15712 Advanced Operating and Distributed System
Android and iOS Platform Study
Final Report

Lucas Tan
lucastan@cmu.edu

Fuyao Zhao
fuyaoz@cmu.edu

Xu Zhang
xuzhang@cmu.edu

1 Introduction

With smart phones becoming more ubiquitous than ever before, lots of applications are being developed for these devices instead of traditional computing platforms such as the desktop. It is thus worthwhile and interesting to conduct a study on mobile platforms and applications.

Better understanding the performance characteristics of the platforms open the path for future improvements by system designers, and might even lead to new classes of applications, for example, applications that can better utilize mobile hardware resources.

Helping mobile developers to understand better the behavior of their applications could provide opportunities for improvement in performance and perhaps even user experience. An application might have different performance characteristics and behavior on each platform even though it could come from the same developer. Any difference in performance could be attributed to the fundamental differences between the platforms.

Our study is based on Apple iOS and Google Android, the two most popular mobile platforms which have a combined market share of 77% [4]. More specifically, our goal is to find out how the differences between both platforms can cause applications to behave in a certain way and how the same application behave on both platforms. Our study focuses on file I/O, and is based on analyzing traces of system calls made by the same set of applications running on each platform.

During the course of the study, we encountered difficulties collecting system call traces for iOS. We thus developed our own system call tracing tool called iTrace. We show that iTrace performs faster than existing tools due to a novel tracing technique we used. We also think it is possible to port iTrace over to Android and even other Linux-based operating systems.

To summarize, our contributions are the findings from the collected traces, iTrace tool and the technique involved, and the traces we collected.

1.1 Related work

It is especially interesting to conduct a study on iOS from a systems perspective since so far there is no such study as far as we know. Most of the studies on iOS are in the fields of privacy, security and application designs [9]. Furthermore, the only work we know that compares the two platforms, provides only a high-level overview and lacks concrete statistics [1]. Our study attempts to fill in this gap in existing literature. We believe the lack of research in this area for iOS can be attributed to the difficulty of collecting traces due to the closed-source nature of the platform.

On the other hand, due to the open nature of the platform, much research have been done on Android in many areas including power management, network performance, security, and storage system.

A recent paper [10] investigates the I/O behavior of common productivity applications on Mac OS X. Though we would be using the same methodologies in analyzing our traces, our work differs in that we investigate mobile applications, as well as a different operating system, Android which is based on Linux, rather than the BSD-derived Mac OS X.

Another recent paper focuses on the storage system [13]. It shows that the underlying flash storage device from various manufacturers could have a significant impact on the performance of commonly used applications. The authors monitored the `/proc` virtual file system periodically to get various system statistics and used the `blktrace` tool [3] to trace block-level events for device I/O. In contrast, we use the `strace` tool [7] to trace file system calls, which is at a logical level rather than physical level. Thus, we can gather information about files and the range of bytes accessed, instead of the physical disk blocks. Furthermore, we aim to provide more fine-grained statistics and findings, as opposed to aggregated and throughput statistics.

[8] also uses strace, in fact, strace is a common tool in Android research. But as far as we can tell, most uses of strace is for analyzing a programs system calls frequency for detecting malware.

[14] and [15] investigated the various factors in web browsers performance on smart phones and the effectiveness of web browser caches. They show several interesting findings but as the research is based only on Android, the results are not generic enough to convince the reader that their suggestions for improvements would also be effective on other platforms.

Yet another study [11] compares various platforms, but only in the area of network performance.

DTrace is a powerful tracing tool that is available on Mac OS X but unfortunately not on iOS. Consequently, we had to use the official Instruments tool [5] from Apple initially. Even though it can dynamically trace and profile both Mac OS X and iOS applications, there are three problems we experienced that prompted us to develop iTrace. First, Instruments exports traces in an unknown proprietary binary format. Second, Instruments can only display the address of a string argument for a system call, it does not display the actual contents of the string. Third, it is very slow as tracing is done over a USB connection with a computer. iTrace is thus developed to solve all of these problems.

1.2 Software Architecture

We provide a brief high-level overview of the software architecture of both platforms.

An iOS application is developed in the Objective-C language and is compiled into native ARM instructions. An application binary and all of its required resource files are stored in a single directory in `/var/mobile/Applications`. Each application gets its own directory. All files are stored individually as such in the file system.

On the other hand, an Android application is developed in Java and is eventually translated to Dalvik bytecodes for execution in the Dalvik Virtual Machine. Unlike iOS, an application and all of its required resources are packaged as a single APK file which is essentially a ZIP file.

On both platforms, applications use either directly or indirectly through frameworks, the lightweight database library SQLite to manage application data.

1.3 Overview

The rest of the report is organized as follows: Section 2 describes our methodology and how we collect system call traces on Android and iOS. Details of our experiments and findings are

presented in sections 3 and 4 respectively. A discussion of how iTrace works is in section 5. Section 6 concludes and section 7 lists future work.

2 Method

We collected system call traces for the following popular applications: Facebook, Dropbox, Angry Birds (ad supported version) and the native web browser on each platform. They represent a good mix of the types of applications.

We used methodologies from [10] to analyze the traces. In section 4, we compare our findings with those presented in the paper.

2.1 Limitations

We summarize the limitations of our study as follows.

Runtime environment and programming model. We assume the differences between the runtime environment and programming model for each platform have little, if any at all, impact on file I/O statistics.

No timing involved. Due to the differences in the runtime environment and the fact that strace on Android has a much higher overhead than iTrace (as explained in section 5), we do not have any reference to timing in our analysis.

Incomplete tracing. Since we could not figure out how to start an iOS application programmatically, we could only activate tracing after manually starting an application. This might potentially omit much important information during the application initialization phase.

Lastly, though our study is by no means complete, we attempt to present as much information as possible. More work is needed to extract further useful information from the traces.

2.2 Android

strace is a tool available on many Linux-based operating systems. It relies on the *ptrace* [6] system call for its functionalities. strace can attach to an existing process or fork a new process for tracing. It can trace all the system calls made by each thread in a process, complete with timestamps, call arguments and return values.

The Android Open Source project contains a port of the strace tool. We downloaded the source code, modified and re-compiled strace in order to suppress a harmless (yet annoying) error message that is written to `stderr` every time the strace process tries to read an invalid memory region in the process being traced. This modification speeds up the tracing slightly.

A partial sample of the trace log file when we attached strace to the web browser application, is as shown below:

```
1345 22:02:36.843001 open("/data/data/com.android.browser/databases",
                          O_RDONLY|O_LARGEFILE) = 46
1345 22:02:36.843220 fcntl64(46, F_GETFD)    = 0
1345 22:02:36.843663 lseek(36, 0, SEEK_SET)  = 0
. . .  . . .
1345 22:02:48.640407 ioctl(47, 0xc01c67230x81, 1, MSG_OOB) = 0
```

The first column is the thread ID, the second the timestamp with up to microseconds precision and the third the system call along with its arguments and return value.

Initially we also used the Dalvik Debug Monitor Server (DDMS) to collect VM method calls, that is, calls to the Android API framework. However, since there is no equivalent way in iOS, we abandoned the use of DDMS.

2.3 iOS

We used our own iTrace tool to collect system call traces on iOS. iTrace logs system calls to a file in the exact same format as strace. A discussion of how iTrace works is in section 5.

3 Experiments

3.1 Equipment

We used iPhone 4 and Samsung Nexus S for all our experiments. Both phones are chosen as they have similar hardware characteristics. Both phones use the ARM architecture due to its low power usage. Table 1 shows the comparison of the hardware and software components of both phones:

	iPhone 4	Samsung Nexus S
OS	iOS 4.1	Android 2.3.6 (Linux 2.6.35)
CPU	Cortex A8, 1 GHz (ARM architecture)	
RAM	512 MB	
Graphics	PowerVR SGX 535 / 540 (200 MHz)	
File System	HFS+	ext4 (`/system` and `/data` partitions) YAFFS2 (`/efs` and `/cache` partitions) vfat (SD card partition)

Table 1: Comparison between iPhone 4 and Nexus S.

We are not sure of the exact file system used in iOS, since there is no official information available. Several sources suggested a variant of HFS. The Android OS uses ext4 for system files and data including applications, YAFFS2 for infrequently accessed files such as recovery and update files, and vfat for the SD card partition which contain user files. It should be noted that the SD card partition is a logical one as the phone has no physical SD card. Out of all the filesystems, only YAFFS2 is log-based and designed specifically for flash storage.

The iPhone is jailbroken so that GDB (GNU Debugger) and SSH (Secured SHell) could be installed. We could then launch an SSH session to issue commands on the iPhone. We used SuperOneClick to gain root access to the Android phone. We were then able to launch a regular Linux terminal session on our computer to communicate with the phone, using the `adb` tool provided by the Android SDK.

3.2 Workload

We decided on the actions to perform for each application when tracing is activated. The actions are intended to represent common usage. We manually execute each action since there is no programmatic nor automatic way to do so in iOS. Thus it is hard to be very precise on the timing of each action but we believe any small variation in timings should have negligible impact on the traces collected.

Before we start collecting traces, we used a fresh installation of each application so that any cached files would be removed. For the web browser application, we manually cleared the history and cache since it cannot be re-installed.

The workload for each application is summarized below:

Application	Actions
Dropbox	When the user interface is ready, open text files of the following sizes in sequence: 64B, 256B, 1KB, 4KB, 16KB and 64KB. Opening a new file not in the cache will trigger downloading of the file.
Facebook	Login and visit 3 friends' profiles. Wait for each profile to be fully loaded.
Angry Birds	Enter level 1 and finish the game with 0 point. Stop right after the score screen is shown.
Web Browser	Load `"http://rng.io/"` to perform the Ringmark mobile browser benchmark test. Wait for the test to finish.

Table 2: Experiment Workload.

Dropbox behaves differently on the two platforms: the iOS version would display PDF documents and multimedia files within the same process (app), while the Android version would display them in an external process. strace would not be able to attach to the external process for an unknown reason, although normally it would. We suspect the new process is started using the Android API framework which strace cannot capture. So for Dropbox, we only test by using plain-text files, which could be displayed in-process on both platforms.

4 Analysis

Our analysis aims to answer the following questions:

- What are the types of files accessed?
- Are files still accessed sequentially most of the time?
- How often is `fsync` used?

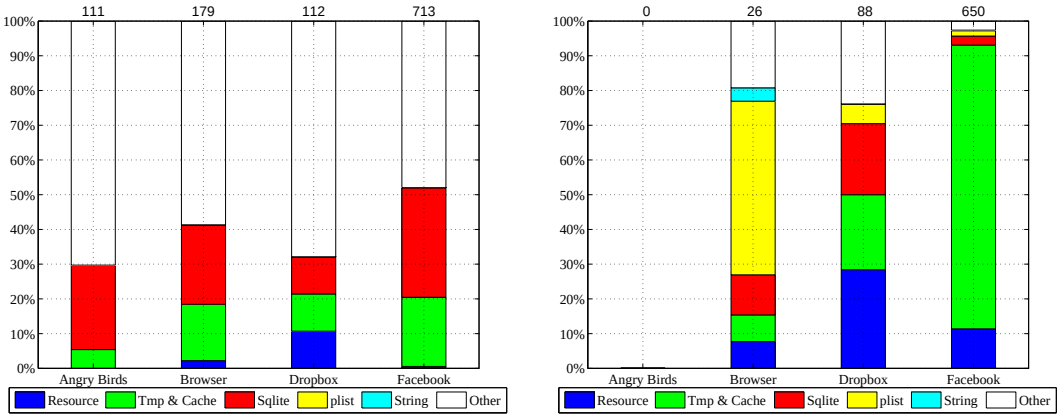
4.1 File types

We group the files into six types: *Resource*: non-executable such as image, audio, and video files. *Temp*: temporary or cache files. *SQLite*: database and journal files. Journal files are used for write-ahead logging and are created and deleted within a second. *plist*: store serialized objects and settings (iOS only). *Strings*: files containing localized application text (iOS only). *Other*: all other file types including binary, plain-text and log files. The Android system frequently uses `/dev/ashmem` which is a virtual device that can be used to create shared memory amongst processes. This is also included in the *Other* category. For the Dropbox application on both platforms, this also includes the workload files.

Figure 1 shows the frequencies with which each application open and access files of each type for both platforms. On Android, the four applications have similar pattern on the files accessed: *other*, *SQLite*, and *temp* are the most frequently accessed file types. The Android web browser stores cache files individually on the file system and metadata in one or two *SQLite* databases. This results in a larger number of files accessed as compared to AngryBirds and Dropbox. Facebook stores every picture, news feed posts, and etc. individually as a cache file

on the file system. As a result, it accessed the most number of files. Database files occupy a significant percentage as well because write-ahead logging occurred very frequently. *Other* files are the most common among the four applications due to the fact that most of them are shared memory files.

On iOS, every application has very different file type distribution. Like the Android version, iOS Facebook accessed a lot of cache files. However there is much less write-ahead logging activity. For Web Browser, *plist* files are accessed many times. It is due to the fact that the browser loaded many plugins which store their settings in *plist* files. Cookies are also stored in *plist* files. However cache files are stored in only one or two *SQLite* databases. HTML5 has a local persistent storage technology that results in a *localstorage* database file to be created for each web domain that uses it. The Ringmark test creates a *localstorage* container as one of the tests. For the iOS version of AngryBirds, all of the file I/O is done during the startup phase, and hence there is no captured file activity after that. For the Android version, there is some activity due to diagnostics logging and a different advertising system it uses.



(a) Android (b) iOS

Figure 1: **Types of Files Accessed By Number of Opens.** This plot shows the relative frequency with which file descriptors are opened upon different file types. The number at the end of each bar indicates the total number of unique file descriptors opened on files

Summary: On Android, the applications usually have similar file types distribution, and shared memory files take a large part in all applications. It would be interesting to find out what shared memory is used exactly for. We believe it might be used for sharing memory with the graphics device, that is, transferring of graphics data for rendering. For iOS, the distribution of file types differ widely for each application. When compared with [10], we find that only the iOS web browser accesses many *plist* files. This might be because the web browser is the only one out of the four applications produced by Apple.

4.2 Access Patterns

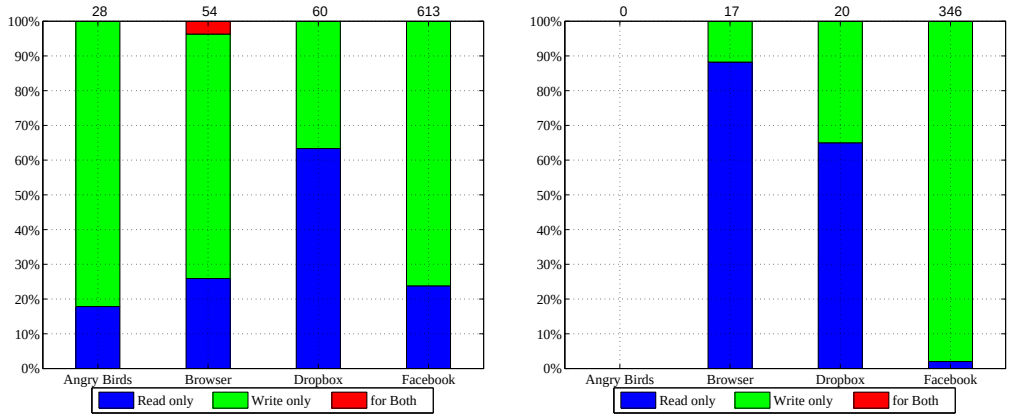
In this section, we examine the file natures in the two platform. More specifically, we study the patterns whether the file used for reading, writing or for both. We also measured how files are accessed, sequentially or randomly,

4.2.1 File Access

One file could be used for reading, writing, and for both. If the file only be used for reading, or only used for writing, the system may have more chance to do optimization in the file system.

As we know, the file could be opened for reading, writing, or both by passing a flag to the open function. We also divide the file descriptors into three types, for read, for write, and for both. However, we cannot use the flags in the open function since they cant reflect the real behavior, since the programmer may open all file by using `O_RDWR` flag, but they only use the file descriptor for reading or only for writing. We also measured how many bytes performed on each type of file descriptor.

Figure 2 shows how many file descriptor for each type of accesses for different application on two platform. For both platforms, the nearly all the file descriptors are used only for reading or writing. For Android, write only file descriptor is the most common. For iOS read only file descriptor is more common on the Browser application and Dropbox, but Facebook use lots of write only file because of writing the cache files.



(a) Android (b) iOS

Figure 2: **Read/Write Distribution By File Descriptor.** This plot shows the percentage of the file descriptors in which category: read, write or both. This plot is based on the real usage, not open function flags

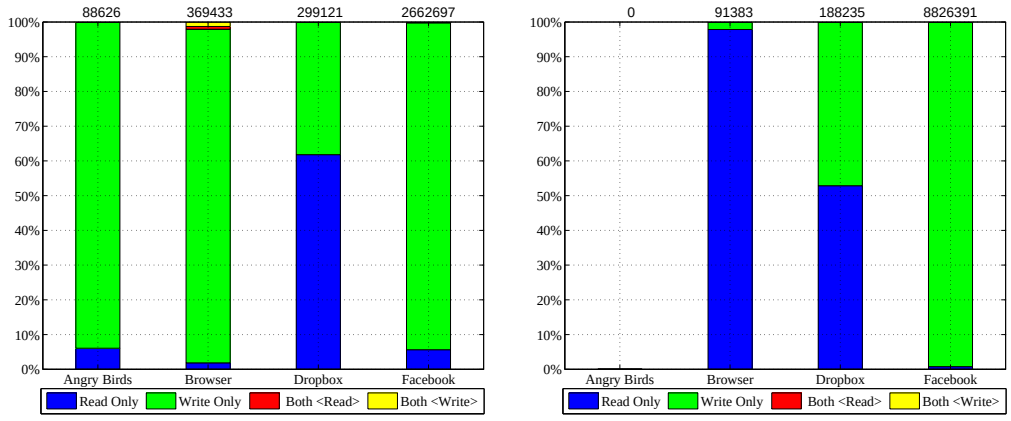
Figure 3 shows how much I/O performed on each type of file descriptor. We found the pattern is similar to the pattern in the Figure 6(a) and Figure 2 . The most majority of I/O are performed on write only file descriptor, and the write-only file descriptor occupied more I/O amount than the read-only descriptor.

Summary: File open flags cannot be used to predict how a file will be accessed. On mobile device platform, there are few file descriptor both for reading and writing.

4.2.2 Sequentiality

Historically, files are usually been read or written entirely sequentially. Recently, the study on Apple desktop shows the similar results on modern desktop, and they used a new metric called nearly sequential. In this section, we will examine the sequentiality on mobile device.

Figure 4 and Figure 5 shows the measurement results for read sequentiality and write sequentiality. For read operations on iOS device, we observe high read sequentiality for all applications. But for Android platform, less sequentiality observed on Browser and Angry Bird. For write



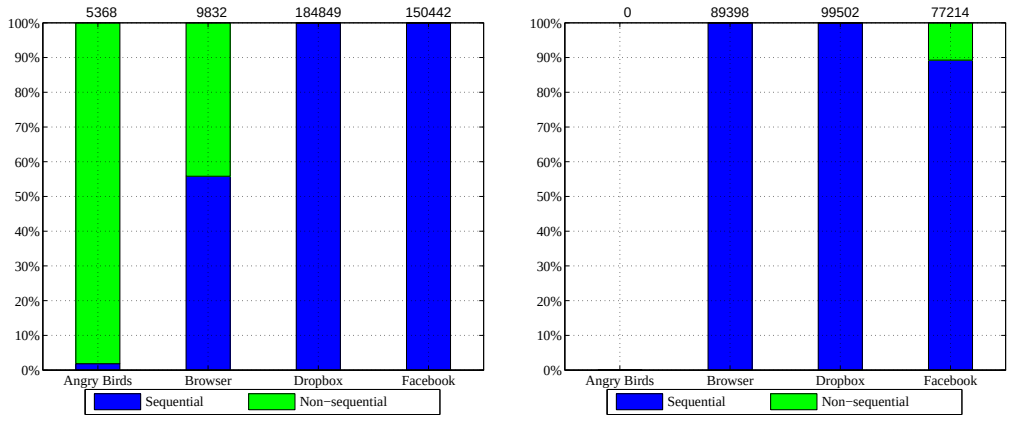
(a) Android (b) iOS

Figure 3: **Read/Write Distribution By Bytes.** This plot shows how I/O bytes are distributed among the three access categories

operations, we could observe the similar results. But the we observe more random write happens on iOS application, which may cause by using database to manage the cache data.

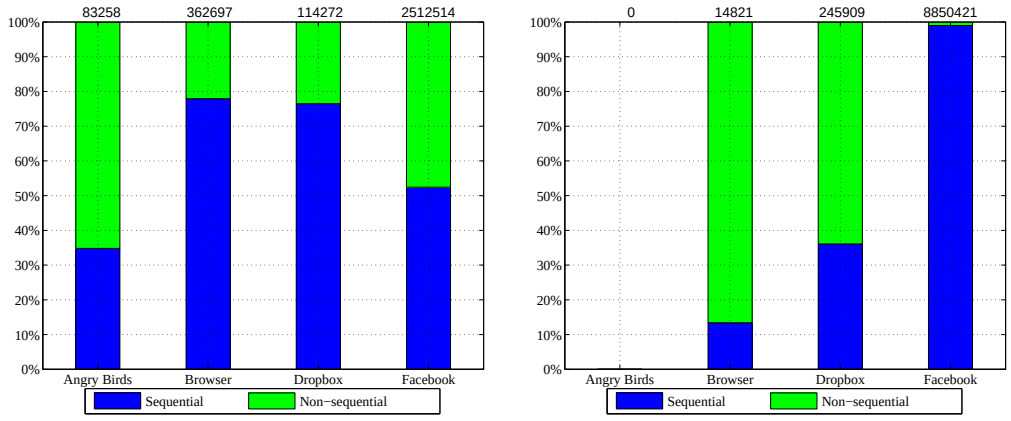
For both platform, we hardly observe nearly-sequential access pattern. That because the nearly-sequential access result from metadata stored at the beginning of complex multimedia files, and the applications often touch a few bytes at the beginning of the file before it sequentially reading or writing the bulk of the file. But our workload dont have too many complex large multimedia files. The most complex file might be the resource image files in Facebook, but they are usually small size.

Summary: the applications on two platform contain lots of purely sequential accesses but few nearly sequential access.



(a) Android (b) iOS

Figure 4: **Read Sequentiality.** This plot shows the portion of file read accesses (weighted by bytes) that sequentially accessed



(a) Android (b) iOS

Figure 5: **Write Sequentiality.** This plot shows the portion of file write accesses (weighted by bytes) that sequentially accessed

4.2.3 Durability

For write operations, the system has write-back cache to improve the write performance. But some application may require the data to be flush to the disk for higher durability. From the recent study on modern Apple desktop, developers are more likely to intentionally use `fsync` operations to ensure data enters stable storage, and some of `fsync` operations are only for small amounts of data[10].

For our study, we also measured how the system perform `fsync` operations on mobile device applications. Figure 6 shows total number of `fsync` operations and the distribution of `fsync` sizes, where we group `fsync` calls based on the amount of I/O performed on each file descriptor when `fsync` is called.

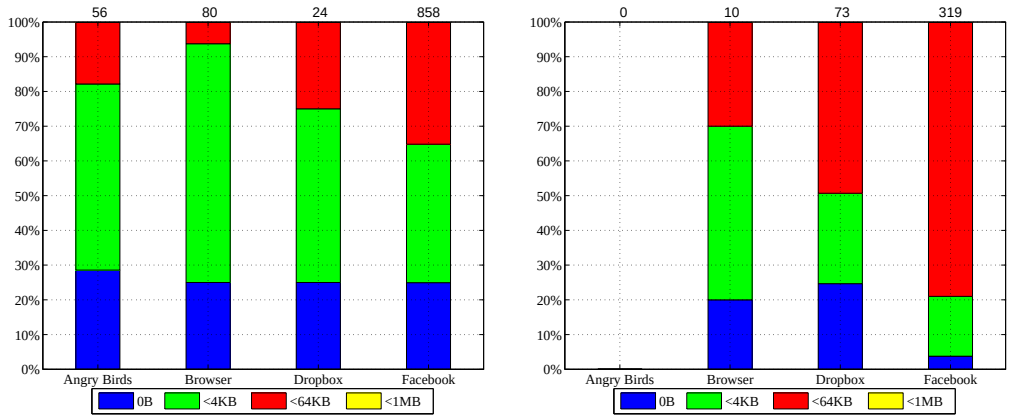
Since on both iOS and Android, applications use `SQLite` data for medata read and write, which might cause lot of `fsync` calls. For both platform, we observe all the `fsync` operations are all for the files which are less than 64KB. For android, more than 60

Summary: we observe the similar behavior on mobile device system. Developers want to ensure data durability, the applications frequently issue `fsync` operations, and most of them are only for small size of data. Most of them are caused by the side effect of using database. And we observe iOS issues less `fsync` operations for larger data than Android.

5 iTrace

The official Instruments tool from Apple can collect traces of system calls and their respective arguments, but as mentioned earlier, there are some problems that could not be resolved. As such we developed our own iTrace tool.

The main technique used in iTrace is *in-process tracing*. We first present an overview on existing tools before we discuss this technique in depth.



(a) Android (b) iOS

Figure 6: **Fsync Sizes.** This plot shows a distribution of fsync sizes. The total number of fsync calls appears at the end of the bars.

5.1 Overview of tracing tools

strace is a popular tool for system call tracing and uses the *ptrace* system call found in many Linux-based operating systems. *ptrace* allows reading and writing of memory from the tracees address space as well as notification of the entry and exit of every system call. The *tracee* is the process to be traced. However, in iOS, the ptrace implementation excludes the ability to monitor system call.

strace uses `ptrace` to attach to the tracee so that it can use the capabilities of ptrace. The following steps describe how strace interacts with the tracee through the use of ptrace:

- strace uses the `PTRACE_SYSCALL` request to inform the kernel to suspend the tracee and to notify strace whenever the tracee enters a system call.
- When the tracee enters a system call, its process is suspended and a `SIGCHLD` signal is sent to the strace process.
- strace then uses ptrace to read the system call arguments off the tracee address space.
- strace resumes the tracee until the tracee is suspended again when it is about to exit the system call.

Clearly, at least two context switches are involved at the points of entry and exit. Using ptrace to read the system call arguments is also inefficient, since ptrace itself is a system call and ptrace can only read a machine word at a time, that is, 4 bytes for a 32-bit machine. There is a lot of overhead involved when recording a system call and its arguments.

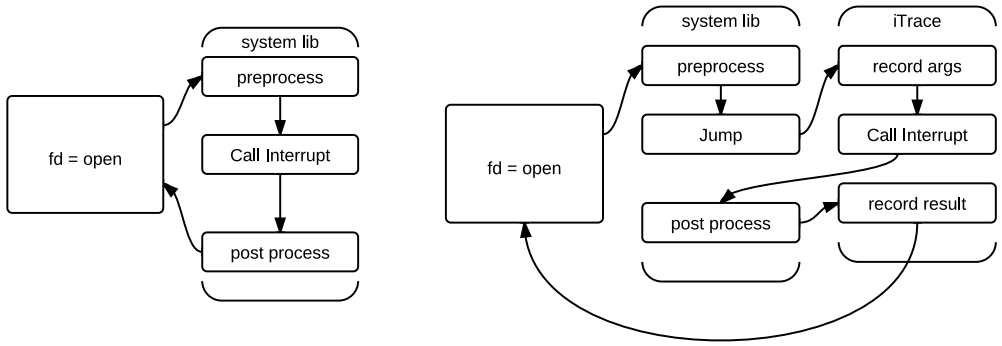
A relatively new entrant to this instrumentation space is utrace. [12] Utrace works by placing tracepoints at strategic points in kernel code. For traced threads, these tracepoints yield calls into the registered utrace clients. These callbacks, though happening in the context of a user process, happen when the process is executing in kernel mode. In other words, utrace clients run in the kernel. In some operating systems, ptrace is a utrace client that lives in the kernel. Other clients – especially those used for ad hoc instrumentation – may be implemented as kernel modules.

5.2 In-process tracing

The key idea of in-process tracing is to load a tracing library into the tracee address space and to redirect system call stubs to the tracing library. There are four benefits. First, recording of each system call and its arguments in the userspace, that is, in the context of the tracee process, is much faster. Second, since there is no need to write custom kernel modules (as in the case of utrace), this technique provides a safer and easier way to collect traces. Third, there is *no* performance penalty at all when tracing is not activated, since the stubs are redirected only when tracing is activated. This is unlike utrace which has to explicitly check for registered callbacks even when tracing is not activated. Fourth, iTrace allows specific system calls to be traced unlike strace and utrace which trace all system calls. We discuss the technique in the context of iOS since it is the first target platform for which iTrace is developed. We plan to port iTrace to Linux in the future.

Every program in iOS is linked against the system call stubs library `libSystem.B.dylib`. A system call stub does a software interrupt to invoke the relevant kernel routine, and after returning from the routine, the stub checks for errors and if necessary, returns the appropriate error code as the return value.

iTrace uses GDB to attach to the tracee and to load our tracing library into its address space. GDB is also used to rewrite parts of a system call stub to redirect the execution control to our tracing library before the software interrupt is invoked. The library then records the system call arguments, and invokes the software interrupt on behalf of the stub. Control is then transferred back to the stub so that the appropriate return value can be set and captured by our tracing library. This is necessary as we do not know how the error code is determined when error occurrs. Figure 7 shows how iTrace changes the program control flow without changing the original system call semantics.



(a) Before change (b) After change

Figure 7: How the original control flow of calling `open` is changed by iTrace.

Reliance on GDB could be removed in the future by extracting the relevant features iTrace requires.

5.3 Performance comparison

Unfortunately, in the interests of time, we were unable to compare with utrace. We timed the overhead of calling the `gettid` system call 100,000 times. `gettid` is chosen as we believe its implementation is simple and it has no argument, and hence the timing would be fairly stable

and largely consists of the tracing tool overhead. For a fair comparison, iTrace and strace both log to a file in the exact same format.

The average across ten runs are reported below. All timings are in milliseconds.

	No tracing	With tracing	Overhead
iTrace (iOS)	80	870	11x
strace (Android)	72	9900	138x

Even though both tools run on different platforms with different hardware, from the timing without tracing, we can see that both platforms have similar performance for the `gettid` system call. Furthermore, we believe that the differences between both platforms and hardware could not possibly contribute a lot to the large difference in the overhead ratios. The disparity in overhead ratios should be due to the in-process tracing technique we used.

For a more accurate timing, we could turn off logging for both tools, which is impossible for strace currently. We could also use a version of iTrace ported to Android. Yet another possibility is to time how long it would take to produce the same log file on both platforms without doing any tracing at all, and subtract it off the timings shown in the table above. We delegate these as a future work.

6 Conclusions

Our study shares similar findings with [10]:

- *Auxiliary files dominate:* On iOS, applications access resource, temp, and plist files very often. This is especially true for Facebook which uses a large number of cache files. We are not sure why this is the case since small pieces of data like comments and news feed could be stored in a database. Also for iOS resource files such as icons and thumbnails are stored individually on the file system.

- *Writes are forced:* This is true for both platforms, especially for the Facebook application. On iOS, Facebook calls `fsync` even on cache files, resulting in the largest number of `fsync` calls out of the applications. On Android, `fsync` is called for each temporary write-ahead logging journal files, which we think is reasonable. However, we think that the Facebook app on Android has much more write-ahead logging due to the use of frameworks since write-ahead logging is typically not controlled by the application itself.

- *Frameworks influence I/O:* The SQLite database library is used often on both platforms to store cache files and metadata. Since the library supports atomic transactions, there is a need for write-ahead logging which creates many short-lived temporary journal files and calls `fsync` often.

From the analysis, it is clear that the same application has very different I/O behavior on each platform. This might be due to the difference in frameworks being used or the developer using a different implementation (logically) to achieve the same functionalities. The latter is unlikely due to development time and maintenance issues. We believe a lot of undesirable I/O behavior can be resolved through careful application design and programming. But the fact that this is not done suggests that mobile devices today are sufficiently powerful that optimizations in I/O are not required. We think the single most significant conclusion is that mobile applications behave very similarly to their desktop counterparts, in terms of I/O, and yet mobile devices are more typically hardware-constrained. Furthermore, we did not notice any poor user experience,

such as lagging or hanging, when we run the applications without tracing. This means that mobile devices today are already powerful enough to run day-to-day, albeit *simple* productive applications. It may just be a matter of time before more complex applications like Microsoft Word could run on mobile devices.

Applications like Facebook and browser which create many temporary files might have a negative impact on the *durability* of the flash storage device. File system designers could keep this mind when designing a file system for flash devices. Also, creating many files result in storage fragmentation. The temporary files usually reside in system-designated temporary directories. Perhaps file system designers could use this as a hint to optimize storage for temporary files.

We collected system call traces for commonly-used applications and summarized and presented our findings from these traces. Our hope is that they shed light on application behavior and open up areas for further investigation.

We also demonstrated the use of iTrace to collect traces on iOS. This creates opportunities for many types of research that were previously impossible to conduct on iOS. iTrace superior speed enables time-sensitive analysis to be conducted as well.

7 Future Work

- *Deeper analysis.* We would like to analyze the traces in greater depth. Specifically, we are interested in these questions: how does the use of `mmap` affect I/O, what are the pros and cons of packaging an app as an APK file, what are the underlying reasons behind the different I/O behavior between Android and iOS applications, how does the use of Dalvik VM affect the performance of an application. Furthermore, there are some interesting areas in the traces that are worth analyzing further.
- *Broader study.* We would also like to investigate other areas such as network activities and memory management. We already have some interesting observations, for example, iOS has a high failure rate for reading from a socket.
- *Tracing.* We wish to be able to activate tracing the instant an iOS application is launched. Also we would like to port iTrace over to Android.
- *Experiments.* More apps from other categories such as Youtube and music player could be included so as to make the study more complete. We also wish to automate the workload (actions) that we carried out manually in this study. On Android, we could use the Monkey script [2] runner which can execute predefined actions in any application. We could develop a similar tool for iOS.
- *File system.* It is interesting that both platforms do not use a log-filesystem. Instead they use traditional filesystems designed for desktops. Android does use YAFFS2 which is designed specifically for flash storage, but it is only used for non-critical partitions. It is worthwhile to investigate the impact of filesystem on the flash storage device.
- *Other platforms.* We wish to extend our study to other platforms such as Windows mobile. Also we want to investigate whether newer devices such as iPhone 4S / 5 and Google Nexus would produce different findings.

References

[1] Analysis and comparison with android and iphone operating system. `http://www.eecs.ucf.edu/~dcm/Teaching/COP5611Spring2010/Project/AmberChang-Project.pdf`.

[2] Android SDK tools. http://developer.android.com/guide/developing/tools/index.html.

[3] blktrace manual page. http://linux.die.net/man/8/blktrace.

[4] comScore reports december 2011 u.s. mobile subscriber market share. http://www.comscore.com/Press_Events/Press_Releases/2012/2/comScore_Reports_December_2011_U.S._Mobile_Subscriber_Market_Share.

[5] instruments. https://developer.apple.com/library/mac/#documentation/developertools/conceptual/InstrumentsUserGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004652-CH1-SW1.

[6] ptrace manual page. http://linux.die.net/man/2/ptrace.

[7] strace manual page. http://linux.die.net/man/1/strace.

[8] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.

[9] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2011).

[10] 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. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 71–83.

[11] Huang, J., Xu, Q., Tiwana, B., Mao, Z. M., Zhang, M., and Bahl, P. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 165–178.

[12] Keniston, J. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Proceedings of the 2007 Linux Symposium*, OSL'07.

[13] Kim, H., Agrawal, N., and Ungureanu, C. Revisiting storage for smartphones. In *10th USENIX Conference on File and Storage Technologies* (2012), FAST '12.

[14] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. How effective is mobile browser cache? In *Proceedings of the 3rd ACM workshop on Wireless of the students, by the students, for the students* (New York, NY, USA, 2011), S3 '11, ACM, pp. 17–20.

[15] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. Why are web browsers slow on smartphones? In *ACM HotMobile 11* (2011), 11.