Regular Paper

I/O Scheduler using splitting mechanism for improving Real-Time performance in eMMC

Shinnosuke Koshiba[1,a)] Yutaka Matsubara[1] Hiroaki Takada[1]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: Embedded MultiMedia Card (eMMC) is widely used in many real-time systems, such as in automotive applications. In embedded systems, it is common for applications to share storage resources, which creates a demand for protecting real-time processes from interference by other processes. Since eMMC lacks internal command queuing, the I/O scheduler becomes a critical component in ensuring performance. Under these conditions, priority-based schedulers for real-time systems have been developed, successfully protecting real-time processes. However, our experiments revealed that while these schedulers can handle the bandwidth load caused by a large number of small-sized I/O requests, they fail to manage the load caused by a small number of large-sized I/O requests. In such cases, tail latency of real-time process significantly worsens to levels comparable to systems without a scheduler. In this research, we propose an I/O scheduler equipped with an I/O splitting mechanism that converts a small number of large-sized I/O operations into a large number of small-sized I/O operations. Experiments demonstrated that, compared to existing priority-based schedulers, this approach successfully reduced 99th percentile latency, achieving up to a 5-fold reduction in read latency and up to a 3-fold reduction in write latency.

Keywords: I/O scheduler, eMMC

1. Introduction

In recent years, storage devices have made remarkable advancements, with flash-based storage such as Embedded MultiMedia Card (eMMC) and Solid-State Drive (SSD) achieving higher throughput and lower latency compared to traditional Hard Disk Drive (HDD). Among them, eMMC is widely adopted in many embedded systems due to its low power consumption and compact size, and it is extensively used in systems that require real-time performance, such as automotive applications.

Furthermore, as the number of applications in embedded systems increases, it has become common for multiple applications with varying levels of importance to share the same storage device. In such environments, there is a growing demand to protect real-time processes with strict timing constraints from interference by lower-priority processes. Since eMMC lacks internal command queuing, the I/O scheduler plays a critical role in meeting these demands. Many I/O schedulers included in the standard Linux kernel or currently under development primarily focus on fairness and average performance [1, 2, 3, 4, 5]. However, several priority-based I/O schedulers have also been developed for real-time systems [6, 7, 8]. These schedulers have successfully reduced tail latency and protected real-time processes in real-time applications. Experiments have shown, however, that current priority-based schedulers, while capable of protecting real-time processes from the background processes imposed by a large number of small-sized I/O operations, fail to provide protection when dealing with a small number of large-sized I/O operations. This type of load occurs during Linux's readahead process, which issues large asynchronous accesses due to improving performance of sequential read operations(by default, 128KiB in Linux 6.5) [10]. Therefore, this issue with current priority-based schedulers is significant problem, as it can arise in real-world systems.

This study proposes a method to split the background process's I/O operations, converting a small number of large-sized I/O operations into a large number of small-sized I/O operations. This can be achieved by using the max_sectors_kb parameter, a Linux configuration option that limits the maximum I/O size issued to a specific device [11]. However, using max_sectors_kb would affect not only the background process but also the real-time processes. To address this issue, we propose K2-split, a custom I/O scheduler that selectively splits only the I/O operations of background processes.

The experimental results demonstrate that this approach reduces the tail latencies (99th percentile) by a factor of 5 for reads and 3 for writes compared to existing priority-based schedulers and the 'none' scheduler at the expense of throughput. Additionally, compared to using max_sectors_kb, K2-split reduces tail latency by a factor of 3 for reads.

The contributions of this paper are as follows:

- We re-evaluate existing priority-based I/O schedulers with a focus on eMMC and conduct a comprehensive analysis. Through experiments, we demonstrate that existing priority-based schedulers fail to protect real-time processes from background processes under certain bandwidth load pat-

1 Graduate School of Informatics, Nagoya University
a) skoshiba@ertl.jp

terns.

- We propose K2-split, a priority-based I/O scheduler equipped with an I/O splitting mechanism. Experiments confirm that K2-split outperforms existing priority-based I/O schedulers, demonstrating that it is the optimal I/O scheduler for real-time systems equipped with eMMC. The source code for K2-split is available under an open-source license.*1

The rest of the paper is organized as follows: Section 2 explains the background knowledge on topics such as the I/O characteristics of flash-based storage and priority-based I/O scheduler such as K2 and mq-deadline. Section 3 show the problems of existing priority-based I/O scheduler. Section 4 explains our proposal that scheduler used I/O splitting mechanism. Section 5 show the Superiority of our proposal compared with existing priority-based I/O schedulers. Section 6 discusses related research, and Section 7 concludes with a summary of the findings and future challenges.

2. BACKGROUND

In this section, we introduce the characteristics of three representative flash-based storage devices: eMMC, UFS, and SSD. Subsequently, we explain why the I/O scheduler plays a critical role in performance improvement for real-time systems using eMMC, and provide an overview of the Linux block layer and existing priority-based I/O schedulers.

2.1 Flash-Based Storage

Flash-based storage has been widely adopted in many systems as a high-performance alternative to HDDs. While HDDs suffer from the bottleneck of mechanical seek time, flash storage achieves significantly shorter access times by reading and writing data electrically. Representative products of flash-based storage include eMMC, UFS (Universal Flash Storage), and SSDs.

eMMC is a compact and lightweight flash-based storage device widely used in embedded systems. It consists of NAND flash memory and a controller, with the latest eMMC 5.1 achieving transfer speeds of up to 600 MiB/s and a maximum capacity of 512 GB. On the other hand, UFS and SSDs are devices that have achieved greater capacity and performance compared to eMMC, making them promising candidates for adoption in future embedded systems.

Table 1 summarizes the characteristics of these three devices. A key aspect to note is the presence or absence of command queuing support. Unlike UFS and SSDs, eMMC does not support command queuing. In devices with command queuing, the internal device controller ultimately determines the order of access for commands stored in the queue. Consequently, even if an I/O scheduler prioritizes real-time processes, background processes may still take precedence within the device. This means that in such devices, the internal controller plays a central role, and many studies have focused on improving the internal controller to protect real-time processes [16, 17, 18, 19, 20, 21, 22].

In contrast, eMMC lacks both command queuing support and internal scheduling, meaning that device access is strictly executed in the order issued by the I/O scheduler. This makes the I/O scheduler a critical component for protecting real-time processes in eMMC, necessitating a comprehensive evaluation of I/O schedulers for this type of storage. This study is the first to conduct a comprehensive analysis of priority-based I/O schedulers focusing on eMMC.

*1 `https://github.com/ertlnagoya/k2-split`

Table 1: Comparison of flash-based storage

	eMMC	UFS	SSD
Max Throughput[MiB/s]	600	2,900	7,000
Max Capacity[GB]	512	1,000	10,000
Power consumption	more low	low	high
Support command queuing	No	Yes	Yes
Support parallelism	No	Yes	Yes

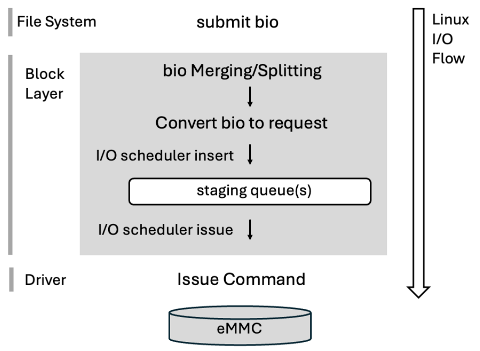


Fig. 1: I/O operation in Linux block layer

2.2 Linux Block Layer

As a key component of the Linux kernel where the I/O scheduler operates, we explain the main processes of the block layer in the Linux kernel (version 6.5.0 used in this study).

Figure 1 illustrates the main processes in the Linux block layer. First, the block layer begins processing when the file system submits a bio—a data structure containing information about the I/O operation (such as data transfer destination and memory page details)—to the block layer. When a bio arrives at the block layer, it attempts optimization by merging or splitting the bio as needed. Bio splitting at this stage follows the setting of max_sectors_kb, a kernel parameter. The bio is then converted into a request, an aggregated structure that the device driver can process. Once the bio is converted to a request, it is queued in the staging queue of the I/O scheduler according to the scheduler-specific insertion policy. Following this, the I/O scheduler retrieves requests from the staging queue according to its scheduling policy and issues them to the device driver. For eMMC, this issuance to the device driver is synchronized with the notification of device access completion.

Thus, the Linux block layer operates in conjunction with the I/O scheduler. Since the I/O scheduler determines the final processing order of requests for eMMC, the block layer functions as the most critical component for eMMC.

2.3 PRIORITY-BASED I/O SCHEDULER

Priority-based scheduling is a long-established method for managing tasks and processes in a system and is widely used in real-time systems. This approach is no exception for I/O resources, and priority-based scheduling is the optimal solution for improving real-time performance in eMMC.

The block layer includes an I/O-specific priority system, distinct from task priority, known as ioprio. The ioprio system comprises three priority classes: the real-time class, best-effort class, and idle class. The real-time class has the highest priority, while the idle class has the lowest. Additionally, within the real-time and best-effort classes, priorities are further distinguished using priority numbers ranging from 0 to 7, with 0 indicating the highest priority.

The two existing priority-based schedulers discussed below perform priority-based scheduling based on the ioprio assigned to each request.

The K2 scheduler is one of the priority-based I/O schedulers implemented as a Linux kernel module [6]. Developed for NVMe SSDs, K2 successfully reduces the tail latency of real-time processes by prioritizing real-time requests while limiting the number of requests issued to the device. When using K2 with eMMC, the lack of command queuing in eMMC means that the number of requests that can be issued to the device is always limited to one. K2 features nine staging queues, of which eight are real-time queues and one is a best-effort queue. When a request arrives at the block layer, it is enqueued in a real-time queue if its ioprio class is real-time; otherwise, it is enqueued in the best-effort queue. For the real-time class, the specific queue among the eight real-time queues is selected based on the priority number. When K2 issues requests to the device driver, it starts with the highest-priority queue (queue with priority number 0 within the real-time class), and requests within the queue are processed in FIFO (First In, First Out) order. Thus, when used with eMMC, K2 functions as a simple and lightweight priority-based I/O scheduler.

MQ-DEADLINE is a simple scheduler designed for HDDs. This scheduler minimizes seek operations by sorting requests in the order of their sector addresses. However, to prevent starvation of requests accessing distant disk regions, a deadline is assigned to each request. Furthermore, priority support was added starting with Linux kernel version 5.9. While the primary goal of mq-deadline is not priority scheduling, it can also function as a priority-based I/O scheduler.

3. EVALUATION OF PRIORITY-BASED I/O SCHEDULER

In this section, we conduct a comprehensive evaluation of priority-based I/O schedulers, which were explained in the previous section, to assess their potential effectiveness in real-time performance in eMMC. To the best of our knowledge, this study is the first to analyze the latency characteristics and suitability of priority-based I/O schedulers for real-time workloads targeting eMMC.

3.1 Bandwidth load

To comprehensively investigate whether existing priority-based I/O schedulers are sufficient for use in real-time systems, we analyzed them using two types of bandwidth loads that could potentially compete with real-time processes at the block layer. At the block layer, there are two bandwidth load patterns that can negatively impact real-time processes:

Request Count-Induced Bandwidth Load : A large number of small-sized I/O requests simultaneously present can degrade the performance of real-time processes. A study analyzing I/O patterns from 18 common applications on a Nexus 5 smartphone with an internal eMMC found that 57.4% of I/O requests were single-page (4KiB) accesses in 15 of the 18 applications [15]. Thus, it is highly likely that the presence of a large number of small-sized I/O requests can have a negative impact on real-time processes in eMMC.

Block Size-Induced Bandwidth Load : A small number of large-sized I/O requests can also degrade the performance of real-time processes. In Linux, the readahead feature is designed to improve throughput during sequential reads. When the kernel determines that an application's access pattern is sequential read, it asynchronously accesses the device and pre-loads the data into the cache. This results in consistent cache hits for the application, significantly improving average performance(excluding Direct I/O). Since readahead accesses the device with a block size of 128KiB(this is the default value and can be configured), the load generated by such large-sized I/O requests can arise. Therefore, a small number of large-sized I/O requests can also be a potential source of bandwidth load.

Table 2: Experimental Environment

OS	Ubuntu 22.04
kernel	Linux kernel 6.5.0 custom
CPU	Intel Pentium Silver N5000(4core)
RAM	8GB
eMMC	FORESEE eMMC 5.1 NCEMAD9D
eMMC Capacity	64GB
eMMC interface	USB3.1

3.2 Evaluation Setup

The experimental environment is outlined in Table 2, and the storage device used is an eMMC 5.1 connected via USB 3.1. Additionally, the parameter nr_requests, which limits the number of requests in the block layer, is set to 256 (the default for eMMC is 2).

In this experiment, we simulate the two types of background process and the real-time process using the Flexible I/O Tester (FIO) version 3.28. The real-time process is configured with synchronous I/O and a block size of 64KiB. The background processes are set up to simulate the two types of bandwidth loads as described above.

Request Count-Induced Bandwidth Load : The block size is kept constant at 4KiB, while the number of outstanding requests is gradually increased to 1, 4, 8, 16, 32, 64, and 128.

Block Size-Induced Bandwidth Load : The number of outstanding requests is kept at 1, while the block size of the requests is gradually increased to 4, 16, 32, 64, 128, 256, and 512 KiB.

To ensure that the latency of drive access is reflected in the results for both the real-time process and the background process, FIO was configured to use Direct I/O, bypassing Linux's buffer cache. Using the Linux ionice command, the real-time process was set to the real-time class, and the background process was set to the idle class. These simulated accesses were comprehensively performed across all access types: random read, sequential read, random write, and sequential write, to analyze the behavior of priority-based schedulers. Additionally, all parameters for mq-deadline are left at their default settings. the real-time process issues about 32,000 I/O requests, which we believe is sufficient for discussing the worst-case scenario (99th percentile latency).

3.3 Evaluation Results : Request Count

Figure 2 shows the experimental results of applying the Request Count-Induced Bandwidth Load to the real-time process. The x-axis represents the number of outstanding requests (with a block size of 4KiB) increasing from 1 to 128, while the y-axis shows the 99th percentile latency of the real-time process. In the random read operation in Figure 2a, it can be seen that the 99th percentile latency of the real-time process under ”none” deteriorates as the number of outstanding requests increases. The latency worsens approximately 40-fold, from 2.2ms with 1 request to 80.2ms with 128 requests. In contrast, both K2 and mq-deadline prevent the latency degradation of the real-time process despite the increase in the number of outstanding requests. Please note that the results for K2 and mq-deadline are displayed overlapping. This is because, with priority-based scheduling, the real-time process is always issued with priority, regardless of the number of background processes. A similar trend can be observed in the sequential read operation in Figure 2b, where the latency under ”none” worsens approximately 37-fold, from 2.1ms with 1 request to 79.1ms with 128 requests. Both K2 and mq-deadline effectively suppress latency degradation in the same way as in the random read case.

On the other hand, Figures 2c and 2d show the results of the same experiment for write operations. In write operations, the behavior is different from that of reads. Similar to the read case, ”none” shows a worsening of the 99th percentile latency of the real-time process as the number of outstanding requests increases, with the latency in random writes worsening approximately 29-fold, from 6.1ms with 1 request to 179.3ms with 128 requests. While mq-deadline was able to prevent latency degradation in read operations similarly to K2, it shows worsening latency in write operations as the number of outstanding requests increases. As explained in the previous section, mq-deadline is not a fully priority-based scheduler; its primary objectives are to reduce seek time and ensure request deadlines. This result indicates that even if high-priority requests exist, lower-priority requests may be prioritized and issued to the device. On the other hand, since the K2 scheduler operates as a fully priority-based scheduler, it can separate the effects of outstanding requests numbers and suppress the degradation of the 99th percentile latency of the real-time process, even during write operations.

3.4 Evaluation Results : Block Size

Figure 3 shows the experimental results when the Block Size-Induced Bandwidth Load was applied. The x-axis represents the block size of the background process (with 1 outstanding request), increasing from 4KiB to 512KiB, and the y-axis shows the 99th percentile latency of the real-time process. The total bandwidth load is adjusted to match the experiment shown in Figure 2, where the maximum load is 4KiB $\times$ 128 = 512KiB in Figure 2, and 512KiB $\times$ 1 = 512KiB in Figure 3. Figure 3a shows the results of random read operations. Under ”none,” the 99th percentile latency worsens from 3.8ms with a 4KiB block size to 8.2ms with a 512KiB block size, a roughly 2-fold increase. Indicating that, the 99th percentile latency of the real-time process cannot be prevented from degrading as the background process block size increases. Similarly, both K2 and mq-deadline fail to mitigate the impact on the real-time process due to the increasing block size of the background process. For K2, the 99th percentile latency worsens by about 5 times as the background process block size increases from 4KiB to 512KiB. Figures 2 and 3 demonstrate that, despite the same total bandwidth load, existing priority-based schedulers can protect real-time processes from load due to an increase in the number of outstanding requests, but they fail to protect them from load caused by an increase in the block size of background processes.

In cases where there are many small I/O requests, as shown in Figure 2, priority-based scheduling ensures that the real-time process is prioritized, so it does not have to wait for a total of 512KiB worth of device accesses consisting of 128 requests of 4KiB each (under maximum bandwidth load conditions). However, this is not the case when there is a single large-sized request, as shown in Figure 3. Since eMMC lacks preemption functionality and parallel access capability, once a device access is issued, the next access is forced to wait in the block layer for the duration of that device access. This means that even with priority-based scheduling, the real-time process may, in the worst case, be delayed by the duration of the background process’s device access, leading to degraded real-time performance. As a result, the 99th percentile latency of the real-time process increases along with the block size of the background process.

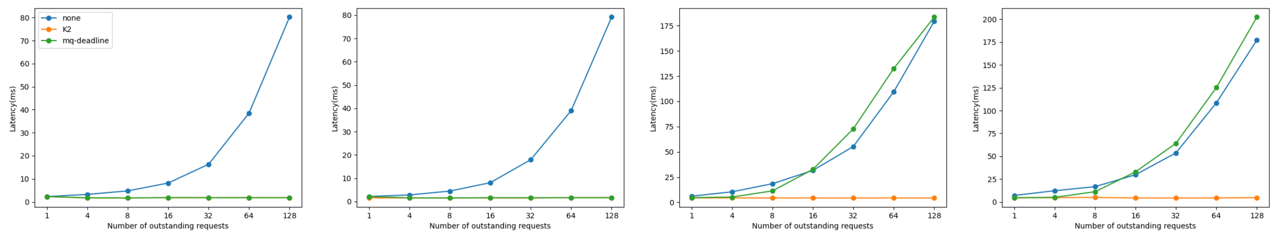
As seen in Figures 3b, 3c, and 3c, no schedulers can protect the real-time process from the increased block size of background processes across all access types, including write operations.

3.5 summary

In summary, with the Request Count-Induced Bandwidth Load, K2 was able to protect the real-time process from the impact of an increasing number of outstanding requests across all access types through priority-based scheduling. However, since priority-based scheduling is not the primary objective of mq-deadline, the 99th percentile latency of the real-time process increased along with the number of outstanding requests during write operations, similar to ”none”. With the Block Size-Induced Bandwidth Load, even fully priority-based schedulers like K2 saw an increase in the 99th percentile latency of the real-time process as the block size of the background process increased. In conclusion, current priority-based schedulers can protect real-time processes from the load caused by a large number of small-sized I/O requests, but they cannot protect real-time processes from the load caused by a small number of large-sized I/O requests.

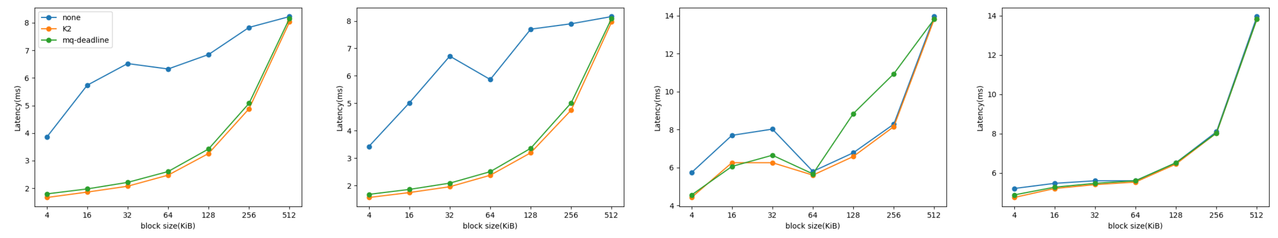
4. The I/O SPLITTING MECHANISM

In the previous section, we found that existing priority-based



(a) random read (b) sequential read (c) random write (d) sequential write

Fig. 2: 99th percentile latency of Real-Time process with Request Count-Induced Bandwidth Load



(a) random read (b) sequential read (c) random write (d) sequential write

Fig. 3: 99th percentile latency of Real-Time process with Block Size-Induced Bandwidth Load

I/O schedulers, such as K2, can protect real-time processes from bandwidth load caused by a large number of small-sized I/O requests but cannot protect them from the bandwidth load caused by a small number of large-sized I/O requests. Therefore, in this section, we explain our proposal, which uses an I/O splitting mechanism to convert a small number of large-sized I/O requests into a large number of small-sized I/O requests, thereby protecting real-time processes even from the bandwidth load caused by a small number of large-sized I/O requests.

4.1 max_sectors_kb

max_sectors_kb is a critical Linux configuration parameter that allows setting the maximum block size for requests issued to the device. This parameter can be used to split I/O. When a bio with a block size greater than the value set in max_sectors_kb arrives at the block layer, the kernel splits the bio into multiple bios, each with a block size equal to the value set in max_sectors_kb. By appropriately configuring max_sectors_kb, it is possible to convert a small number of large-sized I/O requests into a large number of small-sized I/O requests. As mentioned in Section 2.2, bio splitting using max_sectors_kb is performed before the I/O scheduler operates.

However, there is a significant challenge in using max_sectors_kb for I/O splitting mechanism: it applies to all storage accesses. As a result, not only low-priority background processes but also high-priority real-time processes are subject to splitting. This is undesirable because splitting decreases throughput and increases total latency, which can negatively impact real-time processes.

4.2 K2-split : I/O Scheduler with Splitting Mechanism

We propose an I/O scheduler equipped with an I/O splitting mechanism. The main operations of an I/O scheduler are divided into two main tasks: inserting requests that arrive at the block layer into the staging queue, and issuing requests from the staging queue to the device driver. Our proposed I/O scheduler performs I/O splitting for non-real-time processes just before insertion into the staging queue. By employing the same structure for the staging queue and the same method for issuing requests from the staging queue to the device driver as in the K2 scheduler, we have implemented a priority-based scheduler equipped with an I/O splitting mechanism.

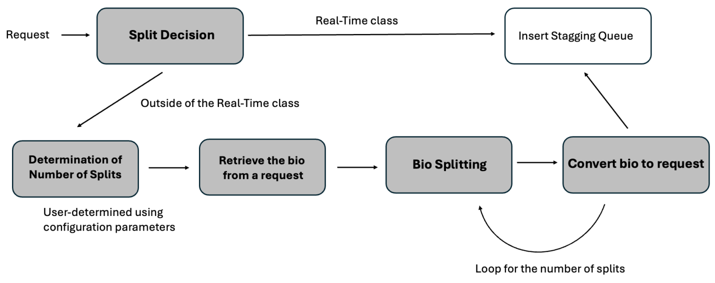

Request
Split Decision
Real-Time class
Insert Stagging Queue
Outside of the Real-Time class
Determination of Number of Splits
User-determined using configuration parameters
Retrieve the bio from a request
Bio Splitting
Convert bio to request
Loop for the number of splits

Fig. 4: Overview of the splitting process in K2-split

The detailed implementation of the splitting mechanism is illustrated in Figure 4. If a request that arrives at the staging queue belongs to the real-time class, it is directly inserted into the staging queue. Otherwise, if the request belongs to any other priority class, the request is split into the specified size before being inserted into the staging queue. Additionally, we have implemented a configuration parameter that allows setting the maximum issuance size, and the number of splits is determined based on the maximum issuance size specified by the user. The actual splitting process is implemented based on the bio splitting function within the Linux kernel. At the point when the I/O arrives at the staging queue, it has already been converted from a bio structure into a request structure. In the added splitting mechanism, the bio is first extracted from the request. Then, using the bio splitting function employed within the kernel, the bio is split into the specified size, converted back into a request, and inserted into the staging queue. Through this process, an I/O scheduler equipped with an I/O splitting mechanism is implemented.

We implemented the K2 scheduler with the above-described I/O splitting mechanism as "K2-split" in the form of a Linux kernel module. The detailed implementation of K2-split is available on GitHub. It is important to note that to implement the splitting mechanism as a kernel module, we modified a part of the source code of Linux kernel version 6.5.0. Specifically, we added an EXPORT_SYMBOL declaration to the kernel functions blk_mq_get_new_requests, blk_mq_bio_to_request, and bio_set_ioprio, which are used in the splitting mechanism, and then rebuilt the kernel.

5. EVALUATION OF SPLITTING MECHANISM

In this section, we evaluate the I/O splitting method described in the previous section. The experimental environment is the same as in Table 2, and the experiments were conducted using an eMMC connected via USB 3.1.

5.1 Comparison of K2 and K2 with a splitting mechanism

In the initial evaluation, we assess the system using the Block Size-Induced Bandwidth Load, which K2 could not handle effectively to maintain real-time performance, as discussed in Section 3.4. The experimental method follows that of Section 3.4, where FIO is used to run the real-time process and background process concurrently. The real-time process issues synchronous I/O with a block size of 64KiB, while the background process keeps the number of outstanding requests at 1 and increases the block size from 4KiB to 512KiB. Additionally, the max_sectors_kb value is set to 4KiB, and the parameter representing the maximum issuance size for K2-split is also set to 4KiB.

Figure 5a shows the results for random reads, with the x-axis representing the block size of the background process and the y-axis showing the 99th percentile latency of the real-time process. While the 99th percentile latency of the real-time process increases with the background process block size under K2, K2-split maintains a consistent latency regardless of the background process size. At a block size of 512KiB, the 99th percentile latency of the real-time process under K2 is 8.0ms, whereas K2-split achieves a significantly lower latency of 1.6ms, reducing latency by approximately 5 times. This is achieved by the splitting mechanism described in Section 4, which ensures that the I/O of background processes is always split into 4KiB block size. As a result, the real-time process only has to wait for 4KiB block size, regardless of block size that background process issued, allowing K2-split to protect real-time processes from the increasing block size of background processes. Although the splitting with max_sectors_kb maintains a consistent 99th percentile latency regardless of the block size of the background process, the latency remains higher compared to K2-split. This increased latency is due to the splitting of the real-time process as well. Since the max_sectors_kb value is set to 4KiB while the real-time process uses a block size of 64KiB, the real-time process is always split into 16 parts, resulting in approximately three times higher latency compared to K2-split.

Figure 5b shows the results for sequential reads, where K2-split successfully protects the real-time process from the increasing block size of background processes, achieving approximately a 4.9-fold reduction in latency compared to K2 with a background process block size of 512KiB. Figure 5c and 5d show the results for write operations, where K2-split similarly protects the real-time process from the increasing block size of background processes, just as it does with read operations. For random writes, K2-split achieves a 3.1-fold reduction in 99th percentile latency compared to K2, and for sequential reads, it achieves a 2.8-fold reduction in latency.

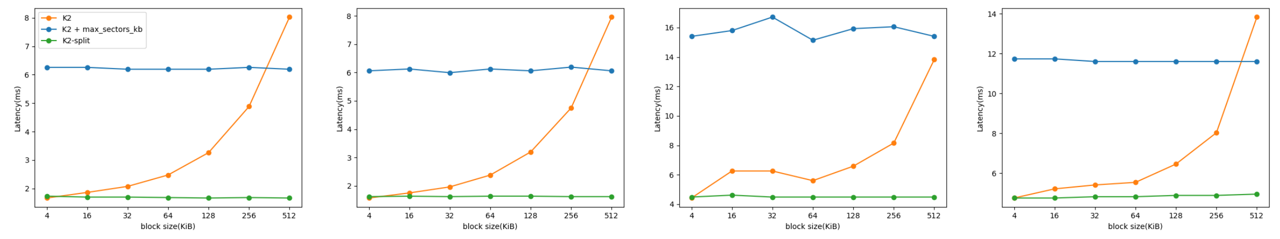
Figure 6 shows the complementary cumulative distribution function (CCDF) of the real-time process latency when the background process issues 512KiB block size requests. The x-axis represents latency in milliseconds, and the y-axis shows the probability of the latency being equal to or greater than the corresponding x value, indicating the predictability (real-time performance) of the latency. From the perspective of real-time performance, K2-split demonstrates significantly better latency predictability across all access types compared to K2 and K2+max_sectors_kb. Please note that for values below the 99.99% threshold (10^{-4} on the y-axis), the observation frequency is low and unstable, reducing the reliability of the data.

In summary, K2-split can minimize the tail latency of real-time processes regardless of the block size of competing background requests. Additionally, priority-based scheduling minimizes the tail latency of real-time processes regardless of the number of outstanding requests from background processes (as explained in Section 3.3). For these reasons, K2-split can be considered the optimal I/O scheduler for real-time systems equipped with eMMC.

5.2 Comparison of K2-split and max_sectors_kb

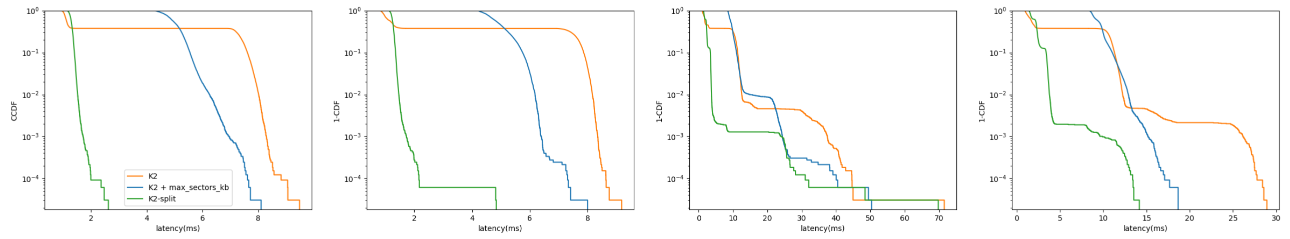
In the previous section, we confirmed that the I/O splitting mechanisms achieved superior performance in terms of 99th percentile latency compared to the conventional K2 scheduler. In this section, we conduct a comparative evaluation of the two splitting methods: I/O splitting with max_sectors_kb and I/O splitting with the K2-split scheduler. As in previous experiments, we use FIO to run the real-time process and background process concurrently. The background process generates 512KiB block size requests, and the real-time process is evaluated with four block sizes: 4KiB, 64KiB, 128KiB, and 512KiB. Both the background and real-time processes use synchronous direct I/O with a single process running concurrently. The maximum issuance size for both max_sectors_kb and K2-split is varied from 512KiB (no splitting of the background process) to 4KiB (128 splits of the background process), and their behaviors are compared.

Figure 7 shows the evaluation results for different block sizes of the real-time process. The x-axis represents the maximum issuance size (number of splits), and the y-axis represents the 99th percentile latency of the real-time process. As shown in In Figure 7a, when the real-time process uses the minimum block size



(a) random read (b) sequential read (c) random write (d) sequential write

Fig. 5: 99th percentile latency of real-time process with Block Size-Induced Bandwidth Load



(a) random read (b) sequential read (c) random write (d) sequential write

Fig. 6: Complementary Cumulative Distribution Function (CCDF) of the real-time process latency with a background process block size of 512KiB

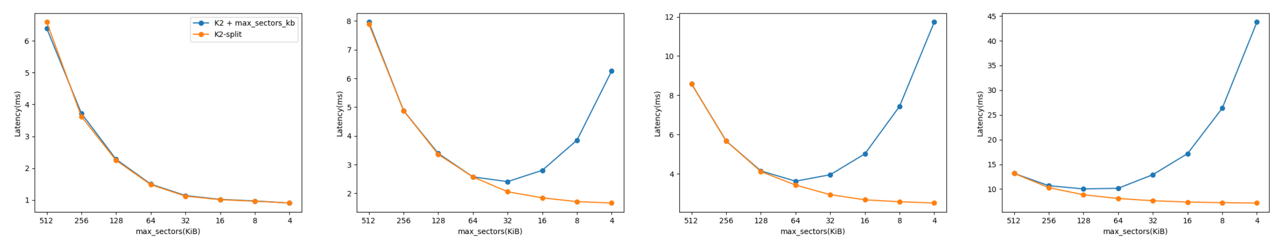
(4KiB), even with splitting using max_sectors_kb, the real-time process is not split, and only the background process is split. Therefore, both max_sectors_kb and K2-split achieve the same results. For both splitting methods, the 99th percentile latency of the real-time process decreases as the maximum issuance size is reduced. With a maximum issuance size of 512KiB (no splitting), the 99th percentile latency with K2-split is 6.5ms, while with a 4KiB maximum issuance size (128 splits), the latency is reduced to 0.9ms—a 7.2x improvement.

Figure 7b shows the results when the real-time process issues requests with a block size of 64KiB. Unlike in Figure 7a, with max_sectors_kb, setting the maximum issuance size smaller than 64KiB results in the real-time process being split along with the background process, causing the 99th percentile latency to worsen. For example, with a 4KiB maximum issuance size (where the real-time process is split into 16 parts), the 99th percentile latency with max_sectors_kb is 6.2ms, which is 3.8x worse than the 1.6ms latency achieved by K2-split. Figures 7c and 7d show similar trends, with the 99th percentile latency for a 4KiB maximum issuance size worsening to 43.7ms with max_sectors_kb, compared to 7.1ms with K2-split—an approximately 6.1x difference. These results demonstrate that splitting the real-time process with max_sectors_kb significantly increases 99th percentile latency.

Next, we discuss the optimal settings for the maximum issuance size for each splitting method. As shown in Figure 7, the optimal setting for K2-split is always the minimum value of 4KiB, regardless of the block size of the real-time process. In contrast, the optimal setting for max_sectors_kb varies depending on the block size of the real-time process. For 4KiB, the optimal setting is 4KiB, the same as with K2-split. For 64KiB, the optimal setting is 32KiB, achieving a 99th percentile latency of 2.4ms; however, this is still 1.4x worse than the 1.6ms achieved by K2-split with a 4KiB setting. The optimal setting for 128KiB is 64KiB, and for 512KiB, it is 128KiB.

These results indicate that with max_sectors_kb, the optimal maximum issuance size depends on the block size of the real-time process. Therefore, in practical systems, users must identify the block size of high-priority applications and configure the maximum issuance size accordingly. However, identifying the block size of an application is challenging, and applications may not always issue requests with a single block size. For example, in the case of the CameraVideo application, it was found that while the maximum access size was 10,104KB, the average access size was 244KB—a nearly 50-fold difference [15]. Thus, determining the optimal settings for max_sectors_kb becomes difficult based on the block size and issuance characteristics of the real-time process, making it impractical to use max_sectors_kb as a splitting mechanism in real-world real-time systems. On the other hand, with K2-split, simply setting the maximum issuance size to the minimum value of 4KiB consistently achieves the lowest latency, regardless of the block size or issuance characteristics of the real-time process. Moreover, K2-split guarantees that the only factor affecting the latency of the real-time process is waiting for 4KiB block sized device access of background processes. This provides higher latency predictability (real-time performance) compared to max_sectors_kb and other schedulers, making K2-split a more suitable choice for practical real-time systems.

Next, we compare each splitting method with K2 for total throughput. Table 3 presents the total throughput and 99th percentile latency for all access types with K2, K2+max_sectors_kb, and K2-split (where max=4KiB indicates the maximum issuance size is set to 4KiB). Block size of background process is 512KiB. Across all access types, K2-split records lower 99th percentile



(a) RT-4KiB (b) RT-64KiB (c) RT-128KiB (d) RT-512KiB

Fig. 7: 99th percentile latency for each maximum issuance size according to the block size of each real-time process. The max_sectors on the x-axis represents the configured maximum issuance size for each splitting method.

Table 3: Total Throughput [MiB/s] (at real-time process 99th percentile [ms])

	read				**write**			
	random		**sequential**		**random**		**sequential**	
	RT-4KiB	**RT-64KiB**	**RT-4KiB**	**RT-64KiB**	**RT-4KiB**	**RT-64KiB**	**RT-4KiB**	**RT-64KiB**
K2	81.02 (16.32)	75.63 (3.26)	85.93 (6.65)	79.24 (3.19)	51.98 (10.16)	48.04 (6.59)	55.49 (10.16)	51.83 (6.46)
K2 + max_sectors_kb(max=64KiB)	55.50 (1.50)	71.47 (2.61)	58.36 (1.42)	70.53 (2.61)	31.28 (4.11)	45.26 (5.21)	32.32 (4.08)	44.32 (5.21)
K2 + max_sectors_kb(max=16KiB)	28.20 (0.99)	51.97 (2.77)	30.37 (0.93)	53.93 (2.67)	13.65 (1.94)	27.55 (6.91)	14.26 (1.94)	24.63 (7.18)
K2 + max_sectors_kb(max=4KiB)	17.08 (0.85)	22.52 (6.26)	16.34 (0.84)	22.60 (6.19)	8.79 (1.25)	12.30 (14.75)	8.96 (1.24)	12.12 (12.91)
K2-split(max=64KiB)	55.55 (1.50)	73.12 (2.38)	58.84 (1.42)	74.84 (2.34)	31.79 (4.11)	44.61 (5.41)	31.92 (4.11)	47.85 (5.08)
K2-split(max=14KiB)	28.20 (1.00)	72.30 (1.70)	30.43 (0.93)	74.01 (1.63)	13.67 (1.94)	36.88 (4.36)	14.03 (1.94)	38.60 (4.36)
K2-split(max=4KiB)	17.08 (0.85)	62.05 (1.53)	16.40 (0.83)	65.53 (1.47)	8.67 (1.25)	35.12 (3.72)	8.96 (1.24)	35.16 (3.78)

latency compared to K2, and the 99th percentile latency decreases as the maximum issuance size is reduced. However, the total throughput deteriorates with both K2+max_sectors_kb and K2-split compared to K2. For the real-time process with a block size of 64KiB, with max=4KiB, the throughput with K2+max_sectors_kb drops to 22.52 MiB/s, which is approximately 26% of the throughput achieved by K2. In contrast, K2-split maintains a throughput of 62.05 MiB/s, approximately 72% of K2's performance. With K2-split, reducing the maximum issuance size improves the 99th percentile latency across all access types, but at the cost of degraded throughput. Therefore, it is essential for users to determine the optimal maximum issuance size based on the importance of throughput versus latency in their specific use case.

5.3 Application Benchmark

Thus far, we have generated storage load and conducted evaluations using simulated workloads with fio. In the final evaluation, however, we use the MySQL benchmark from sysbench to evaluate the I/O splitting mechanism with storage access patterns closer to those of real applications. We use sysbench version 1.0.20 in the OLTP read-only configuration. This configuration emulates a database query workload using the MySQL InnoDB engine, performing random reads and small amounts of writes for transaction logs.

In the experiment, sysbench was executed as a real-time process, while two Unix dd processes with a block size of 512KiB were run concurrently as background processes. Both dd processes were configured to use Direct I/O. The maximum issuance size for both max_sectors_kb and K2-split was set to 4KiB.

Figure 8 compares the output of sysbench under different schedulers when the dd processes performed read-only operations. The length of the bars indicate the completion latencies of database queries, where each query accesses the storage multiple times. It shows that schedulers with the I/O splitting mechanism (K2+max_sectors_kb and K2-split) achieved lower latency across all metrics—minimum, median, 95th percentile latency, and maximum—compared to K2 and none. Additionally, when comparing max_sectors_kb and K2-split, K2-split achieved lower latency across all metrics. This is because, unlike max_sectors_kb, K2-split avoids splitting the real-time process, which helps reduce latency further.

Figure 9 presents the sysbench output when the dd processes performed write operations. Similar to the read scenario, K2-split achieved the lowest latency among all schedulers.

These results demonstrate that the I/O splitting mechanism provides significant benefits not only in macro benchmarks but also under complex database workloads. Furthermore, the K2-split achieves lower latency compared to max_sectors_kb-based splitting. These results support our assertion from the previous section that max_sectors_kb is challenging to use in real-world applications where the block size of real-time processes is not known, and that K2-split is more suitable for practical application use.

6. RELATED WORK

There are many studies on timing constraints in flash-based storage that are related to our research. We classify these existing studies into two groups: those that support timing constraints by improving the internal controller of SSDs and UFS, and those that support timing constraints through the I/O scheduler at the block layer.

6.1 Improvements to the internal controller

As explained in Section 2, among the three representative flash-based storage devices, UFS and SSD have an internal controller called the Flash Translation Layer (FTL). In SSDs and

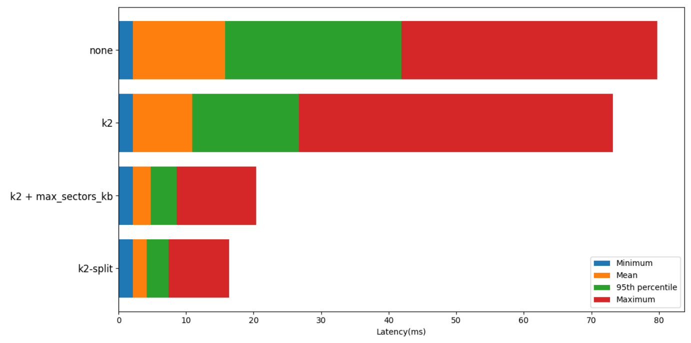


Fig. 8: Event processing times of the Sysbench SQL benchmark for different I/O schedulers with a read background load.

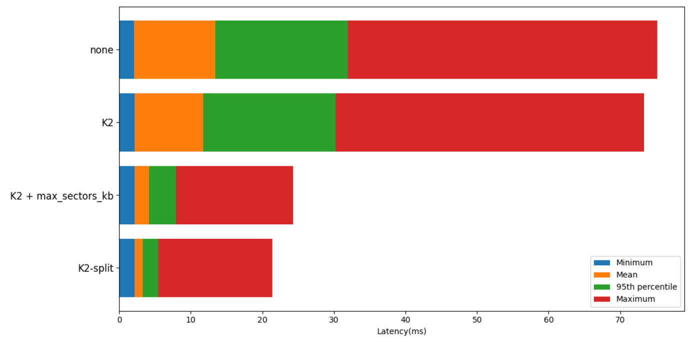


Fig. 9: Event processing times of the Sysbench SQL benchmark for different I/O schedulers with a write background load.

UFS, the FTL is the key to managing timing constraints, and there have been several studies focused on improving the FTL to support such constraints.

PaRT-FTL takes an approach that leverages internal parallelism [16]. By storing read operations and write operations (including garbage collection) on separate chips, it prevents read operations from being delayed by write operations.

RTFTL protects other operations from garbage collection and improves worst-case scenarios [17]. Specifically, it divides the garbage collection process into four steps: block selection, reading valid data, writing data to another block, and finally erasing the block. These steps are executed when normal read/write operations are completed, thereby protecting normal operations from being interrupted by garbage collection.

While the previous two methods focus on protecting processes from write operations, Janus-FTL improves the predictability of write operations themselves [18]. Janus-FTL changes the storage area depending on the type of write request: frequently accessed data is stored in a page-mapping area, while less frequently accessed data is stored in a block-mapping area. This improves the predictability of write operations.

6.2 Improvements to I/O scheduler on Linux block layer

Several I/O schedulers have been developed for real-time systems.

Kim et al. introduced the Completely Fair Scheduler (CFS), a CPU scheduling technique, into the I/O layer [8]. They introduced priority queues between the software queues, which exist for each process in the block layer, and the hardware queues, which exist for each device. There are as many priority queues as there are process priorities, and each request is inserted into a priority queue according to the process's priority. Each priority queue is assigned tokens, and requests are issued to the hardware queue based on the number of tokens.

Kyber, an I/O scheduler standard in Linux, was developed for SSDs [14]. categorizes requests into three categories: read, synchronous write, and others. Each category is assigned a specific number of tokens, which represent the number of requests allowed within the device. By properly configuring these tokens, Kyber can control latency for each category. As a result, Kyber successfully protects read requests from being delayed by write requests.

As demonstrated, there are several studies aimed at improving the real-time performance of flash-based storage. However, no existing research addresses the degradation of real-time performance in real-time processes caused by large-sized requests, which is related to the lack of preemption in eMMC. I/O scheduler with splitting mechanism that proposed in this study is the first study to tackle this issue.

7. CONCLUSION

This study evaluated two existing priority-based I/O schedulers targeting eMMC. The results revealed that neither scheduler could protect real-time processes from bandwidth loads caused by a small number of large-sized I/O requests, leading to a significant deterioration in 99th-percentile latency. To address this issue, we proposed K2-split, an I/O scheduler equipped with an I/O splitting mechanism that divides large-sized I/O requests into multiple smaller-sized I/O requests.

K2-split successfully protected real-time processes from bandwidth loads caused by large-sized I/O requests, which existing priority-based schedulers could not handle, achieving up to a 5x reduction in latency compared to K2. Furthermore, K2-split provided superior latency predictability compared to the splitting method using max_sectors_kb, making it more suitable for real-time systems, as confirmed by our experiments.

Future directions of this research are as follows: (1) Verifying the effectiveness of the proposed method not only for eMMC but also for the latest NVMe SSDs and UFS. Given their higher performance compared to eMMC, NVMe SSDs and UFS may become the first choice for real-time embedded systems in the future. (2) Introduction of a dynamic splitting mechanism. While K2-split achieved significantly lower latency for real-time processes compared to K2, it resulted in a reduction in throughput. Currently, the split count is determined statically, and background process I/O is always split regardless of the presence of real-time processes. It is necessary to consider a mechanism that dynamically determines the use and number of splits based on the storage access cycle of the real-time process and the performance degradation of background processes. This approach could maintain low latency for real-time processes while minimizing the performance degradation of background processes.

References

[1] B. Jun and D. Shin:Workload-aware budget compensation scheduling for NVMe solid state drives, in Proceedings of the 2015 Non-Volatile Memory System and Applications Symposium (NVMSA), Hong Kong, China: IEEE, Aug. 2015, pp. 19–24.

[2] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh.:Parameter-aware I/O management for solid state disks (SSDs), IEEE Transactions on Computers, vol. 61, no. 5, pp. 636–649, May 2012.

[3] A. Tavakkol, M. Sadrosadati, S. Ghose, J. S. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu.:FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives, in Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA: IEEE, Jun. 2018, pp. 397–410.

[4] S. Park and K. Shen.:FIOS: A fair, efficient flash I/O scheduler, in Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST), San Jose, CA, USA: USENIX, Feb. 2012, pp. 155–170.

[5] 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.

[6] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig.:K2: Work-Constraining Scheduling of NVMe-Attached Storage, In Proceedings of 2019 IEEE Real-Time Systems Symposium (RTSS '19), pages 56–68, 2019.

[7] F. Blagojevic, C. Guyot, Q. Wang, T. Tsai, R. Mateescu and Z. Bandic.:Priority IO scheduling in the cloud, Proc. USENIX Conf. Hot Topics Cloud Comput., pp. 1-6, 2013-Jun.

[8] Kyusik Kim, Seungkyu Hong, and Taeseok Kim.:Supporting the priorities in the multi-queue block I/O layer for NVMe SSDs, J. Semiconduct. Technol. Sci. 20 (02 2020), 55–62.

[9] MQ-Deadline Implementation,https://elixir.bootlin.com/linux/latest/source/block/mq-deadline.c,Accessed: 2024-10-02.

[10] F. Wu, H. Xi, and J. Li.:Linux readahead: less tricks for more, in Proceedings of the Linux Symposium, 2007.

[11] Queue sysfs,`https://www.kernel.org/doc/Documentation/block/queue-sysfs.txt`, Accessed: 2024-10-02.

[12] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak.:Do we still need io schedulers for low-latency disks?, In 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 23), HotStorage '23, 2023.

[13] Ren, Zebin, Krijn Doekemeijer, Nick Tehrany, and Animesh Trivedi.:BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era. ,In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, pp. 154-165. 2024.

[14] O. Sandoval, blk-mq: Introduce Kyber multiqueue I/O scheduler, Commit message for the Linux kernel, 2017. [Online]. Available: https://patchwork.kernel.org/patch/9672023 Accessed: 2024-10-02.

[15] D. Zhou, W. Pan, W. Wang and T. Xie.:I/O characteristics of smartphone applications and their implications for eMMC design, Proc. of IEEE IISWC, 2015.

[16] Missimer, Katherine, and Richard West.:Partitioned real-time NAND flash storage.,2018 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2018.

[17] 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.

[18] Lee, Jongmin, et al.:Real-time flash memory storage with Janus-FTL, Proceedings of the 27th Annual ACM Symposium on Applied Computing. 2012.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] eBPF Documentation. `https://ebpf.io/what-is-ebpf/`. Accessed: 2024-10-02.

[24] bpftrace. `https://bpftrace.org/`.Accessed: 2024-10-02.

Shinnosuke Koshiba graduated from the College of Science and Engineering, Kanazawa University, with a major in Electronics, Information, and Communication in 2023. He is currently enrolled in the Master's Program at the Graduate School of Informatics, Nagoya University. His research focuses on technologies to improve the real-time performance of storage devices.

Yutaka Matsubara is an Associate Professor at the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from Nagoya University in 2011. From 2009 to 2018, he was a Researcher, and then an Assistant Professor at the Center of Embedded Computing Systems(NCES), Nagoya university. His research interests include real-time operating systems, real-time scheduling theory, system safety and security for embedded systems, dependability of System of Systems (SoS). He is a member of IEEE, IEICE and JSAE.

Hiroaki Takada is a professor at Institutes of Innovation for Future Society, Nagoya University. He is also a professor and the Executive Director of the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from University of Tokyo in 1996. He was a Research Associate at University of Tokyo from 1989 to 1997, and was a Lecturer and then an Associate Professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and automotive embedded system. He is a fellow of JSSST, and is a member of IEEE, ACM, IEICE, and JSAE.