. 40–49, Feb. 2021.

[35] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading communication with computing near storage," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 219–231.

[36] J. Li, X. Xu, X. Peng, and J. Liao, "Pattern-based write scheduling and read balance-oriented wear-leveling for solid state drivers," in *Proc. 35th Symp. Mass Storage Syst. Technol. (MSST)*, May 2019, pp. 126–133.

[37] C.-Y. Liu, J. B. Kotra, M. Jung, M. T. Kandemir, and C. R. Das, "SOML read: Rethinking the read operation granularity of 3D NAND SSDs," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2019, pp. 955–969.

[38] R. Liu, X. Chen, Y. Tan, R. Zhang, L. Liang, and D. Liu, "SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 966–975.

[39] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 257–270.

[40] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-M. Hwu, "DeepStore: In-storage acceleration for intelligent queries," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 224–238.

[41] B. Mao, S. Wu, and L. Duan, "Improving the SSD performance by exploiting request characteristics and internal parallelism," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 2, pp. 472–484, Feb. 2018.

[42] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (O3): An out-of-order flash memory controller architecture," *IEEE Trans. Comput.*, vol. 60, no. 5, pp. 653–666, May 2011.

[43] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: Analysis of tradeoffs," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 145–158.

[44] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based SSDs," *IEEE Comput. Archit. Lett.*, vol. 9, no. 1, pp. 9–12, Jan. 2010.

[45] S. Pei, J. Yang, and Q. Yang, "REGISTOR: A platform for unstructured data processing inside SSD storage," *ACM Trans. Storage*, vol. 15, no. 1, pp. 1–24, Apr. 2019.

[46] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing in-storage computing system for emerging high-performance drive," in *Proc. Annu. Tech. Conf.*, 2019, pp. 379–394.

[47] H. Sun, S. Dai, J. Huang, and X. Qin, "Co-active: A workload-aware collaborative cache management scheme for NVMe SSDs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1437–1451, Jun. 2021.

[48] S. Talebpour and S. Alinezhad, "AAS: Address aware scheduler for enhancing the performance of NVMe SSDs in Iot applications," in *Proc. 10th Int. Conf. Comput. Knowl. Eng. (ICCKE)*, Oct. 2020, pp. 148–153.

[49] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gomez-Luna, and O. Mutlu, "FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 397–410.

[50] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson, "SSD in-storage computing for search engines," *IEEE Trans. Comput.*, early access, Sep. 13, 2016, doi: 10.1109/TC.2016.2608818.

[51] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-direct fair queueing for NVMe SSDs," in *Proc. 19th USENIX Conf. File Storage Technol.*, Feb. 2021, pp. 403–415.

[52] F. Wu, Z. Lu, Y. Zhou, X. He, Z. Tan, and C. Xie, "OSPADA: One-shot programming aware data allocation policy to improve 3D NAND flash read performance," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 51–58.

[53] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 253–266.

[54] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, vol. 2, 2012, p. 3.

[55] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 563–568.

[56] X. Xie, L. Xiao, D. Wei, Q. Li, Z. Song, and X. Ge, "Pinpointing and scheduling access conflicts to improve internal resource utilization in solid-state drives," *Frontiers Comput. Sci.*, vol. 13, pp. 35–50, Apr. 2019.

[57] H. Yan, Y. Huang, X. Zhou, and Y. Lei, "An efficient and non-time-sensitive file-aware garbage collection algorithm for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 65, no. 1, pp. 73–79, Feb. 2019.

[58] T. Yang, P. Huang, W. Zhang, H. Wu, and L. Lin, "CARS: A multi-layer conflict-aware request scheduler for NVMe SSDs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1293–1296.

[59] J. Yoon, W. S. Jeong, and W. W. Ro, "Check-in: In-storage checkpointing for key-value store system leveraging flash-based SSDs," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 693–706.

[60] Q. Zhang, D. Feng, F. Wang, and Y. Xie, "An efficient, QoS-aware I/O scheduler for solid state drive," in *Proc. IEEE 10th Int. Conf. High Perform. Comput. Commun.*, Nov. 2013, pp. 1408–1415.

[61] W. Zhang, Q. Cao, H. Jiang, and J. Yao, "Improving overall performance of TLC SSD by exploiting dissimilarity of flash pages," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 332–346, Feb. 2020.

JOOHYEONG YOON (Student Member, IEEE) received the B.S. and M.S. degrees in information and communication engineering from Hanyang University, Seoul, South Korea, in 2005 and 2007, respectively. He is currently pursuing the Ph.D. degree with the Embedded Systems and Computer Architecture Laboratory, School of Electrical and Electronic Engineering, Yonsei University. He has been a Principal Engineer with the Controller Development Team, Memory Business, Samsung Electronics. His current research interests include in-storage processing and SSD microarchitecture design.

YOONJIN LEE received the B.S. degree in electronic and electric engineering from Hongik University, Seoul, South Korea, in 2017, and the M.S. degree in electrical and electronic engineering from Yonsei University, Seoul, in 2020. He is currently with Memory Business, Samsung Electronics Inc. His current research interests include SSD controller design and key-value store in SSDs.

WON SEOB JEONG (Member, IEEE) received the B.S. degree in electrical and electronic engineering from Chung-Ang University, Seoul, South Korea, in 2010, and the Ph.D. degree in electrical and electronic engineering from Yonsei University, Seoul, in 2020. He is currently with Memory Business, Samsung Electronics Inc. His current research interests include in-storage-processing architecture design and storage systems.

WON WOO RO (Senior Member, IEEE) received the B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California, in 1999 and 2004, respectively. He worked as a Research Scientist with the Electrical Engineering and Computer Science Department, University of California at Irvine. He currently works as a Professor with the School of Electrical and Electronic Engineering, Yonsei University. Prior to joining Yonsei University, he worked as an Assistant Professor with the Department of Electrical and Computer Engineering, California State University, Northridge. His industry experience also includes a College Internship with Apple Computer Inc., and a Contract Software Engineer with ARM Inc. His current research interests include high-performance microprocessor design, GPU micro-architectures, compiler optimization, and machine learning accelerators.