electronics

MDPI

Article

Analysis of the K2 Scheduler for a Real-Time System with an SSD

Sanghyeok Park and Jinkyu Lee *

Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Suwon 16419, Korea; tkdgur5273@skku.edu

* Correspondence: jinkyu.lee@skku.edu; Tel.: +82-31-290-7691

Abstract: While an SSD (Solid State Drive) has been widely used for storage in many computing systems due to its small average latency, how to provide timing guarantees of a delay-sensitive (real-time) task on a real-time system equipped with an SSD has not been fully explored. A recent study has proposed a work-constraining I/O scheduler, called K2, which has succeeded in reducing the tail latency of a real-time task at the expense of compromising the total bandwidth for real-time and non-real-time tasks. Although the queue length bound parameter of the K2 scheduler is a key to regulate the tradeoff between a decrease in the tail latency of a real-time task and an increase in penalty of the total bandwidth, the parameter's impact on the tradeoff has not been thoroughly investigated. In particular, no studies have addressed how the case of a fully occupied SSD that incurs garbage collection changes the performance of the K2 scheduler in terms of the tail latency of the real-time task and the total bandwidth. In this paper, we systematically analyze the performance of the K2 scheduler for different I/O operation types, based on experiments on Linux. We investigate how the performance is changed on a fully occupied SSD due to garbage collection. Utilizing the investigation, we draw general guidelines on how to select a proper setting of the queue length bound for better performance. Finally, we propose how to apply the guidelines to achieve target objectives that optimize the tail latency of the real-time task and the total bandwidth at the same time, which has not been achieved by previous studies.

Keywords: SSD (Solid State Drive); I/O scheduler in Linux Kernel; the K2 scheduler; timing guarantees; real-time systems; experiments

Citation: Park, S.; Lee, J. Analysis of the K2 Scheduler for a Real-Time System with an SSD. *Electronics* **2021**, *10*, 865. https://doi.org/10.3390/electronics10070865

Academic Editor: Paolo Torroni

Received: 15 March 2021
Accepted: 2 April 2021
Published: 6 April 2021

1. Introduction

Nowadays, an SSD (Solid State Drive) is widely deployed in computing systems from small embedded devices to large-scale high-performance servers. In particular, many computing systems in which the operating system is Linux are also equipped with an SSD as the main storage instead of an HDD (Hard Disk Drive). While most existing studies for an SSD have been confined to improving the average performance [1–13], only a few studies have aimed at addressing timing guarantees on an SSD [14–19]. Timing guarantees of real-time tasks are essential for a real-time system in which the system's validity depends not only on the functional correctness, but also on the temporal correctness (i.e., whether a given task is finished before its predefined deadline). The existing studies that support a real-time task on an SSD have focused on the latency of internal I/O operations and modified mechanisms of the FTL (Flash Translation Layer); however, modifying the FTL is usually not applicable to commercial SSDs.

A recent study has focused on the I/O scheduler in Linux Kernel and developed a work-constraining I/O scheduler, called K2 [20]. Compared to modifying the FTL, modifying the I/O scheduler in Linux Kernel is more effective in terms of ease of implementation and wide applicability to most (if not all) SSDs. The core mechanism of the K2 scheduler is to restrict the queue length bound, which is the maximum number of I/O requests in flight processed at the same time in the NVMe (Non-Volatile Memory Express) storage

device. Targeting the situation where an SSD is not fully occupied, the mechanism enables a real-time task to reduce its tail latency significantly at the expense of compromising the total bandwidth of the real-time task and non-real-time tasks. Although the queue length bound parameter of the K2 scheduler is a key to regulate the tradeoff between a decrease in the tail latency of a real-time task and an increase in penalty of the total bandwidth, the parameter's impact on the tradeoff has not been thoroughly investigated. Especially, it has not been addressed how K2's performance is changed on a fully occupied SSD in which garbage collection occurs.

In this paper, we perform various experiments on a computing system equipped with an SSD in which the operating system is Linux. Based on the experiments, we systematically analyze the performance of the K2 scheduler for different settings of the queue length bound, different I/O operation types and even different SSD occupancy (i.e., whether an SSD is partially or fully occupied). To this end, we first analyze the performance on a partially occupied SSD and then investigate how the performance is changed on a fully occupied SSD due to garbage collection. Based on the investigation, we draw general guidelines on how to select a proper setting of the queue length bound for better performance. Finally, we propose how to exploit the guidelines to achieving the typical objectives: to maximize the total bandwidth while guaranteeing a target tail latency of the real-time task, and to minimize the tail latency of the real-time task while guaranteeing a target total bandwidth.

In summary, this paper makes the following contributions.

C1. We systematically analyze the queue length bound parameter of the K2 scheduler in affecting the tail latency of a real-time task and the total bandwidth, based on real experiments on Linux Kernel.
C2. We investigate how the performance for the tail latency and the total bandwidth is changed on a fully occupied SSD in which garbage collection occurs.
C3. We develop general guidelines on how to tune the queue length bound parameter to improve the performance of the K2 scheduler.
C4. Utilizing the guidelines, we achieve target objectives that optimize the tail latency of the real-time task and the total bandwidth simultaneously, which has not been achieved by any existing studies. This is accomplished based on C2 and C3 in conjunction with C1, which also differentiates our study from existing studies.

The rest of the paper is organized as follows. Section 2 explains the basics of an SSD, including related work that addresses timing guarantees of real-time tasks on an SSD. Section 3 analyzes how different settings of the queue length bound affect the performance of the K2 scheduler for different types of the SSD operation, and Section 4 applies the analysis to the case of a fully occupied SSD that invokes garbage collection. Based on the analysis, Section 5 develops guidelines on how to set the queue length bound parameter to optimize the performance of the K2 scheduler. Finally, Section 6 concludes the paper.

2. Background and Related Work

In this section, we explain the basics of an SSD and recapitulate how existing studies utilized the characteristics of an SSD to provide timing guarantees of real-time tasks on an SSD. Next, focusing on the I/O scheduler in Linux Kernel for an SSD, we summarize the existing schedulers (including I/O schedulers that are oblivious to real-time tasks) and present details of our target I/O scheduler in Linux Kernel for an SSD (i.e., K2), which is designed to shorten the tail latency of a real-time task.

2.1. Basics of SSD

In this subsection, we explain architectures of an SSD as well as operations in an SSD. We then summarize existing studies for a real-time system equipped with an SSD.

2.1.1. Architectures of SSD

Different from an HDD that uses a mechanical spinning platter, an SSD uses NAND flash memory that operates electrically. As shown in Figure 1, the NAND flash memory area consists of a lot of pages, and a page is the smallest unit for a read/write operation. A block is a unit for an erase operation, which is comprised of 128 to 512 pages. Several blocks compose a die, which is the smallest unit for executing operations in a parallel manner. Likewise, several dies compose a chip, and several chips compose a channel.

In addition to the NAND flash memory area, an SSD has its own DRAM and an SSD controller that manages all SSD functions. The SSD controller consists of hardware and software components. The former is a processor to process operations, and the latter is a FTL (Flash Translation Layer) that takes charge of internal algorithms. The FTL has several functionalities. First, the FTL has an address mapping table to translate a logical block address of an I/O request to a physical block address in the SSD. Second, the FTL controls executions of garbage collection and wear-leveling for even lifespan of NAND flash cells in the SSD.

A storage device communicates with a driver via a host controller interface. Since HDDs use SATA (Serial-ATA) interface, SSDs in the early stage use it for compatibility. To achieve the potential for low latency, recent SSDs employ NVMe interface that allows to exploit internal SSD parallelism.

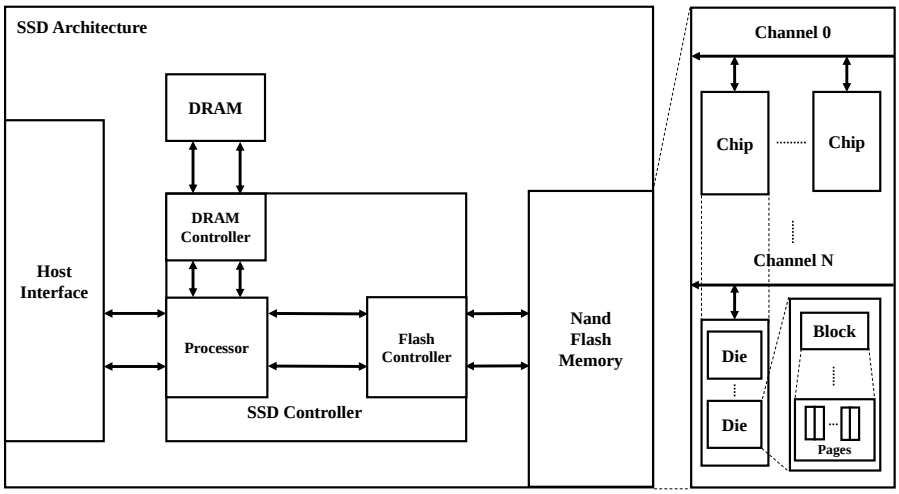


Figure 1. SSD architecture.

2.1.2. Operations in SSD

An SSD has the following basic operations: read, write and erase. As we mentioned in Section 2.1.1 the smallest unit for a read/write operation and that for an erase operation are a page and a block, respectively. Unlike an HDD, an SSD does not allow to overwrite data. To solve this problem, an SSD needs to perform garbage collection, which consists of the following three steps: (i) find blocks to be erased, (ii) write valid data from the selected blocks to be erased to other blocks, and (iii) erase the selected blocks and delete data from the address mapping table in the FTL.

In (i), the SSD controller finds proper blocks to be erased using its own selection algorithm which varies with SSD vendors. For example, a simple algorithm chooses blocks with the smallest number of valid pages, which can minimize overhead of copying valid pages to other blocks. While (ii) can be omitted in case of no valid page for all the selected blocks, most cases need to perform (ii). After (ii), the SSD controller erases the selected blocks, which necessitates deletion of address data from those blocks in (iii). Therefore, the FTL that takes charge of the address mapping table removes address data of the selected blocks. Due to (i)–(iii), garbage collection requires much more time than a read/write

operation. The time taken for garbage collection gets longer if a user updates data frequently, which can be partially solved by the TRIM command.

The TRIM command enables an SSD to efficiently handle garbage collection, which is a function for the operating system to inform the SSD of the pages that include invalid data. For example, when a file is erased, an SSD marks the NAND flash area for the file as invalid, instead of actually erasing the file data. This can minimize the latency of the erase operation, and the invalid pages are erased later when an SSD is idle.

Also, there exists a way to improve write performance, which is to add a write buffer to an SSD. Write buffering is also called SLC caching and Turbowrite. Using write buffering, write performance can be improved as long as the write buffer is not completely full, and the degree of the performance improvement depends on the write buffer capacity. Therefore, if an SSD executes write operations whose data size is larger than the write buffer capacity, the write operations return to exhibit normal write performance as if there is no write buffer. Table 1 shows an example of write performance improvement.

Table 1. Write performance in Samsung SSD 970 EVO Plus 250 GB [21].

Specification	Write Buffering Enabled	Write Buffering Disabled
250 GB sequential write	2300 MB/s	400 MB/s
250 GB random write (QD32 Thread 4)	550 K IOPS	100 K IOPS
500 GB sequential write	3200 MB/s	900 MB/s
500 GB random write (QD32 Thread 4)	550 K IOPS	200 K IOPS
1 TB sequential write	3300 MB/s	1700 MB/s
1 TB random write (QD32 Thread 4)	550 K IOPS	400 K IOPS
2 TB sequential write	3300 MB/s	1750 MB/s
2 TB random write (QD32 Thread 4)	560 K IOPS	420 K IOPS

2.1.3. Existing Studies for Real-Time System with SSD

While most existing studies for an SSD have focused on improving average performance [1–13], there exist only a few studies that aim at addressing timing guarantees on an SSD [14–19]. Chang et al. developed mechanisms that support timing guarantees of a real-time system with an SSD, by limiting the number of periodic read/write operations [14]. In detail, the study develops a predictable block recycling policy and a free-page replenishment mechanism to provide timing guarantees of real-time tasks. Also, the study employs an additional mechanism that considers the performance of non-real-time tasks.

The RFTL (Real-Time Flash Translation Layer) paper improves the worst-case performance when garbage collection is performed [15]. While the traditional garbage collection selects victim blocks, reads their valid data pages, writes them in other blocks and finally erases the victim blocks, RFTL splits the garbage collection process into several sub-processes and performs each process at the end of normal read/write operations that do not trigger garbage collection. To this end, the study develops a new FTL called RFTL, which enables a distributed garbage collection at the expense of reserving a large storage volume. RFTL is an improved version of the study that reduces the latency of garbage collection by utilizing a write buffer and developing a partial block cleaning policy [16].

There also exists a different study that modifies the FTL, called WAO-GC [17], which also aims at reducing the latency caused by garbage collection. WAO-GC postpones garbage collection; this minimizes valid data pages of victim blocks, which in turn reduces the effect of garbage collection on the latency. In addition, WAO-GC employs a page-level mapping table, which allows a partial page copy for garbage collection. Although reducing the worst-case latency, this approach incurs high memory overhead due to usage of the page-level mapping table. In addition, another study tries to guarantee the worst-case latency by limiting the number of log block merges for hybrid-level mapping on the FTL [18].

Missimer and West utilize SSD internal parallelism to minimize the effect of garbage collection [19]. Focusing on read/write operations, the study proposes a partitioned FTL,

which divides the internal NAND flash into two chips (for read and write operations); then both chips can execute their designated operations in a parallel manner. Since chips for read operations and those for write operations are separated, a read operation cannot be blocked by a write operation. In particular, while a write operation may incur a long latency due to garbage collection under vanilla FTL, the study can eliminate the effect of garbage collection triggered by a write operation on a read operation, which yields a lower latency.

2.2. I/O Schedulers in Linux Kernel for an SSD

As we explained in Section 2.1, there are interesting characteristics in different components for an SSD. To support real-time tasks, we target the I/O scheduler in Linux Kernel for an SSD as a component to be modified, which allows to utilize the following two advantages. First, it is convenient to modify the I/O scheduler in Linux Kernel, and second, the modification is collectively applied to most (if not all) SSDs regardless of their manufacturers. This is different from modifying components within a target SSD; for example, if we need to modify the internal scheduling algorithm of a target SSD, we should (i) disclose details of the algorithm used in the SSD, which is usually unreleased to the public and (ii) find a way to modify internals of the SSD, which varies with SSD manufacturers (or is not permitted). In this section, we first outline the block layer in Linux Kernel, which manages I/O schedulers. We then explain existing I/O schedulers in Linux Kernel for an SSD, designed for maximizing average performance of general-purpose tasks.

2.2.1. Block Layer in Linux Kernel

The block layer is a part of Linux Kernel, which is an interface that allows userspace applications to access various storage devices. In the block layer, there exists I/O schedulers that have their own algorithms to manage I/O requests. In the single-queue block layer, the I/O scheduler usually has a single staging queue to control the order of I/O requests to be processed; I/O requests in the staging queue can be merged to improve performance. The single-queue block layer is used in existing Linux systems for processing I/O requests from userspace to HDDs and SATA SSDs. However, with advent of an NVMe SSD which exhibits much lower latency than conventional storage devices, the single-queue block layer with only one queue and one lock becomes a bottleneck. Thus, the multi-queue block layer, which has multiple queues and multiple locks, was developed to solve the bottleneck problem of the single-queue block layer. Since our target storage is an NVMe SSD, we assume to use the multi-queue block layer, which is shown in Figure 2.

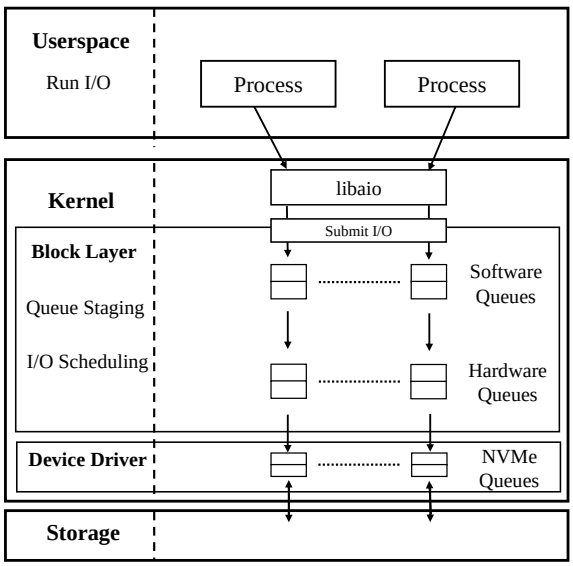


Figure 2. Multi-queue block layer in Linux Kernel.

2.2.2. I/O Schedulers for General-Purpose Tasks

In Linux Kernel, there exist several I/O schedulers for an SSD, which are developed for improving average performance of general-purpose tasks. Now, we explain the following I/O schedulers: the none [20], BFQ (Budget Fair Queuing) [22], MQ-Deadline (Multi-Queue Deadline) [23], and Kyber [24] schedulers.

The none scheduler [20], as the name indicates, implies there is no specific I/O scheduler that regulates the mechanism of block layers. That is, the none scheduler passes I/O requests to the NVMe queue in the FIFO (First-In-First-Out) manner. Since a latency in an SSD is much smaller than that in an HDD, even a simple operation in Linux Kernel could result in significant overhead. Therefore, to maximize the performance by avoiding unnecessary overhead as much as possible, the none scheduler has been widely employed.

The BFQ scheduler [22] tailors the CFQ (Completely Fair Queuing) scheduler in the single-queue block layer, to the multi-queue block layer. Although the BFQ scheduler fairly divides storage device resources using weight, it requires complex operations. This incurs large overhead, and in turns makes it difficult to employ the BFQ scheduler in practical systems.

The MQ-Deadline scheduler [23] targets the Deadline scheduler for the single-queue block layer, and modifies it for the multi-queue block layer. Note that the Deadline scheduler reduces HDD seek time by storing I/O requests in multiple software queues. In the MQ-Deadline scheduler, if an I/O request is not performed within its deadline, it will be promoted to the highest priority; otherwise, each I/O request is performed according to the order of each sector number.

The Kyber scheduler [24] uses token-bucket algorithms and assigns a different target latency for each operation. By default, the target latency is 2 ms for read requests and 10ms for write requests. To complete each I/O request within its target latency, the number of tokens limits the number of I/O requests in flight in NVMe queues. Also, there exist sixteen latency buckets for each core, consisting of eight buckets to track read latency and other eight buckets to track write latency. Each latency bucket has its own range of latency between 1/4 and 8/4 of its target latency value. The number of tokens is adjusted by the scheduler, according to the bucket index of the 99th percentile sample latency. In other words, if the 99th percentile sample latency exceeds the target latency value, the scheduler allocates more tokens. Although the latency of each operation is controlled by the scheduler, the latency of each application cannot be directly controlled.

In summary, there exist different I/O schedulers in Linux Kernel for an SSD, designed for maximizing average performance of general-purpose tasks. Although some of the explained schedulers have features to control I/O requests, it is not clear how to utilize the features for timing guarantees of real-time tasks. In Section 3.1, we will explain our target I/O scheduler, called K2, specialized for reducing the latency of a real-time task.

3. Analysis of the K2 Scheduler for Partially Occupied SSD

In this section, we investigate how different settings for the queue length bound affect the performance of the K2 scheduler with different I/O operations, when the SSD is not fully occupied. To this end, we first explain our target I/O scheduler, called K2. We next explain experiment settings for the K2 scheduler. Finally, we analyze the tradeoff between a decrease in the tail latency of a real-time task and an increase in penalty of the total bandwidth.

3.1. K2 Scheduler

As we mentioned in Section 2.1.3, existing studies that support real-time tasks on an SSD focused on the latency of internal I/O operations and modified mechanisms of the FTL. However, modifying the FTL is usually not applicable to commercial SSDs, because the mechanisms of the FTL developed by the manufacturers of SSDs themselves or those of their controllers are usually neither (i) open to the public nor (ii) modifiable. On the other

hand, the K2 scheduler [20] supports timing guarantees of a real-time task, by modifying the I/O scheduler in Linux Kernel, which satisfies both (i) and (ii).

The K2 scheduler targets the situation where an SSD is not fully occupied; in the target situation, benchmark results are affected by the write buffer, but not by garbage collection. Also, the K2 scheduler targets an NVMe SSD, and therefore it is developed by considering the Linux multi-queue block layer. The core mechanism of the K2 scheduler is to limit the queue length bound, which is the maximum number of I/O requests in flight processed at the same time in the NVMe storage device; the queue length bound is a tunable parameter.

The K2 scheduler has nine staging queues: eight for existing Linux real-time I/O priorities and one for all non-real-time I/O requests. Whenever a userspace application generates I/O requests, the requests are sent to the Linux Kernel block layer and enqueued to proper staging queues based on their metadata. When the current number of requests in flight is less than the queue length bound, the real-time staging queue which has the highest priority starts to pass its I/O requests to NVMe queues. If all real-time staging queues have no I/O request, the non-real-time staging queue sends its I/O requests to the NVMe queues. However, when the current number of requests in flight is the same as the queue length bound, the K2 scheduler prevents I/O requests from spreading to NVMe queues. When the NVMe driver sends a completion signal to the scheduler, the scheduler decreases the current number of requests in flight and resumes to propagate I/O requests in their staging queues to NVMe queues. This mechanism allows to prioritize I/O requests in real-time staging queues, yielding a reduced tail latency of a real-time task at the expense of limiting the overall bandwidth. Due to the queue length bound that controls the overall bandwidth performance of an SSD, the K2 scheduler is also called a work-constraining scheduler. As expected, tuning the queue length bound is a key to regulate the tradeoff between a decrease in the tail latency of a real-time task and an increase in penalty of the total bandwidth; however, the effect of the tuning on the tradeoff has not been thoroughly investigated, especially for the situation where an SSD is fully occupied. In Sections 3.3 and 4.2, we will analyze the performance of the K2 scheduler on a partially occupied SSD and fully occupied SSD, respectively, in terms of the effect.

3.2. Experiment Settings for Partially Occupied SSD

To analyze the performance of the K2 scheduler, we use the source kernel and the scheduler provided by the K2 scheduler paper [20]. The target operating system is Ubuntu 18.04, but the source kernel includes LTTng, an open source program that allows to trace I/O latency in Linux Kernel. We utilize LTTng tracepoints to measure SSD I/O latency, which aims at recording request submission, request propagation between queues, and request completion. Our target computing system is equipped with Intel(R) Core(TM) i5-9400 CPU @ 2.90Hz, 16GB memory and Samsung SSD 970 EVO Plus 250GB. We enable the option of blk-mq, which maximizes the performance of an NVMe SSD in Linux. We summarize our experimental environments in Table 2.

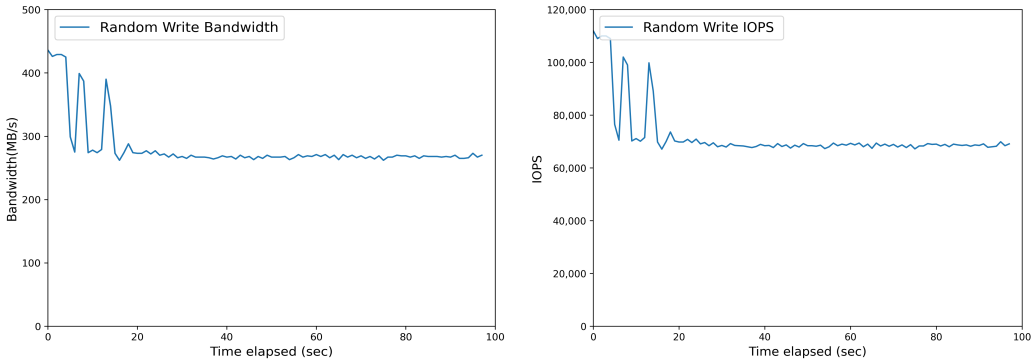
Table 2. Experiment environments.

Specification	Software and Hardware Environments
CPU	Intel(R) Core(TM) i5-9400 CPU@2.90Hz
Memory	16 GB
SSD	Samsung SSD 970 EVO Plus 250 GB
SSD interface	NVM express
Operating system	Ubuntu 18.04.2 LTS
Kernel version	4.15.0 tracepoint modified version
Kernel Blk-mq option	Enabled

When it comes to benchmark, we target the FIO benchmark with 3.25 version. For the FIO benchmark, we execute three non-real-time tasks and one real-time task, and measure

the latency of the real-time task and the total bandwidth. Also, the non-real-time tasks and the real-time task are distinguished by assigning different priorities. For the non-real-time tasks, we execute three asynchronous processes so as to obtain their maximum bandwidth. For the real-time task, we execute a synchronous process, and its consecutive requests are separated with a 2-ms-long issuing term. To accurately measure the latency on an SSD, we use the direct I/O mode, which eliminates the effect of the buffer cache in Linux. We fill about half of the blocks in the target SSD, which is different from the settings to be applied in Section 4.

The above benchmark settings are the same as those in the K2 scheduler paper [20] except the followings. First, we did not establish the target bandwidth since we are interested in the maximum performance. Second, we set the experiment time of our benchmark as 100 s, which is different from 10 s in the K2 scheduler paper. This is because, the experiment time less than 20 s can result in unstable bandwidth and IOPS (I/O Per Second) due to the effect of write buffering, as shown in Figure 3.



(**a**) The total bandwidth (**b**) IOPS

Figure 3. The total bandwidth and IOPS for random write operations on a partially occupied SSD.

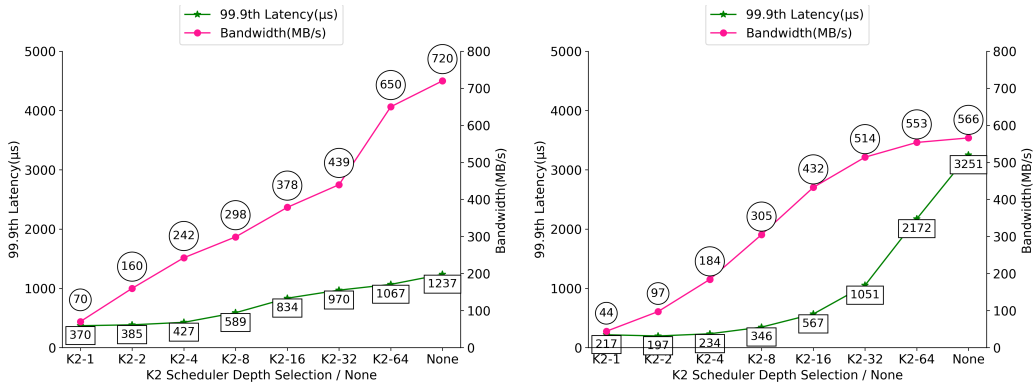
Under the above experimental environments, the SSD is exercised with (i) a sequential read, (ii) a random read, (iii) a sequential write and (iv) a random write, all with 4 KB block size. To avoid unexpected outlier values, we repeat five experiments for each operation under the same experimental environments and show the average of the results for the five experiments, which will be presented in the next subsection.

3.3. Investigation of Latency and Bandwidth on Partially Occupied SSD

As we mentioned in Section 3.1, the K2 scheduler can significantly reduce the worst-case latency (99.9th percentile) of a real-time task. In addition to the capability of the K2 scheduler in achieving a low latency for a target task, we systematically analyze key factors that affect the latency of the real-time task and the total bandwidth of the real-time task and non-real-time tasks. To this end, we target the K2 scheduler with different settings for the queue length bound (and the none scheduler as a reference), and do experiments that show the performance according to different operation types (i.e., random/sequential read/write operations). To express the K2 scheduler with different settings for the queue length bound, let K2-x denote the K2 scheduler with the queue length bound of x. For example, K2-8 implies the K2 scheduler which sets the queue length bound to 8.

Figure 4a,b show the worst-case latency (99.9th percentile) of the real-time task and the total bandwidth of the real-time and non-real-time tasks, when the target operation is sequential read and random read, respectively. If we focus on a sequential read operation shown in Figure 4a, the none scheduler yields 1237 µs for the worst-case latency of the real-time task. The worst-case latency decreases as the queue length bound of the K2 scheduler decreases; if the queue length bound is 1 (i.e., K2-1), the latency reduces to 370 µs, resulting in 3.3x latency reduction compared to the none scheduler. On the other hand, the

total bandwidth is reduced from 720 MB/s (for the none scheduler) to 70 MB/s (for the K2-1 scheduler). This implies that reducing the latency of the real-time task incurs much overhead, yielding a significant drop of the total bandwidth.



(**a**) Sequential read (**b**) Random read

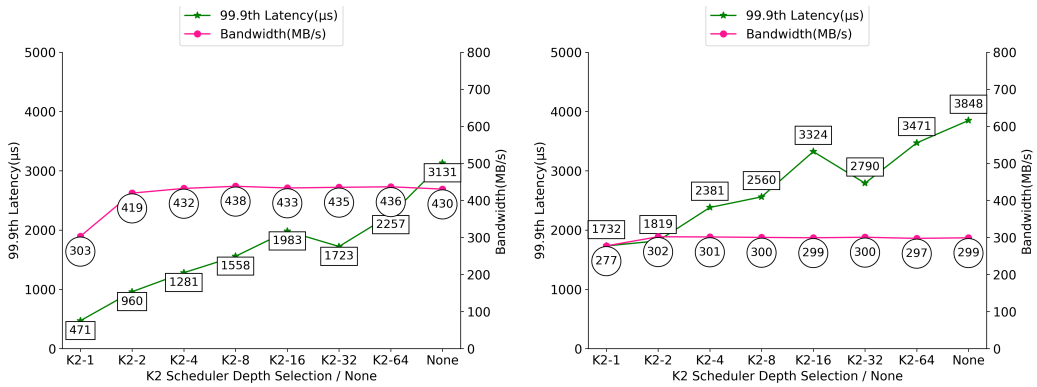
Figure 4. Performance of sequential/random read operations on a partially occupied SSD.

The performance trend of the worst-case latency of the real-time task and the total bandwidth of the real-time and non-real-time tasks for random read operations is similar to that for random read operations, as shown in Figure 4b. That is, by minimizing the queue length bound, the K2-1 scheduler yields 217 µs for the worst-case latency of the real-time task, which is 15.0x reduction compared to the none scheduler (that yields 3251 µs). On the other hand, minimizing the queue length bound makes the K2 scheduler limit its total bandwidth significantly from 566 MB/s (the none scheduler) to 44 MB/s (the K2-1 scheduler), which is 12.9x performance loss.

We would like to mention that there is an interesting result for the random read operation. If we compare the worst-case latency of the real-time task under K2-1 and that under K2-2, the former (i.e., 217 µs) does not yield a lower latency than the latter (i.e., 197 µs). As the queue length bound decreases, the time to process the read operation in an SSD gets shorter. Hence, the time is eventually smaller than the time to pass the operation from the staging queue to the SSD under a small value of the queue length bound. Under K2-1, the overhead of the I/O scheduler dominates the advantage of the I/O scheduler in processing the real-time task, yielding a longer worst-case latency of the real-time task than K2-2. Note that this phenomenon cannot be observed when the target operation is sequential read, as shown in Figure 4a. This is because sequential read requests in the staging queue tend to be merged with high probability, which reduces the I/O scheduler overhead by reducing the number of actual requests to be passed from the staging queue to the SSD.

Now, we analyze the write operation case. Figure 5a,b show the worst-case latency (99.9th percentile) of the real-time task and the total bandwidth of the real-time and non-real-time tasks, when the target operation is sequential write and random write, respectively. As shown in Figure 5a that represents the sequential write operation case, the bandwidth drop according to reduction of the queue length is not as significant as the sequential read operation case. That is, while the none scheduler exhibits 430 MB/s total bandwidth, the K2-1 scheduler exhibits 303 MB/s total bandwidth, resulting in only 1.42x decrease compared with the none scheduler; in addition, the K2 scheduler with other settings for the queue length bound does not yield a meaningful bandwidth drop. Since a write operation takes longer than a read operation, write operations are not much affected by decreasing the queue length bound as long as the bound is no smaller than the threshold (which is two in this case). On the other hand, the worst-case latency of the real-time task is 3131 µs, 960 µs, 471 µs, respectively for the none, K2-2 and K2-1 schedulers; K2-2 and K2-1 yield 3.3x and 6.6x latency reduction, compared to the none scheduler. This means, the

K2 scheduler with the queue length bound larger than one (e.g., K2-2) yields a significant latency reduction of the real-time task with negligible total bandwidth loss. Therefore, for sequential write operations, it is possible to support timing guarantees of the real-time task without compromising the total bandwidth.



(**a**) Sequential write (**b**) Random write

Figure 5. Performance of sequential/random write operations on a partially occupied SSD.

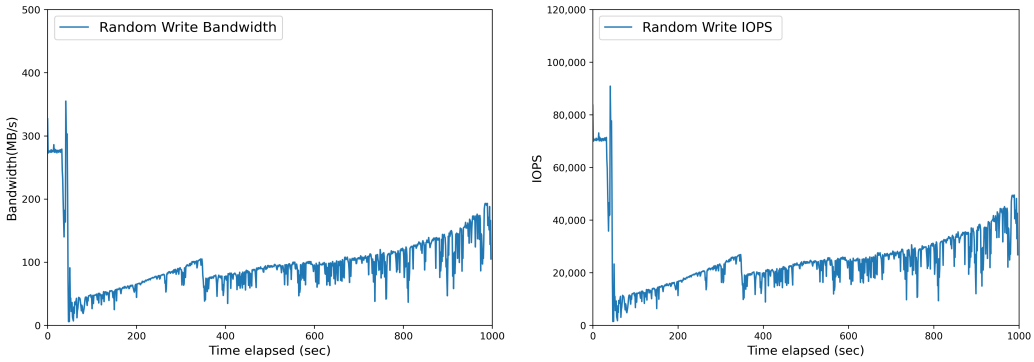
The results for the random write operation case are similar to those for the sequential write operation case, as shown in Figure 5b. The worst-case latency decreases as the queue length bound decreases: from 3848 µs under the none scheduler and 3471 µs under K2-64 to 1732 µs under K2-1, which is 2.2x latency reduction compared to the none scheduler. The total bandwidth of K2 with the queue length bound larger than 1 (i.e., 297–302 MB/s) is similar to that of the none scheduler (i.e., 299 MB/s), while that of K2-1 is slightly smaller (i.e., 277 MB/s).

4. Analysis of the K2 Scheduler for Fully Occupied SSD

While the K2 scheduler paper [20] targets a partially occupied SSD, a fully occupied SSD is expected to yield a longer latency of the real-time task due to garbage collection. In this section, we analyze the performance of the K2 scheduler on a fully occupied SSD.

4.1. Experiment Settings for Fully Occupied SSD

The experiment settings for a fully occupied SSD are similar to that for a partially occupied SSD explained in Section 3.2. The differences are to enable to perform garbage collection by occupying all blocks of the SSD and to execute the benchmark for a sufficiently long duration (i.e., 1000 s); the latter makes it possible to observe the effect of garbage collection more accurately. To check whether the experiment settings successfully incur garbage collection, we plot the total bandwidth and IOPS of the none scheduler on a fully occupied SSD in Figure 6; note that those of the K2 scheduler on a fully occupied SSD exhibit similar behaviors. It is confirmed that the total bandwidth and IOPS of random write operations decrease sharply at 50 s after the start of the target benchmark, which is different from the partially occupied SSD case in Figure 3. That is, while the total bandwidth and IOPS under a partially occupied SSD (shown in Figure 3) are respectively about 300 MB/s and 70,000, those under a fully occupied SSD (shown in Figure 6) are much smaller, which indicates that garbage collection occurs in our experiment settings on a fully occupied SSD.



(**a**) The total bandwidth (**b**) IOPS

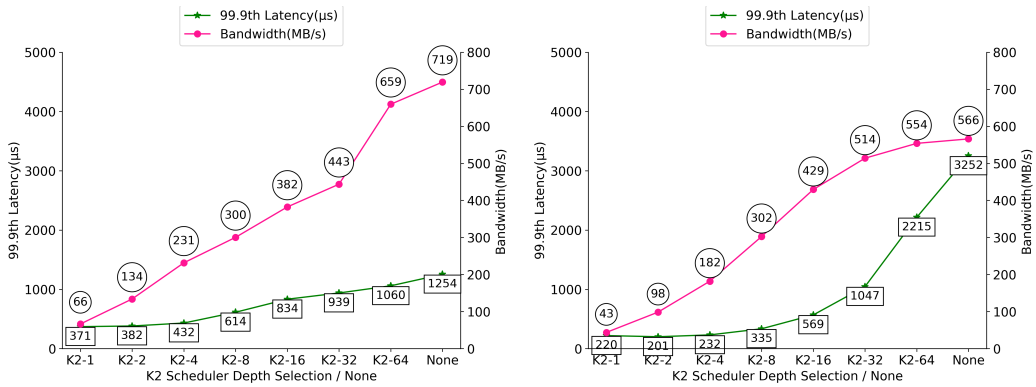
Figure 6. The total bandwidth and IOPS for random write operations on a fully occupied SSD.

4.2. Investigation of Latency and Bandwidth on Fully Occupied SSD

We observe that the performance of read operations on a fully occupied SSD shown in Figure 7 is almost the same as that on a partially occupied SSD shown in Figure 4. This is because, read operations do not trigger garbage collection even if the SSD is fully occupied. Therefore, there is no meaningful difference between Figures 4 and 7 for both sequential and random read operations.

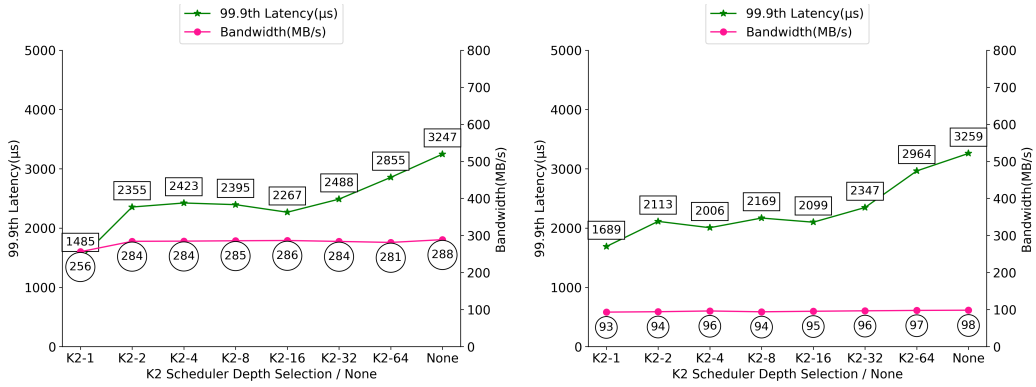
On the other hand, we make the following observations for the performance of write operations in Figure 8. First, the total bandwidth does not significantly vary with different values of the queue length bound for both sequential and random write operations. For example, while there is a gap between the total bandwidth for K2-1 and that for K2 with other queue length bounds for sequential write operations under a partially occupied SSD (i.e., 303 MB/s versus 419–438 MB/s shown in Figure 5a), the same cannot be said true under a fully occupied SSD in that the total bandwidth under K2 with any queue length bounds (as well as that of the none scheduler) is between 256 MB/s and 288 MB/s shown in Figure 8a. A similar trend also holds for the random write operation; while there is about 10% difference between the total bandwidth for K2-1 and that for K2 with other queue length bounds under a partially occupied SSD (see Figure 5b), there is no meaningful difference under a fully occupied SSD (see Figure 8b). This implies that garbage collection lowers the usable bandwidth, yielding a less bandwidth change for different queue length bounds. The difference between the sequential write case and the random write case comes from the difference between the processing time for sequential write operations and that for random write operations. That is, sequential write operations incur more request merges, which yields a shorter processing time and makes it possible to be affected by the queue length bound setting more significantly. We observe that random write operations under garbage collection yield the lowest processing time, which results in little bandwidth change according to different settings for the queue length bound as shown in Figure 8b.

Second, the worst-case latency of the real-time task decreases as the queue length bound decreases; in addition, the degree of the latency reduction for the sequential write case is similar to that for the random write case, which is different from the results for the corresponding sequential/random write cases under a partially occupied SSD. For example, the worst-case latency for the none scheduler for sequential write operations is 3247 µs, which is reduced to 1485 µs for K2-1; for random write operations, the worst-case latency for the none scheduler is 3259 µs, which is reduced to 1689 µs for K2-1. This is because, garbage collection makes it difficult to reduce the tail latency of the real-time task as much as the situation where an SSD is partially occupied without garbage collection.



(**a**) Sequential read (**b**) Random read

Figure 7. Performance of sequential/random read operations on a fully occupied SSD.



(**a**) Sequential write (**b**) Random write

Figure 8. Performance of sequential/random write operations on a fully occupied SSD.

5. Optimizing the K2 Scheduler: Queue Length Bound Selection

Through Sections 3 and 4, we observe that the queue length bound parameter regulates the tradeoff between a decrease in the tail latency of the real-time task and an increase in penalty of the total bandwidth, and the tradeoff varies with the I/O operation type (i.e., sequential/random read/write) and the SSD occupancy (partially or fully occupied). In this section, we draw general guidelines on how to select a proper setting for the queue length bound that optimizes the performance of the K2 scheduler. Also, we propose how to apply the guidelines to the following typical objectives:

O1. For given $0 < \alpha \leq 100$, to maximize the total bandwidth while guaranteeing at most α% of the tail latency of the real-time task under the none scheduler.
O2. For given $0 < \beta \leq 100$, to minimize the tail latency of the real-time task while guaranteeing at least β% of the total bandwidth under the none scheduler.

We now focus on the read operation case and summarize its characteristics explained in Sections 3 and 4 as follows.

R1. The SSD occupancy does not affect the tail latency of the real-time task and the total bandwidth.
R2. The total bandwidth decreases as the queue length bound decreases.
R3. The tail latency of the real-time task and the total bandwidth mostly decrease as the queue length bound decreases, but this does not hold for random read operations under K2-1.

Considering R1–R3, general guidelines for the queue length bound selection are presented as follows. First, we do not need to distinguish whether the SSD is partially or fully occupied, according to R1. Second, to increase the total bandwidth, we should increase the queue length bound, according to R2. Third, to decrease the tail latency of the real-time task, we should decrease the queue length bound, but no more decrement is needed for random read operations if the queue length bound is 2, which comes from R3.

The general guidelines can be applied to achieving O1 and O2 as follows. To achieve O1, we first find a set of candidates for the queue length bound setting, which guarantee at most α% of the tail latency of the real-time task under the none scheduler; the set depends on the operation type (i.e., sequential read or random read), but not on the SSD occupancy (i.e., partially or fully occupied SSD). Second, we choose the largest queue length bound among the candidates. For example, if $\alpha = 50\%$, the candidates for the queue length bound setting that satisfy the condition for the tail latency are {1, 2, 4, 8} and {1, 2, 4, 8, 16, 32}, respectively for the sequential and random read operation cases regardless of the SSD occupancy. Then, we choose the largest queue length bound, which is 8 and 32, respectively for the sequential and random read operation cases. The way of achieving O2 is similar to that of achieving O1. For example, if $\beta = 50\%$, we find that the queue length bound of {16, 32, 64} and {8, 16, 32, 64} can achieve at least 50% total bandwidth under the none scheduler, respectively for the sequential and random read operation cases regardless of the SSD occupancy. Then, we choose the smallest queue length bound, which is 16 and 8, respectively for the sequential and random read operation cases.

Next, when it comes to the write operation case, the characteristics analyzed in Sections 3 and 4 can be summarized as follows.

W1. The SSD occupancy has an impact on the tail latency of the real-time task and the total bandwidth.
W2. The tail latency of the real-time task mostly decreases as the queue length decreases.
W3. While the queue length bound does not affect the total bandwidth in most cases, K2-1 yields a drop of the total bandwidth under some settings.

According to W1, we need to draw different guidelines of the queue length bound selection for W2 and W3, according to the SSD occupancy. If we want to maximize the total bandwidth on a partially occupied SSD, we choose any queue length bound (i.e., K2-2, K2-4, ..., K2-64) except one (i.e., K2-1); on a fully occupied SSD, while the same holds for the sequential write case, we can choose any queue length bound (including one) for the random write case. To decrease the tail latency of the real-time task, we need to decrease the queue length bound regardless of the SSD occupancy; however, we should consider some odd points which do not yield a smaller tail latency than that with a larger queue length bound, which vary with the operation type and the SSD occupancy.

The guidelines for the write operation can be applied to achieving O1 and O2 as follows. For O1, we find a set of settings for the queue length bound that yield at most α% of the tail latency of the real-time task under the none scheduler, and then choose one of the settings other than 1 (note that for the random write case on a fully occupied SSD, we can choose any setting). For example, for the sequential write case on a partially occupied SSD, {1, 2, 4, 8} is a set of settings for the queue length bound that guarantee at most $\alpha = 50\%$ of the tail latency of the real-time task under the none scheduler. Among the settings, we can choose any of {2, 4, 8} to maximize the total bandwidth. For O2, if the total bandwidth under K2-1 is more than β% of the total bandwidth under the none scheduler, we choose K2-1; otherwise, we choose K2-2. For example, if $\beta = 50\%$ for the sequential write case on a partially occupied SSD, we choose K2-1; if $\beta = 80\%$, we choose K2-2.

Note that different experimental environments may yield different specific points in the guidelines; for example, in R3 and W3, the K2 scheduler with the queue length bound of "one" exhibits a different behavior from that with other settings. However, the guidelines of R1–R3 and W1–W3 are generally applicable to different experimental environments, as long as we tailor the guidelines to a specific experimental environment.

6. Conclusions and Discussion

In this paper, we performed various experiments on a computing system equipped with an SSD under different experimental environments in order to demonstrate importance of tuning a proper queue length bound for the K2 scheduler. Based on the experiments, we systematically analyzed the tail latency of the real-time task and the total bandwidth not only on a partially occupied SSD, but also on a fully occupied SSD in which garbage collection occurs. We drew general guidelines on how to select a proper setting of the queue length bound for better performance, and succeeded to achieve the typical objectives that optimize the tail latency of the real-time task and the total bandwidth at the same time.

Despite the advantages explained so far, the proposed approaches have the following limitations. First, we need to perform new experiments if we do not have any experimental results for a target situation. This makes it difficult to use our approaches in a situation where a target SSD or a target application is unknown in advance. Second, our approaches are confined to a tail latency of a real-time task executed on an SSD, but they cannot consider a total latency of a real-time task executed on a computing unit (e.g., CPU) and a storage (e.g., SSD).

Those limitations suggest a direction for future work as follows. First, we would like to develop a method that adaptively adjusts the queue length bound at runtime without relying on previous experiment results. Second, it is interesting to utilize the analysis developed in this paper for end-to-end timing guarantees of a real-time task, which necessitate consideration of delays not only in an SSD, but also in CPU and networks.

Author Contributions: Conceptualization, S.P. and J.L.; software, S.P.; experiments, S.P.; data curation, S.P. and J.L.; writing—original draft preparation, S.P. and J.L.; writing—review and editing, J.L.; supervision, J.L.; project administration, J.L.; funding acquisition J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2021R1A2B5B02001758, 2019R1A2B5B02001794, 2017H1D8A2031628).

Data Availability Statement: The data presented in this study are available on request from the first author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gupta, A.; Kim, Y.; Urgaonkar, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, USA, 7–11 March 2009.
2. Kim, J.; Oh, Y.; Kim, E.; Choi, J.; Lee, D.; Noh, S.H. Disk schedulers for solid state drivers. In Proceedings of the ACM & IEEE International Conference on Embedded Software, Grenoble, France, 12–16 October 2009.
3. Wu, C.-H.; Kuo, T.-W. An adaptive two-level management for the flash translation layer in embedded systems. In Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA, USA, 5–9 November 2006.
4. Qin, Z.; Wang, Y.; Liu, D.; Shao, Z.; Guan, Y. MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems. In Proceedings of the Design Automation Conference (DAC), San Diego, CA, USA, 5–9 June 2011.
5. Jung, M.; Wilson, E.H.; Kandemir, M. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In Proceedings of the Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 9–13 June 2012
6. Jung, D.; Kang, J.; Jo, H.; Kim, J.S.; Lee, J. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Trans. Embed. Comput. Syst.* **2010**, *9*, 1–29. [CrossRef]
7. Jung, M.; Choi, W.; Srikantaiah, S.; Yoo, J.; Kandemir, M.T. HIOS: A host interface I/O scheduler for Solid State Disks. In Proceedings of the Annual International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 14–18 June 2014.
8. Jung, M.; Prabhakar, R.; Kandemir, M.T. Taking garbage collection overheads off the critical path in SSDs. In Proceedings of the 13th International Middleware Conference, Montreal, QC, Canada, 3–7 December 2012.
9. Jung, S.; Song, Y.H. Garbage Collection for Low Performance Variation in NAND Flash Storage Systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2015**, *34*, 16–28. [CrossRef]
10. Lee, J.; Kim, Y.; Shipman, G.M.; Oral, S.; Kim, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2013**, *32*, 247–260. [CrossRef]

11. Zhang, J.; Shu, J.; Lu, Y. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In Proceedings of the USENIX Annual Technical Conference (ATC), Denver, CO, USA, 22–24 June 2016.
12. Chen, T.Y.; Chang, Y.H.; Ho, C.C.; Chen, S.H. Enabling sub-blocks erase management to boost the performance of 3D NAND flash memory. In Proceedings of the Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016.
13. Liu, C.Y.; Kotra, J.; Jung, M.; Kandemir, M. PEN: Design and evaluation of partial-erase for 3D NAND-based high density SSDs. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST), Oakland, CA, USA, 12–15 February 2018.
14. Chang, L.P.; Kuo, T.W.; Lo, S. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* **2004**, *3*, 661–863. [CrossRef]
15. Qin, Z.; Wang, Y.; Liu, D.; Shao, Z. Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems. In Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS), Beijing, China, 16–19 April 2012.
16. Choudhuri, S.; Givargis, T.D. Deterministic service guarantees for nand flash using partial block cleaning. In Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Atlanta, GA, USA, 19–24 October 2008.
17. Zhang, Q.; Li, X.; Wang, L.; Zhang, T.; Wang, Y.; Shao, Z. Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems. In Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS), Seattle, WA, USA, 13–16 April 2015.
18. Cho, H.; Shin, D.; Eom, Y.I. KAST: K-associative sector translation for NAND flash memory in real-time systems. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Nice, France, 20–24 April 2009.
19. Missimer, K.; West, R. Partitioned Real-Time NAND Flash Storage. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Nashville, TN, USA, 11–14 December 2018.
20. Miemietz, T.; Weisbach, H.; Roitzsch, M.; Härtig, H. K2: Work-Constraining Scheduling of NVMe-Attached Storage. In Proceedings of the IEEE Real-Time Systems Symposium (RTSS), Hong Kong, China, 3–6 December 2019.
21. Samsung. Samsung V-NAND SSD 970 EVO Plus Data Sheet. 2019. Available online: https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Samsung_NVMe_SSD_970_EVO_Plus_Data_Sheet_Rev-2-0.pdf (accessed on 9 March 2021).
22. Valente, P.; Andreolini, M. Improving application responsiveness with the BFQ disk I/O scheduler. In Proceedings of the 5th Annual International Systems and Storage Conference, Haifa, Israel, 4–6 June 2012.
23. Axboe, J. mq-Deadline: Add blk-mq Adaptation of the Deadline IO Scheduler, Commit Message for the Linux Kernel. Available online: https://lore.kernel.org/patchwork/patch/750212/ (accessed on 9 March 2021).
24. Sandoval, O. blk-mq: Introduce Kyber Multiqueue I/O Scheduler, Commit Message for the Linux Kernel. Available online: https://patchwork.kernel.org/patch/9672023 (accessed on 9 March 2021).