I/O Characteristics of Smartphone Applications and Their Implications for eMMC Design

Deng Zhou
San Diego State University
Email: zhoud@rohan.sdsu.edu

Wen Pan, Wei Wang
San Diego State University
Email: {pan, wang}@rohan.sdsu.edu

Tao Xie
San Diego State University
Email: txie@mail.sdsu.edu

***Abstract*—A vast majority of smartphones use eMMC (embedded multimedia card) devices as their storage subsystems. Recent studies reveal that storage subsystem is a significant contributor to the performance of smartphone applications. Nevertheless, smartphone applications' block-level I/O characteristics and their implications on eMMC design are still poorly understood. In this research, we collect and analyze block-level I/O traces from 18 common applications (e.g., Email and Twitter) on a Nexus 5 smartphone. We observe some I/O characteristics from which several implications for eMMC design are derived. For example, we find that in 15 out of the 18 traces majority requests (44.9%-57.4%) are small single-page (4KB) requests. The implication is that small requests should be served rapidly so that the overall performance of an eMMC device can be boosted. Next, we conduct a case study to demonstrate how to apply the implications to optimize eMMC design. Inspired by two implications, we propose a hybrid-page-size (HPS) eMMC. Experimental results show that the HPS scheme can reduce mean response time by up to 86% while improving space utilization by up to 24.2%.**

I. Introduction

The storage subsystem of a smartphone normally uses NAND flash memory through either a flash file system [1] or a block file system like Ext4 [2]. Early Android-based smartphones widely used YAFFS2 as their default file system, which is a dedicated file system designed for storing files on raw flash memory. After the release of Android 2.3 (Gingerbread) platform in 2010, the default file system of an Android-based smartphone was switched to Ext4 [3] when the underlying storage subsystem has been changed from raw flash memory to an eMMC device. eMMC has a uniform protocol and provides Ext4 with a conventional block device interface. Its controller locally processes address mapping, wear-leveling, and garbage collection, which largely relieves the burden of the file system above. Fig. 1 shows the I/O stack of a smartphone like Nexus 5. Most of smartphone applications' files and data are managed by the SQLite library, which is the default database management system (DBMS) for an Android-base smartphone. Typically, one I/O activity of an application result in multiple SQLite I/O requests that are served by the virtual file system (VFS) at the kernel layer. Each SQLite I/O request normally invokes a system call function provided by a block file system like Ext4, which further generates I/O requests and sends them to the block layer. The block layer is responsible for scheduling these I/O requests, which are finally dispatched to an eMMC driver. The eMMC driver translates each I/O request into commands that are understood by an eMMC controller, which executes all commands to fulfill the I/O request.

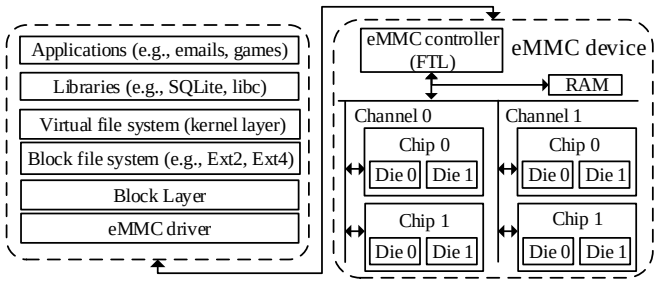


Fig. 1: Android I/O stack.

Similar to a flash memory based solid-state drive (SSD), an eMMC device behaves like a traditional block device as its own flash translation layer (FTL) hides all flash constraints like *erase-before-write*. Due to a smartphone's limitations in space, power, and cost, an eMMC device has a simpler FTL and architecture as well as a smaller RAM buffer compared to an SSD. Thus, the performance of an eMMC device is much lower than that of an SSD [4]. On the other hand, smartphone technologies in both software and hardware have advanced rapidly in recent years. Several mobile operating systems such as Google Android, Apple iOS and Microsoft Windows Mobile have been developed to improve the management of hardware and the support for applications. On the hardware aspect, the performance of an embedded multicore processor becomes comparable to that of a personal computer CPU and gigabyte scale RAM has been introduced [5][6]. These new advances make an eMMC device become a performance bottleneck of contemporary smartphones [7][8]. Obviously, the performance of eMMC needs to be further improved.

To develop a high-performance eMMC device, a better understanding of smartphone applications' block-level I/O characteristics is indispensable. Although a few studies [9][7][4][10] on the storage subsystems of smartphones have been reported recently in the literature, none of them focuses on how to optimize storage subsystem design based on smartphone applications' I/O characteristics. To the best of our knowledge, the only work close to this research is [10], which analyzes the I/O behaviors of 14 Android applications on a Nexus S smartphone. Its major finding is that the combined operations of SQLite and Ext4 generate unnecessarily excessive write operations to the NAND-based storage, which not only degrades I/O performance but also significantly reduces the lifetime of the underlying NAND flash

storage [10]. Thus, it suggests that SQLite and Ext4 need to be optimized in an integrated manner so that redundant efforts are eliminated. Instead of focusing on understanding the interaction across software layers [10], in this research we aim to utilize the implications of block-level I/O characteristics to optimize eMMC design.

To achieve this goal, we first implement an I/O monitor software tool, which enables us to collect 25 block-level I/O traces from 18 applications (i.e., 18 individual traces) and their combinations (i.e., 7 combo traces) on a Nexus 5 smartphone. Next, we quantitatively analyze these traces and discover some interesting smartphone applications' I/O characteristics. For example, we find that in 15 out of the 18 individual traces majority requests (44.9%-57.4%) are small single-page (4KB) requests. In addition, in most traces more than 80% requests can be immediately served once they arrive. Further, several important implications for eMMC design are derived based on the I/O trace characteristics that we find. For instance, an implication of the discovery that majority requests are small requests is that these small requests should be served rapidly so that the overall performance of an eMMC device can be boosted. Finally, we conduct a case study to demonstrate how to apply the implications to optimize eMMC design. Inspired by two implications, we propose a hybrid-page-size (HPS) eMMC scheme. Experimental results show that the HPS scheme can reduce mean response time by up to 86% compared with a conventional pure-4KB-page-size structure. Compared with an existing pure-8KB-page-size architecture, HPS can improve space utilization by up to 24.2%.

The rest of this paper is organized as follows. In the next section, we present the details of trace collecting. Section III analyzes the characteristics of the 25 traces in terms of size and timing. The implications of these characteristics are provided in section IV. Section V presents a case study to illustrate how the implications can be exploited to optimize eMMC design. We briefly summarize related work in section VI. Finally, we conclude this research in section VII.

II. Trace Collecting

In this section, we first introduce the environment setup for trace collecting. Next, we present the design and implementation of an I/O monitor called BIOtracer (**B**lock-level **I/O tracer**), which collects 25 block-level I/O traces. Finally, we explain how we use the 18 common applications in both individual and combined ways so that their block-level I/O activities are recorded.

A. Environment Setup

We use a Nexus 5 as our mobile platform to collect all the traces. Its storage subsystem is a 32 GB SanDisk INAND eMMC 4.51 without an external SDcard. This smartphone has Android v4.4 (KitKat). On the Nexus 5, our customized kernel is installed and we use the default configurations. An I/O record buffer is created to log I/O activities during each trace collecting process and its size is set to 32 KB, which can store about 300 request records. A log file is generated at the beginning of each trace collecting process so that I/O records stored in the I/O record buffer can be periodically flushed into it. For each trace collecting process, the system is restarted and the log file is cleared to minimize the disturbance from other processes.

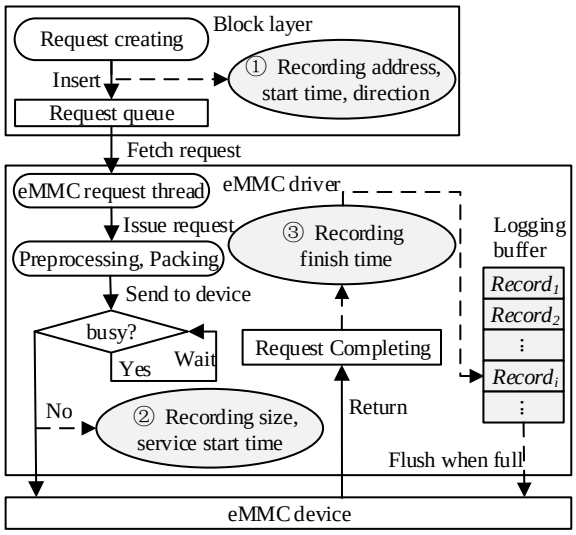


Fig. 2: Workflow of the BIOtracer.

B. The BIOtracer

We implement a trace collector tool named BIOtracer in the Linux kernel 3.4.0 to collect I/O information from both the block layer and the eMMC driver layer (see Fig. 1). Its architecture is shown in Fig. 2. For each request, it records its starting time, logical address, request size, and access type (read or write) at the block layer (see step 1 in Fig. 2). After the request is created at the block layer, it is inserted into the request queue in the block layer. The request is then fetched by the eMMC request thread, which in turn issues it to the pre-processing function after a series of status checking. The pre-processing function transforms the block request into an eMMC request. The packing function merges multiple write requests into a large one if possible.

Finally, the packed request is sent to the eMMC device if it is not busy. Otherwise, the eMMC request thread has to wait until the device is in the free status. BIOtracer records the request's service start time when the request is indeed sent to the device (see step 2 in Fig. 2). When the device driver completes the request, BIOtracer records its finish time (see step 3 in Fig. 2). The timestamps of the request are combined into a record, which is stored into a 32 KB I/O record buffer in the host's RAM (see Fig. 2). When the buffer is full, all records are flushed into a log file on the eMMC device.

C. BIOtracer Overhead Analysis

In this section, we analyze the overhead of the I/O monitor. There are two parts of overhead introduced by the I/O monitor. The first part is its code execution time, which consists of the time for recording timestamps and the time spent for storing them to the logging buffer in main memory. The timestamp recording operations take place in three different layers as shown in Fig. 2. However, at each layer, the I/O monitor only executes a couple of lines of code without peripheral accessing,

TABLE I: Selected applications

Application	Definition
Idle	Smartphone in idle state
CallIn	Answering an incoming call
CallOut	Making a phone call
Booting	Smartphone booting process
Movie	Watching a movie on the smartphone
Music	Listening songs on the smartphone
AngryBirds	Playing the AngryBirds game
CameraVideo	Recording a video clip
GoogleMaps	Road map and navigation
Messaging	Receiving/sending/viewing messages
Twitter	Reading and posting tweets
Email	Receiving/sending/viewing emails
Facebook	Viewing pictures/adding comments/etc.
Amazon	Mobile online shopping
YouTube	Watching videos on the YouTube
Radio	Listening to online radio
Installing	Installing applications from Google Play
WebBrowsing	Reading news on the TIME website

waiting, and thread switching. Thus, this part of CPU time can be safely ignored because the amount of code for timestamp recording is extremely trivial compared with thousands of lines of code for request processing in the kernel.

The second part is the logging buffer flushing cost. In order to obtain a stable I/O pattern, for each trace the I/O monitor collects I/O data for a relatively long period of time including application launching, running, and closing. As a result, the I/O monitor periodically flushes the buffer to the eMMC device. In our trace collecting experiments, the buffer size is set to 32 KB, which can accommodate the records of around 300 requests. Whenever the buffer is full, all data in the buffer are stored to the eMMC device. We observe that a flushing operation always generates 5-7 extra I/O operations (e.g., synchronously opening, appending, and closing the log file). Thus, the number of extra I/O operations caused by the I/O monitor is on average about 6, which is only 2% (i.e., 6/300 = 2%) of the number of normal I/O requests.

D. Application Selection

Among a large number of smartphone applications, we select 14 frequently-used applications. In addition, we collect I/O activities for 4 system functions including CallIn, CallOut, Idle, and Booting. For simplicity, the four system functions are called applications as well. Table I summarizes the 18 applications and Table II shows how these 18 applications are used to generate the 25 traces. The background services of the Nexus 5 smartphone include email and message receiving services and some basic smartphone functions like network connection. Except the CallIn and CallOut applications, all these background services are enabled when an application is running for trace collecting.

III. Trace Analyzing

In this section, we analyze the 25 traces including 18 individual traces and 7 combo traces. While each individual trace comes from a particular application, a combo trace (e.g., Music/WB) is generated by two concurrently running applications such as Music and WebBrowsing. In order to fully understand I/O patterns, we not only study request size, request type, and request locality but also investigate request response

TABLE II: Trace collecting details

Idle *(10pm - 6am)*: Idle status.
Booting *(30 seconds)*: Launching the smartphone.
CallIn,CallOut *(1 hour)*: Mimicking a phone interview including answering, talking, listening, and hanging out.
CameraVideo, AngryBrid, GoogleMaps *(0.5 - 1 hour)*: Recording a video, playing games, driving navigating.
Facebook, Twitter, Amazon, Email, Messaging *(10 - 20 minutes)*: Viewing comments, searching people or items, viewing pictures, and composing replies.
WebBrowsing, YouTube, Radio, Music *(1 - 1.5 hours)*: Reading news, watching online videos, listening radio, and listening music.
Movie, Installing *(10 minutes)*: Watching locally stored movie, installing game applications via WIFI connection.
Combo traces except FB/Msg *(10 minutes to half an hour)*: Using Facebook, Messaging, or Browsing online news while listening Radio or Music.
FB/Msg *(12 minutes)*: Using Facebook, switching to read message whenever a new massage comes, continuing to use Facebook after replying it.

time, service time, and inter-arrival time. In addition, we also analyze the degree of request parallelism.

A. Throughput of eMMC

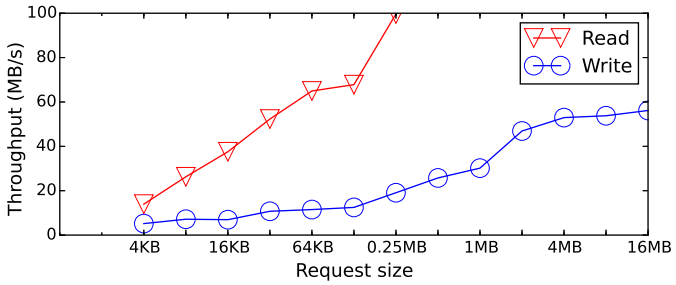


Fig. 3: The impact of request size on throughput.

Fig. 3 shows that request size significantly impacts the performance of eMMC in terms of throughput. In this figure, the throughput of a particular request size is obtained by calculating the average access rate of requests with that size in all traces. In the 25 traces that are collected, the largest size of a read request is 256 KB, whereas the largest size of a write request is 16 MB. That is why the read throughput curve terminates at about 100 MB/s (see Fig. 3). Fig. 3 shows that the read throughput changes from 13.94 MB/s to 99.65 MB/s, whereas the write throughput varies from 5.18 MB/s to 56.15 MB/s when request size increases from 4 KB to 16 MB. When the request size is 256 KB, read throughput achieves its maximum value (i.e., 99.65 MB/s) while write throughput is only 19 MB/s. This is because reading a page in flash memory is much faster than writing a page. We also find that larger size requests (e.g., above 1 MB) result in a higher throughput in both read and write, which is attributed to the packing command at the eMMC driver layer. In addition, Fig. 3 also confirms that the performance of eMMC is obviously lower than that of an SSD, which can achieve 550 MB/s and 520 MB/s on sequential read and write, respectively [11].

B. Size-Related Analysis

The request size-related characteristics of the 25 collected traces are showed in Table III. Fig. 4 illustrates request size distributions of the 18 individual traces. In Table III, the *Data Size* column and the *Number of Reqs.* column store the

TABLE III: Size-related statistics of the 25 traces

Application Name	Data Size (KB)	Number of Reqs.	Max Size (KB)	Ave. Size (KB)	Ave. R Size (KB)	Ave. W Size (KB)	Write Reqs. Pct.(%)	Write Size Pct.(%)
Idle	123,220	6,932	1,536	17.5	39.5	15.0	88.94	75.41
CallIn	27,300	1,491	1,536	18.0	12.0	18.0	99.93	99.96
CallOut	27,364	1,569	1,536	17.0	10.0	17.5	98.92	99.37
Booting	982,200	18,417	20,816	53.0	61.0	37.5	33.07	23.26
Movie	130,420	4,781	512	27.0	27.5	17.0	5.40	3.37
Music	240,060	6,913	940	34.5	62.5	9.5	52.80	14.48
AngryBrid	94,684	3,215	3,940	29.0	51.0	25.0	84.51	73.12
CameraVideo	2,283,184	9,348	10,104	244.0	38.5	736.5	29.46	88.85
GoogleMaps	197,808	12,603	8,174	15.5	28.5	13.5	86.78	75.90
Messaging	63,668	5,702	128	11.0	23.0	10.5	97.30	94.38
Twitter	187,540	13,807	2,216	13.5	35.5	10.5	88.48	69.86
Email	59,276	2,906	388	20.0	14.5	22.5	70.37	78.62
Facebook	97,436	3,897	2,680	25.0	28.5	23.5	74.42	70.70
Amazon	67,412	3,272	1,392	20.5	24.5	18.0	63.02	55.07
YouTube	28,692	2,080	1,536	13.5	19.5	13.5	97.50	96.46
Radio	115,972	5,820	11,164	19.5	36.0	19.5	98.68	97.59
Installing	1,653,900	17,952	22,144	92.0	22.0	93.0	98.26	99.58
WebBrowsing	95,908	4,090	1,536	23.0	21.5	23.5	80.71	81.95
Music/WB	289,280	12,603	1,544	21.5	50.5	15.0	81.68	57.36
Radio/WB	269,932	5,702	2,716	22.5	29.0	19.5	72.02	63.65
Music/FB	442,388	13,807	2,424	12.5	38.0	8.5	87.67	62.34
Radio/FB	153,776	2,906	1,368	14.5	23.0	13.5	91.68	86.92
Music/Msg	234,000	3,897	472	14.0	56.0	11.5	94.43	77.96
Radio/Msg	150,344	3,272	1,536	13.5	17.5	13.0	98.15	97.55
FB/Msg	182,632	2,080	732	11.5	21.5	9.5	84.72	71.72

total size of data accessed and the total number of requests, respectively. The *Max Size* column records the largest request size found in a trace. The *Write Reqs. Pct.* column records the percentage of write requests in each trace while *Write Size Pct.* shows the percentage of written data amount to the total size of data accessed.

Due to the packaging command, the largest requests in most traces are larger than 512 KB, which is the largest allowed size for a request in Linux kernel. In addition, the *Ave. Size* field shows that data- intensive applications (e.g., CameraVideo and Installing) have a much larger average request size than others. Most of the 18 individual applications have an average request size from 11 KB to 34.5 KB.

The values in the last two columns in Table III indicate that most of the traces are write-dominate except Booting, Movie, and Music. During booting, many read requests are issued to load the program and configuration files so that a smartphone is initialized. Music and Movie also need to fetch media data from the eMMC device and thus generate a great number of read operations. CallIn and CallOut generate very few read operations (less than 1% and 2%, respectively). This is due to the fact that when a user is calling somebody all other activities will be in a pending status except a few logging file updates. AngryBrid continuously updates its logs and playing statuses, which generates a great deal of write operations. In addition, CameraVideo has a 88.85% of data written while its write request percentage is only 29.46% because it mainly issues large and sequential requests, which may further be packaged at driver level. Online applications such as Email mainly fetch data from the Internet. In order to update logs and cache the data to the eMMC device, these applications normally issue a large amount of write operations.

Characteristic 1: Most smartphone applications are write-dominant. The percentages of write requests in 15 out of the 18 individual traces vary from 52.8% to 99.9%, among which 6 of them exceed 90%.

The request size distributions of the 18 applications are shown in Fig. 4. Requests are categorized into different ranges (e.g., smaller than or equal to 4 KB) based on their sizes. Although a range represents a continuous region, the possible sizes can only be multiples of four because all the request sizes are aligned to flash page size (i.e., 4 KB) at file system level. Fig. 4 shows that small requests take major part of the total requests in 16 out of the 18 individual traces except Movie and Booting. Still, applications like Booting and Movie generate a significant number of large requests. The general trend is that small requests are majority and the numbers of large requests are generally few. Applications related to the Internet (i.e., the last 10 applications in Fig. 4) show similar distributions and generally follow the same trend. We find that data-intensive traces have their unique request size distributions while others share a similar pattern. For example, Movie has a large mount of requests with sizes between 16 KB and 64 KB.

Characteristic 2: Small size requests take a significant percentage of the total number of requests in most applications. In 15 out of the 18 individual traces, majority requests (44.9%-57.4%) are small requests (i.e., 4 KB).

C. Timing-Related Analysis

In this section, timing-related I/O characteristics of the 18 applications are summarized in Table IV, Fig. 5, and Fig. 6. In Table IV, the values in the *Recording Duration* field are measured from the start time to the finish time of each trace. The duration of Booting is decided by system performance, whereas the duration times of other applications are determined by a user. Although the I/O pattern of an application can be largely impacted by a user's habits, we ensure that for each application its duration time is long enough so that a stable I/O pattern can be recorded.

While the arrival rate is taken as the number of requests arrived per second, the data access rate is defined as the

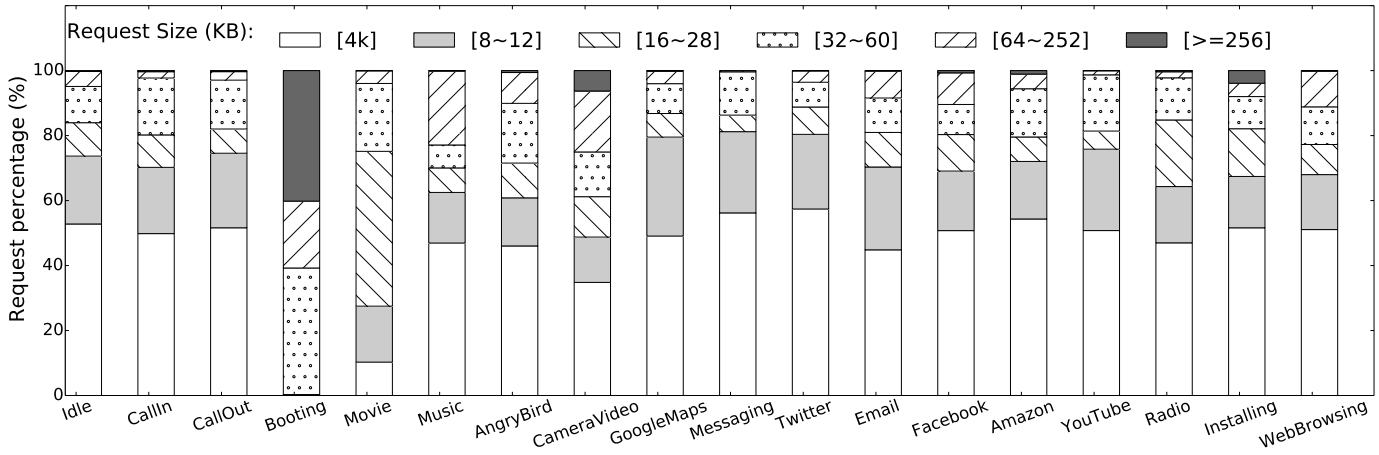


Fig. 4: Request size distributions.

TABLE IV: Timing-related statistics of the 25 traces

Application Name	Recording Duration (second)	Arrival Rate (Reqs./s)	Access Rate (KB/s)	NoWait Req. Ratio	Mean. Serv. (ms)	Mean. Resp. (ms)	Spatial Locality (%)	Temporal Locality (%)
Idle	29,363	0.24	4.20	89	7.42	9.24	25.32	34.22
CallIn	3,767	0.40	7.25	98	5.61	6.18	29.59	31.00
CallOut	3,700	0.42	7.40	94	5.57	6.07	27.29	35.14
Booting	40	460.40	24,555.00	58	1.65	4.93	28.19	19.70
Movie	998	4.79	130.68	23	2.13	6.28	17.25	1.72
Music	3,801	1.82	63.16	64	2.38	3.45	21.51	31.86
AngryBrid	2,023	1.59	46.80	84	3.44	4.06	30.08	26.07
CameraVideo	3,417	2.74	668.18	47	8.07	11.61	20.34	16.30
GoogleMaps	1,720	7.33	117.76	85	1.40	2.23	21.10	42.78
Messaging	589	9.68	108.10	86	1.68	1.88	28.85	50.82
Twitter	856	16.13	219.09	84	1.72	2.07	26.57	52.90
Email	740	3.93	80.10	63	3.01	4.09	14.49	34.87
Facebook	1,112	3.50	87.62	69	2.99	4.08	19.89	34.21
Amazon	819	3.90	84.29	73	1.45	4.70	17.79	26.38
YouTube	4,690	0.44	6.12	96	6.90	7.19	47.61	16.35
Radio	4,454	1.31	26.04	82	3.54	6.62	23.90	29.18
Installing	977	18.37	1,692.84	80	3.64	10.04	22.59	49.57
WebBrowsing	4,901	0.83	19.57	79	4.33	5.20	23.77	30.83
Music/WB	2,165	6.10	133.62	65	1.70	3.61	18.40	38.40
Radio/WB	1,227	9.78	219.99	69	1.86	3.30	18.66	28.48
Music/FB	2,026	17.34	218.36	70	1.13	2.09	14.19	60.50
Radio/FB	900	11.66	170.86	78	1.64	2.58	19.12	52.70
Music/Msg	926	17.82	252.70	74	1.36	2.19	20.68	53.84
Radio/Msg	660	16.82	227.79	89	1.63	2.04	27.25	49.48
FB/Msg	699	22.32	261.28	72	1.23	1.90	15.80	54.04

average data amount accessed (i.e., both read and write) per second. The CallIn, CallOut, Idle, and YouTube show a very low request arrival rate (e.g., fewer than one request per second) and data access rate (e.g., less than 10 KB/s). Booting only lasts 40 seconds. Still, it generates a large number of read requests. It has the highest request access rate and data access rate. The Installing trace also shows a high request arrival rate and data access rate because an installing process incurs software downloading and installing. Together with the average size in Table III, we can find that the applications with higher data access rates (e.g., Booting, CameraVideo, and Installing) also have larger request sizes. Further more, we find that 15 out of the 18 individual traces have an arrival rate less than 10 requests per second. The *NoWait Req. Ratio* represents the percentage of requests that do not need to wait when they arrive. The implication is that these requests can be immediately served because there is no ongoing request that is being served. Table IV shows that at least 63% of requests in 15 out of the 18 individual traces belongs to this category and 10 out of the 18 individual traces have more than 80% requests that can be served immediately.

Characteristic 3: Most of requests can be served immediately once they arrive. In other words, there are very few simultaneously arriving requests.

The data in *Mean Serv.* (i.e., mean service time) and *Mean Resp.* (i.e., mean response time) in Table IV indicate that the requests from Booting, Movie, Amazon, and Installing have a longer request queue for they have a higher ratio of mean response time to mean service time. Combined with the fact that Amazon is not a data-intensive application (see Table III), we can conclude that Amazon has a different I/O pattern from others. In addition, we notice that the request arrival rates of Idle, CallIn, CallOut, YouTube, and WebBrowsing are lower than 1 request per second. They also have higher mean response times and mean service times compared to

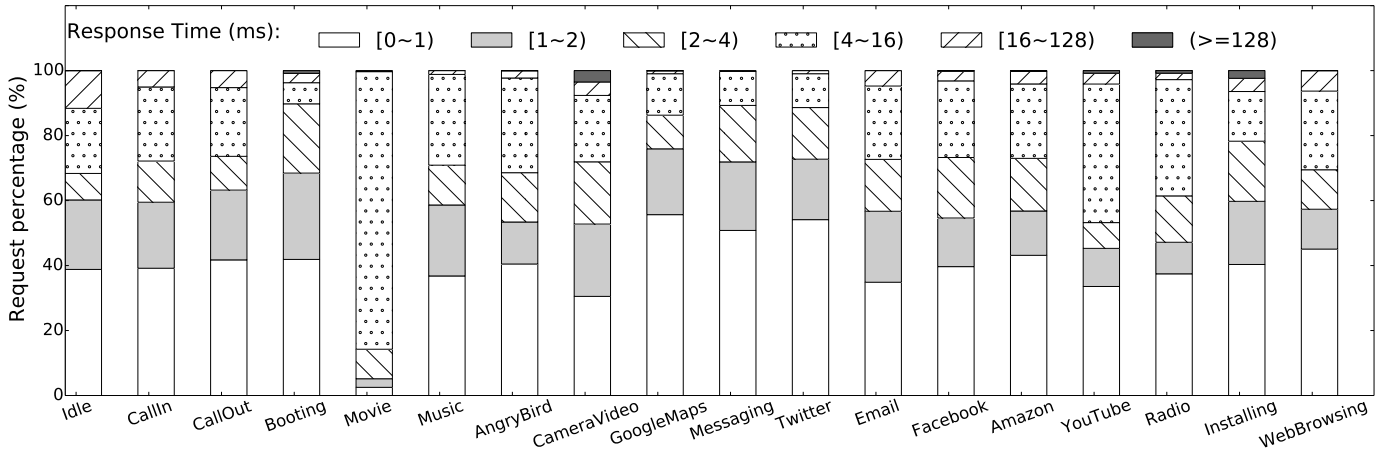


Fig. 5: Request response time distributions.

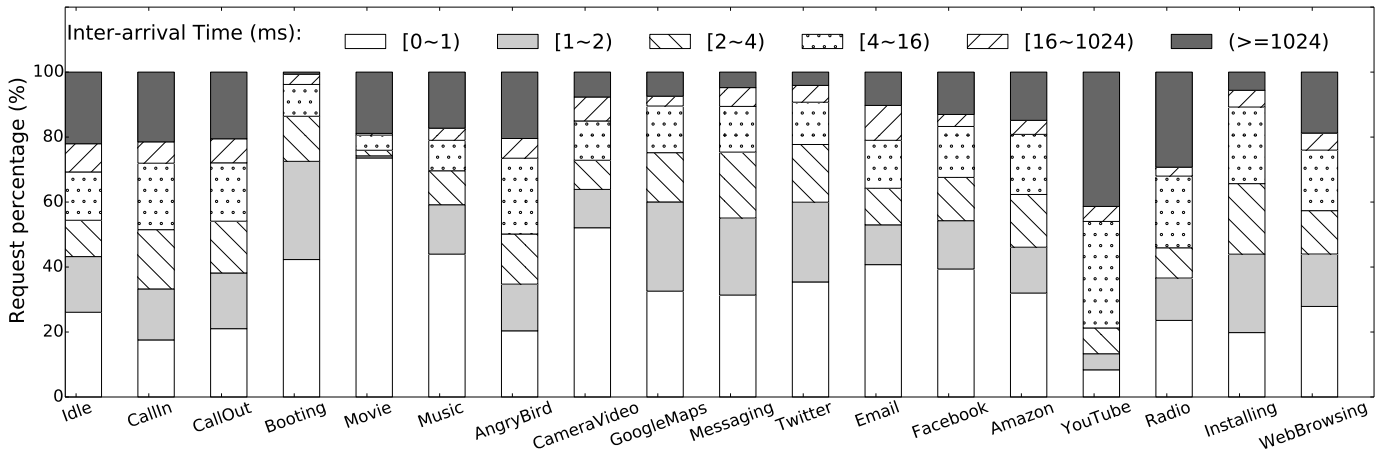


Fig. 6: Request inter-arrival time distributions.

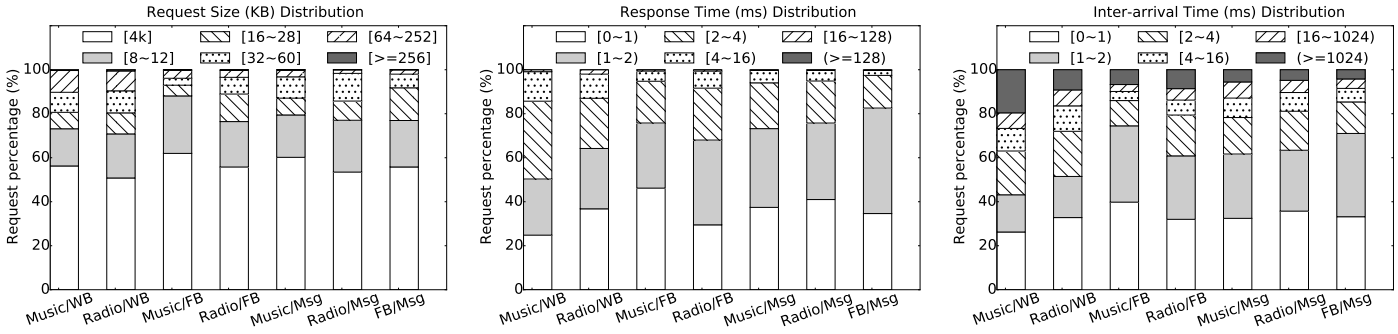
others (e.g., Music, Email, Facebook). This is because that the eMMC device will enter into a low-power mode when no request arrives for a certain period of time. Consequently, a relatively long warm-up time is needed for the eMMC device to serve a newly arrived request. In most of other traces, the values of mean response time are about two times of mean service times. This fact together with the high *NoWait Req. Ratio* values imply that the time spent at the software layer is significant and thus should be further optimized.

Characteristic 4: An eMMC device will enter into a low-power mode if the request inter-arrival time is longer than its power-saving threshold. Thus, in some applications periodic mode switching may happen. Frequent mode switching, however, increases request mean response times.

The last two columns record the localities of the 18 individual traces. Spatial locality is defined as the percentage of sequential request accesses over the total number of requests in a trace. A sequential request access happens when the starting address of the current request is next to the ending address of its predecessor. Temporal locality is the percentage of the number of address hits out of the total number of requests. The number of address hits is increased by one when an address is re-accessed. In terms of spatial locality, 16 out of 18 individual traces have a spatial locality lower than 30% and all the 18 traces have spatial localities lower than 48%. Compared to spatial locality, the temporal locality are slightly better in general (e.g., 11 out of the 18 traces have a temporal locality ranging from 30.83% to 52.9%). However, all these localities are relatively low compared to the 20/80 rule of locality that has been observed in some server-class applications.

Characteristic 5: The localities of the 18 applications are generally weak. In addition, the spatial localities are lower than temporal localities.

Fig. 5 provides some information about response time that cannot be observed from Table IV. The overall trend in Fig. 5 is that most requests can be completed within 2 *ms*. In addition, a vast majority of requests can be processed within 16 ms. There are few long-response-time (i.e., above 128 *ms*) requests existed in the 18 traces. We find that the response time distributions are strongly correlated to the request size distributions. The high correlation indicates that the response time of a request is largely determined by its size, which



(a) (b) (c)

Fig. 7: I/O patterns of the 7 combo traces.

further implies that there are few requests waiting in the request queue. For instance, Movie shows a high percentage (i.e., over 65%) of requests whose sizes are between 16 and 64 KB. It has nearly 70% response times in the range from 4 *ms* to 8 *ms*.

Fig. 6 offers the details of the inter-arrival time distributions. Large inter-arrival times in CallIn and CallOut indicate that there are fewer I/O activities during a phone call process. WebBrowsing, YouTube, and Radio also show that they do not heavily stress the eMMC device as they have larger inter-arrival times. In addition, most inter-arrival times are smaller than 1 ms when watching a movie (see Fig. 6). This figure further supports the claim that Internet-related applications have similar I/O patterns as they show a similar distribution of inter-arrival times. Local applications (e.g., Booting, Movie, Music, CameraVideo) exhibit smaller inter-arrival times compared with online applications like YouTube and Radio.

Characteristic 6: The average request inter-arrival times are long in most applications. 13 out of the 18 applications have an average request inter-arrival time at least 200 *ms*. In 10 out of the 18 traces, more than 20% inter-arrival times are larger than 16 *ms*.

D. I/O Patterns of Concurrent Applications

In most cases, a smartphone user normally runs one application (e.g., YouTube) at a time with background services enabled. Still, it is not uncommon for a user to launch multiple applications concurrently. For example, a user could read news on a website (i.e., WebBrowsing) while listening to music (i.e., Music). Another example is that while a user is using Facebook (i.e., Facebook) he may suddenly switch to read an incoming message (i.e., Messaging). And then he could continue to use Facebook after he replies the message. Therefore, in addition to collecting the 18 individual traces, each from one of the 18 applications, we also gather 7 combo traces from 7 common application combinations in order to understand their I/O patterns. The 7 application combinations include {Music, Radio} $\times$ {Facebook (FB), Message (Msg), WebBrowsing (WB)} as well as task switching between Facebook and Messaging (i.e., FB/Msg). Music and Radio are selected because they are popular companions of other applications such as WebBrowsing. Fig. 7 shows the characteristics of the 7 combo traces in terms of request size and timing. Fig. 7a illustrates that Music-included combo traces have a higher percentage of 4 KB size requests compared to Radio-included combo traces. However, the overall request size distributions of the 7 combo traces are similar to that of the 15 individual traces(see Fig. 4). On the other hand, Fig. 7b indicates that the response times of a combo trace do not obviously increase compared with an individual application case (see Fig. 5). For example, Music/WB has an average response time of 3.61 *ms* while Music alone has a 3.45 *ms* and WebBrowsing has a 5.2 *ms* average response time. In addition, the inter-arrival times of the 7 combo traces are generally large as all of them have more than 20% of inter-arrival times longer than 4 *ms* except Music/FB (see Fig. 7c).

The average inter-arrival times of the 7 combo traces range from 44.8 *ms* to 164 *ms*. After comparing data under request arrival rate and data access rate (see Table IV columns 3 and 4), one can find that generally a combo trace exhibits a higher value in these two items than the sum of the two individual traces due to limited shared resources like memory buffer. For example, while the access rate of Music/FB is 218.36 KB/s, the sum of access rates of Music (i.e., 63.16 KB/s) and Facebook (i.e., 87.62 KB/s) is only 150.78 KB/s. However, the high values of *NoWaitReq.Ratio* (see Table IV column 5) hints that a parallel processing mechanism for multiple requests that arrive at the same time is still not necessary for concurrent applications. In summary, the 7 combo traces from application combinations confirm that the I/O characteristics of smartphone applications are generally stable even when multiple applications are running simultaneously.

IV. Implications for eMMC Design

Based on the 6 characteristics, we are in a position to provide eMMC designers with the following implications:

Implication 1: More than 80% requests can be immediately served once they arrive in most traces, which implies that very few requests arrive simultaneously (characteristic 3). The implication is that enhancing parallelism on device level (e.g., using an external SDcard) or providing parallel request queues at OS layer does not help for performance improvement. This is because the performance of an external SDcard on a smartphone is obviously lower than that of

an internal eMMC device. For example, we observe that the performance of the eMMC on the Nexus 5 is roughly triple of the best performance tested in [7] from 8 SDcards provided by main stream manufacturers. For most traces, using an external SDcard could unexpectedly degrade overall performance because the slower external SDcard negatively affect the overall performance when the internal eMMC device alone can process most requests in time. Reducing large-size requests' service times can significantly improve the overall performance. Further, Characteristic 3 also implies that the response time of a request is mainly decided by its service time. This fact indicates that large-size requests have longer response times because existing eMMC devices normally have a very limited number of channels. For example, the latest eMMC product from SanDisk only provides two channels [12]. As a result, multiple sub-requests (e.g., more than 2 sub-requests) split from a large-size request cannot be processed in a complete parallel manner.

Implication 2: FTL in eMMC needs to be tailored to match the I/O characteristics of smartphones (from characteristics 3 and 6). We find that 13 of 18 traces have an average request inter-arrival time more than 200 milliseconds, which is long enough for a garbage collection process to complete. Thus, garbage collection mechanism in the FTL should be redesigned so that garbage collections are launched during the execution of these non-data-intensive applications. In this way, users cannot perceive performance degradation due to garbage collection. In an SSD FTL, a garbage collection is normally triggered when the number of free blocks reaches a predefined threshold. FTL for eMMC should not adopt the same strategy as there are plenty of opportunities to carry out garbage collections before the number of free blocks becomes significantly low.

Implication 3: We observe that both the temporal locality and spatial locality are weak in almost all traces (from characteristic 5). For instance, the spatial localities are less than 30% and the temporal localities are less than 40% in most traces (see Table IV). Therefore, a large size RAM buffer inside an eMMC device may not be beneficial for performance optimization because of a low hit rate.

Implication 4: From characteristic 5, we also notice that 14 of 18 traces have a temporal locality under 40% and the spatial locality of 16 traces is below 30% (see Table IV). The low localities indicate that I/O requests at the eMMC device level tend to access different locations on flash memory. Therefore, we argue that a simple wear-leveling strategy is sufficient for an eMMC device.

Implication 5: Characteristic 2 discloses that in 15 out of the 18 traces majority requests (44.9%-57.4%) are small single-page (4 KB) requests. The implication is that serving the large amounts of small requests quickly could improve the overall performance of an eMMC device. One feasible way to better serve these small requests is to use SLC (single-level cell) flash, which has a better read/write performance and higher price than that of MLC flash. Fortunately, an MLC flash cell can work in the SLC mode by selectively using its fast pages, and thus, obtains an SLC-like performance [13][14]. Thus, the performance gain is achieved at the cost of 50% capacity loss.

V. Case Study

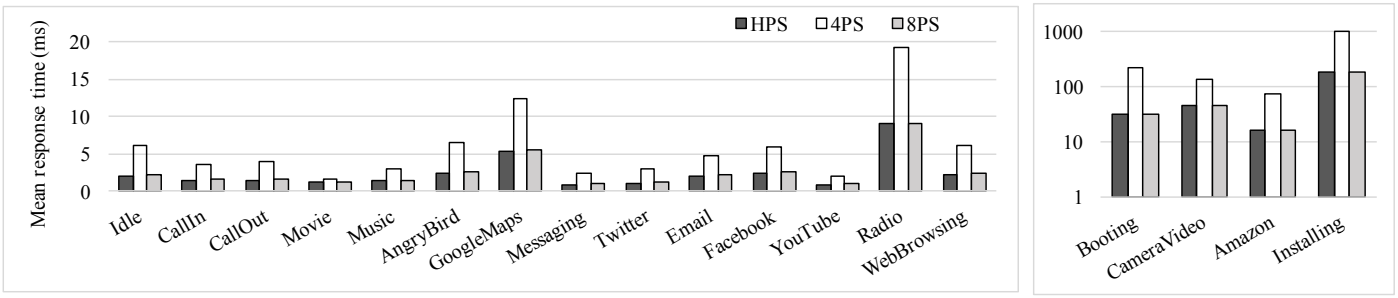
In this section, we conduct a case study to demonstrate how to utilize the insights provided by the implications to optimize the design of an eMMC device. Implication 1 suggests that speeding up large-size requests service times can significantly improve the overall performance of an eMMC device. Clearly, an eMMC device with a large page size is desirable as it can efficiently process large-size requests. In fact, modern SSDs tend to use a larger flash page size [15]. On the other hand, implication 5 suggests that small-size requests should be also quickly served because they are major requests in most smartphone applications. However, an eMMC device with a large page size is inappropriate to process small-size requests as it may degrade both the performance and lifetime of an eMMC device. For example, when majority requests in a trace are random 4 KB writes the performance of a large page size (e.g., 8 KB) eMMC could be lower than that of a small page size (e.g., 4 KB) eMMC because writing a 4 KB data to an 8 KB page takes a longer time than writing it to a 4 KB page. Besides, when the two eMMC devices have the same total capacity (e.g., both 32 GB) the 8KB-page-size eMMC has a much fewer number of pages compared with the 4KB-page-size eMMC. Thus, it will have more garbage collection (GC) operations after its limited number of free pages are quickly consumed by the small random write requests. More GC operations further lowers the performance and shrinks the lifetime of the device. Therefore, a small-page-size eMMC device is needed for processing small-size requests. Thus, the Implication 1 together with the Implication 5 motivate us to propose a hybrid-page-size (HPS) eMMC to effectively serve both small-size and large-size requests simultaneously. The basic idea of HPS is that all blocks in an eMMC have the same number of pages (e.g., 1,024 in our experiments) and all pages in a block have the same size (e.g., 4 KB). However, page size may vary across different blocks in a die (see Fig. 1). To the best of our knowledge, an HPS eMMC device does not exist. Therefore, simulation becomes the only way to verify its effectiveness.

TABLE V: Configurations of the three eMMC devices

	4PS	8PS	HPS
Page read latency (us)	160	244	N/A
Page write latency (us)	1,385	1,491	N/A
Block erase latency (us)	3,800	3,800	3,800
Channel×chip×die×plane	2×1×2×2	2×1×2×2	2×1×2×2
Blocks per plane	1,024	512	512 4KB-page blks + 256 8KB-page blks
Pages per block	1,024	1,024	1024
Total capacity	32 GB	32 GB	32 GB

A. Simulation Setup

Due to its cost limit, an eMMC device normally has no more than two channels. One or multiple MLC flash chips are attached on each channel. Each chip consists of multiple dies and each die has thousands of blocks. Each block have hundreds of pages. Since an eMMC device can viewed as a light-weight SSD, we use a validated SSD simulator called SSDsim [16] to mimic an eMMC device. SSDsim is an event-driven and highly accurate simulator. We implement the HPS scheme in SSDsim so that it can support different page-size blocks in one die. We configure several hybrid-page-size chips



(a) (b)

Fig. 8: Performance comparisons among the three schemes.

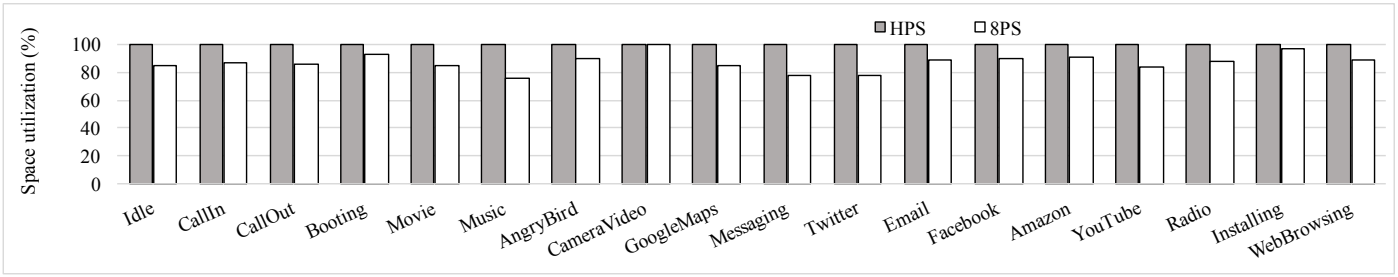


Fig. 9: Space utilization comparisons between 8PS and HPS.

in an eMMC device and a new request distributor is developed to dispatch a read/write request into an appropriate page. In the HPS scheme, two sizes of pages are used (i.e., 4 KB and 8 KB). Latency-related parameters are obtained from Micron datasheets [17][18].

For the HPS scheme, we assume that page size varies among different blocks inside a plane. For example, in the simulator configuration, we assume that there are 512 4KB blocks and 256 8KB blocks inside a plane (see Fig. 10). We also configure a pure 4 KB page scheme (4PS) and a pure 8 KB page scheme (8PS) as two baseline eMMC devices. The detailed configurations of HPS and two pure schemes are shown in Table V. Notice that all three schemes have same number of channels, chips, dies, and planes so that internal parallelism will have same effects to the performance of the three schemes. Based on the configurations, the total capacities of three schemes are also the same (see Table V).

The request distributor splits a request into multiple pages. For a read request, data is retrieved based on a mapping table maintained by the eMMC controller. For a write request, it will be processed in different ways depending on its size. For example, when the size of a write request is 20 KB, it will be divided into two 8-KB sub-requests and one 4-KB sub-request. On a single-page-sized eMMC device, some flash space is wasted. For example, if we use a 8KB-page-size chip, then 3 sub-requests (totally 24 KB) are needed to complete the 20 KB write request. As a result, 4 KB flash space is wasted. The space utilization of the write request is defined as 20/24, which is equal to 83.3%. The space utilization of a trace is defined as the ratio of its total amount of data written to total amount of flash space consumed. Obviously, a higher space utilization indicates a longer lifetime of an eMMC device.

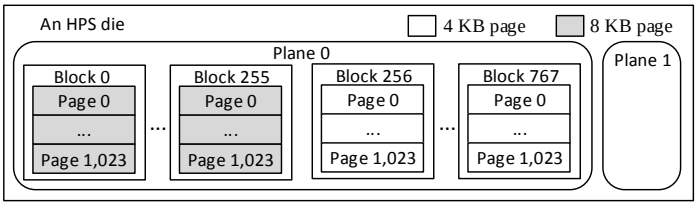


Fig. 10: The structure of an HPS die.

B. Experimental Results

We measure the performance of each scheme in terms of mean response time (MRT) and space utilization. All traces are replayed on a simulated brand new eMMC device. The RAM buffer layer of the simulator is disabled to eliminate its performance impact. Fig. 8 demonstrates that the HPS scheme outperforms 4PS for all the 18 traces. Since MRTs of Booting, CameraVideo, Amazon, and Installing are much higher than that of others, their results are represented in a log scale and separated in Fig. 8b. The rest 14 results are shown in Fig. 8a. Compared to 4PS, HPS obtains its most significant improvement (86%) in terms of MRT in Booting. Even in the worst case, the Movie trace, HPS can still reduce MRT by 24.0%. On average, HPS can achieve 61.9% performance improvement. The 8PS scheme has a very similar performance to HPS in terms of MRT as expected.

Fig. 9 shows the space utilization of the two schemes. We use 4PS as the baseline scheme. The results of HPS and 8PS are all normalized to that of 4PS. Since the HPS scheme always achieves same space utilization as 4PS does, we ignore the results of 4PS in Fig. 9. Compared with the 8PS, HPS

obtains the most significant improvement in terms of space utilization in the Music trace, which is 24.2%. On average, the HPS scheme achieves 13.1% improvement in terms of space utilization compared to 8PS.

VI. Related Work

As smartphones have become more and more popular, understanding the relationship between their storage subsystems and system performance started to draw attention from both industry and academia recently. As of September 2010, Android-based smartphone sales numbered 200,000 per day versus 80,000 per day for iPhone iOS due to the fact that Android is an open-source operating system [19]. Thus, almost all existing work on smartphone storage subsystems has been carried out on Android-based platforms. However, to the best of our knowledge, there are only a few studies investigated Android-based storage subsystems. Kim *et. al.* found that storage performance does affect the performance of several common applications such as Web browsing, GoogleMaps, application install, email, and Facebook [7]. They observed that just by varying the underlying flash storage, performance over WiFi can typically vary between 100% and 300% across applications. Kim and Shin confirmed the conclusion of [7] after investigating the internal features of eMMC. In particular, they studied the effects of LSB backup, packed command, and flex group on Android-based smartphones. They concluded that storage subsystem like eMMC device needs further optimizations [4]. Kim and Ramachandran re-examined the OS storage software stack of smartphone to improve the storage performance [8]. They implemented a framework called Fjord that can provide a fine-grained control mechanism to trade-off reliability for performance. None of [7][8] and [4] focuses on analyzing eMMC level I/O traces and their implications.

Lee and Won analyzed the I/O behaviors of a total of 14 Android applications from six different categories [10]. They focused on the interaction across software layers: applications, Android OS, filesystem, and underlying storage device [10]. They discovered that the operations of SQLite and Ext4 greatly burden the storage as they generate unnecessarily excessive write operations [9]. Although they also analyzed some I/O features like I/O size and randomness as well as their impact on storage device, their main focus lies in understanding I/O activities generated by applications and OS layer, and then, how to optimize these software layers. On the contrary, we concentrate on how to apply the implications derived from I/O characteristics for eMMC design.

VII. Conclusions

The performance of an eMMC device noticeably impacts the performance of smartphone applications [7][8]. Unfortunately, little research has been done in quantitatively analyzing block-level smartphone I/O characteristics and their implications on eMMC design. To understand the smartphone applications' I/O patterns, we implement an I/O monitor tool called BIOtracer and integrate it into Android kernel 3.4 on a Nexus 5. Next, we conduct a comprehensive analysis on 25 traces. Six I/O characteristics have been observed. Next, 5 implications for eMMC design are derived based on the characteristics. Finally, we conduct a case study to demonstrate how to apply the implications to optimize eMMC design.

Acknowledgment

This work is sponsored by the U.S. National Science Foundation under grant CNS-1320738.

References

[1] W. Wang, D. Zhou, and T. Xie, "An embedded storage framework abstracting each raw flash device as an mtd," in *Proceedings of the 8th ACM Interational Systems and Storage Conference*. ACM, 2015, p. 7.

[2] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[3] T. H. open, "Android 2.3 gingerbread to use ext4 file system," 2014, accessed: 2015-4-26. [Online]. Available: http://www.h-online.com/open/news/item/Android-2-3-Gingerbread-to-use-Ext4-file-system-1152775.html

[4] H. Kim and D. Shin, "Optimizing storage performance of android smartphone," in *Proceedings of the 7th Int'l Conf. on Ubiquitous Information Management and Communication*. ACM, 2013, p. 95.

[5] Kenlo, "Ram on smartphones and tablets: Everything you need to know," 2014, accessed: 2015-4-26. [Online]. Available: http://community.giffgaff.com/t5/Blog/RAM-On-Smartphones-amp-Tablets-Everything-You-Need-To-Know/ba-p/7950298

[6] PHONEGG, "Fastest processor," 2014, accessed: 2015-4-26. [Online]. Available: http://us.phonegg.com/top/53-Fastest-Processor

[7] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Transactions on Storage*, vol. 8, no. 4, p. 14, 2012.

[8] H. Kim and U. Ramachandran, "Fjord: Informed storage management for smartphones," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–5.

[9] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/o stack optimization for smartphones," in *Presented as part of the 2013 USENIX Annual Technical Conference*. USENIX, 2013, pp. 309–320.

[10] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the tenth ACM Int'l Conf. on Embedded software*. ACM, 2012, pp. 23–32.

[11] Samsung, "Samsung 850 pro ssd," 2014, accessed: 2015-4-26. [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/SSD/us/html/ssd850pro/overview.html?gclid=CjwKEAjwmfKpBRC8tb3Mh5rs23ASJACWy1QPGj8I7uQtfTq_hMohfl7GF7ecW1b5usExP9lUZ1NtfxoCHk3w_wcB

[12] SanDisk, "Sandisk inand extreme in 2014," 2014, accessed: 2015-4-26. [Online]. Available: http://www.anandtech.com/show/7790/sandisk-inand-extreme-in-2014-finally-a-good-emmc-solution-for-mobile

[13] S. Im and D. Shin, "Comboftl: Improving performance and lifespan of mlc flash memory using slc flash buffer," *Journal of Systems Architecture*, vol. 56, no. 12, pp. 641–653, 2010.

[14] W. Wang, T. Xie, and D. Zhou, "Understanding the impact of threshold voltage on mlc flash memory performance and reliability," in *Proceedings of the 28th ACM Int'l Conf. on Supercomputing*. ACM, 2014, pp. 201–210.

[15] Micron, "Mt29f32g08cba datasheet," 2013, accessed: 2015-4-26. [Online]. Available: http://www.datasheetspdf.com/datasheet/mt29f32g08cba.html

[16] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the Int'l Conf. on Supercomputing*. ACM, 2011, pp. 96–107.

[17] Micron, "Mt29f128g08cbcbb datasheet," 2008, accessed: 2015-4-26. [Online]. Available: http://www.micron.com/~/media/documents/products/data-sheet/nand-flash/80-series/l85c_plus_128gb_256gb_512gb_1tb_2tb_async_sync_nand.pdf

[18] ——, "Mt29f64g08cbaa datasheet," 2009, accessed: 2015-4-26. [Online]. Available: http://www.datasheetarchive.com/dl/Datasheets-IS23/DSA00448784.pdf

[19] M. Butler, "Android: changing the mobile landscape," *Pervasive Computing, IEEE*, vol. 10, no. 1, pp. 4–7, 2011.