Androtrace: Framework for Tracing and Analyzing IOs on Android *

Eunryoung Lim, Seongjin Lee and Youjip Won
Hanyang University, Korea
{erlim | insight | yjwon}@hanyang.ac.kr

ABSTRACT

In this work, we develop IO trace and analysis framework, Androtrace, which is specifically tailored for Android platform. Unlike earlier works that required prolonged post processing procedures, Androtrace not only traces with low overhead, but also provides efficient solution for storage within mobile devices. Captured IO trace is temporarily stored in main memory and storage device, and they are transferred to Androtrace server when the device is connected to WiFi. We use server and client model to support and analyze multiple Android users. Using the framework, we find that write IOs are dominant in mobile workload.

Keywords

Framework, IO Trace Utilities, Android, Real-time, SQLite, Synchronous Write, IO Analysis

1. INTRODUCTION

The performance of IO subsystem is one of the main governing factors for a wide range of computing systems including PC and smartphone [17]. The ideal way of characterizing the IO subsystem is to collect and analyze the IO traces from the real-world environment. For the past several decades, a fair number of works were performed to extensively analyze various characteristics of IOs; [3, 6, 31], OLTP server [23], WEB Server [2, 24], Desktop PC [4, 12, 27, 33], and Mac OS X [11]. A work examined the journaling behavior of different file systems [28] insights in designing file system and storage stack, e.g. LFS, FFS [7, 26, 28]. Despite the popularity of smart devices, our understanding on IO characteristics of smartphone is not as comprehensive as the ones we used to have on the other platforms. This tardiness is partly due to the nature of the device; the smartphone is mobile, personal, and private. Smartphone users are reluctant to share their IO traces because the IO traces, unless they are properly anonymized, certainly reveals the detailed information about the behavior of the user, e.g. the frequently accessed applications, personal preference, to list a few. Also, due to the limited storage capacity and CPU speed, the collected IO traces should be often transferred to the server for storage and analysis. Transferring IO traces can shorten the battery lifespan and consume the monthly data limit or through external memory devices, both of which smartphone users are very sensitive about and not to mention the cumbersome procedures in accessing the data. In addition to that, existing IO trace collecting utilities are not designed for Android platform or long-term IO traces.

A few recent works analyze the Android IO in a controlled environment with synthetic workloads [16, 18, 22, 25]. While these studies reveal important IO characteristics of individual smartphone applications, we still do not know the IO characteristics of the overall real-world smartphone usage. As a prerequisite step to solve the problem, we develop a trace utility. Androtrace is tailored for acquiring and analyzing the IO behavior of Android platforms. In this work, we present the Androtrace and the preliminary analysis result.

The contribution of this paper is as follows:

- **Androtrace Framework:** We develop a low overhead real-time IO tracing and analyzing framework that can collect IOs indefinitely. It is based on server-client model, thus it allows analyzing IO traces of multiple users without having to concern the storage capacity.
- **No More Synthetic Workloads:** We collect IO trace of four users in real environment and verify that all users suffered from heavy write IOs and journaling of journal problem.
- **`fdatasync()`, Heavy Synchronous IOs Generator:** It is formerly known that synchronous IOs are the most important IO characteristics of Android devices, yet the caller of such IOs were unknown. The framework shows that about 56% of all synchronous IOs are issued by `fdatasync()`.

2. MOTIVATION

2.1 Utilities for Tracing and Analyzing IO

Although there are fair amount of utilities [5, 8, 10, 13, 19, 32] available for analyzing the performance of a system, each of them is designed in such a way that the scope of

*Source code is publicly available at `https://github.com/ESOS-Lab/AndroTrace`

INFLOW'15, October 04-07, 2015, Monterey, CA, USA
@ 2015 ACM. ISBN 978-1-4503-3945-2/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2819001.2819007.

the utilities is limited to one or two layers of the system. For example, `strace` [19] captures the trace of system call interfaces, `iostat` [8] measures IO performance exploiting block device interfaces, and `blktrace` [13] captures the block level IO accesses. `blktrace` and `MOST` [21] are used to capture the essence of IO characteristics of given workload. However, the limits in these utilities are that one needs to use separate utilities for post processing and visualization of the data for analysis. For example, `MOST` exploits `debugfs` to find out the file and block types of given IO, and the captured IO trace needs to be manipulated before it can be visualized.

2.2 IO Behavior of SQLite

The use of SQLite is prevalent in not only mobile devices but also in many of the essential desktop applications; some of the famous applications that exploits SQLite include Apple Safari, Dropbox, Firefox, Chrome, and Microsoft Windows 10 [29]. The popularity comes from the fact that it is server-less, light, and reliable. As it is a well-known fact that SQLite causes a lot of IOs overhead for committing its data and journal files to the storage, it can be said that all applications and systems that make use of SQLite and journaling file system suffer from journaling of journal problem [15]. The existing utilities such as `blktrace` and `MOST` do not provide the method to identify the IOs that are caused by SQLite.

2.3 Source of Synchronous IO

Jeong et al. [15], Lee et al. [22], and Kim et al. [18] have shown that Android and Tizen devices suffer from significant overhead of synchronous IO operation. However, what we do not know is the source of such synchronous IOs. For example, `blktrace` cannot properly identify the process name which originally issue an IO on Android because all processes delegate the IO requests to `mmcqd` process. `mmcqd` is a daemon that manages eMMC card device driver in the Android platform. Moreover, existing utilities cannot distinguish the difference between IOs issued by `fsync()` and `fdatasync()` system calls. The IO characteristics of `fsync()` and `fdatasync()` is completely different from each other because `fsync()` enforces to flush data, metadata and journal every time it is called, but `fdatasync()` flushes metadata and journal only when it is necessary.

2.4 Limitation of the Existing Trace Utilities

There exists a number of issues in the existing trace utilities [5, 8, 10, 13, 19, 32] which makes these utilities practically infeasible in collecting traces in smartphone. The first issue is storage capacity. Most of the existing utilities are designed to store the collected IO traces in its local storage. In legacy computing platforms, e,g. server or desktop PC, it is trivial to install additional storage device to store collected trace. In smartphone, it is not equipped with additional storage slot or the external storage, e,g. SDCARD, is much slower than its internal storage, eMMC, where writing the trace log to the external storage can significantly interfere with the application performance. The second issue is post-processing overhead. To properly capture the correlational behavior between the block device level IO accesses and the file system level file access characteristics, we should be able to reverse map the block address to the file name (or inode number) where the block belongs. MOST [21] rely on debugfs [20], reverse mapping utility, to obtain this information. To properly characterize the IO characteristics in Android IO platform, it is mandatory that each block accesses are properly mapped to the original file where it belongs. However, reverse mapping is a very time consuming task. In the existing utilities [13, 21], the overhead of reverse mapping is prohibitively large and cannot be used in mobile device. The third issue is limited connectivity. Existing utilities do not pay much concern on transporting the IO trace to the server. They are designed to process the IOs locally or migrating the trace data to different host is not interfering task since they are connected to network. Migrating the trace data in Android service to the other platform in very cumbersome chore. It practically prohibits the trace collecting utilities from being deployed to wider public.

2.5 IO Trace Utility for Android

The ideal IO trace utilities on mobile devices should at least include following design paradigms in consideration. First, it should be able to distinguish the original process name for IOs. Second, it should be able to identify the file name and file type where the block belongs to. It is practically infeasible to store file type or file name due to its storage overhead. The trace utilities should be able to filter the file name and file type which are determined to deserve attention from trace analysis point of view. Third, the log should contain file system block type information. In EXT4, a block in a file system can be metadata, e,g. inode and bitmap, journal or data. Fourth, it should be able to trace with low memory and IO overhead, as well as low storage space. It also should be able to trace without human intervention and with minimal interference. Fifth, it should be able to collect IO traces from different users and also be able to compare IO behaviors of acquired traces. Sixth, it should be able to guarantee the privacy of user. In this paper, we try to address all of these requirements in building the ideal IO trace utilities and analyzer for mobile devices.

3. DESIGN OF THE FRAMEWORK

The storage capacity of a mobile device is considered as scarce resource when it comes to long term IO tracing. Thus, we decided to transfer the acquired user IO trace over the network to Androtrace server which conducts the analysis and visualization of the IO trace. Fig. 1 shows the architecture of Androtrace. It consists of Server and Client side. In this section, we explore the design of each component in both sides of the Androtrace.

3.1 IO Acquisition

There are three most important things that should be considered in the Collector in Fig. 1, which is the main daemon of Androtrace client for acquiring the IOs. First, what kind of information is Androtrace going to acquire. Second, how should data be stored in the storage so that it would not interfere with user IO streams. Third, the application must provide privacy.

Table 1 shows the list of fields Androtrace acquires, and it also gives a comparison with other IO trace utilities. Androtrace is able to distinguish file name, file type and block type of given IO. It also can distinguish whether corresponding IOs are issued by `fsync()`/`fdatasync()` and SQLite. One of other interesting information Androtrace manages is lifespan of a file. When `unlink()` is called for the file and the link count is zero, then the Collector examines the creation time

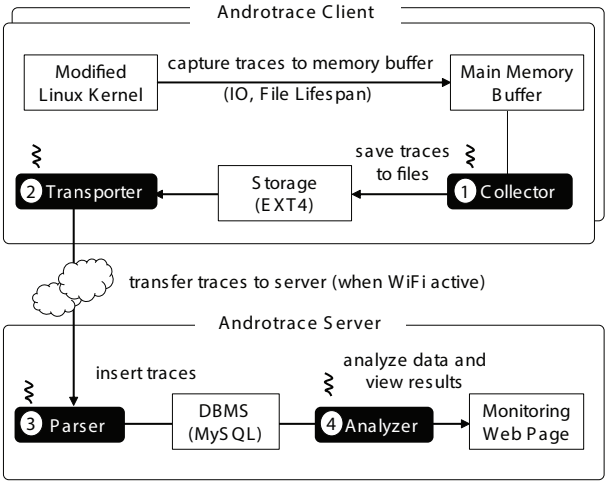

Figure 1: Architecture of Androtrace Framework

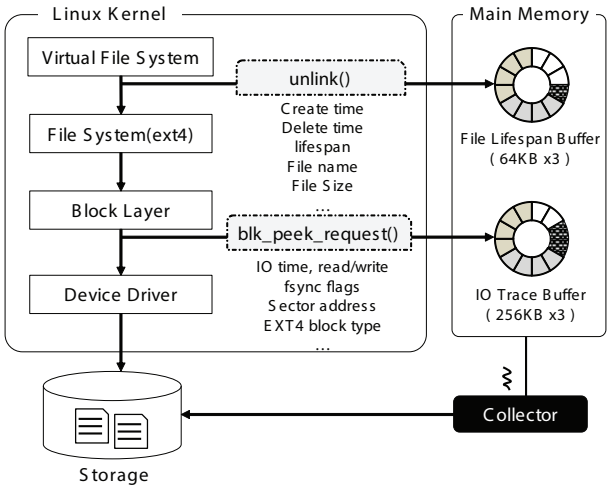

Figure 2: Androtrace Acquisition Process

of the file and computes the lifespan of the file.

Table 1: Feature Comparison: Androtrace, `MOST` [14] vs. `blktrace` [13]. (off-line indicates that the information is produced via off-line processing.)

Information	Androtrace	MOST	blktrace
Time	◯	◯	◯
Operation Type	◯	◯	◯
Device Number	◯	◯	◯
Block Type	◯	off-line	×
Sector Number	◯	◯	◯
IO Size	◯	◯	◯
Process Name	◯	◯	×
File Name	◯	off-line	×
File Type	◯	off-line	×
`fsync`	◯	×	×
`fdatasync`	◯	×	×
SQLite IO	◯	×	×
File Lifespan	◯	×	×

In order to minimize the performance interference with ongoing IOs of applications, it logs the IO trace to memory buffer instead of directly recording it in a file. As shown in Fig. 2, the collector exploits three 64 KB file-lifespan circular buffers and three 256 KB IO trace circular buffers. The size of buffer is determined to avoid log-bottleneck and buffer bottleneck. One of the three buffers for both circular buffers takes a role of active buffer to store the incoming IO trace. When an active buffer is full, the collector closes and flushes collected data to log file while the next buffer becomes an active buffer. Instead of using one large log file, the trace information is maintained at the multiple of small log files (1 MByte by default).

The trace logging mechanism of Androtrace shares its idea with `blktrace` [13]. Androtrace collects the IO trace when the IO scheduler dispatches a single request to the storage device, and it uses `blk_peek_request()` to log the IO trace.

Finally, we acquire written consent from Androtrace users before beginning to collect the trace at the mobile device. Androtrace guarantees the privacy of users by generating MD5 on device ID from the client side of the Androtrace. The hashed value is only used to distinguish different users of the Androtrace.

3.2 Transmission Over the Network

After the Collector stores number of logs and IO trace files, the Transporter in Fig. 1 periodically sends files to the server through air-link. The Transporter is a separate thread to transmit the trace data to the server. There are two options in transmitting the file over network. The first option enforces Androtrace to use only WiFi connection and the second option enforces Androtrace to transfer the data only when the device is connected to a power source and WiFi. This is to avoid using paid bandwidth (e.g., 3G/LTE).

According to our experiments, transferring the log to the Androtrace server is subject to frequent interruption; not only because the air-link is unstable, but also because many of the smartphones have short battery life, e,g. eight hours is the average battery life even if they are on top ten smartphones [30]. Since we believe one of the main sources of power consumption is data network components, frequently issuing the data transmission requests will further reduce the battery life. The overhead may be amplified if packets are lost during the transmission because it has to retransmit the data. To reduce the retransmission overhead, we limit the size of the IO trace file to 1 MB. The log file is removed after it is sent to the server.

3.3 Data Analysis and Presentation

Server of Androtrace uses Apache server as an interface with the client. It uses MySQL DBMS to manage the traces. The salient feature of Androtrace is the real-time monitoring page. The real-time trace statistics are published to the web to provide information on how much read/write IOs a user generates, file and block types, and synchronous IOs are generated on each user. This web page is updated every hour. Fig. 3 illustrates the screen shot of trace statistics web page. This real-time monitoring page allows to understand brief preliminary IO characteristics as a basis for deeper analysis.

4. IMPLEMENTATION

In this section, we explain the details of IO acquisition

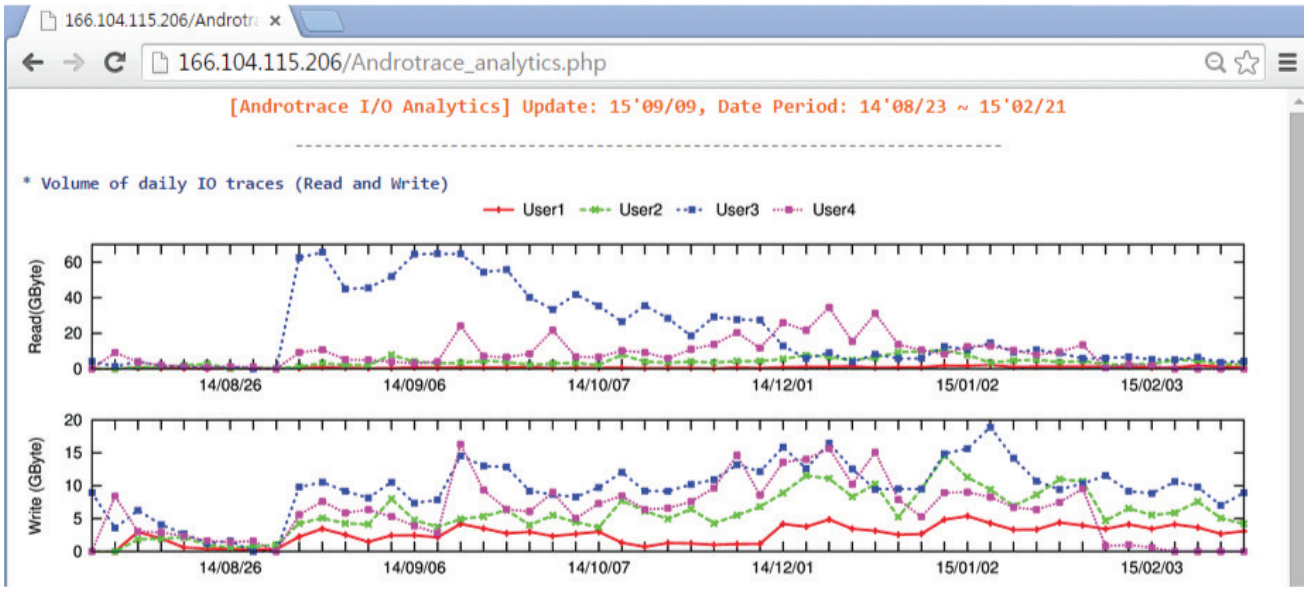

Figure 3: Androtrace monitoring web-page

process in Androtrace. As shown in Table 1, Androtrace keeps account of several fields of information that are not readily available in other utilities. Androtrace is based on Linux Kernel Ver. 3.4.0 and Android 4.4 KitKat platform. We augmented four kernel data structures, namely `bio`, `address_space`, `transaction_s`, and `inode`. Fig. 4 illustrates the structure of Androtrace-enabled kernel and key data structures. The newly added fields in existing data structures are annotated with '+' symbol.

4.1 Original Process Name

In many cases, knowing the correlation between the original name of the process and the issued IO gives clear understanding of what is going on in the IO stack and gives better idea of exact IO behavior of applications. In order to collect the name of the original process, we add an unsigned integer field, `process_id`, to `bio` structure which is initialized in `submit_bio()`. Upon calling of `blk_peek_request()`, Androtrace consults `process_id` to find out the process name in a set of task structures. It records the information of an IO along with other fields on the IO trace circular buffer.

4.2 `fsync()`, `fdatasync()` and SQLite

Previous studies show that major causes for excessive IOs on Android and Tizen are synchronous write IO, SQLite and file system journal [15, 18]. When a new record is inserted in the SQLite database, it is synchronized with `fsync()` to flush the data and SQLite journal file. This system call enforces EXT4 file system to flush the dirty page cache entries and to commit EXT4 journal transaction. In the case of PERSIST SQLite journal mode, `fsync()` is called four to five times and each call flushes file system metadata and journal [15].

We add a field to both `address_space` and `transaction_s` kernel structures to identify if a given IO is issued by `fsync()` or by `fdatasync()` and if a given IO is for SQLite related files. `address_space` data structure manages page cache entries of a given file, and `transaction_s` maintains dirty

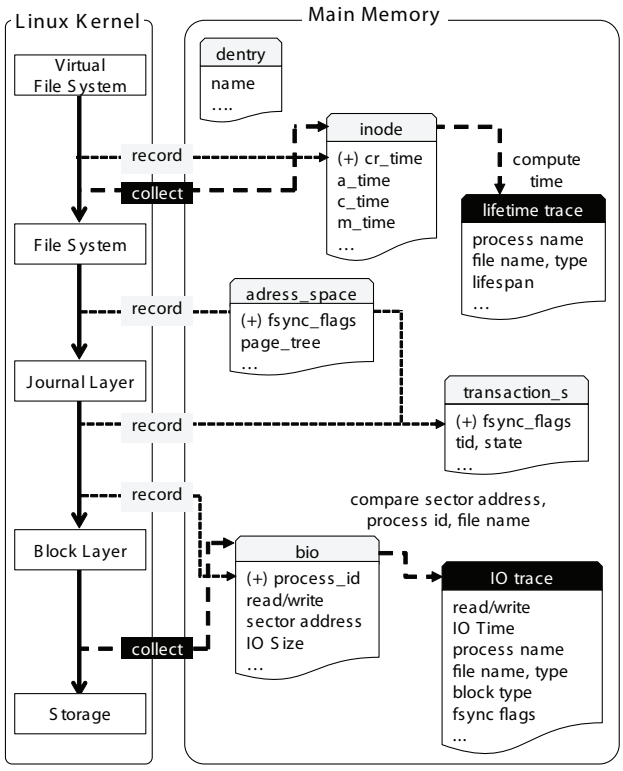

Figure 4: Modified Kernel Structure: Virtual File System, File System (EXT4), journal, and block layer. '+' symbol denotes newly added field in existing kernel structure. The dashed line (record) and dark-dashed line (collect) indicate where Androtrace marks IO related information and collects traces, respectively. It also present what Androtrace operate when it collects IO traces.

metadata cache entry for a single EXT4 journal transaction.

Let us look at Fig. 5 for detailed step by step overview. There are five steps, 1 and 2 are for flushing dirty page cache entries and 3, 4 and 5 are for committing journal transaction. First, when `fsync()` or `fdatasync()` is called, EXT4 examines `address_space` structure to flush dirty data pages. At this point, we set fsync flags for the newly added field in the `address_space`. After data write completes, EXT4 wakes `JBD2` thread up to commit the journal transaction. Androtrace marks fsync flags in `transaction_s` in order to notify `JBD2` thread that the requested transaction is triggered by `fsync()` or `fdatsync()` or SQLite. When `JBD2` thread starts journal commit, Androtrace sets fsync flags in `address_space` which maintains dirty metadata with the same value extracted from `transaction_s`. The collector examines the `address_space` data structure associated with a given `bio` structure to determine if a given IO is triggered by `fsync()` or if a given IO is related to SQLite. The collector sets the appropriate field.

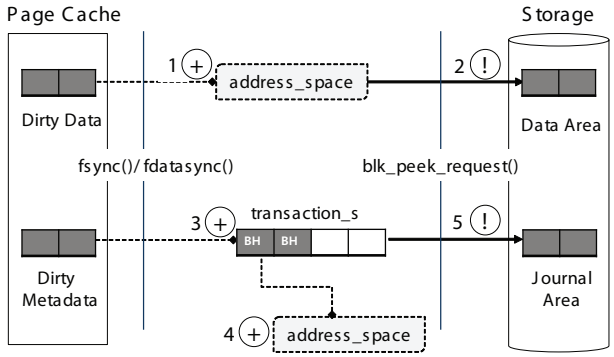

Figure 5: Determining `fsync()`, `fdatasync()`, and SQLite related IOs: '+' and '!' symbol indicate when Androtrace records fsync flags to `address_space`, and collects other fsync related IO information.

4.3 File and Block Type Identification

Androtrace records the file name and file type for each IO. It acquires `inode` object of a given IO from `bio` structure to extract the file name from `dentry` structure. Using the name, we determine the file type by their file name extension. Androtrace defines nine file types for IO analysis. The IO behaviors of SQLite are at the core of the Android IO stack. Androtrace puts much effort on providing detailed IO analysis with respect to SQLite. Four of nine types are SQLite related files, which are database, rollback journal, WAL file and temporary SQLite files. The rest of the file types are Multimedia, Executable, Cache, Temp and Other. The detailed list is shown in Table 2.

Androtrace also collects the file system block type for each IO. It categorizes the EXT4 file system blocks into three types: EXT4 Journal, EXT4 metadata and EXT4 data. We develop an efficient method to classify the file system block type in on-line manner. EXT4 file system reserves the first n inode numbers for special purpose file. In EXT4, inode number starts from 9[1]. If the inode number of a given `bio` is greater than or equal to 9, Androtrace categorizes it as data block. EXT4 file system defines a system wide magic

[1]This number varies subject to file system

Table 2: File Type Categories

file extension	file type
`.db`	SQLite db
`.db-journal`	SQLite journal
`.db-wal`	SQLite wal
`.db-shm`, `.db-mjxxxx`	SQLite temp
`.jpg`, `.3gp`, `.mp3`, `.thumb`,..	Multimedia
`.apk`, `.so`, `.dex`, `.odex`	Executable
`.localstorage`, `.xml`, `.cache`, ...	Cache
`.temp`, `.tmp`, `.bak`	Temp
others	Other

number for EXT4 journal transaction. EXT4 journal transaction consists of the descriptor blocks and commit mark block. Each of these forms a single IO. For both of these IOs, the first four bytes contain the magic number. Androtrace identifies a block as the EXT4 journal block examining this magic number. If it is neither data block nor journal block, it categorizes an IO as file system metadata block.

4.4 File Lifespan

Lifespan of a file in the smartphone provides an important measure on the frequency of file system metadata update. Androtrace logs the file attributes for unlinked file along with the lifespan of the file. To obtain the file lifespan, we add a creation time field, in the `inode` structure. This field is initialized when the file is created by `do_sys_open()` with `O_CREAT` flag. We modify the `unlink` system call to compute the file lifespan and collect traces, e.g. file size, file name.

5. EXPERIMENT

5.1 Environment Setup

We implement Androtrace in Nexus 5 [9], Google's reference smartphone, using Android 4.4 (Kitkat) with 16 GByte of internal Flash memory. Nexus 5 has twenty nine partitions. The most important partitions are `/system` (1 GByte read-only), `/data` (13 GByte) and `/cache` (700 MByte).

Androtrace is currently under pilot test before full deployment. We deploy Androtrace equipped smartphone to four volunteers. We present preliminary results using four users to get brief understanding of capabilities of Androtrace.

5.2 Overhead Analysis

As on-line multi user IO Trace and Analysis utilities, the first goal of Androtrace is not to interfere with the foreground application. We examine how the Androtrace affects the IO performance. We examine the IO performance in three set-ups: stock NEXUS 5, Androtrace enabled NEXUS 5, `blktrace` enabled NEXUS 5. We use open source benchmark, `Mobibench` [14] to examine the overhead.

We perform random and sequential write with buffered, synchronous (`write()` followed by `fsync()`) and Direct IO write. The IO size is 128 KByte and 4 KByte for sequential write and random write, respectively. The file size is set to 512 MByte. Fig. 6 illustrates the result. As we can see, Androtrace is not interfering. The performance of Androtrace loaded smartphone is almost on par with the one without any trace utility. On the other hand, the `blktrace` loaded smartphone exhibits as much as 72% performance

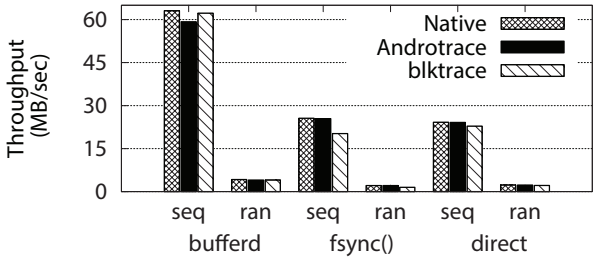

Figure 6: Androtracs vs. IO Performance : Native, Androtrace vs. blktrace, File Size: 512MB, IO Size: 128KB (sequential) 4KB (random), Device: Nexus5, Partition: /data, EXT4 File System.

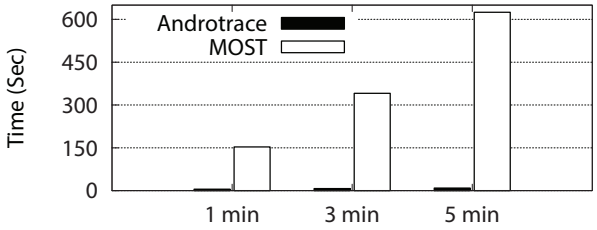

Figure 7: Androtracs vs. Processing Time : Androtrace vs. MOST IO's are generated from Monkey utilities (executing web, youtube, video, contacts, etc.)

hit against the one without any trace utilities. Especially, `blktrace` is sensitive to the synchronous IO.

Both `MOST` and `blktrace` require the identical post processing step to extract some information, e.g. file system block type. On the other hand, Androtrace does not require any post processing. We compare the overhead of post-processing of Androtrace and `MOST`. We used `Monkey` [1] to generate the workloads for varying intervals (1 min, 3 min, and 5 min) and measured the time for post-processing the collected traces. Fig. 7 illustrates the result. `MOST` requires off-line processing to identify the EXT4 file system block type using `debugfs` [20] and file type information.

As the trace size increases, the time to post-process the IO traces also increases in `MOST`, while Androtrace remains unaffected by the length of the trace. Furthermore, the cost of post processing is very high. It takes twice as long as IO acquisition period. This overhead makes `MOST` practically infeasible to be used in prolonged time of IO trace study, e,g. several month.

5.3 IO Analysis

We use the collected user IO trace to prove some of the key advantages of using Androtrace.

Read vs. Write: Fig. 8 shows the number of average read and write IOs observed on a day. Having profound understanding of IO usage patterns of a device is important for both device manufacturers and software developers because it allows predicting the lifespan of a Flash based storage device and also enlightens areas of improvement in terms of IO performance.

The result shows that write IOs are dominant in Android platform. Write IO counts on User 1 are eight times larger than that of read IOs. In other users, the number of Write IOs are at least twice as large as the number of Read IOs.

To have better understanding of how file system works, Fig. 9 illustrates block type information; it shows ratio of File System Metadata, Journal and Data on a daily portion. In both user 3 and user 4, the read activity prevails. In user 3 and user 4, we find that 75% and 63% of all read IOs on data block are for executable files such as `.apk`, `.dex`, and `.so` respectively. In general, these files are often read at the beginning of an application launch. We find that the executable files, especially `.apk` are frequently read while applications are running. Top three rank of applications for User3 is Angry Bird (game), Atlan (navigation application for automobile), and Afreeca TV (streaming video application). On the other hand, User4 shows that top 2 ranks are RoadMovie (video editor) and Wonder Camera. Although access to executable files are subject to investigation, we believe that caching policy for such applications should be reconsidered to relieve the read IO overhead.

Is the JOJ Problem Prevalent in Real World?: SQLite database and EXT4 file system both provide consistency of data through journaling; however, when two different IO layers are stacked on top of each other, it seems that it is inevitable to face the journaling of journal (JOJ) problem [15].

We compare the generated journal IO overhead with respect to the IO count and the volume. IOs issued for `.db-journal`, `.db-mj`, `.db-wal` and `.db-shm` files account for SQLite Journal related IO. IOs observed only on journal block accounts for EXT4 Journal IO. SQLite journal related file accesses account for 41% (count) and 25% (volume) of the total data block accesses in the file system. The accesses to the EXT4 Journal blocks account for 24% (count) and 40% (volume) of the total file system block accesses. The IOs to EXT4 journal region is larger than the IOs to the file system data region. This is because the size of a journal entry is 4KB and a journal transaction is composed of at least three 4KB blocks; journal header, journal descriptor block and journal commit block.

Finding New IO Characteristic: While many works have reported that SQLite creates excessive writes [15, 18, 22], the problem of the previous works is that they concluded their findings using only brief amount of time and limited number of applications. Thus, the questions still remain at hand are who is responsible for synchronous IOs and how `fsync()` and `fdatasync()` are issued; how the real smartphone applications use SQLite and how many synchronous IOs such as `fsync()` and `fdatasync()` are issued while using the device. They are almost impossible to figure out using by `blktrace` and `MOST`. Androtrace not only allows figuring out the sources and callers of issued IO but also it allows identifying the most active database in terms of issuing synchronous IOs. Table. 3 shows the ratio of synchronous writes and buffered writes observed in four users, and it shows that average of 73% of writes are synchronous which is consistent with earlier studies [15, 22].

Table 3: Sync Write vs. Buffred Write on IO Count

User	User 1	User 2	User 3	User 4
Sync Write	76%	73%	67%	77%
Buffered Write	24%	27%	33%	23%

We examine how much fraction of synchronous write is

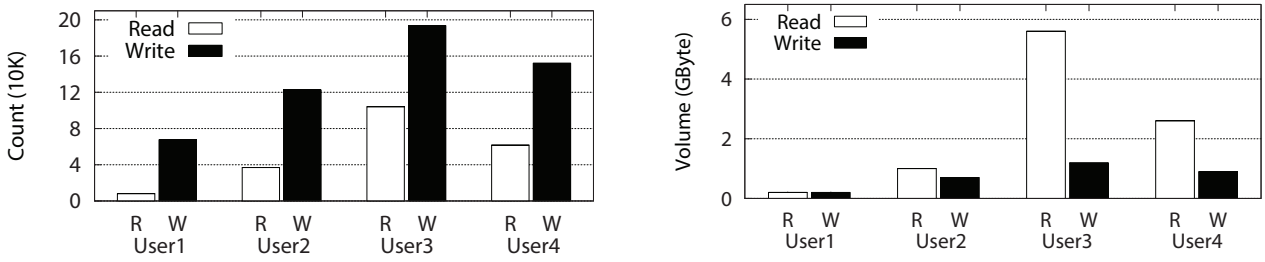

Figure 8: Daily Average IO Count and Volume

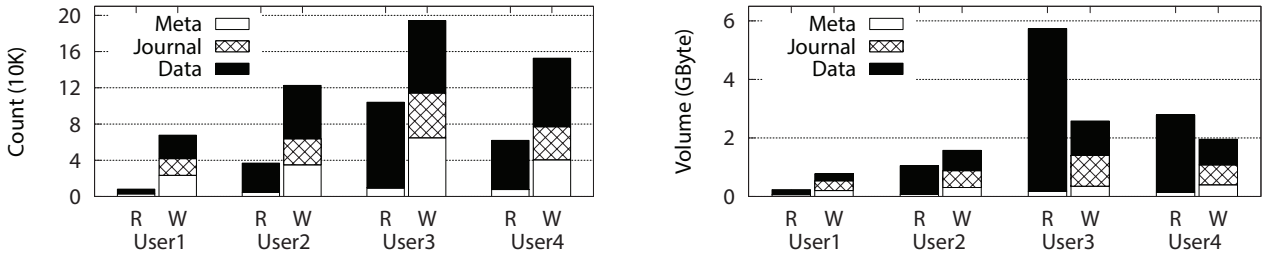

Figure 9: Daily Average IO Count and Volume with Repective to Block Types

caused by `fsync()` or `fdatasync()`. They include not only the filesystem data blocks but also the filesystem journal blocks. Fig. 10 illustrates the result. Among the synchronous writes, `fsync()` and `fdatasync()` account for 70% of all synchronous write. In Android, `fsync()` and `fdatasync()` are the main cause for synchronous write. Another interesting finding is the ratio between `fsync()` and `fdatasync()`. `fdatasync()` provides clear performance advantage over – `fsync()`. The ratio between the IO count from `fsync()` and one from `fdatasync()` is 1:4 (16% vs. 56%).

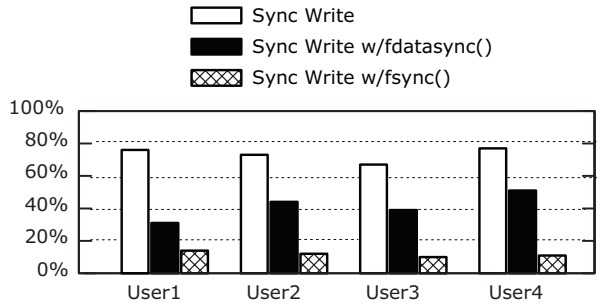

Figure 10: The Ratio of Synchronous Write.

6. CONCLUSION AND FUTURE WORK

As the numbers of mobile users are increasing, the importance of having in depth understanding of user IO behavior is also increasing. However, many of the existing utilities are short on providing a framework for Android device to acquire multiple user IO traces and analyzing them. In this paper we developed Androtrace. Androtrace is a cross layer IO trace utilities that collects information from various levels of software stack including application, system call, virtual file system, EXT4 and JBD2. Androtrace opens a new door in analyzing IO trace and in optimizing mobile devices. We presented basic design framework.

The number of people participated are not enough to draw concrete and general IO behaviors of Android devices, but we have shown that it is possible with Androtrace to acquire user IO behavior for extensive period of time.

IO characteristics which are impossible with existing utilities. While we show limited use case of analyzing user patterns on Read/Write IOs with given IO trace, it is also possible to analyze daily and hourly usage of the device using the time-stamp of the trace. Since one of the killer applications is messaging in mobile devices, we can also restrict the environment to set groups of two or more people in group messaging sessions to observe difference between groups of IO behaviors.

We find write traffic is dominant and that indeed SQLite and EXT4 journal poses journaling of journal problem in real world cases, and addressed that caching mechanism for Android executable files e,g. `.apk` needs much attention.

In the limited experiment, JOJ problem in mobile devices are for real, and most of synchronous IOs are issued with `fdatasync()` IOs and SQLite is responsible for issuing them. The challenge is to identify the top most synchronous IO application and when the IOs are generated. We reserve this research for future work and it needs attention in optimizing the IO performance.

Understanding the IO behavior when it crashes and recovers from the crash is important in both designing the application and the platform itself. It would be certainly interesting to reproduce the behaviors of events occurred just before the crash. Since crash is not often in the device, it would be unwise to record all of the relevant events produced; thus, it is a challenging work to design a method to only make few seconds or minutes of information before the crash persistent.

Acknowledgement

We would like to thank Joongwoo Hwang at Hanyang University for developing transport module in Androtrace. We also would like to thanks our colleague Dam Quang Tuan for his help in preparing the manuscript. This work is sponsored by IT R&D program from MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device) and by ICT R&D program of MSIP/IITP (No.1I2221-14- 1005) and by ITRC(Information Technology Research Center) support program (IITP-2015- H8501-15-1006).

7. REFERENCES

[1] Android. Monkey, 2015. `http://developer.android.com/tools/help/monkey.html`.

[2] Arlitt, M. F., and Williamson, C. L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw. 5*, 5 (Oct. 1997), 631–645.

[3] Chen, P. M., and Patterson, D. A. A new approach to I/O performance evaluation: Self-scaling I/O benchmarks, predicted I/O performance. *SIGMETRICS Perform. Eval. Rev. 21*, 1 (June 1993), 1–12.

[4] Chi-ming, C., and Mutka, M. W. Characteristics of user file-usage patterns. *Journal of Systems and Software 23*, 3 (1993), 257 – 268.

[5] Cho, M., Hwang, S. J., Lee, H. J., Kim, M., and Kim, S. W. Androscope for detailed performance study of the android platform and its applications. In *Consumer Electronics (ICCE), 2012 IEEE International Conference on* (Jan 2012), pp. 408–409.

[6] Crandall, P. E., Aydt, R. A., Chien, A. A., and Reed, D. A. Input/output characteristics of scalable parallel applications. In *Proc. of ACM/IEEE Conference on Supercomputing* (1995), Supercomputing '95.

[7] Douglis, F., and Ousterhout, J. Log-structured file systems. In *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.* (1989), IEEE, pp. 124–129.

[8] Godard, S. iostat. `http://linux.die.net/man/1/iostat`.

[9] Google. Nexus 5. `http://www.google.com/nexus/5`.

[10] Gregg, B. iotop. `http://linux.die.net/man/1/iotop`.

[11] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. A file is not a file: Understanding the I/O behavior of apple desktop applications. *ACM Trans. Comput. Syst. 30*, 3 (Aug. 2012), 10:1–10:39.

[12] Hsu, W., and Smith, A. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal 42*, 2 (2003), 347–372.

[13] Jens Axboe, A. D. B., and Scott., N. blktrace, 2006. `http://linux.die.net/man/8/blktrace`.

[14] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. Androstep: Android storage performance analysis tool. In *Software Engineering* (2013), pp. 327–340.

[15] Jeong, S., Lee, K., Lee, S., Son, S., and Won, Y. I/O stack optimization for smartphones. In *Proc. of USENIX Conference* (2013), ATC '13, pp. 309–320.

[16] Kim, H., Agrawal, N., and Ungureanu, C. Examining storage performance on mobile devices. In *Proc. of ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds* (2011), MobiHeld '11, pp. 6:1–6:6.

[17] Kim, H., Agrawal, N., and Ungureanu, C. Revisiting storage for smartphones. *Trans. Storage 8*, 4 (Dec. 2012), 14:1–14:25.

[18] Kim, M., Lee, S., and Won, Y. IO workload characterization of Tizen based consumer electronics. In *Consumer Electronics (ISCE 2014), The 18th IEEE International Symposium on* (June 2014), pp. 1–4.

[19] Kranenburg, P. strace. `http://linux.die.net/man/1/strace`.

[20] Kroah-Hartman, G. debugfs. `http://linux.die.net/man/8/debugfs`.

[21] Lee, K. Mobile storage analyzer (MOST). `https://github.com/ESOS-Lab/MOST`.

[22] Lee, K., and Won, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proc. of EMSOFT 2012*.

[23] Lee, S., Moon, B., and Park, C. Advances in flash memory SSD technology for enterprise database applications. In *Proc. of ACM SIGMOD International Conference on Management of Data* (2009), SIGMOD'09, pp. 863–870.

[24] Leung, A. W., Pasupathy, S., Goodson, G., and Miller, E. L. Measurement and analysis of large-scale network file system workloads. In *Proc. of USENIX Conference* (2008), ATC '08, pp. 213–226.

[25] Lim, S., Lee, S., and Ahn, W. Applications IO profiling and analysis for smart devices. *Journal of Systems Architecture 59*, 9 (2013), 740 – 747.

[26] McVoy, L. W., and Kleiman, S. R. Extent-like performance from a unix file system. In *USENIX Winter* (1991), vol. 91.

[27] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *Proc. of USENIX Conference* (2005), ATC '05, pp. 8–8.

[28] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *Proc. of USENIX Conference* (2000), ATC '00, pp. 4–4.

[29] SQLite.org. Well-known users of sqlite. `https://www.sqlite.org/famous.html`.

[30] tom'sGuide. Smartphones with the longest battery life, 2015. `http://www.tomsguide.com/us/smartphones-best-battery-life,review-2857.html`.

[31] Vogels, W. File system usage in Windows NT 4.0. *SIGOPS Operating Systems Review 33*, 5 (Dec. 1999), 93–109.

[32] Ward, G. rtrace. `http://radsite.lbl.gov/radiance/man_html/rtrace.1.html`.

[33] Zhou, M., and Smith, A. Analysis of personal computer workloads. In *Proc. of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (1999), MASCOTS '99, pp. 208–217.