Fjord: Informed Storage Management for Smartphones

Hyojun Kim
IBM Research
California, USA
Email: hyojun@us.ibm.com

Umakishore Ramachandran
Georgia Institute of Technology
Georgia, USA
Email: rama@cc.gatech.edu

***Abstract*—Smartphone applications are becoming more sophisticated and require high storage performance. Unfortunately, the OS storage software stack is not well engineered to support flash-based storage used in smartphones. On top of that, storage software stack is configured to be too conservative due to the fear of sudden power failures. We believe that this conservatism with respect to data reliability is misplaced considering that many of the popular apps (e.g., Web browsing, Facebook, Gmail) that run on today's smartphones are cloud-backed, and the local storage on the smartphone is often used as a cache for cloud data.**

In this paper, we propose *Informed Storage Management* framework, named *Fjord*, for mobile platforms. The key insight is to use *system-wide dynamic context information* to improve the storage performance on mobile platforms. We implement a set of mechanisms (write buffering, logging, and fine-grained reliability control), and through judicious use of these mechanisms based on system context, we show how we can achieve significant improvement in storage performance. As proof of concept, we implement Fjord in two Android smartphones and experimentally validate the performance advantage of informed storage management with multiple smartphone applications.

I. Introduction

Smartphones have become an essential part of our daily life. Today, more smartphones than PCs are being shipped [1], and the number of people subscribing to mobile phones is bigger than the number of people subscribing to electricity and safe drinking water [2].

In contrast to regular desktop and server systems, the storage subsystem can easily become a performance bottleneck in smartphones. Smartphones use low-end flash storage devices such as embedded Multimedia Card (eMMC) chips and microSDHC cards, and their performance is relatively limited compared to regular computer storage devices such as Solid-State Drives (SSDs) and Hard Disk Drives (HDDs). For instance, sequential write throughputs are about 400MB/second and 150MB/second on a recent SSD and HDD, respectively [3], but they are only about 15MB/second and 12MB/second on eMMC and microSDHC card, respectively; the performance gap is even bigger for random write throughputs [4], [5].

The storage performance is limited also by conservative configurations of the smartphone OS; the safer but slow choices normally win over faster but risky ones in designing smartphones. As a concrete example, Google Android 4.0.4 uses write barrier enabled EXT4 journaling file system instead of the faster EXT2 file system, and uses very conservative configurations to minimize the possibility of losing dirty page content due to unexpected system crashes - `dirty_expire_centisecs` and `dirty_background_ratio` values are reduced from 30 seconds to 2 seconds and from 10 percent to 5 percent, respectively. Of course, the configurations limit the capability of Linux page cache, and it can be a limitation to the performance of smartphone storage. Consequently, the low-end flash storage easily becomes the Achilles' heel when it comes to performance of mobile platforms [6], [4].

However, such a conservative approach that sacrifices performance for storage reliability is unwarranted for all applications. In many cases, smartphones are terminal devices for cloud contents, and local storage is used mostly as a cache for data that already resides safely in the cloud. Facebook, web browser, Twitter, Google Maps are good examples. For such applications, users may prefer performance to reliability; loss of the cached content is not catastrophic since the original content is safely in the cloud.

However, there is no systematic method to selectively control the conservative storage configurations of the smartphone OS for such applications that can use relaxed semantics for reliability to gain higher performance. If smartphone OS can provide a fine-grained control mechanism to tradeoff reliability for performance, then it will be possible to get better performance for some applications without losing reliability for critical applications (e.g., bank transactions).

We propose a novel *Informed Storage Management* framework, named Fjord[1]. Fjord includes three key components: 1) controlled write buffering layer based on logging and RAM based write-back buffering, 2) fine-grained control mechanism to trade off reliability for performance, and 3) a framework to control these solutions dynamically. The third component, namely, a framework for dynamic control will not be covered in this paper due to space constraints.

We implement and evaluate Fjord on two real Android smartphones, and demonstrate significant performance gains with SQLite benchmark application as well as Email and Web Browsing clients. For instance, the elapsed time for running the Email test case is reduced from 34.6 seconds to 16.1 seconds on Samsung Galaxy Note phone with Fjord. Overall, we see a performance improvement by a factor of 2 for many test cases.

[1]The English word Fjord is derived from a Scandinavian word that signifies a narrow and often shallow area in a river for crossing from one side to the other on foot...an analogy for our thin system software layer that allows safely moving data from higher levels of system software to the storage device.

The rest of the paper is structured as follows. Section II provides a brief background and related works. Section III presents the key design concepts of Fjord. Our evaluation results are presented in Section IV, and we conclude in Section V.

II. Background and Related Work

Flash memory has brought about a drastic change in storage technology recently, and there are a number of studies devoted to a deeper understanding of that technology. Most of the studies are about regular or enterprise level SSDs while only few studies deal with low-end flash storage for mobile platforms. Smartphones just a few years back used to have a bare NAND flash memory chip with a flash native file system like YAFFS2 [7], but the latest ones use eMMC devices rather than bare NAND flash memories. The upshot is that each eMMC chip has Flash Translation Layer (FTL) [8], [9] internally by using System-On-Chip technology. Eliminating the need for FTL and/or flash native file system (such as YAFFS2) on the host side helps rapid development, and the unified interfaces (at the storage system software level) can be used by the mobile platform both for an internal eMMC chip and for an external flash memory card like microSDHC. Naturally, small flash memory cards (including eMMC) suffer from many limitations. Only small amount of RAM is available for internal FTL, and these flash memory cards have severe limitations when it comes to response time and power consumption. As a result, most inexpensive flash storage devices show very poor performances especially for small random write requests, and as a consequence, inexpensive flash storage devices remain as the main source of performance bottleneck on mobile platforms [4].

Related with file system reliability, journaling [10] is being popularly used in modern file systems. To fix inconsistent file system states, journaling file system keeps track of file system changes as a journal, and uses it for a recovery. Soft-update [11], [12] was proposed to provide stronger reliability guarantees than journaling, and it attacks the meta-data update problem by guaranteeing that blocks are written to the disk in their required order without using synchronous disk I/Os; Seltzer *et al.* have compared the file system performance of Soft-update and journaling [13].

Ensuring write ordering is an essential part of both Soft-update and journaling file systems. Most storage devices have volatile on-board write buffer to improve storage performance, and consequently write ordering is not guaranteed. In other words, the storage devices internally decide as to when the writes pending in the on-board write buffer are committed to the physical medium. To enforce write ordering under these circumstances, storage devices expose a *write barrier* interface to the OS. Whenever a storage device receives a write barrier from the upper layers of the OS, it has to ensure that the content of the on-board buffer is written to the physical storage media safely. That is, a write barrier limits the capability of on-board write buffer, and thus frequent use of write barrier can degrade the overall file system performance significantly [14].

EXT4 file system is used as a default file system in today's Android smartphones, and provides three different data modes related with file system consistency: *write-back*, *ordered*, and

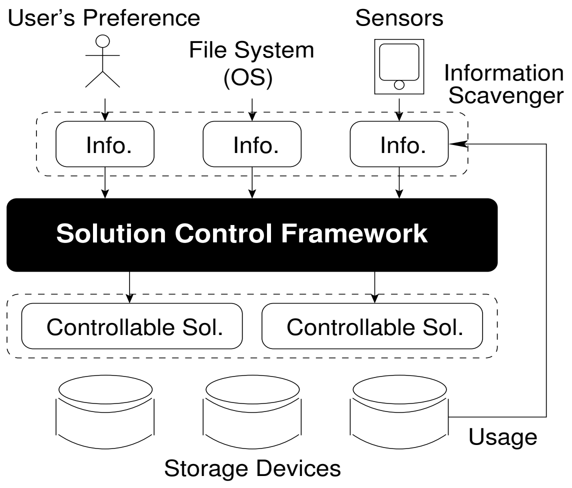

Fig. 1. Informed Storage Management:*solutions are selectively, dynamically, and systematically used based on various context information.*

journal [15]. With the fastest *write-back* mode, EXT4 does not journal user-data at all, and provides only file system metadata consistency. That is, a crash can cause incorrect data to appear in files, which were written shortly before the crash. When the *ordered* mode (which is the default) is used in EXT4, once again only the metadata is written to the journal, but the metadata update happens only after the associated data blocks have been written to the storage first. With the safest - but slowest - *journal* mode, both user-data and file system metadata are written to the journal first, than written to their final locations. Within Android version 4.0.4, EXT4 file system is being used with the default `data=ordered, barrier=1` options, and the performance of the option is between *write-back* and *journal* options. In EXT4 file system, it is not possible to use different journaling options for different files even if each file may have different levels of reliability requirements.

III. Informed Storage Management

Informed Storage Management (ISM) is aimed at providing a dynamic decision-making framework for mobile system design, specifically targeted to storage systems. The goal is maximizing the performance benefits while minimizing the side-effects of the design choice.

ISM consists of three major components: 1) controllable software modules that represent different design choices that can be harnessed on the fly, and whose behavior can be altered on the fly, 2) a control framework for selecting a software solution on the fly, and 3) a scavenger that gathers fine-grained information to feed to the control framework for just-in-time decision-making. Figure 1 shows this concept.

As a concrete example of ISM, we build Fjord within the Android platform.

Figure 2 shows the overall architecture of Fjord and its relationship to the Android system. We add a new *controlled write buffering layer* between the file system and block device layers, and this layer optimizes write requests considering the general performance characteristics of flash storage. We also modify Android, Linux page cache, and the file system layers

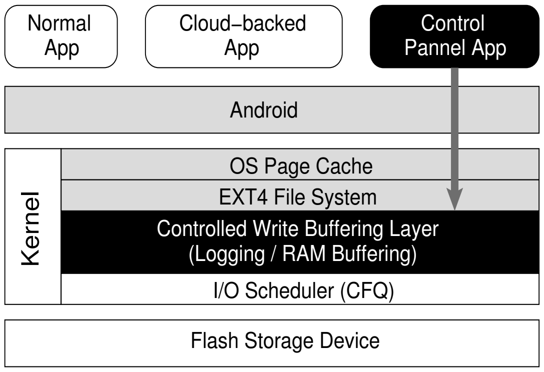

Fig. 2. Fjord Architecture: *Black boxes represent newly inserted components, and gray boxes represent existing Android components with our modifications.*

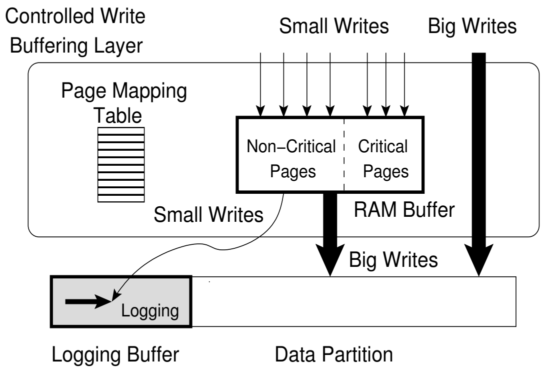

Fig. 3. Controlled Write Buffering Layer: *Non-critical pages are allowed to stay in RAM buffer ignoring a write barrier.*

to provide fine-grained reliability control. All files are not equally important, and it is especially true on mobile platforms because the applications are tightly coupled with the cloud storage in many cases. Fjord can relax reliability constraint selectively for such files to improve storage performance. On top of these solutions, Fjord has an Android App, which controls the solutions based on various dynamic context information.

A. Controlled Write-back Buffering

Figure 3 shows our design for controlled write buffering layer, which combines non-volatile logging and RAM buffering together. This layer is designed to improve write performance and acts as follows. When a write request is given, it is acted upon based on its size. If the request size is big enough, the request bypasses our buffering layer because flash storage can handle big-sized write requests efficiently. Besides, a big-sized request will consume more space in our buffering layer. In the Fjord framework, a write request is tagged to indicate whether the request is for a critical or non-critical file (this will be explained in the following Section III-B). Both non-critical and critical small-sized write requests go to the RAM buffer, but the tagging information is conveyed to the RAM buffer to guarantee that the critical data are flushed upon a write barrier from the upper layer. While staying in the RAM buffer write requests are reordered and merged for better performance. When data pages are evicted from the RAM buffer, they can go to the non-volatile logging buffer or the final flash storage based on the their size. That is, the RAM buffer acts as a staging area for small write requests; spatially near, but temporally separated write requests are merged to a big write request, and directly written to the final location.

B. Fine-grained Reliability Control

One fundamental question is how to distinguish cloud-backed applications from others. For a long-term view, we believe that the smartphone OS should provide proper interface (APIs) to let developers define their applications as cloud-backed applications. However, such an approach is difficult to implement for the purposes of this study because it requires modifications to both the smartphone OS and the applications. Instead, for the purposes of this study, we maintain a *whitelist* (opposite to the concept of *blacklist*) in our prototype, and manually configure a particular application as a cloud-backed application. In the Android system, each application writes only to well separated individual application storage space under `/data/data/` directory, and thus, by providing the home directory name of an Android application, we can easily find all files from the applications. The whitelist serves as the set of file names that are buffer-able from the point of view of our fine-grained reliability control.

Fjord also distinguishes file system metadata from user-data, and applies relaxed reliability semantics only for the user-data of the files within the whitelist. It can be done of course by file systems, but for the purposes of this study, we take a different approach. Instead of modifying the file system code, we modify the page cache related function of Linux kernel, `generic_perform_write()`, within `mm/filemap.c`. The function is called when a file is written by the file system write APIs, and it eventually copies the written data to a page frame in the Linux page cache. Since the page in this case is holding user-data of a file, we annotate this information within the page data structure. For this purpose, we add a new page bit flag (`include/linux/page-flags.h`) and set the bit within the `generic_perform_write()` function.

In addition to the controlled write-back buffering, Fjord includes a simple but very effective mechanism, which is selectively ignoring fsync() requests only for non-critical files within the whitelist. We modify `ext4_sync_file()` function within `fs/ext4/fsync.c` file to return without syncing the file when the file is marked as buffer-able.

IV. Evaluation

To see the performance effect of Fjord on various hardware, we choose two smartphones: Google Nexus One having Android 2.3.7 and Samsung Galaxy Note N7000 with Android 4.0.4. The Nexus One has a single-core 1 GHz Qualcomm QSD8250 Snapdragon processor, 512MB RAM, and 512MB internal flash storage while the Galaxy Note has a dual-core 1.4GHz Samsung Exynos processor, 1GB main memory, and 16GB eMMC storage; both phones have external memory card slots. We also choose two typical storage devices for smartphones: 1) the eMMC device used in the Galaxy Note phone, and 2) 8GB sized class 10 microSDHC card from Samsung.

We assign 16MB for the RAM buffer[2], and 128MB for

[2]We have evaluated with different sizes of the RAM buffer, and there were no meaningful differences in the observed results.

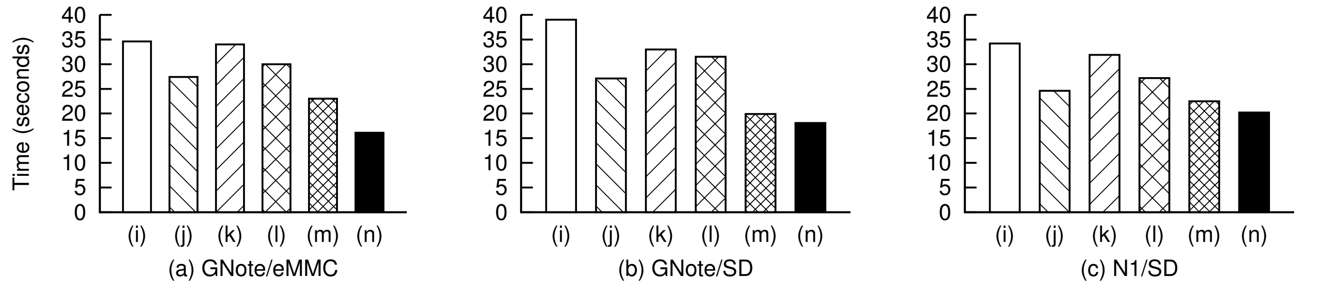

Fig. 4. **Email**: *(i) Native, (j) Logging, (k) RAMBuf, (l) Both, (m) FjordRAM, (n) FjordBoth. Fjord eliminates between 41 - 55% of the execution time.*

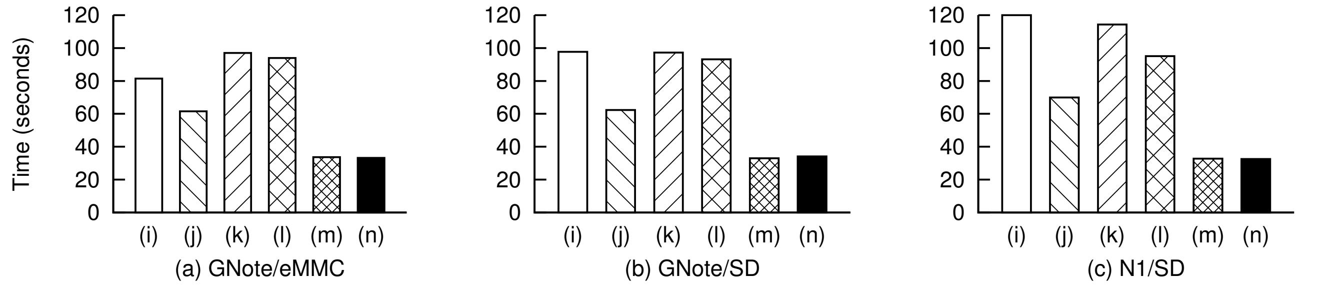

Fig. 5. **RL Benchmark**: *(i) Native, (j) Logging, (k) RAMBuf, (l) Both, (m) FjordRAM, (n) FjordBoth. Fjord eliminates between 59 - 73% of the execution time.*

the non-volatile logging buffer. All other configurations use Android Open Source Project (AOSP) [16] defaults: EXT4 file system and CFQ I/O scheduler, etc.

We compare the performance for six different storage software configurations: (i) the original storage (Native), (j) only the logging buffer is enabled (Logging), (k) only the RAM buffer is enabled (RAMBuf), (l) Both the RAM and the logging buffers are enabled (Both), (m) Fjord selective buffering is used with the RAM buffer (FjordRAM), and (n) Fjord selective buffering is used with both the RAM and the logging buffers (FjordBoth). Note that the first four configurations do not sacrifice reliability while the last two configurations (FjordRAM and FjordBoth) sacrifice reliability for chosen application files. All tests have been repeated 3 times, and the average numbers are reported in our results; the variations were small enough to be ignored.

Our first test case is the default Email App of Android OS. Email is perhaps the most popular application on smartphones, and its performance is highly reliant on network as well as local storage performance because it downloads emails to the local storage. To obtain repeatable results, we prepare an email account having about 150 emails in its inbox. We launch Email App, input information about the email account, and then measure the initial email downloading time. The time is measured by monitoring CPU usage rate; CPU rate is between 50-90% during email downloading, and drops down to around 10% when the task completes. We automate the measurement process by using Android Monkey Runner framework [17] as well as our custom utility, which monitors CPU utilization in the background.

Figures 4 (a), (b), and (c) show the measured run-times on the Galaxy Note with the eMMC, Galaxy Note with microSDHC, and Nexus One with microSDHC, respectively. Let us first consider the native results on the different platforms ("(i)" on the x-axis). On the Galaxy Note, the result with eMMC (Figure 4-(a)) is faster (34.6 seconds) than the result with microSDHC card (Figure 4-(b), 39.9 seconds) as expected. Interestingly, the old Nexus One (Figure 4-(c)) shows the shortest execution time (34.2 seconds) than the newer and faster Galaxy Note phone. The difference could be due to several reasons: Linux Kernel, Android OS, or Email App itself.

Next we consider the results with the logging solution ("(j)" on the x-axis). Small sized ($\leq$ 4KB) write requests are written sequentially to the non-volatile logging buffer regardless of its final destination address. The solution improves the performance very effectively, and it is more effective with microSDHC card because its performance is more sensitive to the write pattern.

The results with the RAM based write buffer are shown as the third bars from the left with the legend "(k)" on the x-axis in Figures 4-(a)-(c). The RAM buffer could also help to change the write pattern to be more sequential, but its capability is limited by write barriers because it has to flush out all its content to the non-volatile storage to obey write barrier semantics, and EXT4 file system generates write barriers very frequently.

The fourth bars from the left represent the results with both the volatile RAM and the non-volatile logging buffers. All written data go to the RAM buffer first, and move to non-volatile logging buffer selectively. It can be seen that the results are worse than the logging only solution. This is because the results for the logging only solution (bars with the legend "(j)" on the x-axis) do not include garbage collection overhead since the reported numbers for that experiment are obtained when there is enough free space in the logging buffer.

The next two bars (legends "(m)" and "(n)" in Figures 4-(a)-(c)) show the results with our precise write buffer controlling mechanisms. For cloud-backed applications, the local dirty file content (which is really the meta-data of the App for manipulating the real data objects in the cloud) is not that critical because the real content needed by the App (e.g., widgets displayed by a browser) are safely in the cloud, and can be easily recovered over the network. By allowing non-critical dirty data to stay in the RAM buffer, the storage write amount can be reduced, and the performance can be improved significantly. On the Galaxy Note, the written data amount is reduced from about 35MB to 11MB for both (m) and (n) result bars in Figures 4-(a)-(c). The rightmost bars in Figure 4 (solid black bars in each graph cluster) show the results with our final integrated solution; the run-time has been reduced by 53%, 55%, 41% on three smartphone/ storage configurations, respectively.

Our second test case is a database benchmark named *RL Benchmark: SQLite* [18]. This benchmark measures the database performance of Android system by running synthetic database queries. In the latest study about Android application performance [4], the database is known to be a key performance contributor, and thus, this benchmark is very useful to see the performance effects of our storage solution. This benchmark is freely available, and produces repeatable results. Figure 5 shows very similar trend to the email results in Figure 4, but there are a few points worthy of elaboration.

The first point is shown in Figure 5 (a); RAM buffering only and both RAM / logging buffering solutions show worse performance than the native performance on Galaxy Note with eMMC. This is due to the CPU and memory copy overheads since the total amount of storage access is unchanged. RAM buffering necessarily incurs some processing overhead compared to the Native configuration but the hope is that the improvement in storage performance will more than compensate for this overhead. The write performance of the eMMC device is less sensitive to request ordering, and thus RAM buffering on the eMMC device results in worse performance than the Native configuration.

The second interesting point is regarding FjordRAM and FjordBoth solutions: the measured run-times are almost the same for FjordRAM and for FjordBoth solutions (legends "(m)" and "(n)", respectively, on the x-axis in Figures 5-(a)-(c)). Recall that there is a clear difference between the two solutions for the Email test case. Non-volatile logging requires additional space in the storage, and it may not be readily available always. This result suggests that FjordRAM solution could be a convenient alternative to FjordBoth when storage space is limited.

The final point is that our integrated storage solution removes the dependencies on storage sophistication; Nexus One with microSDHC card is about 50% slower than Galaxy Note with eMMC for this benchmark in the Native configuration. However, with Fjord, Nexus One becomes even slightly faster than Galaxy Note. This is a good example showing that when we have the right OS support (Fjord), we can achieve better performance than using a hardware solution (eMMC) to circumvent the performance issues of mobile flash storage.

V. Conclusion

Due to size, power, and cost considerations, smartphones will continue to deploy low-end flash memories as the primary storage. Therefore, it is important to consider what can be done in the OS to enhance the performance of flash based storage systems. In this paper, we propose multiple solutions from different levels of storage software stack to improve the storage performance of mobile platforms. Based on the understanding of low-end flash storage devices, we re-design two typical storage solutions (logging and RAM buffering) for smartphones. We also introspect on the right level of reliability requirement for cloud-backed applications on smartphones. We modify Linux page cache and file system layers to provide fine-grained control for caching and buffering, and demonstrate that we can effectively improve the performance of chosen cloud-backed applications without compromising the integrity of other applications or the Android system itself. Finally, we implement our integrated solution into real Android smartphones, and show that the solution can effectively improve application performance. For example, Email downloading time improves almost by a factor of two compared to the Native configuration. Even though we are focusing on smartphone storage in this paper, we believe that some ideas have potentials beyond smartphones for other types of storage systems.

References

[1] "Worldwide quarterly mobile phone tracker," 2011, http://www.idc.com/research/viewfactsheet.jsp?containerId=IDC_P8397.

[2] C. Sharma, "Global Mobile Market Update," http://www.chetansharma.com/GlobalMobileMarketUpdate2012.htm, 2012.

[3] tom's hardware, "Performance Charts Hard Drives and SSDs," http://www.tomshardware.com/charts/hard-drives-and-ssds,3.html.

[4] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. of the 10th USENIX conference on File and storage technologies*, 2012.

[5] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012.

[6] H. Kim, N. Agrawal, and C. Ungureanu, "Examining storage performance on mobile devices," in *Proc. of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, 2011.

[7] A. O. Ltd, "Yaffs: A NAND-Flash Filesystem," http://www.yaffs.net.

[8] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," White Paper, http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proc. of the USENIX 1995 Technical Conference Proceedings*.

[10] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proc. of the annual conference on USENIX Annual Technical Conference*, 2005.

[11] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," *ACM Trans. Comput. Syst.*, vol. 18, no. 2, May 2000.

[12] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *Proc. of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[13] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein, "Journaling versus soft updates: asynchronous meta-data protection in file systems," in *Proc. of the annual conference on USENIX Annual Technical Conference*, 2000.

[14] J. Corbet, "Barriers and journaling filesystems," http://lwn.net/Articles/283161/, May 2008.

[15] "Documentation/filesystems/ext4.txt," http://lwn.net/Articles/203915/.

[16] "Android Open Source Project," http://source.android.com/index.html.

[17] "MonkeyRunner for Android Developers," http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

[18] RedLicense Labs, "RL Benchmark: SQLite," https://market.android.com/details?id=com.redlicense.benchmark.sqlite.