Understanding and Reducing the Energy Consumption of F2FS on Smartphones

LIHUA YANG, RUIBO WANG* and YONG DONG, WEI ZHANG, HUIJUN WU, College of Computer Science and Technology, National University of Defense Technology, China
FANG WANG* and YANG XIAO, Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, China
KAI LU, College of Computer Science and Technology, National University of Defense Technology, China

Smartphones are everyday necessities with limited power supply due to portability. Charging a smartphone twice a day or more affects user experience. Flash friendly file system (F2FS) is a widely-used file system for smartphones. However, there is limited understanding of the energy consumption of novel smartphones using F2FS with software and hardware upgrades. We explore the energy consumption of F2FS by completing I/O requests in different scenarios to understand the energy consumption of F2FS on smartphones. We observe that the energy consumption of CPU (Central Processing Unit) and UFS (Universal Flash Storage) completing I/Os increases with the number of free space fragments. Free space fragmentation of F2FS mainly consists of invalid blocks. F2FS reclaims invalid blocks by garbage collection (GC). We observe that the energy consumption of one background GC is more than 100 mJ but its effect on reducing free space fragments is limited. These motivate us to improve the energy efficiency of GC reducing free space fragmentation. We reassess how much free space is a free space fragment based on data analysis, and use the free space fragmentation factor as a metric to measure the degree of free space fragmentation quickly. Then we suggest the free space Fragmentation Aware GC scheme (FAGC). FAGC optimizes the selection for victim segments and the migration for valid blocks to reduce GCs and improve the energy efficiency of each GC. Experiments on the real platform show that FAGC reduces GC count by 82.68% and 74.51% respectively than traditional F2FS and the latest GC optimization of F2FS, ATGC. FAGC reduces the energy consumption by 164.37 J and 100.64 J compared to traditional F2FS and ATGC respectively for a synthetic benchmark. FAGC also reduces the running time by 24.4%-40.9% for large-scale applications.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Software and its engineering** → **File systems management**;

Additional Key Words and Phrases: F2FS, energy consumption, free space fragmentation, garbage collection

*Ruibo Wang and Fang Wang is the corresponding author.

Authors' addresses: Lihua Yang, Ruibo Wang; Yong Dong, Wei Zhang, Huijun Wu, College of Computer Science and Technology, National University of Defense Technology, Deya Road 109, Changsha, China, 410073, {yanglihua,ruibo, yongdong,weizhang,wuhuijun}@nudt.edu.cn; Fang Wang; Yang Xiao, Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, China, 430074, {wangfang,menguozi}@hust.edu.cn; Kai Lu, College of Computer Science and Technology, National University of Defense Technology, Deya Road 109, Changsha, China, 410073, kailu@nudt.edu.cn.

XXXX-XXXX/2023/12-ART00 $15.00
https://doi.org/0000000.0000000

ACM Reference Format:
Lihua Yang, Ruibo Wang, Yong Dong, Wei Zhang, Huijun Wu, Fang Wang, Yang Xiao, and Kai Lu. 2023. Understanding and Reducing the Energy Consumption of F2FS on Smartphones. *ACM Trans. Arch. Code Optim.* 0, 0, Article 00 (December 2023), 23 pages. https://doi.org/0000000.0000000

1 INTRODUCTION

Smartphones provide limited battery power due to portability. Charging a smartphone twice a day or more affects user experience. It is expected that software and hardware components of smartphones to reduce energy consumption. In the past decade, previous studies have attempted to explore the energy consumption model of smartphones [1–4], discover bottlenecks of energy consumption [5–8], and propose optimization schemes based on usage scenarios. Counter-intuitively, the energy consumption of smartphone storage subsystems for I/O intensive workloads is similar to the energy consumption of screen display and network connection, accounting for about 30% [9]. Flash friendly file system (F2FS) [10] is a typical log-structured file system designed for Flash characteristics. The performance of F2FS relies on contiguous free space. Maintaining contiguous free space requires expensive garbage collection (GC) to reclaim invalid blocks, which introduces a large number of reads and writes from migrating valid blocks. As a widely-used file system for smartphones, there is little research to understand the energy consumption characteristics of F2FS. It is urgent to explore its energy consumption completing I/O requests in different scenarios and inspire feasible solutions to reduce energy consumption. Understanding the energy consumption of F2FS I/O activities is significantly important for system architects, file system developers, and smartphone users.

There is file and free space fragmentation in F2FS. File fragmentation causes sequential read performance to degrade. Free space fragmentation of F2FS mainly consists of invalid blocks. Free space fragmentation leads to write performance degradation directly and causes read performance degradation of newly written files indirectly, and aging the underlying SSD devices [11]. In addition, free space fragmentation results in large blocks are not available for writes with increased GC overheads. There are two ways to manage free space in the traditional log-structured file system: threading write and copying data [12]. The method of copying data selects a segment as the cleaning object first, identifies the valid blocks in the segment, copies the valid blocks to a new segment, and reclaims these cleaned objects as idle segments for subsequent data writing, i.e., the garbage collection. The method of threading write treats invalid blocks in the file system as idle blocks and overwrites directly.

We evaluate the impact of file system space utilization, file and free space fragmentation on F2FS. We construct three states of pristine, full, and full fragmented with different space utilization of file system and number of free space fragments. We measure the energy consumption of sequential/random reading and writing, sequential reading continuous and fragmented files, and launching applications. Through an in-depth study to understand the energy consumption of storage systems using F2FS, we observe that free space fragmentation leads to an increase in the energy consumption of CPU and UFS modules to complete I/O tasks. When the continuous free space of file system is exhausted, there are a large number of free space fragments. It is necessary to trigger garbage collection to obtain idle segments or use the threaded logging write scheme to overwrite free space fragments. We compare the energy consumption of foreground GC (FGGC) and background GC (BGGC) and explore the effect of GC on reducing free space fragments. Importantly, the energy consumption of one BGGC is relatively large and its effect on reducing free space fragments is limited. We are motivated to reduce GC number and make each GC reduce free space fragments effectively. We recommend improving the energy efficiency of GC recycling invalid data.

We propose a GC scheme that is aware of the degree of free space fragmentation. There are file and free space fragmentation in file systems that cause performance degradation. Extensive research [13–15] optimizes file fragmentation while little research studies free space fragmentation. F2FS rearranges free space fragments through GC traditionally. We reassess the size of a free space fragment based on data analysis, design the metric of free space fragmentation **f**actor (f) to measure the degree of free space fragmentation in F2FS and propose the FAGC scheme. FAGC modifies the algorithm for selecting victim segments to trade off among segment utilization, age, and free space fragmentation. FAGC migrates valid blocks using threaded logging writes. The experimental results show that the GC count of FAGC is 82.68% lower than that of traditional F2FS and 74.51% lower than that of the latest GC optimization scheme of F2FS, ATGC (Age-Threshold based Garbage Collection) [16]. Compared to traditional F2FS and ATGC, FAGC reduces energy consumption by 164.37 J and 100.64 J for a synthetic benchmark, respectively. FAGC can reduce the time to service large application requests by 24.4%-40.9%.

The rest of this paper is organized as follows. We introduce the background in Section 2. Section 3 presents the experimental methodology and Section 4 discusses observations of energy consumption. The FAGC scheme is proposed in Section 5 and evaluated in Section 6. Related work is described in Section 7. We conclude in Section 8.

2 BACKGROUND

The hierarchical structure of a smartphone storage system is shown in Fig. 1. User space processes use system calls to access files, e.g., *open*, *read*, and *write*. User space processes are usually initiated by various applications, resulting in a heavy read and write pressure for the storage subsystem. SQLite is the default relational database used by Android. The virtual file system (VFS) services as an abstraction layer introduced over a specific file system. VFS sends file operations to the actual file system, here is F2FS.

F2FS is a typical log-structured file system that is widely used in mobile devices due to its data layout and management mechanism designed for Flash memory characteristics. Its data layout is divided into zones, sections, segments, and logical blocks. This is to facilitate users to align these cells with physical cells of the flash chip, such as die, plane, flash block, and flash page when formatting the file system. A logical block is the basic unit of read and write in F2FS which is 4 KB in size. 512 logical blocks make up a segment with a size of 2 MB. One or more segments form a section, which is the unit of F2FS garbage collection. The F2FS used by us sets that a section consists of a segment. One or more sections form a zone. The data organization is arranged flexibly to align as much as possible with the flash operation unit.

Table 1 lists commercial mobile devices that use F2FS as the file system. There are already a large number of commodity devices that use F2FS as a filesystem. Starting with the mobile operating system (Android 11) released in September 2020, the dynamic system update [17] mechanism of the Android Open Source Project (AOSP [18]) requires data partition using F2FS or the fourth extended file system (ext4 [19]). Since F2FS provides better read and write performance than ext4 [10], we recommend using F2FS.

A smartphone is powered by one battery that is limited in size and capacity so managing energy well is crucial. Effective energy management requires a good understanding of energy consumption. We come up with targeted proven solutions to reduce energy consumption naturally. Researchers have been exploring energy consumption models for mobile devices [20, 21] and combining energy consumption reduction with scenarios [22, 23]. However, there is a lack of analysis and optimization of energy consumption in nowadays smartphones using F2FS. Recently, researchers turn to the impact of free space fragmentation on F2FS energy consumption [24].

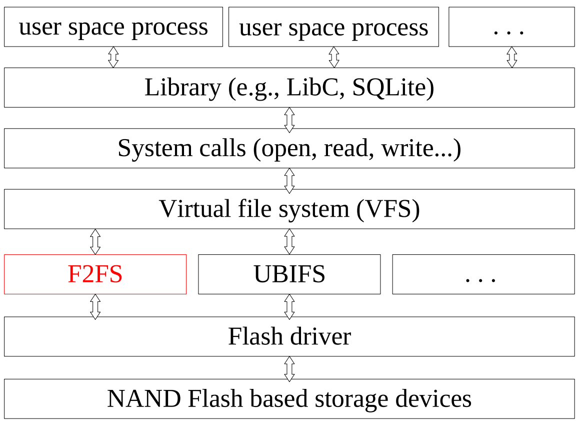
user space process
user space process
. . .
Library (e.g., LibC, SQLite)
System calls (open, read, write...)
Virtual file system (VFS)
F2FS
UBIFS
. . .
Flash driver
NAND Flash based storage devices

Fig. 1. The architecture of smartphone storage system

Table 1. Commercial mobile devices using F2FS

year of production	commercial mobile devices
2012-2015	MOTOROLA MSM8960 JBBL Droid/ G/ X Series
2014-2016	Nexus 9, MOTOROLA E LTE/Z, OnePlus 3T, Huawei Honor 8/ V8/ P9/ Mate 9
2018-2019	Google Pixel 3/3 XL, ZTE Axon 10 Pro, Samsung Galaxy Note 10/Tab S6
from 2020	Google Pixel series with Android 11 and above

Table 2. Three states of an F2FS

status	utilization	free space	description
pristine	2%	216 GB	An initialized system that has just been refreshed. We install six applications of Baidu, Angry Birds, WeChat, Tencent Video, and Taobao.
full	100%	1.5 GB	In the pristine state of the file system, we use fio [25] to randomly write a file of 2 GB, then copy and fill the file system until the remaining free space is 1.5 GB. There is no deletion in this process.
full fragmented	100%	1.5 GB	In the pristine state of the file system, we use fio to fragment the file system, and the remaining free space is 1.5 GB.

3 METHODOLOGY

To observe the impact of space utilization and free space fragmentation, we construct three states of F2FS: pristine, full, and full fragmented. How to construct the three states are shown in Table 2. To construct the state of full fragmented, we use fio to fill the system with files of 1 GB, 2 MB, 4 KB, and 8 KB in order and delete files at intervals after writing corresponding files. Referring to the classification of file fragment size in paper [14], we design the classification of free space fragment size. The size of free space fragments from L1 to L9 is shown in Table 3. We classify to L9 because the number of fragments larger than 512 KB is relatively small. The distribution of

Table 3. The size of different levels fragments of free space

fragment size	**the size of a free space fragment x (unit: KB)**
L1	$0 < x \leq 4$
L2	$4 < x \leq 8$
L3	$8 < x \leq 16$
L4	$16 < x \leq 32$
L5	$32 < x \leq 64$
L6	$64 < x \leq 128$
L7	$128 < x \leq 256$
L8	$256 < x \leq 512$
L9	$x > 512$

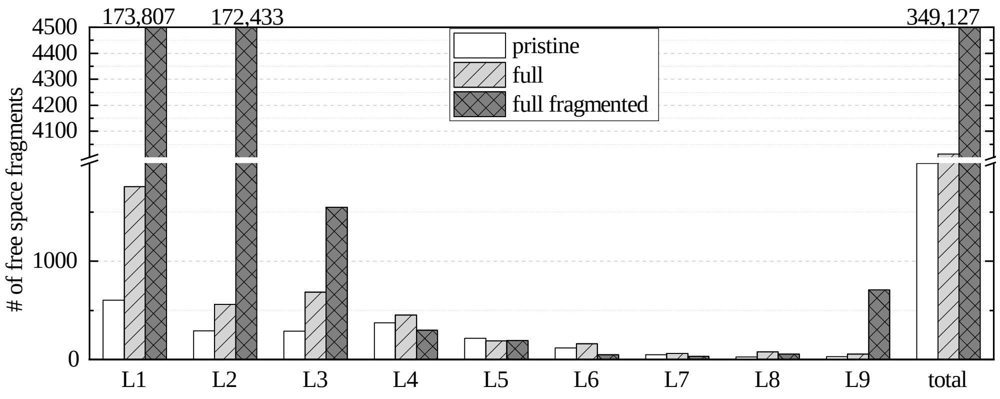

Fig. 2. The distribution of free space fragments in the three states

free space fragments for the pristine, full, and full fragmented state is shown in Fig. 2. Although the space utilization of the full state is much higher than that of the pristine state (100% vs. 2%, i.e., 50 ×), the number of its free space fragments is not so much more than that in the pristine state. The total number of free space fragments in the full state is only 2.01 times as much as that of the pristine state because there is no deletion while constructing the full state. The number of free space fragments in the full fragmented state is much higher compared to the full state since constructing the full fragmented state creates and deletes a large number of files. According to Table 2 and Fig. 2, although the free space of the full fragmented and the full state is the same, the total number of free space fragments in the full fragmented state is 87.02 times as much as that of the full state. The L1 (4 KB) and L2 (8 KB) fragments of the full fragmented state are much more (98.81 × and 306.82 ×) than that of the full state.

The parameters of our experimental platform are shown in Table 4. We use our developed energy consumption measurement system based on Kirin 9000 [26]. The system consists of a power supply, an energy consumption board, three monitoring boards of energy consumption, recording software on the host computer, and conversion tables. The software includes application interfaces, configuration files, and drivers. We use the power supply to provide a regulated voltage of 4.2 volts and monitor the current. The energy consumption board uses two data cables to interact with the computer. A data cable transmits energy consumption data and the software displays the curve

Table 4. Parameters of the experimental platform

Device	Kirin 9000
Process	5 nm
CPU	1x Cortex-A77@3.13 GHZ, 3x Cortex-A77@2.54 GHZ, 4x Cortex-A55@2.05 GHZ
GPU	24-core Mali-G78, Kirin Gaming+ 3.0
AI	HUAWEI Da Vinci Architecture 2.0 Ascend Lite×2 + Ascend Tiny×1
System Cache	8 MB
Memory	LDPPR 5/4X
Memory size	8 GB
Storage size	256 GB (Data: 220 GB)
Android version	10
Linux Kernel	4.14.116

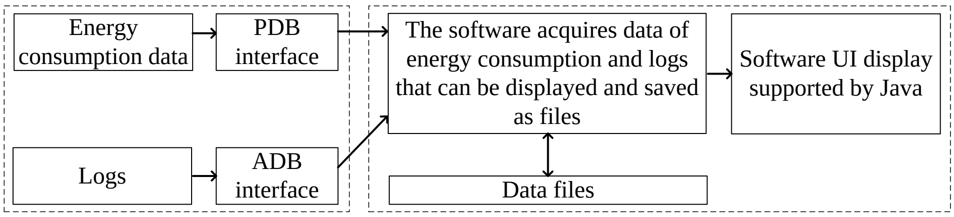
Energy consumption data
PDB interface
Logs
ADB interface
The software acquires data of energy consumption and logs that can be displayed and saved as files
Software UI display supported by Java
Data files

Fig. 3. Schematic diagram of the structure of the energy consumption measurement system

of voltage, current, and power in real-time. Another data cable transmits ADB (android debug bridge) commands and the energy consumption board executes operations corresponding to the command. The monitoring board of energy consumption inserts detection resistors in series into the power supply line of the system and calculates the current by measuring the voltage across the resistor. The schematic diagram of the structure of the energy consumption measurement system is shown in Fig. 3. The left side shows the functions of the energy consumption board and monitoring boards of energy consumption, and the right side shows the functions of the recording software and conversion tables. Our measurement system of energy consumption can decompose the energy consumption of each functional module of a smartphone in a certain scenario.

The data files derived can be converted by the conversion table to obtain the energy consumption of each module. In the model of Kirin 9000, the energy consumption of CPU, DDR (Double Data Rate Synchronous Dynamic Random Access Memory), UFS, GPU (Graphics Processing Unit), NPU (Neural-Network Processing Unit), LCD (Liquid Crystal Display), 5G (5th Generation Mobile Communication Technology), WiFi, and Bluetooth can be obtained. We can calculate energy consumption using voltage, current, and time to complete I/O tasks that can be captured in curves displayed by the software in the computer. The 5G, WiFi, and Bluetooth consume little energy because the network is unavailable in closed labs. Moreover, *our study focuses on I/O-intensive workloads which comprehensively explores I/Os of F2FS without complicated calculation*, the energy consumption of GPU and NPU is extremely small. The energy consumption of the LCD screen varies with scenarios. It is woken up when launching applications while is off for other I/O tasks and GCs. Finally, we focus on the energy consumption of CPU, DDR, and UFS modules that are closely related to F2FS working.

4 RESULT ANALYSIS AND DISCUSSION

We measure the energy consumption of sequential/random reading and writing, sequential reading continuous and fragmented files, launching applications, and GC, and the effect of GC on reorganizing free space fragments.

4.1 Sequential/random read and write

Considering that sequential/random read and write have different performances, we measure the latency and energy consumption of sequential/random read and write. The file size is 128 MB, and the I/O size of sequential read and write is 1 MB while that of random read and write is 4 KB. We restart the system to avoid the influence of cached data. The energy consumption of sequential/random reading and writing 1 MB of data and their runtime are shown in Fig. 4. The screen is off and the network is disconnected during these I/O-intensive workloads, so the energy consumption of CPU, DDR, and UFS accounts for a large proportion. In Fig. 4, *CPU, DDR*, and *UFS* means the energy consumption of CPU, DDR, and UFS modules respectively where *all* means the energy consumption of the energy consumption board working.

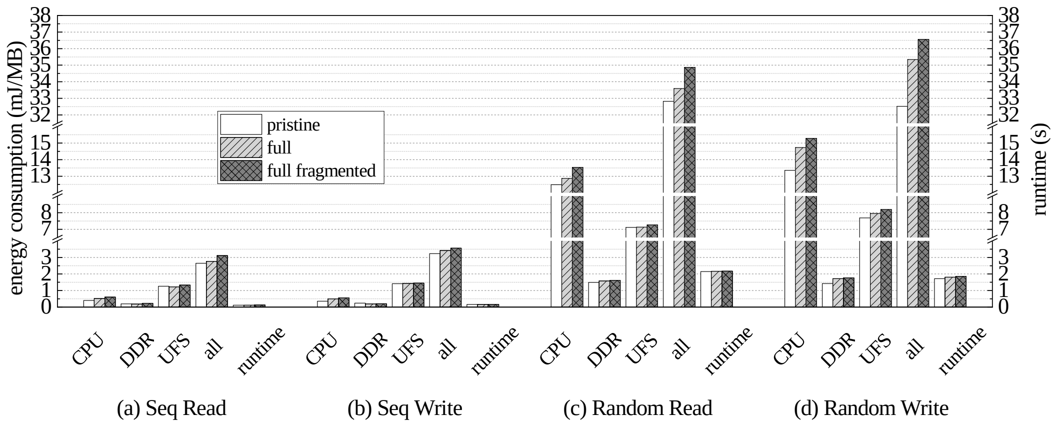

Fig. 4. Latency and energy consumption of sequential/random reading and writing a 128 MB file

The average energy consumption of the three states for the board to sequential read, sequential write, random read, and random write 1 MB of data is 2.84 millijoule (mJ), 3.4 mJ, 33.76 mJ, and 34.81 mJ, respectively. The time of random read for the three states is 17.57 times as much as that of sequential read on average, and the time of random write is 10.81 times as much as that of sequential write on average. The energy consumption of the UFS module accounts for the highest proportion while sequential reading and writing. The UFS energy consumption of sequential read and write accounts for 44.77% and 42.07% respectively while that of random read and write accounts for 21.23% and 22.81% respectively. In contrast, the energy consumption of the CPU module accounts for the highest proportion while random reading and writing. The proportion of CPU energy consumption for random read and write is 38.49% and 41.53% respectively while that for sequential read and write is 17.91% and 13.95% respectively. The runtime and energy consumption of the full fragmented state are the highest among the three states.

Furthermore, we compare the power of sequential/random read and write. The sequential read power is 1.48 times as large as the random read power for the three states on average, and the sequential write power is 1.06 times as large as the random write power. Compared to sequential read and write, random read and write take longer, their power is slightly lower and the overall energy consumption is higher. The CPU power of sequential read and write in the full fragmented state is 1.3 times and 1.5 times higher than that of sequential read and write in the pristine state, respectively.

Discussion 1. *The energy consumption of random read and write is much higher than that of sequential read and write. It is recommended to make I/O requests as continuous as possible. However, file and free space fragments make I/O requests split. We propose to reduce file and free space fragmentation to guarantee the continuity of I/O requests.*

The energy consumption of UFS working increases with space utilization and free space fragmentation. The proportion of UFS energy consumption for sequential I/Os is greater than that for random I/Os, by contrast, the CPU energy consumption for random read and write accounts for a large proportion. We would recommend exploring the root causes and reducing UFS energy consumption in sequential I/Os. Researchers could explore what influences CPU energy consumption in random I/Os and optimizes performance for key I/Os.

4.2 Sequential reading continuous and fragmented files

Since file fragmentation affects the performance of sequential reads [27–29], we measure the runtime and energy consumption of sequential reading continuous and fragmented files. We sequentially write the continuous file of 128 MB while randomly writing the fragmented file. Meanwhile, we both randomly write 7 other files of 128 MB. We restart the system and repeat the measurement 5 times. The energy consumption and runtime are shown in Fig. 5.

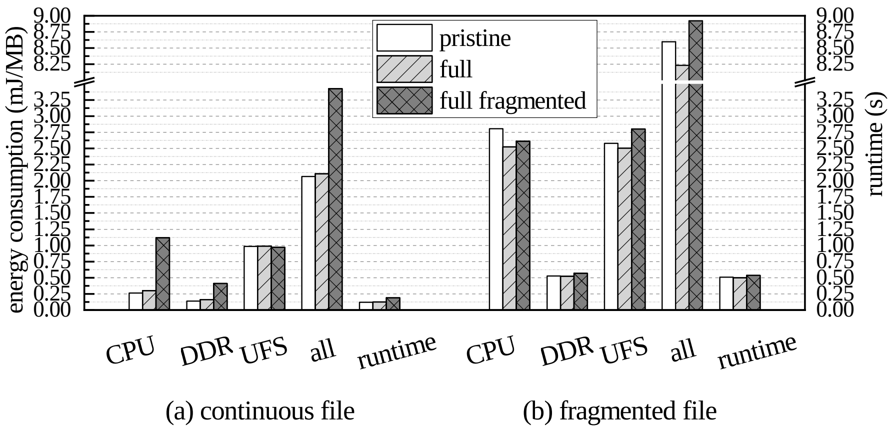

Fig. 5. Sequential reading continuous and fragmented files

The energy consumption for the smartphone to sequential read 1 MB of data for continuous files and fragmented files is 2.53 mJ and 8.58 mJ for the three states on average, respectively. The energy consumption for sequential reading 1 MB of data from fragmented files is 3.39 times as much as that for sequential reading continuous files. The time spent reading fragmented files sequentially and their energy consumption is significantly increased. Compared to continuous file, the time to read fragmented file sequentially increases by 254.38%, its CPU energy consumption increases by 371.92%, its DDR energy consumption increases by 128.65%, and UFS energy consumption increases by 167.72% on average. The increase in CPU energy consumption is the largest and it is necessary to optimize the CPU energy consumption increased by fragmentation. Compared to the full state, the latency and energy consumption of sequential read in the full fragmented state increase, indicating that free space fragmentation affects the sequential read performance negatively.

Compared to the pristine state, the energy consumption to read fragmented files sequentially in the full state decreases instead. Because the number of file fragments in fragmented files is relatively large, the latency and energy consumption of sequentially reading fragmented file in the pristine state are relatively large. Since no files are deleted during the process from the pristine state to the full state, there are few free space fragments, which only increases the space utilization of file system. This shows that characteristics of read and write operations and the occurrence of fragmentation in the system are more related to the performance of file system than the increase in the space utilization.

Discussion 2. *The energy consumption of sequential reading fragmented files is higher than that of contiguous files. The CPU energy consumption increases significantly with the number of file and free space fragments. Alleviating file and free space fragmentation is suggested to optimize performance and energy consumption.*

4.3 Launching applications

Application launch time is an important factor affecting experience of smartphone users. The launch time and energy consumption of launching applications are shown in Fig. 6. We can see energy consumption and launch time of baidu, game (Angry Birds here), video (Tencent Video Player here), taobao, wechat, and the average of these 5 applications. The energy consumption of CPU and DDR is more than that of UFS while launching applications, and the CPU energy consumption increases the most due to fragmentation. The launch time of the full and the full fragmented state increases by 2.57% and 4.36% respectively for the 5 applications on average compared to the pristine state. The energy consumption of the whole system in the full and the full fragmented state increases by 57.45% and 62.08% respectively. In contrast, the CPU energy consumption of the full and the full fragmented state increases by 94.84% and 100.8% respectively compared to the pristine state. The bad impact of fragmentation is urgently needed to be improved.

Discussion 3. *The time for I/O tasks and launching applications increases with the file system space utilization and the number of free space fragments. We observe that a small increase in runtime results in a large increase in energy consumption. We suggest reducing the time to launch applications because the short time induces the energy consumption reduction of all modules that is large overall. It is noteworthy that a prominent increase in CPU energy consumption due to the increase of fragments.*

4.4 Energy Consumption of GC

F2FS relies on garbage collection to reorganize free space fragments to obtain idle segments. We measure the energy consumption of FGGC versus BGGC, and measure the effectiveness of garbage collection on reducing free space fragmentation. F2FS uses the *gc_thread_func()* to increase or decrease the time interval for waking up the thread for BGGC according to the actual situation of file system. F2FS uses the *__get_victim()* to call the *select_policy()* to determine whether to use the greedy algorithm or the Cost-Benefit algorithm when selecting a victim segment. Then F2FS uses different methods in *get_gc_cost()* to calculate the migration cost according to the policy. Finally, F2FS migrates the corresponding segment with *do_garbage_collect()* and modifies the corresponding metadata structure and count.

We modify the kernel code to trigger a foreground GC by outputting the maximum time interval without GC under the */sys/fs/f2fs/blkid/*, i.e., *cat gc_no_gc_sleep_time*. We calculate the time and energy consumption to do 10 FGGCs differentially. We achieve different trigger frequencies of BGGC by modifying the sleep time. Taking the energy consumption of the whole system where the

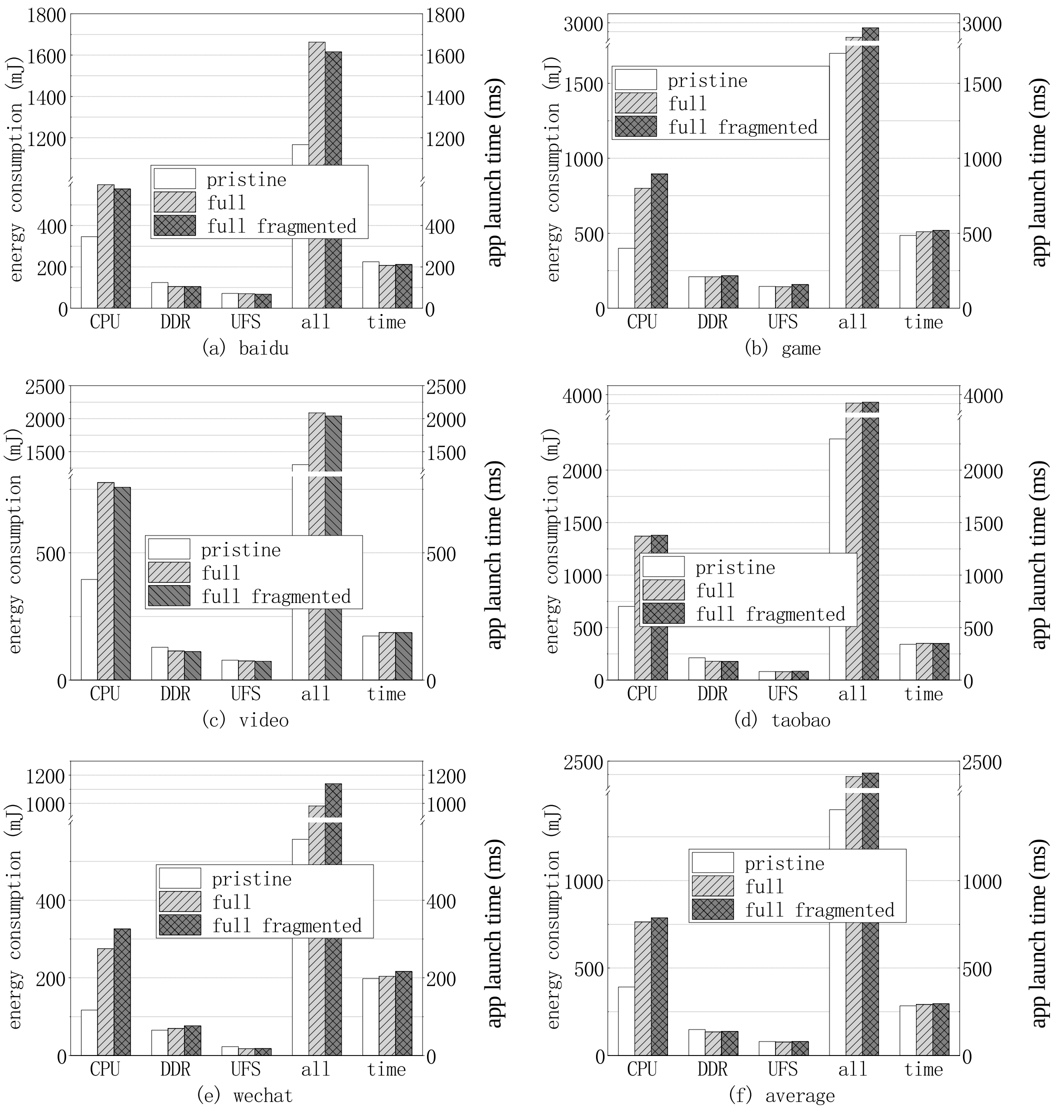

Fig. 6. Launching applications

sleep time is 120 s with one BGGC in the pristine state as the baseline, we calculate the increased energy consumption due to increased BGGCs. There is no other I/O activities in the system during this period. The increased energy consumption of BGGCs with different trigger frequencies after 120 s is shown in Fig. 7. The sleep time inteval of BGGC are 1000 ms, 100 ms, and 10 ms, respectively. The energy consumption of the whole system with the sleep time of 1000 ms is 17.5% lower than that of 100 ms and 55.46% lower than that of 10 ms for three states on average. From the pristine state to the full fragmented state, the energy consumption of the whole system is increased significantly due to the increase of space utilization of file system and number of free space fragments.

The energy consumption of one FGGC and one BGGC with different trigger frequencies is shown in Fig. 8. It takes 4.619 s, 4.923 s, and 5.389 s to complete 10 FGGCs in the pristine, full, and full

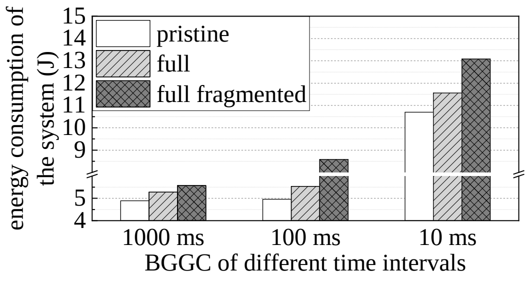

Fig. 7. Increased energy consumption after 120s BGGC

fragmented state, respectively. The energy consumption of one FGGC, one BGGC with the sleep time of 1000 ms, 100 ms, and 10 ms is 438.95 mJ, 123.02 mJ, 134.67 mJ, and 184.3 mJ, respectively, for three states on average. The energy for one FGGC can be used to do 3 BGGCs with the sleep time of 100 ms. The energy consumption of one BGGC with the sleep time of 1000 ms is 8.66% lower than that of one BGGC with the sleep time of 100 ms. The energy consumption of one BGGC with the sleep time of 10 ms is 36.85% higher than that of one BGGC with the sleep time of 100 ms and its CPU energy consumption is relatively high.

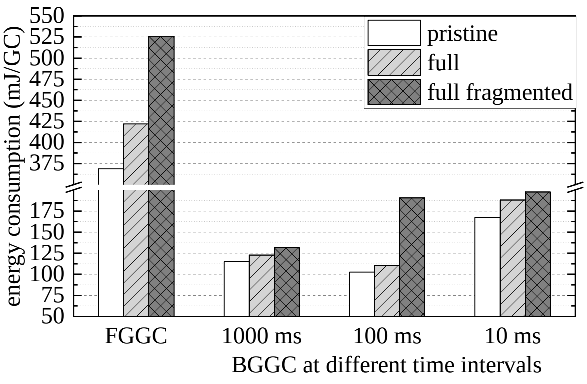

Fig. 8. Energy consumption of one time GC

The energy consumption of one FGGC is prohibitively high and FGGC is only triggered when necessary. Compared to the energy consumption for random reading or writing 1 MB of data, the energy consumption of one BGGC cannot be ignored. The least energy-consuming BGGC in Fig. 8 also consumes more than 100 mJ energy that can complete random reading or writing about 3 MB of data. The minimum energy consumption of FGGC and BGGC is 368.85 mJ and 102.79 mJ, respectively. The battery capacity of a mid-end smartphone is 3000 mAh and the smartphone battery voltage in industry is commonly 3.8 V. The battery can provide 41,040 J energy when fully charged. Assuming that a full-charged smartphone can be used for 24 hours, it consumes 475 mJ energy per second. If adding 1000 FGGCs per hour, the standby time is shortened by 4.26 hours. If adding 1000 BGGCs per hour, the standby time is shortened by 1.36 hours.

Discussion 4. *The energy consumption of one FGGC or one BGGC is relatively large. We would trigger GC lazily under the premise of ensuring the normal work of system. When the trigger frequency* *of BGGC is high, its CPU energy consumption is extremely large. Reducing BGGC by adjusting their trigger frequency needs to consider the CPU energy consumption, e.g., RLBC [30], MOBC [31].*

4.5 The effect of GC on reducing free space fragmentation

We measure and analyze the effect of energy-intensive GC on reducing free space fragmentation. The number of GCs and free space fragments with different sizes change with GCs in the full fragmented state are shown in Fig. 9. The trigger frequency of FGGC in Fig. 9 is lower than that in Fig. 8 and its FGGC is triggered by system. The sleep time of BGGC is 10 ms. We measure 800 s BGGC to trigger more GCs because the number of the 600 s FGGC is relatively large.

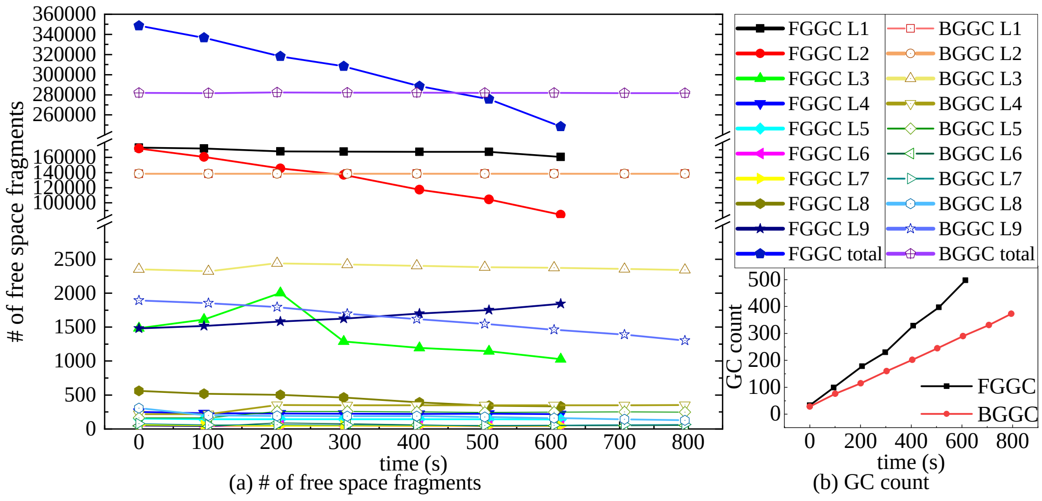

Fig. 9. Changes in the number of free space fragments with GCs

As the number of FGGC increases, the number of free space fragments for L1, L2, L3, L8, and total decreases obviously in Fig. 9, while the number of L9 fragments increases. By contrast, as the number of BGGC increases, the number of L9 fragments decreases, the number of other fragments and the total number do not change significantly. The effect of energy-intensive BGGC on reducing free space fragments is limited based on changes in the number of free space fragments. We calculate the theoretical value of energy consumed by GC using the number of GCs in Fig. 9 (b) and the energy consumption of one GC in the full fragmented state in Fig. 8. The energy consumption of the 600 s FGGC is 261.36 J and the energy consumption of the 800 s BGGC is 73.68 J. In fact, the energy consumption of the 600s FGGC is slightly smaller because its trigger frequency is lower than that in Fig. 8. The energy consumption of FGGC is large and its effect on reducing free space fragments is better than that of BGGC. The FGGC of F2FS is only triggered when free space is insufficient. The energy consumption of BGGC is also relatively large. F2FS wakes up the thread to trigger BGGC periodically, but the ability of BGGC with great energy consumption to reduce free space fragments is limited.

In addition, we analyze changes in the capacity of free space with different sizes for the above scenario since GC can obtain continuous free space. The capacity changes of continuous free space with different sizes are shown in Table 5 where 733.79↓ means that the capacity of continuous free space less than 128 KB reduces by 733.79 MB after 600 s FGGC. The capacity of free space of 2 MB and above increases by 694.1 MB after 800 s BGGC so that BGGC can obtain free space.

FGGC reduces the capacity of free space that generally places invalid data less than 128 KB more effectively compared to BGGC. BGGC mainly reduces free space between 128 KB and 1 MB. FGGC and BGGC respectively select segments where free space fragments of these two sizes are located as victim segments.

Table 5. Capacity changes of free space with different sizes after GC (MB)

Size	**[0, 128 KB)**	**[128 KB, 1 MB)**	**[1 MB, 2 MB)**	**2 MB and above**
FGGC	733.79 ↓	165.94 ↓	7.8 ↓	907.53 ↑
BGGC	0.84 ↑	681.11 ↓	13.83 ↓	694.1 ↑

Discussion 5. *The initial goal of file system GC is to obtain continuous free space to serve new I/O requests. An FGGC is more expensive than a BGGC, however, the effect of BGGC on reducing the number of free space fragments is not as good as that of FGGC. We notice that the size of free space mainly reduced by FGGC and BGGC is different. It is advised to improve the energy efficiency of BGGC to reduce free space fragmentation.*

5 REDUCING ENERGY CONSUMPTION

There are usually two ways to reduce energy consumption, i.e., reducing energy consumption directly and improving energy efficiency by increasing the revenue per unit of energy. A large number of research work [32–34] has been devoted to reducing energy consumption of storage systems directly, while there are few methods to improve the efficiency of a unit of energy consumed. File system design and implementation have a significant effect on CPU/disk utilization, and hence on performance and power [35]. Through the analysis of aforementioned experimental results, increased file and free space fragments are important reasons for the increase in energy consumption of the whole system completing I/O tasks. There are a lot of studies to reduce file fragmentation [13–15, 36–38] while there is few work related to reducing free space fragmentation. Compared with the energy consumed by random writing 1 MB of data, the energy consumed by a BGGC can complete random writing about 3 MB of data. The energy consumption of BGGC should not be underestimated. The existing method of F2FS to reorganize free space fragments is garbage collection, but the BGGC with high energy consumption is limited in reducing free space fragments. We want to improve the efficiency of BGGC that recover more free space fragments with the same or less GC and achieve energy-efficient recycling of invalid blocks.

GC must be triggered to obtain free space and ensure the normal working of file system. We observe CPU and UFS energy consumption increase with the number of free space fragments, however, the expensive BGGC reduces free space fragments inefficiently. This inspires us to propose a GC scheme that is aware of the degree of free space fragmentation. The traditional definition regards completely continuous free space as no free space fragments, and does not clearly indicate how much free space is a free space fragment. In addition, there is a lack of a method to measure the degree of free space fragmentation in file system. F2FS urgently needs a method to recover invalid blocks effectively with a small amount of energy consumption. The existing GC strategy only considers the number of valid blocks and segment age while selecting victim segments, and does not consider the degree of free space fragmentation, which may lead to less gains after migrating valid data. For example, there is 1 free space piece of 1 MB on the candidate segment and 256 free space pieces of 4 KB on another candidate segment, both of which will get one idle segment after GC. We choose the latter as the victim segment that GC can better alleviate free space fragmentation. The benefit of reclaiming a segment is greater when there are more and smaller free space fragments on the segment.

5.1 Reassessment of free space fragments

How to distinguish between free space and free space fragment becomes a key issue in selecting victim segments. Since most I/O requests in smartphones are small [39], a relatively **large free space piece** can satisfy most I/O requests that should be regarded as **available free space** and cannot be regarded as a free space fragment. According to the traditional definition of free space fragmentation, only free space is completely continuous to be considered as having no fragmentation. In order to ensure the continuity of free space, GC of the log-structured file system would be aggressive, resulting in enormous GC overheads. We need to define a reasonable size of free space fragment and consider free space fragments during GC to defragmentation and ensure user experience.

There are three segments have the same number of valid blocks but different degrees of free space fragmentation as shown in Fig. 10. No matter which segment is selected as the victim segment, migration overheads are the same, i.e., migrating 8 valid blocks. If an I/O request that writing a 16 KB file comes before GC, it would be split into four I/O requests in segment A and two I/O requests in segment B. Only one I/O request is required in segment C because its free space is 16 KB in size. Since the continuous free space in segment C is large enough to satisfy a lot of requests with one I/O, its free space cannot be called fragmented. Segment C does not gain much as a victim segment. We need to reassess the size of a free space fragment.

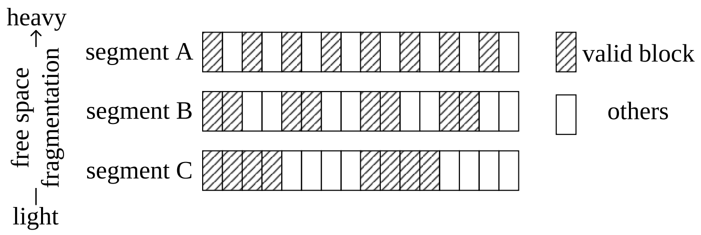

Fig. 10. Different degrees of free space fragmentation

We define the free space less than 128 KB as a free space fragment considering the following three aspects. (1) Setting 128 KB as the threshold is because that 128 KB is the default size for read ahead in Linux kernel. Pre-read data can be placed in a free space of 128 KB and a read request can be satisfied by one I/O instead of being split into multiple I/O requests. (2) After analyzing traces of launching applications, we find that there are a large number of 64 KB synchronous and asynchronous reads that free space of 128 KB can satisfy these read requests with one I/O. (3) We use MobiBench [39] to capture traces of third-party and system applications used in a day and analyze the number of reads and writes of different sizes. Read and write I/Os less than or equal to 128 KB account for 85% and 87%, respectively. We also count the large requests specially in some applications. Only 2% of requests in Baidu Maps are 512 KB. The 128 KB requests in Gallery, QQ, and Taobao account for 4%, 2%, and 6%, respectively. In other words, I/O requests smaller than or equal to 128 KB account for the majority, and I/O requests larger than 128 KB are rare in applications. I/O requests in smartphones are mostly small. Since that a free space of 128 KB can satisfy most requests with one I/O, 128 KB could be the threshold for free space fragment. BGGC mainly reduces the free space between 128 KB and 1 MB as shown in Table 5, but free space of these sizes can satisfy most I/O requests which cannot be regarded as free space fragment. It is expected that GC reduces free space fragments smaller than 128 KB. It is intuitive to choose the segment with more severe free space fragmentation under same migration overheads.

5.2 Measurement of free space fragmentation

We need a metric to indicate the severity of free space fragmentation like that filefrag [40] reports how badly fragmented a particular file is. We design the free space fragmentation **f**actor (f) to measure the degree of free space fragmentation. The f is calculated with *valid_map* in segment information table where bit 0 means the block is invalid or free. We show the calculation of f in Algorithm 1. The threshold for a free space fragment is 128 KB, i.e., $m = 128$, divided by the size of a block, 4 KB, then the value of threshold is 32. There are two parts to obtain f. (1) Judging. A free space greater than 128 KB, i.e., there is greater than or equal to 32 consecutive zeros in the bitmap, is not a fragment (Line 4-7); (2) Calculating f. There are X consecutive zeros of a free space fragment in the bitmap, denoted as X ($X < 32$). Assuming that there are n free space fragments in a segment, firstly $f = \sum_{i=1}^{n} \frac{1}{X_i}$. The larger f is, the more fragmented the free space is (Line 8-12). Since floating-point arithmetic is not recommended in the kernel, we make f an integer via $f = \sum_{i=1}^{n} \frac{1}{X_i} \times (m-1) \times (m-2)$. Multiplying by $(m-1)$ is for rounding and enlarging by $(m-2)$ times is to distinguish free space fragments of different sizes quickly. The f takes values from 0 to 4,096,512. The worst case of 4,096,512 is a valid block interleaves with a free or invalid block where 256 fragments of 4 KB are stored at intervals. We can use f to calculate the degree of free space fragmentation of file systems other than F2FS. Free space fragments from 1 to 127 KB have unique values of f.

ALGORITHM 1: Calculating free space fragmentation factor

```
Input: $bitmap$, $m$
Output: the free space fragmentation factor of the segment, i.e., $f$
 1: Initialize f, $f \leftarrow 0$;
 2: Convert the segment's bitmap, $bitmap$, from hexadecimal to binary;
 3: while Bit of the segment has not been judged do
 4:     Count the number of free or invalid blocks for a continuous free space, i.e., the number of consecutive
        zeros, $X$;
 5:     if $X \geq \frac{m}{4}$ then
 6:         Do not count as a free space fragment;
 7:         $X \leftarrow 0$, used to count the size of a new free space;
 8:     else
 9:         For X consecutive zero(s), $f \leftarrow f + \frac{1}{X} \times (m-1) \times (m-2)$;
10:         $X \leftarrow 0$, used to count the size of a new free space;
11:     end if
12: end while
```

Assuming that the size of a segment and a block in Fig. 10 is 64 KB and 4 KB respectively, the f of segment A is $\frac{1}{1} \times 126 \times 127 \times 8 = 128,016$, the f of segment B is $\frac{1}{2} \times 126 \times 127 \times 4 = 32,004$, and the f of segment C is $\frac{1}{4} \times 126 \times 127 \times 2 = 8,001$. Their degrees of free space fragmentation can be distinguished clearly according to f that segment A with the most severe free space fragmentation has the largest f.

5.3 Design of FAGC

How FAGC works is shown in Algorithm 2. The total time (*total_time*) equals to the difference between the maximum and minimum update time. We determine the range of segments to be compared based on a similar age, and obtain the number of candidate victim segments, namely *dirty_threshold*. The weight of age (*age_weight*) is 40 in Cost-Benefit algorithm, we design the weight of f (*f_weight*) is the same as the weight of age. The weight of u is 20 that *age_weight* +

$f\_weight + u\_weight = 100$. The larger the age is, the older the segment is; the larger the f is, the more fragmented its free space is; the larger u is, the larger free space of the segment is; then the smaller migration cost is according to $cost = UINT\_MAX - (age + f + u)$. We choose the smallest cost to obtain continuous free space. When multiple candidate victim segments have the same migration costs, we choose the segment with the largest f. If f is also the same, we select the victim segment in the order of the least valid blocks and the oldest age. Age, f, and u are all normalized that migration cost will not tend to be affected by a single factor. They are multiplied by 10,000 (Line 2-4) to enlarge so that the weighted sum result has a suitable value for the difference between the UINT_MAX (Line 5). UINT_MAX is the largest unsigned integer in the kernel. When selecting the victim segment, FAGC trades off among age, utilization, and free space fragmentation. When migrating valid blocks, FAGC selects the target segment with a similar age to the victim segment and migrates valid blocks to the target segment via threaded logging writes so that migrated valid blocks do not occupy new idle segments. FAGC writes invalid or free blocks in the target segment one by one which also reduces free space fragments of the target segment.

ALGORITHM 2: The algorithm of FAGC

```
Input: $max\_mtime$, $mtime$, $total\_time$, $vblocks$, $f$, $max\_f$, $min\_f$, $segno$, $age\_weight$, $f\_weight$, $min\_cost$,
    the number of iterations $iter \leftarrow 0$, the threshold of candidate segment to look for, $dirty\_threshold$
Output: the victim segment number $min\_segno$
 1: while $iter < dirty\_threshold$ do
 2:     Calculate the age of the candidate segment, $age \leftarrow 10,000 \times \frac{max\_mtime - mtime}{total\_time} \times age\_weight$;
 3:     Calculate the free space in the segment, $u \leftarrow 10,000 \times \frac{512 - vblocks}{512} \times (100 - age\_weight - f\_weight)$;
 4:     Adjust f of the victim candidate segment, $f \leftarrow 10,000 \times \frac{f}{max\_f - min\_f} \times f\_weight$;
 5:     Calculate the migration cost, $cost \leftarrow UINT\_MAX - (age + f + u)$;
 6:     Increase the number of iterations, $iter++$;
 7:     if $cost < min\_cost$ then
 8:         Update the minimum migration cost, $min\_cost \leftarrow cost$;
 9:     else if $cost == min\_cost$ then
10:         Select the victim segment in the order of the largest f, the least valid blocks, and the oldest age;
11:     end if
12:     Update the victim segment number with the least migration cost, $min\_segno \leftarrow segno$ ;
13: end while
14: Select the segment with the same age as the victim segment as the target segment;
15: Migrate valid blocks on the victim segment to the target segment through threaded logging writes;
```

Different from traditional F2FS and ATGC, FAGC selects victim segments with severe free space fragmentation. FAGC only performs GC on critical segments that greatly affect system performance to reduce GCs and make each GC reduce free space fragments as much as possible. Traditional F2FS and ATGC may trigger some GCs which find victim segments with large free space, while in this scenario FAGC would not trigger GC because no suitable victim segments are found.

6 EVALUATION OF FAGC

Li et al. propose a multi-level threshold synchronous write technology to alleviate file fragmentation and design a high-frequency detection background segment cleaning (MWHFB) to reduce GC overheads [41]. The latest ATGC [16] proposed by the Linux community optimizes F2FS GC. We use traditional F2FS and them as comparison schemes and deploy these four schemes in Linux kernel 5.4.147. We implement the prototype of FAGC by 830 lines of C code.

6.1 GC overheads

We imitate the evaluation method of ATGC [16] and write 7,000 dirty segments of three types respectively in the pristine state. The f of segments of type A is small while that of type B is large, and the number of valid blocks on segments of type A and B is the same. Segments of type C are target segments where valid blocks of segments of type A and B are migrated. The proportion of invalid data for this synthetic benchmark is 55.53%. The number of GCs, valid blocks migrated, and free space fragments, and the capacity of free space fragments after running this synthetic benchmark are shown in Table 6.

Table 6. The performance of GC and free space fragmentation

Scheme	GC count	Migrated blocks	# of free space fragments	Capacity of free space fragments (MB)
F2FS	1,934	450,790	3,187	64.65
MWHFB	2,105	479,288	3,010	62.29
ATGC	1,314	296,956	3,045	62.78
FAGC	335	77,434	3,282	63.71

FAGC reduces the number of GCs significantly which is 82.68% and 74.51% less than traditional F2FS and ATGC, respectively. The number of valid blocks migrated by FAGC is reduced by 82.82% and 73.92% compared to traditional F2FS and ATGC, respectively. The number of GCs in FAGC is less than that of traditional F2FS and ATGC, so FAGC cleans up fewer free space fragments. The number of free space fragments of FAGC is 2.98% and 7.78% more than traditional F2FS and ATGC, respectively. FAGC increases the number and capacity of free space fragments by a much smaller margin while FAGC decreases the number of GCs greatly. The capacity of free space fragments in FAGC even reduces by 1.45% than traditional F2FS. This indicates that FAGC still cleans up free space fragments despite its GC reduction. Each GC in FAGC selects the segment with severe free space fragmentation and reduces the free space fragmentation as much as possible. Traditional F2FS uses the logging writes to migrate valid blocks that many new segments are written dirty. MWHFB has the largest number of GCs and valid block migrations, and it has the least number and capacity of free space fragments.

We take the minimum energy consumption of BGGC in the pristine state in Fig. 8, 102.79 mJ per BGGC. Traditional F2FS, MWHFB, ATGC, and FAGC consume 198.8 J, 216.37 J, 135.07 J, and 34.43 J, respectively. FAGC consumes 164.37 J and 100.64 J less than traditional F2FS and ATGC to complete this synthetic benchmark, respectively. On the one hand, FAGC takes the free space fragmentation factor into consideration when selecting objects to be cleared. So that there is no need to migrate some segments that meet the requirements of valid blocks and age for other GC schemes but get less free space benefits after migration. The experimental results show that ATGC has more GC times and more valid blocks to migrate. Other GC schemes always service users to write new idle segments, and there are a large number of small I/Os in the mobile storage system. In this case, the energy consumption of writing 256 KB or 512 KB free space is similar to that of 2 MB free space. As long as it is a task completed with one I/O request. On the other hand, FAGC selects the segment with the most serious degree of free space fragmentation at the same migration cost, which can merge small free space fragments as far as possible and increase the performance-cost ratio of acquiring continuous free space. FAGC can effectively reduce the energy consumption of triggering GC and improve the revenue of recycling invalid blocks per unit energy consumption.

6.2 Time to replay application traces

Response time of application request is an important indicator that affects user experience. The time to replay application traces in the full fragmented state is shown in Fig. 11. For Facebook and Twitter traces with small data volume, FAGC has a limited effect on reducing replay time by reducing GC. For Baidu, WeChat, and Honor of Kings (noted as HoK) traces with large data volume [14], FAGC reduces their replay time effectively. Compared to traditional F2FS, FAGC reduces the time to replay traces of Baidu, WeChat, and HoK by 36.83%, 24.4%, and 40.9% respectively. Since application request response time is reduced, energy consumption over time can be reduced significantly.

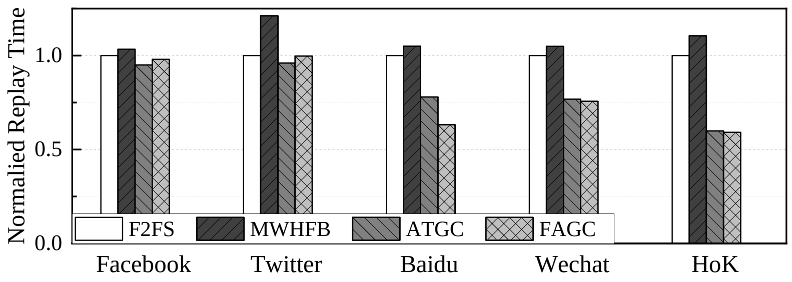

Fig. 11. Normalized time to replay application traces

FAGC can reduce the time to replay application traces effectively at the file system state of high space utilization and a large number of free space fragments. FAGC reduces the number of GCs in that it increases the screening conditions for the degree of free space fragmentation, and does not block regular I/Os from triggering GCs when replaying large application traces. Write requests of large-scale applications in FAGC will not be split due to a large number of small free space fragments or need to trigger GC first in this scenario.

6.3 I/O performance

We want to determine whether FAGC does less GC to achieve the same effect as other GC schemes. We measure the file I/O performance in the full fragmented state to measure the impact of FAGC-reduced GCs. We set *direct* and *sync* I/Os for a 128 MB file and its I/O performance is shown in Fig. 12. We can see that the I/O performance does not decrease significantly while the sequential read performance of FAGC increases by 4.86% compared to traditional F2FS. This indicates that FAGC does not affect the normal read and write performance of files by reducing the number of GCs. This is because that threaded logging writes of FAGC are only used to migrate valid blocks,

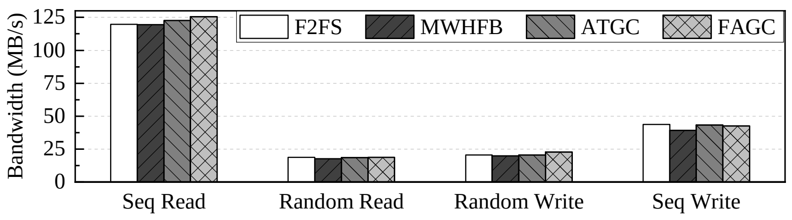

Fig. 12. The I/O performance

and do not affect other data written in a log structure. In addition, FAGC can reduce the number and capacity of free space fragments effectively but with GC lazily, and obtain continuous free space to service write requests of new files.

6.4 Overheads and future improvements

The space overhead of FAGC is to increase a recorded f in the structure, and the calculation overhead is to calculate f when triggering GC to locate a victim segment. The space and calculation overhead are small. FAGC is more effective in reducing GC times and less effective in reducing the number of free space fragments. This is because reducing the number of GC times results in a smaller number of free space fragments to be rearranged by GC. Since FAGC always selects segments with severe fragmentation of free space, the capacity of free space fragmentation is not much different from other strategies.

Combined with application scenarios, we can study methods to directly reduce free space fragmentation in the future. In addition, because it takes a long time to complete GC, it is difficult for us to monitor the reduced energy consumption visually. Currently, we calculate the theoretical value of energy consumption reduction based on GC times. As the tool is upgraded, we will be able to directly observe the reduction in energy consumption. To intuitively reduce energy consumption, we suggest to study the bottleneck of CPU energy consumption and alleviate it. Researchers also need to pay attention to the UFS energy consumption of sequential reads and writes. Although the foreground GC is only triggered when necessary, it consumes much more energy than background GC. We also propose methods for reducing energy consumption of foreground GC directly.

7 RELATED WORK

7.1 Understanding energy consumption.

Carroll and Heiser develop Freerunner's power model and find that the energy consumption of the flash chip is low, but the energy consumed by the components (CPU and RAM) executing flash management is non-negligible [42]. Yoon et al. propose the energy measurement system AppScope which provides detailed information about application energy consumption by monitoring kernel activity requested by hardware components [43]. The storage-intensive microbenchmarks show that storage software can consume 200 times more energy than storage hardware on Android phones and Windows RT tablets. Li et al. establish an energy model, EMOS (Energy MOdeling for Storage) that is a simulation tool to estimate the energy required from storage activities [44]. Olivier et al. propose a three-stage approach to estimate the performance and energy consumption of application I/Os on FFS (Flash File Systems) based storage systems in embedded Linux [45]. The experimental results of Mohan et al. using differential analysis show that in workloads dominated by random I/Os, the storage subsystem on Android smartphones consumes a large amount of energy (36%) [9]. Ferreira et al. present a 4-week study of more than 4,000 people to assess their smartphone charging habits [46]. They describe users' habits of charging their smartphones and the implications on battery life and energy usage. With the upgrading of smartphone hardware and the widespread use of F2FS, we need to update our understanding of mobile phone energy consumption.

7.2 Reducing energy consumption.

Background apps continuously consume the battery power for smartphones, how to balance the application launch delay and battery life becomes a problem. Chung et al. propose an application management framework to terminate unused background apps to save energy and pre-launch beneficial applications to reduce launch delays [47]. Nguyen et al. explore how storage parameters in caches, device drivers, and block layers affect smartphone energy consumption and design a

SmartStorage system [48]. WearDrive utilizes a hybrid low-power network connection of Bluetooth and Wi-Fi on wearable devices to transfer data and computing from the wearable device to the phone. The phone performs energy-intensive tasks while battery-backed RAM performs small energy-efficient tasks locally [49]. Zhong et al. propose Dr.Swap to reduce the energy consumption of DRAM that uses the energy-saving non-volatile memory (NVM) as the Swap area, allowing read requests to read directly from the Swap area [50]. Yan and Fu observe that more than 40% of L2 cache accesses in smartphone applications are kernel accesses. Frequent kernel accesses result in serious interference between user and kernel blocks in the L2 cache. They divide the L2 cache into user and kernel segment. The user segment uses short-reserved STT-RAM and the kernel segment uses long-reserved STT-RAM based on different access behaviors [51]. Yan et al. characterize user behaviors as a sequence of interactions with the same type, analyze their QoS requirements and build corresponding QoS models [52]. They introduce U-ACT, a user behavior aware power management framework, which can select the optimal frequency based on the behavior type and optimize CPU frequencies.

Researchers explore the behavior characteristics and influence of energy consumption of storage systems actively and use high-efficiency devices and technologies to propose energy-efficient solutions. However, we lack an understanding of the energy consumption of modern smartphones based on F2FS and solutions to reduce energy consumption based on this. Our work further understand the principle of energy consumption of F2FS-based storage systems and propose energy consumption reductions.

8 CONCLUSION

Smartphones are powered by batteries, but the size and capacity of batteries are limited. This means that managing energy well is crucial. Efficient energy management requires a good understanding of where and how energy is used. We construct a series of I/O activities to study the impact of the number of fragments and GCs on the energy consumption of the storage system. We observe that free space fragmentation leads to increased energy consumption of CPU and UFS to complete I/O requests. F2FS rearranges free space fragments via GC. We observe that the energy consumption of one BGGC is large, but its effect on reducing free space fragments is limited. We discuss how much free space is a free space fragment based on data analysis, define the free space fragmentation factor to measure the degree of free space fragmentation, propose FAGC to select fragmented victim segments, and use threaded logging writes to migrate valid blocks. The experimental results show that FAGC can reduce the number of GC and improve the energy efficiency of GC to reduce free space fragments. FAGC reduces energy consumption and time to service large application requests.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Program of China under Grant 2021YFB0300101, NSFC (No. U22A2027, 61832020). We thank the anonymous reviewers and editors for their constructive comments and suggestions. We also thank anyone who helped us improve this work.

REFERENCES

[1] Xiangyu Li, Xiao Zhang, Kongyang Chen, and Shengzhong Feng. Measurement and analysis of energy consumption on android smartphones. In *2014 4th IEEE International Conference on Information Science and Technology*, pages 242–245, 2014.

[2] Pijush Kanti Dutta Pramanik, Nilanjan Sinhababu, Bulbul Mukherjee, Sanjeevikumar Padmanaban, Aranyak Maity, Bijoy Kumar Upadhyaya, Jens Bo Holm-Nielsen, and Prasenjit Choudhury. Power consumption analysis, measurement,

management, and issues: A state-of-the-art review of smartphone battery and energy usage. *IEEE Access*, 7:182113–182172, 2019.

[3] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Mohammad Shojafar, Abdelmuttlib Ibrahim Abdalla Ahmed, Sajjad A Madani, Kashif Saleem, and Joel JPC Rodrigues. A survey on energy estimation and power modeling schemes for smartphone applications. *International Journal of Communication Systems*, 30(11):e3234, 2017.

[4] Ding Li, Shuai Hao, Jiaping Gui, and William G.J. Halfond. An empirical study of the energy consumption of android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, 2014.

[5] Chanmin Yoon, Seokjun Lee, Yonghun Choi, Rhan Ha, and Hojung Cha. Accurate power modeling of modern mobile application processors. *Journal of Systems Architecture*, 81:17–31, 2017.

[6] Xiaojing Liu, Fangwei Ding, Jie Li, Haifeng Liu, Zhuo Yang, Juan Chen, and Feng Xia. Phonejoule: An energy management system for android-based smartphones. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 1996–2001, 2013.

[7] Hongjia Wu, Jiao Zhang, Zhiping Cai, Fang Liu, Yangyang Li, and Anfeng Liu. Toward energy-aware caching for intelligent connected vehicles. *IEEE Internet of Things Journal*, 7(9):8157–8166, 2020.

[8] Jian-Bin Fang, Xiang-Ke Liao, Chun Huang, and De-Zun Dong. Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+. *Journal of Computer Science and Technology*, 36:33–43, 2021.

[9] Jayashree Mohan, Dhathri Purohith, Vijay Chidambaram Halpern, Matthew, and Vijay J. Reddi. Storage on your smartphone uses more energy than you think. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 9–15, 2017.

[10] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[11] Saurabh Kadekodi, Vaishnavh Nagarajan, Gregory R. Ganger, and Garth A. Gibson. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–703, 2018.

[12] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[13] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, et al. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[14] Lihua Yang, Zhipeng Tan, Fang Wang, Dan Feng, Hongwei Qin, Jiaxing Qian, et al. Improving f2fs performance in mobile devices with adaptive reserved space based on traceback. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(1):169–182, 2021.

[15] Jonggyu Park and Young Ik Eom. Fragpicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 280–294, 2021.

[16] Chao Yu. f2fs: support age threshold based garbage collection, 2021.

[17] Google. Dynamic system updates, 2020.

[18] Google. Android open source project, 2023.

[19] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–33, 2007.

[20] Jian Liu, Kefei Wang, and Feng Chen. Understanding energy efficiency of databases on single board computers for edge computing. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2021.

[21] Luis Cruz and Rui Abreu. Emaas: Energy measurements as a service for mobile applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019.

[22] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *2013 IEEE International Conference on Software Maintenance*, pages 170–179, 2013.

[23] Eunji Kwon, Sodam Han, Yoonho Park, Young Hwan Kim, and Seokhyeong Kang. Late breaking results: Reinforcement learning-based power management policy for mobile device systems. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2, 2020.

[24] Lihua Yang, Zhipeng Tan, Fang Wang, Yang Xiao, Wei Zhang, and Biao He. Fagc: Free space fragmentation aware gc scheme based on observations of energy consumption. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–2, 2023.

[25] Jens Axboe. fio - flexible i/o tester rev. 3.30, 2017.

[26] Ltd Huawei Technologies Co. Kirin 9000, 2022.

[27] Yu Liang, Chenchen Fu, Yajuan Du, Aosong Deng, Mengying Zhao, Liang Shi, et al. An empirical study of f2fs on mobile devices. In *Proceedings of the 23rd International Conference on Embedded and Real-Time Computing Systems and*

Applications (RTCSA), pages 1–9, 2017.

[28] Cheng Ji, Li-Pin Chang, Sangwook S. Hahn, Sungjin Lee, Riwei Pan, Liang Shi, et al. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 18(9):2062–2076, 2019.

[29] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. Ars: Reducing f2fs fragmentation for smartphones using decision trees. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1061–1066, 2020.

[30] Chao Wu, Cheng Ji, and Chun Jason Xue. Reinforcement learning based background segment cleaning for log-structured file system on mobile devices. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2019.

[31] Chao Wu, Yufei Cui, Cheng Ji, Tei-Wei Kuo, and Chun Jason Xue. Pruning deep reinforcement learning for dual user experience and storage lifetime improvement on mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3993–4005, 2020.

[32] Junghoon Kim, Sundoo Kim, Juseong Yun, and Youjip Won. Energy efficient io stack design for wearable device. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC)*, page 2152–2159, 2019.

[33] Jawad Haj-Yahya, Yanos Sazeides, Mohammed Alser, Efraim Rotem, and Onur Mutlu. Techniques for reducing the connected-standby energy consumption of mobile devices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 623–636, 2020.

[34] Jawad Haj-Yahya, Mohammed Alser, Jeremie Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu. Sysscale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 227–240, 2020.

[35] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*, page 19, 2010.

[36] Jaegeuk Kim. defrag.f2fs relocate blocks in a given area to the specified region, 2015.

[37] Ubuntu Manpage Repository. e4defrag - online defragmenter for ext4 file system, 2019.

[38] Jonggyu Park, Dong H. Kang, and Young I. Eom. File defragmentation scheme for a log-structured file system. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, pages 1–7, 2017.

[39] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, 2013.

[40] Theodore Ts'o. filefrag - report on file fragmentation, 2023.

[41] Qi Li, Aosong Deng, Congming Gao, Yu Liang, Liang Shi, and Edwin H.-M. Sha. Optimizing fragmentation and segment cleaning for cps based storage devices. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, page 242–249, 2019.

[42] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, pages 1–14, 2010.

[43] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application energy metering framework for android smartphone using kernel activity monitoring. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 387–400, 2012.

[44] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, and Qi Zhang. On the energy overhead of mobile storage systems. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 105–118, 2014.

[45] Pierre Olivier, Jalil Boukhobza, Eric Senn, and Hamza Ouarnoughi. A methodology for estimating performance and power consumption of embedded flash file systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4):1–25, 2016.

[46] Denzil Ferreira, Anind K. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In *Pervasive Computing*, pages 19–33. Springer Berlin Heidelberg, 2011.

[47] Yi-Fan Chung, Yin-Tsung Lo, and Chung-Ta King. Enhancing user experiences by exploiting energy and launch delay trade-off of mobile multimedia applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):1–19, 2013.

[48] David T Nguyen, Gang Zhou, Xin Qi, Ge Peng, Jianing Zhao, Tommy Nguyen, and Duy Le. Storage-aware smartphone energy savings. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 677–686, 2013.

[49] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B Nightingale. Weardrive: Fast and energy-efficient storage for wearables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 613–625, 2015.

[50] Kan Zhong, Xiao Zhu, Tianzheng Wang, Dan Zhang, Xianlu Luo, Duo Liu, Weichen Liu, and Edwin H-M Sha. Dr. swap: energy-efficient paging for smartphones. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 81–86. IEEE, 2014.

[51] Kaige Yan and Xin Fu. Energy-efficient cache design in emerging mobile platforms: the implications and optimizations. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 375–380. IEEE, 2015.

[52] Kaige Yan, Jingweijia Tan, and Xin Fu. Improving energy efficiency of mobile devices by characterizing and exploring user behaviors. *Journal of Systems Architecture*, 98:126–134, 2019.

Received xxx 2023; revised xxx 2023; accepted xxx 2023