IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 31, 2020, accepted January 10, 2021, date of publication January 20, 2021, date of current version January 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3053087

Context-Aware File I/O Management System for Mobile Devices

JAEHWAN LEE, SANGHYUCK NAM, SUHWAN KWAK, AND SANGOH PARK, (Member, IEEE)
School of Software, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Sangoh Park (sopark@cau.ac.kr)

This work was supported in part by the Chung-Ang University Graduate Research Scholarship, in 2019, and in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant 2020R1A2C1005265.

ABSTRACT Mobile devices such as smartphones and tablets have become widespread, and studies are being conducted to improve the convenience of smartphone users. The I/O performance is considered an important factor affecting the quality of the smartphone user experience. Therefore, methods for retaining applications with a high launch frequency in the main memory, improving the I/O stack of the smartphone operating system, and a new file system for improving the I/O performance have been actively studied. However, there is no information sharing system between the I/O stacks in a smartphone operating system. Existing studies suffer from limitations in improving the file I/O performance because the upper- and lower-layer information of the I/O stack was not considered simultaneously. In this paper, we propose a context-aware file I/O management system (CAFIO) for analyzing context information collected from various layers of the I/O stack and sharing them between the layers. CAFIO collects smartphone application usage patterns and combines them with the lower-layer I/O information to improve the launch speed and I/O latency of an application. CAFIO exhibits average improvements of 26% in application launch time and 45% in cache hit ratio, as well as a read speed 63% faster than that of existing file I/O management systems.

INDEX TERMS Mobile environments, storage management, file systems management, storage hierarchies, context-aware, pattern recognition.

I. INTRODUCTION

With the development and widespread use of smartphones, users are spending more time using such devices. The share of smartphones in the total mobile phone market is 81% according to statistics from 2019 [1], and users use their smartphones an average of 3 hours and 14 minutes per day. In addition, users typically install dozens to hundreds of applications, and smartphone applications are equipped with more functionalities to improve user convenience. As a result, mobile applications have increased in size and require an increasing number of I/Os to execute an application. Therefore, as the numbers of features and application sizes increase, so does the number of resources that must be fetched from storage. Because it is important for the system to respond quickly to smartphone users, providing users with a fast system response through a high I/O performance is necessary. Thus, studies on optimization of various parts of the file I/O layer have been proposed.

Android smartphones, which account for a high share in the smartphone market, use a Linux kernel-based operating system and an ext4-based file system. However, the ext4 file system was developed for desktop systems, not mobile systems such as smartphones. Furthermore, the I/O stack structure and I/O scheduler of the Linux kernel used in Android do not consider patterns such as the frequency of application launches. They are suitable for a typical multitasking desktop environment with a fair allocation of the I/O time to a large number of applications. Unlike a desktop environment, a smartphone environment does not use a large number of applications concurrently, and the user can only interact with one application at a time. Therefore, the existing I/O scheduler is unsuitable for a smartphone environment. Moreover, ext4 was developed for use in hard disk drives, and it does not consider the characteristics of NAND flash memory used as a permanent smartphone storage [2], [7].

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

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

The number of applications installed on smartphones has gradually been increasing, although applications that are frequently used or that have long foreground times only account for a small portion of them [8]. According to these characteristics, studies have been proposed to analyze and keep frequently used applications in the main memory to improve the application launch time. Not all applications remain in the main memory owing to the limited resources available. In studies on improving the I/O stack to upgrade the performance of a permanent storage device [9], [11], an improved I/O scheduler and a new file system suitable for smartphone storage have been proposed. Because these studies only consider individual parts of the I/O stack, the improvement of the overall file I/O performance is limited.

In this paper, we propose the use of a context-aware file I/O management system (CAFIO) that analyzes the frequency of application launches of a smartphone and shares the analyzed information between the layers of the I/O stack. CAFIO improves the application launch, execution speed, and I/O latency through file caching, reorganized sequential write, file level readahead, and I/O scheduling priority configurations based on analyzed context information such as the launch frequency of an application. The remainder of this article is organized as follows. In section II, we examine the existing Linux kernel I/O stacks, file I/O managements, features, and limitations of the existing file system used in a smartphone environment. We then introduce the proposed file I/O management system CAFIO in Section III and describe the performance evaluation methods along with the results in comparison with the existing system in Section V. Finally, in section VI we provide some concluding remarks and directions for future research.

II. BACKGROUNDS

A. ANDROID AND THE STANDARD I/O SCHEDULER

Android is a collection of a mobile operating system, middlewares, and core applications for mobile platforms such as smartphones and tablets. It is based on a modified Linux kernel suited for Android's mobile environments. Android is composed of 5 representative layers; application layer, android framework and runtime layers for application executions, C/C++ library layer for system components, and the linux kernel layer. The Android's Linux kernel is responsible for the core parts of the operating system such as process management, memory management, networking, and device drivers. It operates mostly similar to the original Linux kernel, with additional functionalities such as energy management and swap operations specific for mobile platforms.

Android applications are written in Java or Kotlin, which are compiled into bytecode running on the Android runtime. The Android runtime converts the Android application bytecode into native instructions and executes it. Each Android application runs as a virtual machine instance of the Android runtime. When a user or a system service launches an application, it creates a new process for the application if the application is not running. The application process is a process of the Linux kernel, and generally, multiple processes can be created when one application is executed.

File I/Os requested by Android applications are delivered to the Linux kernel through system calls of the process. As shown in Fig. 1, file I/O requests sent to the Linux kernel are delivered to Virtual File System (VFS), actual file system implementation, page cache, block I/O scheduler, and block device driver in order. VFS is a layer that abstracts the actual file system implementation such that it allows applications to access different file systems in the same way. The page cache temporarily stores data for file I/Os into memory. The file system converts file I/O requests into block I/O requests and delivers them to the block I/O scheduler. The block I/O scheduler rearranges block I/O requests according to the algorithms (e.g. CFQ, deadline, NOOP) in consideration of performance and load balancing. There is no difference in core code between the block I/O scheduler implemented in Android's Linux kernel and the block I/O scheduler of the original Linux kernel. In addition, they both use the Completely Fair Queueing (CFQ) algorithm by default. CFQ manages block I/O for each process, and assigns a time slice to the block I/O queue for each process so that each process has the same block I/O processing opportunity. However, it is not suitable for mobile environment due to the nature of CFQ that processes I/O of the foreground/background process fairly. In a mobile environment, since the responsiveness of the application being used by the user is most important, I/O schemes taking this into account are necessary.

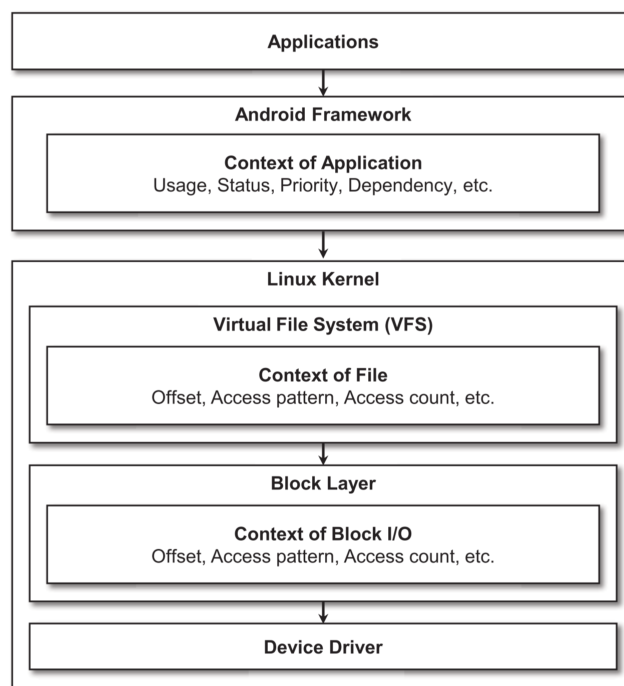

FIGURE 1. Context information from each layer of I/O stack.

B. RELATED STUDIES

Various studies have been conducted to improve the performance of a file I/O for application responsiveness and user experience in smartphones. To this end, I/O scheduling schemes for the Linux kernel [9], [14] were proposed. These studies mitigate the data transfer delay caused by incorrect dependencies of I/O operations [9] or prioritizing I/O requests from applications currently in the foreground [14].

In a study [15] on improving responsiveness through I/O optimization for smartphones, the authors analyzed the I/O characteristics of mobile applications and proposed a modified version of complete fair queueing-based block I/O scheduler which set more weights for read requests. Another study [16] on optimizing I/O stack of smartphones proposed techniques to minimize unnecessary journaling and random writes that can occur on smartphone file systems. The authors analyzed the I/O characteristics of SQLite, which takes part in Android and widely used among mobile applications.

It is reported in that more than 80 applications are installed on a smartphone on average [8], out of which less than 50 applications account for more than 50% of the total number of application launches, and less than 10 applications account for more than 50% of the total foreground time. In other words, only a small number of applications are frequently used. Studies [17], [19] have been proposed for the frequency analysis of open applications residing in the main memory.

However, the aforementioned studies have attempted to improve the performance only within one layer of the I/O stack. This acts as a limitation in improving the overall performance of the system I/O. A performance improvement by sharing context information between layers has yet to be considered. The context information that can be obtained from each layer of the I/O stack is shown in Fig. 1. Firstly, in the Android framework layer, context information such as an application launch and an application foreground and background execution status can be obtained. Secondly, Context information such as an access offset and access patterns for each file can be obtained in the virtual file system (VFS) layer. Lastly, in the block layer, context information such as a block device offset for applying an I/O, and an access pattern of a block device can be obtained. Information that can be acquired is only consumed within each layer. The limitation that the information cannot be transferred and utilized in other layers is due to the Linux kernel design, which is focused on modularization and abstraction of the I/O stack.

The formerly stated problems prevent the VFS layer from recognizing outside contexts such as whether the process currently accessing a file belongs to the foreground application or to a frequently used application. Moreover, it is not possible to receive information in the block layer regarding which file a block request belongs to, which process requested the block, or within which offset a block is located logically in the file. It becomes difficult to improve the I/O performance when it is difficult to share information between the layers of the I/O stack. For example, an improvement of the I/O performance with the block readahead is negligible for blocks related to infrequently used applications or files, even if the readahead scheme is proven effective.

A context-awareness of operating systems is introduced with the emerging of smartphones. Context-awareness in this kind of area includes utilizing smartphone sensing data, user's application usage data, and operating system generated data to optimize the system's resource management. The early concepts of context-awareness on operating systems were proposed [20], [22] for saving energy consumption of smartphones. The concept was applied on memory management [17] and I/O management [14] later. However, to the best of our knowledge, no research has yet been attempted to improve performance of I/O by combining context information from inside and outside the operating system and sharing them between I/O stacks.

In this paper, we propose a file I/O management system that can share the context information collected from each layer of the I/O stack. By utilizing the information sharing between the I/O stack layers, the smartphone application usage pattern, file I/O information, and block I/O information can be combined to enhance the performance of the file I/O.

III. CAFIO: CONTEXT-AWARE FILE I/O MANAGEMENT SYSTEM

In this paper, we propose CAFIO, a context-aware file I/O management system. The architecture of CAFIO is shown in Fig. 2. The context information manager collects context information generated from the Android Framework layer and the Linux kernel I/O stack. The context information analyzer analyzes the information collected from the context information manager and generates information that is utilized by CAFIO sub-modules. The virtual file system to file descriptor (VFS/FD) mapper allows the Android framework layer and the Linux kernel I/O stack layers to share the context information. The context-aware readahead conducts a file readahead based on the analyzed context information, and the context-aware cache queue determines the cache priority to increase cache hit ratio according to the context information. Context-aware defragger improves the read performance with the analyzed context information by sequentially storing frequently used small-sized files in a storage device. The completely fair queue with context-awareness (CFQ-CA) minimizes the response time of the application currently in the foreground by prioritizing I/O requests related to the application being used in the foreground.

A. CONTEXT INFORMATION MANAGER

The *FileContext* information collected for each application *uid* in the context information manager is shown in Table 1. This information is collected as indicated in Algorithm 1. Here, *collectFileOpen* is called by the Linux kernel's VFS when a file is opened, and this procedure increases the read count statistics by one. In addition, *collectFileRead* and *collectFileWrite* are called when file reads and writes are performed, respectively. When called, *collectFileRead*

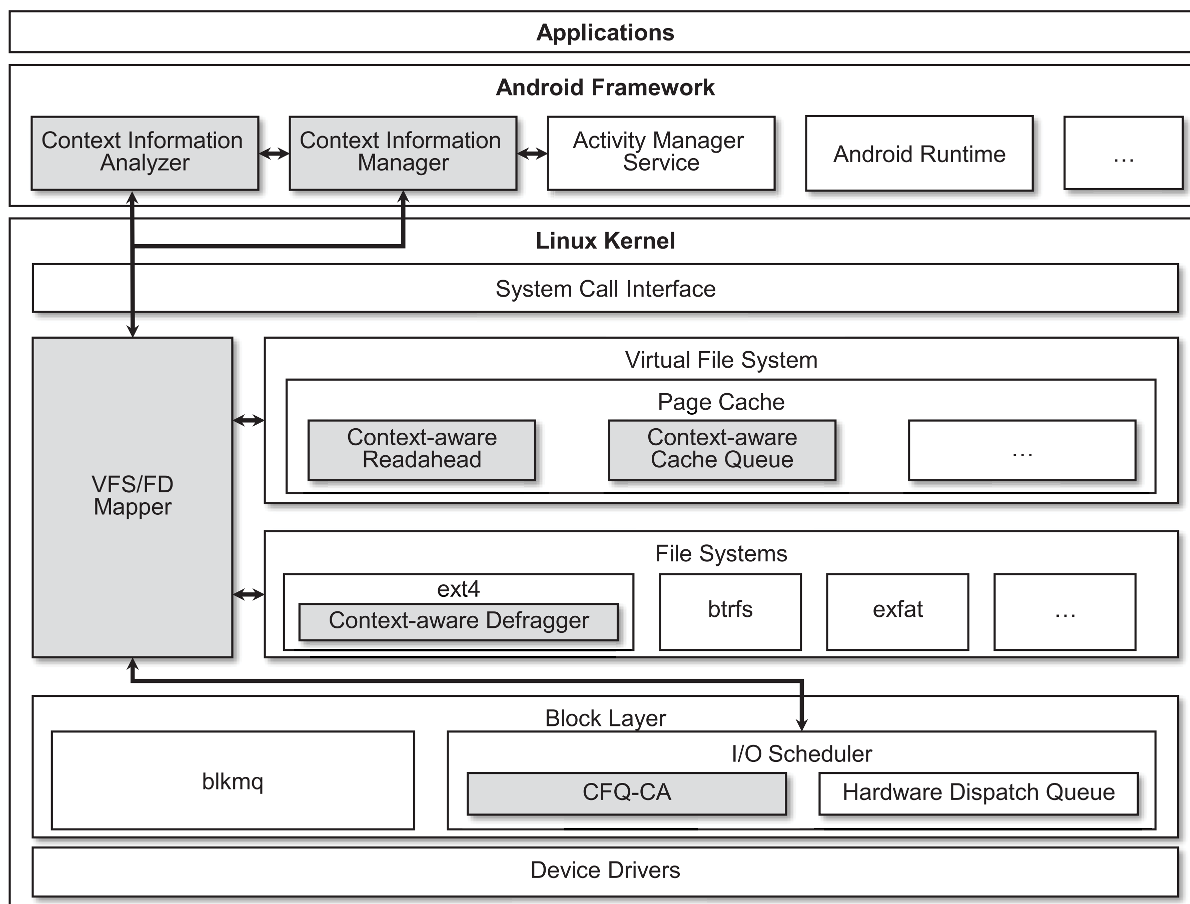

FIGURE 2. Architecture of context-aware file I/O management system.

TABLE 1. Definition of *FileContext*.

Field	Description
$inodeid$	the inode identifier of a file
$nread$	the number of reads on a file
$nwrite$	the number of writes on a file
$rcachehit$	the number of read cache hits on a file
$wcachehit$	the number of write cache hits on a file
$lastread$	last read timestamp on a file
$lastwrite$	last written timestamp on a file
$nextfc$	list of $FileContexts$ of read file right after this file

and *collectFileWrite* save the current timestamp to decide whether the *minwait* time has passed from the last read or write. *minwait* is the Linux kernel's I/O request timeout value in seconds, which is 5 [23]. Since reads and writes on a large file can be burst-called for a short period of time and are proportional to the size of the file, the *nread* and *nwrite* of infrequently used large files should not grow. Therefore, the *minwait* is defined to prevent *collectFileRead* and *collectFileWrite* from increasing *nread* and *nwrite*, respectively, if they are called before the *minwait* has passed. For CAFIO to adapt to dynamically changing file access patterns, the *nread* and *nwrite* are measured as weekly cumulative values.

B. CONTEXT INFORMATION ANALYZER

The context information analyzer analyzes information on cache priorities and sequential file reads using *FileContext* information that is collected from the context information manager. The context-aware readahead and the context-aware defragger use the sequential read information to optimize the file read performance. Cache priority information is utilized by the context-aware cache queue to improve the hit ratio of high-priority caches. When an application *uid* reads a file *inodeid* in the context information analyzer, the corresponding *FileContext* information is added to $history_{uid}$. The $history_{uid}$ is a list of *FileContexts* that manages the files accessed by the application *uid*. Files that are accessed by an application are classified into *hot*, *warm*, and *cold* files according to the *nread* and *nwrite*. The *nread* and *nwrite* represents the file access frequency. The upper-third of the most frequently read files are classified as read *hot* files, the middle third are read *warm*, and the lower third are read *cold*. The same applies to the write hotness of a file with *nwrite*.

Algorithm 1 Collecting File Usage Information of an Application

```
1: procedure collectFileOpen(inodeid)
2:     stat ← FileContext of inodeid
3:     stat.nread ++
4: end procedure
5: procedure collectFileRead(inodeid)
6:     stat ← FileContext of inodeid
7:     timestamp ← current time
8:     if timestamp − stat.lastread ≥ minwait then
9:         stat.nread ++
10:    end if
11:    stat.lastread ← timestamp
12: end procedure
13: procedure collectFileWrite(inodeid)
14:    stat ← FileContext of inodeid
15:    timestamp ← current time
16:    if timestamp − stat.lastwrite ≥ minwait then
17:        stat.nwrite ++
18:    end if
19:    stat.lastwrite ← timestamp
20: end procedure
```

Algorithm 2 When a File Is Read by an Application

```
1: procedure isTraceTarget(stat, timestamp)
2:     if the last element of history_uid is stat.inodeid then
3:         return false
4:     end if
5:     if timestamp − stat.lastread ≥ minwait then
6:         return false
7:     end if
8:     if sizeof stat.inodeid ≥ S_readahead then
9:         return false
10:    end if
11:    return true
12: end procedure
13: procedure traceFileRead(uid, inodeid)
14:    stat ← FileContext of inodeid
15:    timestamp ← current time
16:    if the file of inodeid is hot file then
17:        if isTraceTarget(stat, timestamp) is true then
18:            tr ← last element of history_uid
19:            append stat to tr.nextfc
20:        end if
21:    end if
22:    stat.lastread ← timestamp
23:    append stat to history_uid
24: end procedure
```

When an application process reads a file in the Linux kernel's virtual file system (VFS), the *onFileRead* operation of Algorithm 2 is executed by the context information analyzer. Since the file operation occurs when a user runs an application and the usage data needs to be applied, the proposed analyzer is executed online. File read pattern information is also collected for readahead tasks over multiple sequential files. If an application *uid* shows a sequential read pattern over multiple files, context-aware readahead preloads the file's contents into the page cache. The context information analyzer collects the file read patterns from (a) a *hot* file, (b) a file not most recently used, (c) a frequently accessed file, and (d) a file whose size is smaller than $S_{readahead}$. Condition (a) is required, as indicated in line 19, to reduce the data collection overhead. Condition (b) is required because of the possibility that contents of the most recently read file remain in the main memory. Lines 2–4 are executed to check condition (b). If a file *inodeid* is the last file read by the *uid*, the context information analyzer excludes the file from the collection target. Condition (c) is required because there is little benefit for a file that is accessed intermittently. Even if the file is read in advance, it is likely to be discarded without being used. Accordingly, lines 5–7 are executed to exclude the file that reads again after *minwait* from the last read. Finally, as represented in lines 8–10, condition (d) is for multiple file readaheads because the readahead size of the operating system is $S_{readahead}$. The maximum readahead size $S_{readahead}$ is 512 KB, which is the maximum I/O size of Android's Linux kernel. When the currently read file *inodeid* is determined as the information collection target for sequential reads, *FileContext* of the currently read file is appended to the *nextfc* list, as shown in lines 18 and 19; this *nextfc* belongs to the file that the application has most recently read. In other words, it is recorded in the last file read for which the current file *inodeid* is read next.

Using the updated $history_{uid}$, the cache priorities for each file are analyzed. The analyzed information is used in the context-aware cache queue to apply a priority-based file cache management policy. The file cache priority is determined as high, normal, or low according to Table 2. A file belongs to the high-priority cache group if it is determined to be a *readhot* file or *writehot* file. A file belongs to the normal priority cache group if it is determined to be a *readwarm* file or a *writewarm* file. The other files belong to the low-priority cache group, which are *read* and *write* cold files. Files belonging to the high-priority cache group accumulate their cache hit counts to *rcachehit* and *wcachehit*. These hit counts are used for $CHScore_{inodeid}$, the normalized cache hit ratio of *inodeid* to compare the intra-group priorities. It is obtained using (1), by dividing the read and write hit count of a file *inodeid* by the largest read and write hit count

TABLE 2. Cache priorities for file.

Priority	Description
High	read or write hot file
Normal	read or write warm
Low	read and write cold

amount all files, respectively. Therefore, the absolute hit count $rcachehit_{inodeid}$ and $wcachehit_{inodeid}$ together becomes a normalized value $CHScore_{inodeid}$ between 0 and 2.

$$CHScore_{inodeid} = \frac{rcachehit_{inodeid}}{\max\limits_{\forall iid}(rcachehit_{iid})} + \frac{wcachehit_{inodeid}}{\max\limits_{\forall iid}(wcachehit_{iid})} \quad (1)$$

C. VFS/FD MAPPER

The VFS/FD mapper tracks the I/O on the Android framework and Linux kernel. It maps a file descriptor (fd) of an application process to a VFS entry of the Linux kernel. It also creates an I/O tag to track which application issues a file I/O request, or which application is related to which block I/O.

D. CONTEXT-AWARE READAHEAD

Context-aware readahead reads a file with a high access probability into the memory in advance to enhance the cache hit ratio of the file. There are single file readaheads and multiple file readaheads. A single file readahead reads a portion of or all of the data over a single file, as shown in Fig. 3 (a), whereas a multiple file readahead reads a portion of or all of the data over multiple files, as shown in Fig. 3 (b). If the size from the current position to the readahead position is smaller than the maximum readahead size $S_{readahead}$ as well as the next sequential read information in *FileContext.nextfc*, a multiple file readahead is applied. A single file readahead is conducted for the other cases. The maximum readahead size per file is 512KB. With the Linux kernel's cache policy, context-aware readahead will cache readahead data in memory as long as the memory allows. When the cache space runs out, the cache is evicted according to the priority of CFQ-CA.

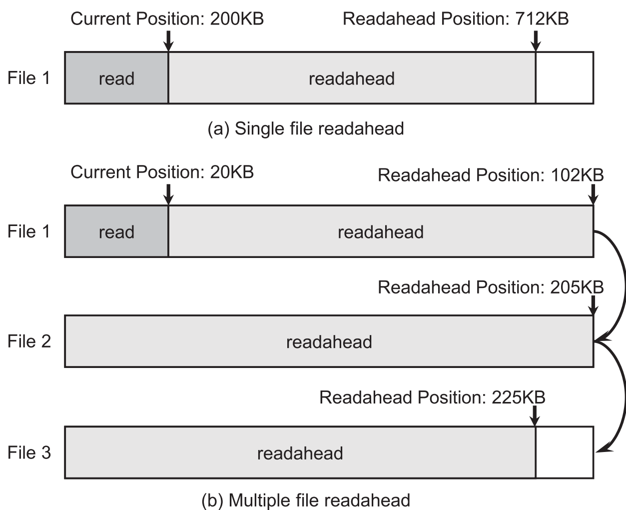

FIGURE 3. Single and multiple file readaheads.

E. CONTEXT-AWARE CACHE QUEUE

The context-aware cache queue manages the file page cache according to Table 2, which is the cache priority analyzed by the context information analyzer. The pages of a file are inserted into three types of cache queues, as shown in Fig. 4, according to the priority of the file. For example, if a file's priority of a page is tagged to be high, then the page would be inserted to the priority queue. The caches of high-priority files are managed by a priority queue to provide a fast response. In the case of normal priority, it is managed by the least recently used (LRU) queue that is deployed in the existing system. Files with low priority are managed in the simplest way regardless of their order using the first-in first-out (FIFO) queue to achieve lower management overhead. Upon a cache replacement, caches from the FIFO queue, where the cache of the lower priority files is kept, are first evicted. Therefore, the caches with a high access probability are kept in the main memory as much as possible.

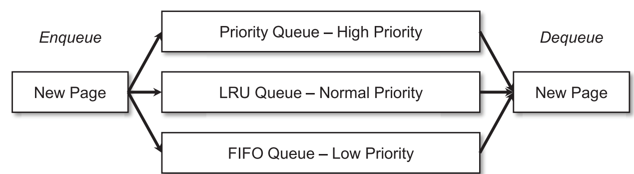

FIGURE 4. Context-aware cache queue for high, normal, and low priorities.

F. CONTEXT-AWARE DEFRAGGER

In a mobile operating system environment such as Android, it has been reported [26] that the probability of fragmentation occurring in a single file is low; however, these observations have a limitation in that the fragmentation of inter-files has not been considered. For instance, let us assume that there is a multiple file readahead task with the files shown in Fig. 5. The I/O for readahead on files 1-3 and 5 can be reduced if those files are consecutively located than separated; thus, the read performance can be improved if files 1–3 and 5 are read in advance as a sequential file access pattern. The existing studies cannot conduct a defragmentation, i.e., a sequential relocation of those files, as such studies cannot detect that access to file 4 is different from that of the other files. Context-aware defragger reads multiple files at the same time when conducting multiple file readaheads and then performs defragmentation that relocates the related files sequentially

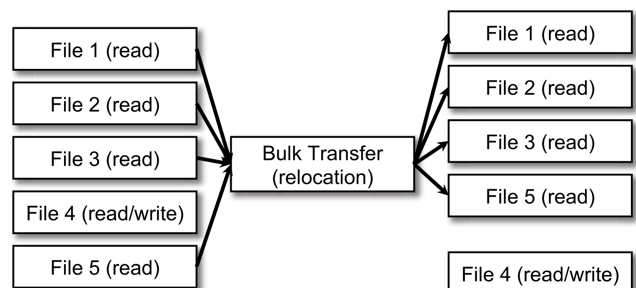

FIGURE 5. Context-aware defragment for multiple file readaheads.

to improve the I/O performance. Context-aware defragger selects readahead candidates using *FileContext.nextfc* and supports as many files to be stored sequentially as possible in the storage device using the bulk transfer function, as shown in Fig. 5. A bulk transfer is conducted for multiple-file readahead targets without write operations.

G. CFQ-CA

CFQ-CA is a scheduling module designed to prioritize I/O requests from applications currently in the foreground. It replaces the existing Linux I/O scheduler. In the VFS/FD mapper, the block I/O request of the application running in the foreground and the block I/O request running in the background are tagged and delivered to CFQ-CA. The weights for the weighted round robin of CFQ-CA are set according to the background/foreground state of application the process belongs to. In order to improve the application responsiveness for users, the weight for the foreground I/Os is set to higher than the weight for the background I/Os. Each process is added to its priority weight in the weighted round robin.

IV. IMPLEMENTATION

The CAFIO's components exist across Android Framework layer and each I/O stack of Linux kernel. A mechanism to communicate between user space and kernel space is required to implement CAFIO. Therefore, the linux netlink mechanism is used to share data between the user space components and the kernel space components.

The VFS/FD mapper in kernel space combines the file descriptors of an application process and its corresponding inode in VFS. When each of open(), read(), and write() is called, its related inode and process uid is collected by the VFS/FD mapper. These two data are then sent up to the CAFIO's context information manager. The memory overhead of VFS/FD mapper is the sum of those two data for every uid-inode pair. Each data is represented by an 8-byte variable on 64-bit architecture so that the memory overhead of each data pair is 16 bytes on the actual implementation.

The *FileContext* is created for each accessed file. In order to create and access *FileContext* structure efficiently, the hashmap is implemented in CAFIO. Therefore, a *FileContext* structure is accessed with its key, which is its *inodeid*. On the other hand, the *FileContext*(s) are managed as a list when accessed through its *nextfc* field. The *nextfc* field is a linked list of *FileContext* of read files right after this file. In the actual implementation on arm64 architecture, each field of *FileContext* data structure is a 64-bit variable such that the memory overhead of *FileContext* is 64 bytes.

V. PERFORMANCE EVALUATION

A. ENVIRONMENTAL SETUP

To evaluate the performance of CAFIO proposed in this paper, the launch time, cache hit ratio, and read speed of the target applications are measured and compared with those for the existing Android system. As a target device to be used for performance measurement, a Google Nexus 6P is selected. In addition, a Google Pixel 3 device is used to confirm the performance evaluation results with other mobile devices. The detailed specifications are shown in Table 3 and 4, respectively.

TABLE 3. Target device specification - Google Nexus 6P.

Model	Google Nexus 6P
Processor	Qualcomm Snapdragon 810 MSM8994 SoC ARM Cortex-A57 2GHz + ARM Cortex-A53 1.55GHz
Memory	3GB LPDDR4 SDRAM
Display	WQHD(2560x1440) AMOLED Touchscreen Display
Storage	32GB Flash Storage (eMMC 5.0) MLC
OS	Android 7.1.2 (Nougat) with 3.10.73

TABLE 4. Target device specification - Google Pixel 3.

Model	Google Pixel 3
Processor	Qualcomm Snapdragon 845 SDM845 SoC ARM Kryo 385 Gold 2.8GHz + ARM Kryo 385 Silver 1.8GHz
Memory	4GB LPDDR4 SDRAM
Display	WQHD(2160x1080) OLED Touchscreen Display
Storage	64GB Flash Storage (UFS 2.1)
OS	Android 9.0.0 (Pie) with 4.9.124

The target applications for performance measurement were selected from the top five popular applications for each category in the Google Play Store. The full list of applications is shown in Table 5. The total execution of applications A–T is defined as a single test run. In each test run, the launch sequence of 20 applications is permuted randomly such that the measured performance of an application is not affected by the predetermined sequence of the application launches. A command to terminate all applications as well as invalidate the caches is explicitly used in the Linux kernel between each test run such that the caches remaining between test runs do not affect the other results.

The file I/O footprints of target applications was measured and shown in Table 6. Each application was executed 10 times independently, such that each footprint value is a mean of 10 independent measures. The average number of files an application accesses, the number of reads and writes an application requests, and the total volume of reads and writes during the execution were measured to validate the experimental results.

The test method used to measure the launch time, cache hit ratio, and read speed of an application is as follows. The application launch time is the time from when the user launches an application to when the application is displayed on the screen. The cache hit ratio represents the cache hit ratio for all I/Os generated during the measurement of the launch time of the application. The read speed is the average speed of all read requests issued by the target application during its launches.

TABLE 5. Tested package set for target platform.

ID	Original Package Name	ID	Original Package Name
A	Naver Band	K	Lineage M
B	Intagram	L	HelloHero Epic Battle
C	Facebook	M	Friends Jam
D	Kakao Story	N	Eternal 7 Cities
E	Naver Cafe	O	Pokemon Quest
F	TikTok	P	Hancom Office Viewer
G	Oksusu	Q	Naver Cloud
H	SBS - OnAir	R	Polaris Office
I	Melon	S	Microsoft Powerpoint
J	Music Mate	T	Microsoft Word

Each experimental result is a mean of 10 test run outputs. It is also possible to measure the read speed with real-world data in which user interactions are recorded. However, collecting the vast amount of user-application interaction data is out of the scope of our research, and the footprints in Table 6 shows that there are diverse form of reads exist in application launches. The launch sequence of applications A-T for each test run is permuted randomly such that the measured performance of an application is not affected by the predetermined sequence of the application launches.

TABLE 6. File I/O footprints of tested package set.

ID	No. of Files	Read Size	No. of Reads	Avg. Read Size
A	15	2.79 MB	446.2	6.246 KB
B	25	1.13 MB	4263.1	0.266 KB
C	41.1	0.98 MB	5081.5	0.192 KB
D	14.3	1.38 MB	217.8	6.335 KB
E	17.1	2.44 MB	392.3	6.216 KB
F	120.7	4.60 MB	1240.9	3.707 KB
G	10.9	0.82 MB	251.9	3.246 KB
H	19.5	1.81 MB	393.4	4.593 KB
I	20	1.73 MB	475.3	3.638 KB
J	25.5	3.65 MB	819.2	4.459 KB
K	34.1	15.00 MB	3592.7	4.176 KB
L	24.2	20.10 MB	4498.4	4.467 KB
M	40.3	187.56 MB	67659.6	2.772 KB
N	32.5	10.14 MB	1133.1	8.947 KB
O	15.2	6.78 MB	1078.2	6.284 KB
P	2.3	0.10 MB	102.8	1.001 KB
Q	7.8	0.79 MB	159.7	4.933 KB
R	74	37.04 MB	16307.1	2.271 KB
S	61.2	10.05 MB	2348.8	4.279 KB
T	55.7	11.04 MB	2539.2	0.004 KB

B. EVALUATION RESULTS

The application launch time results are shown in Fig. 6. The overall launch time of the applications was improved by an average of 26.4%. A performance degradation was observed at −11% and −3% for applications D and L, respectively, whereas an average 30% improvement was observed for the other applications. The result exhibits a trend in the improved performance of Android+CAFIO.

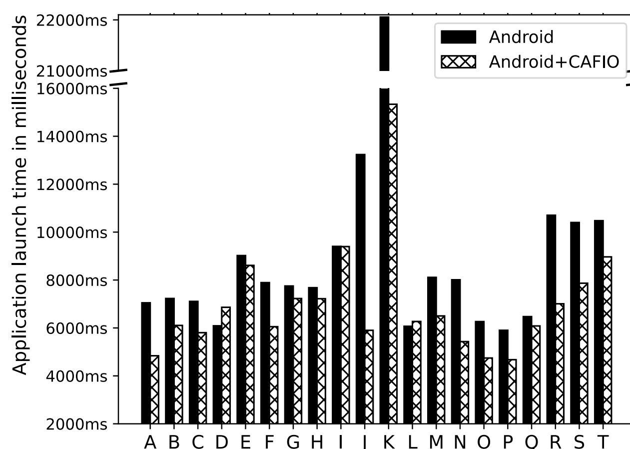

FIGURE 6. Average launch time of each application.

The cache hit ratio of the applications is shown in Fig. 7. An average of 45.4% improvements in cache hit ratio is observed from the result. The application K is shown to be the most positively affected one when using CAFIO with an average improvement ratio of 330.4%. Applications that showed a high hit ratio already on Android were not improved as much as the others since there was not enough room to improve. Application C showed the least performance improvement in the cache hit ratio, and application G showed performance degradation by 0.7%.

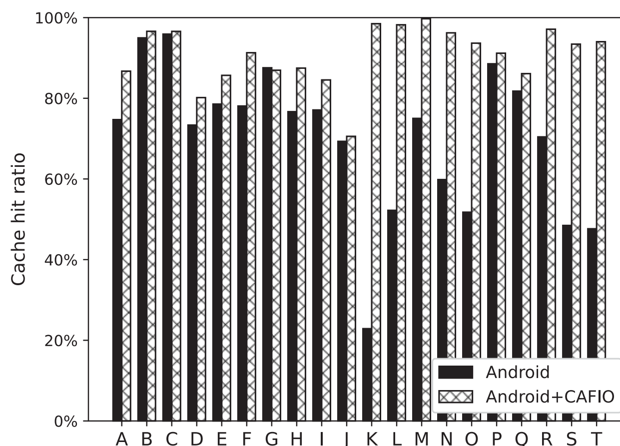

FIGURE 7. Average cache hit ratio of each application.

The read performance measurements of the application file are shown in Fig. 8. The read speed of applications P and Q could not measured because the time spent on read operation was too short. On average, CAFIO showed a performance improvement of 63.1%, with applications B and L showing 7.83- and 2.47-times faster read speeds than that of the

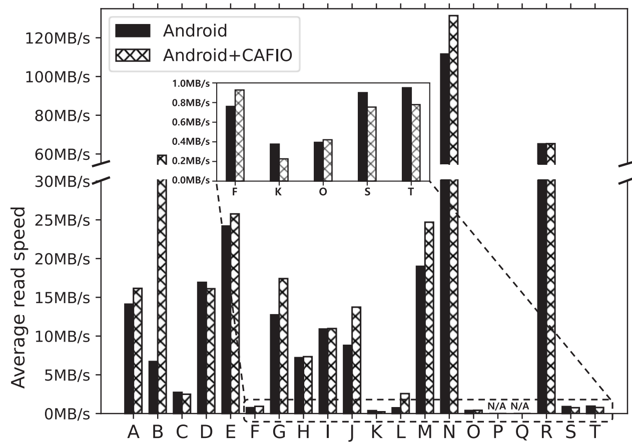

FIGURE 8. Average read speed of each application.

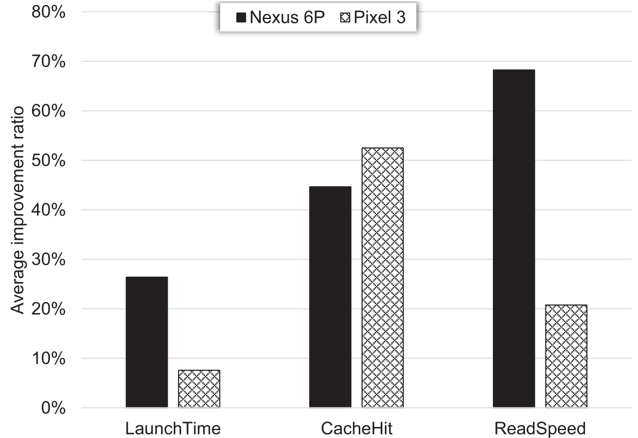

FIGURE 10. Average improvement ratio on different devices.

existing system, respectively. Applications C, D, K, S, and T showed a performance degradation on average of 17.5%.

If either the portion of one of hot, warm, cold file increases, the overall result would close to the case in which all of the files are hot, warm, or cold. Then, the performance improvements would be lessened to the stock Android system. Therefore, we measured the performance on different hot/warm/cold ratios to confirm the hypothesis. The average launch time of applications A-T is shown in Fig. 9. The ratio of 1:1:1 is observed to be the most effective ratio than others.

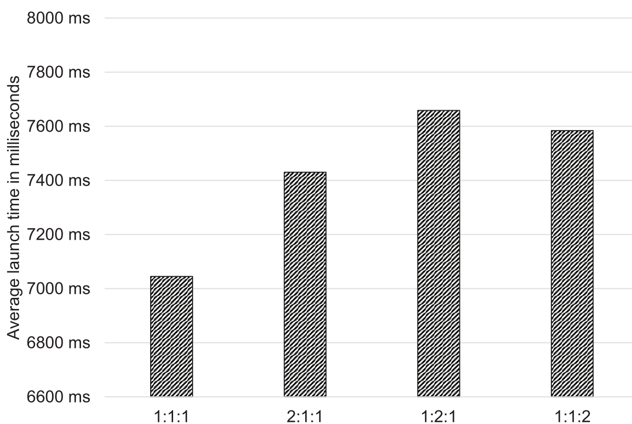

FIGURE 9. Average launch time of applications for different hot/warm/cold ratios.

To confirm the CAFIO's performance improvements in other devices, we conducted the same experiments on Google Pixel 3 device and compared the results to that of Google Nexus 6P. As shown in Table 4, the Pixel 3 device has larger RAM with faster flash storage. The performance measurements are shown in Fig. 10.

The average application launch time improvement was 7.6%, which is lesser than the Nexus 6P. However, the cache hit ratio showed higher improvements than CAFIO on Nexus 6P device. This is mainly because Pixel's RAM with higher capacity can cache more relevant data than Nexus 6P. The read speed improvement ratio is shown to be less than Nexus 6P case. The Pixel 3 device is equipped with high performance storage than Nexus 6P device. Therefore, there was less room for improvement on Pixel 3 device. The positive improvement ratio on both devices means that the experimental results can be confirmed such that CAFIO generally improves the I/O performance.

C. DISCUSSION

The performance measurement results of application launch time, cache hit ratio, and read speed have the tendency to be improved using CAFIO. When launching application, processing and networking can occur as well as file I/O such that some applications do not benefit from CAFIO's optimizations. Various applications of each category were selected from app store for generality. Therefore, for most cases, the proposed CAFIO improved the file I/O performance by enabling context-awareness between each I/O stack. Despite the improvements in cache hit ratio or read speed for the case of applications D and L, their launch time performance was not improved as shown in Fig. 6. These can be the case where processing and networking heavily affect the launch time of an application.

The cache hit ratio of application G decreased compared to the existing system as shown in Fig. 7. These degradations were due to the characteristics of the applications, which issue fewer read requests of the files, more I/Os over the network, or exhibit random file read patterns. Because CAFIO does not apply caching for a random file I/O, the cache hit ratio decreased. However, CAFIO allocates more resources for sequential file access caching instead of random file access caching. Therefore, CAFIO showed an overall performance improvement for all other cases. The readahead of CAFIO was proven to be effective through this cache hit ratio measurement.

The degraded performance is also observed in Fig. 8. This is due to the CAFIO's optimization scheme through context-awareness does not apply for random file I/Os.

Therefore, in case of small sized random I/O dominated application, only the read speed can be degraded while in the opposite case the performance can be dramatically improved. Because applications B and L entailed a large number of small reads, the number of I/O requests can be reduced through the context-aware defragmentation of CAFIO, thereby dramatically improving the read performance.

Despite the empirical results in Fig. 9 that the ratio of 1:1:1 is the most effective than others, the impact of varying ratio may depend on the workload of other applications. This is because the ratios can be infinitely divided and testing all possible workloads for all ratios is practically not possible.

VI. CONCLUSION

As the use of smartphones has increased and smartphones with various functions have been released, the capacity and launch time of their applications have also increased. However, because smartphone users are sensitive to the application launch time, it is important to decrease this time. Various studies have attempted to improve the I/O performance on each layer of the I/O stack; however, such studies have limitations in improving the performance because they consider only a single part of the I/O stack, such as improving the I/O scheduler or developing a new file system suitable for the smartphone's permanent storage device.

In this paper, we proposed CAFIO, a file I/O management system that can share information from each part of the I/O stack to the other parts of the stack. The application launch frequency was analyzed, and the information was utilized for file caching, readahead, and I/O priority configurations. A performance evaluation with various types of applications showed that the performance improved by an average of 26% in terms of the application launch time, 45% in the cache hit ratio, and 63% the read speed compared to the existing file I/O management system.

In the future, we aim to address the performance degradation caused when a random I/O is not cached. We also aim to extend our research to collecting real-world data on user-application interactions. In addition, it is necessary to study a method for improving both the network I/O and the file I/O owing to the highly dependent characteristics of smartphone devices.

REFERENCES

[1] S. Kemp. (2019). *Digital 2019: Global Digital Overview*. Accessed: May 19, 2020. [Online]. Available: https://datareportal.com/reports/digital-2019-global-digital-overview

[2] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.

[3] K. Lee and Y. Won, "Smart layers and dumb result," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, 2012, pp. 23–32.

[4] J. Han, S. Kim, S. Lee, J. Lee, and S. J. Kim, "A hybrid swapping scheme based on per-process reclaim for performance improvement of Android smartphones (August 2018)," *IEEE Access*, vol. 6, pp. 56099–56108, 2018.

[5] J. Kim, C. Kim, and E. Seo, "*ezswap*: Enhanced compressed swap scheme for mobile devices," *IEEE Access*, vol. 7, pp. 139678–139691, 2019.

[6] J. Kim and H. Bahn, "Analysis of smartphone I/O characteristics—Toward efficient swap in a smartphone," *IEEE Access*, vol. 7, pp. 129930–129941, 2019.

[7] J. Kim and H. Bahn, "Maintaining application context of smartphones by selectively supporting swap and kill," *IEEE Access*, vol. 8, pp. 85140–85153, 2020.

[8] H. Ahn, M. E. Wijaya, and B. C. Esmero, "A systemic smartphone usage pattern analysis: Focusing on smartphone addiction issue," *Int. J. Multimedia Ubiquitous Eng.*, vol. 9, no. 6, pp. 9–14, Jun. 2014.

[9] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O path: A holistic approach for application performance," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 345–358.

[10] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim, "Improving user experience of Android smartphones using foreground app-aware I/O management," in *Proc. 8th Asia–Pacific Workshop Syst.*, Sep. 2017, pp. 1–8.

[11] S. Han, S. Lee, I. Yee, D. Ryu, and J. Kim, "Fasttrack: Foreground app-aware I/O management for improving user experience of Android smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 15–28.

[12] M. Ju, H. Kim, M. Kang, and S. Kim, "Efficient memory reclaiming for mitigating sluggish response in mobile devices," in *Proc. IEEE 5th Int. Conf. Consum. Electron.*, Sep. 2015, pp. 232–236.

[13] C. Wu, C. Ji, L. Shi, C. J. Xue, B. Huang, and Y. Wang, "Dynamic merging/splitting for better responsiveness in mobile devices," in *Proc. 5th Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Aug. 2016, pp. 191–202.

[14] S. J. Prasath, "Android application context aware I/O scheduler," Ph.D. dissertation, School Comput. Inform., Decis. Syst. Eng., Arizona State Univ., Phoenix, AZ, USA, 2014.

[15] D. T. Nguyen, "Improving smartphone responsiveness through I/O optimizations," in *Proc. ACM Int. Joint Conf. Pervas. Ubiquitous Comput.*, New york, NY, USA, Sep. 2014, pp. 337–342.

[16] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 309–320.

[17] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 113–126.

[18] P. R. Jelenkovic and A. Radovanovic, "Least-recently-used caching with dependent requests," *Theor. Comput. Sci.*, vol. 326, nos. 1–3, pp. 293–328, 2004.

[19] S.-H. Kim, J. Jeong, and J.-S. Kim, "Application-Aware Swapping for Mobile Systems," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, p. 182, 2017.

[20] K. Ariyapala, M. Conti, and C. Keppitiyagama, "ContextOS: A context aware operating system for mobile devices," in *Proc. IEEE Int. Conf. Green Comput. Commun.*, Aug. 2013, pp. 976–984.

[21] D. Chu, A. Kansal, J. Liu, and F. Zhao, "Mobile apps: It's time to move up to CondOS," in *Proc. 13th USENIX Conf. Hot Topics Oper. Syst.*, 2011, p. 16.

[22] N. Vallina-Rodriguez and J. Crowcroft, "ErdOS: Achieving energy savings in mobile OS," in *Proc. 6th Int. workshop*, 2011, pp. 37–42.

[23] *blk-timeout.c*. [Online]. Available: https://elixir.bootlin.com/linux/v5.7.4/

[24] *When 2MB Turns Into 512KB*. Accessed: Jun. 20, 2020. [Online]. Available: https://kernel.dk/when-2mb-turns-into-512k.pdf

[25] H. Kim and D. Shin, "Optimizing storage performance of Android smartphone," in *Proc. 7th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2013, pp. 1–7.

[26] S. S. Hahn, "Improving file system performance of mobile storage systems using a decoupled defragmenter," *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 759–771.

JAEHWAN LEE received the B.S. degree in computer engineering from the School of Computer Science and Engineering, Chung-Ang University, Seoul, South Korea, in 2015, where he is currently pursuing the Ph.D. degree in software engineering.

His research interests include mobile operating systems, embedded systems, and Linux systems.

SANGHYUCK NAM received the B.S. and M.S. degrees in computer engineering from the School of Computer Science and Engineering, Chung-Ang University, Seoul, South Korea, in 2017 and 2020, respectively. He is currently pursuing the Ph.D. degree in system software engineering.

He was a Research Engineer with CreativeSoft, Seoul. His research interests include mobile systems, Linux systems, and context-aware systems.

SUHWAN KWAK received the B.S. degree from the Division of Media Software, Sungkyul University. He is currently pursuing the master's degree with the School of Computer Science and Engineering, Chung-Ang University.

His research interests include mobile embedded systems, cyber physical systems, and autonomous vehicle systems.

SANGOH PARK (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the School of Computer Science and Engineering, Chung-Ang University, in 2005, 2007, and 2010, respectively.

From 2012 to 2017, he worked as a Senior Researcher of Global Science Experimental Data Hub Center, Korea Institute of Science and Technology Information. He was a Research Professor with the School of Computer Science and Engineering. Since 2017, he has been working as an Associate Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interests include embedded systems, big data systems, cyber physical systems, and Linux systems.