Revisiting Storage for Smartphones

HYOJUN KIM, NITIN AGRAWAL, and CRISTIAN UNGUREANU, NEC Laboratories America

Conventional wisdom holds that storage is not a big contributor to application performance on mobile devices. Flash storage (the type most commonly used today) draws little power, and its performance is thought to exceed that of the network subsystem. In this article, we present evidence that storage performance does indeed affect the performance of several common applications such as Web browsing, maps, application install, email, and Facebook. For several Android smartphones, we find that just by varying the underlying flash storage, performance over WiFi can typically vary between 100% and 300% across applications; in one extreme scenario, the variation jumped to over 2000%. With a faster network (set up over USB), the performance variation rose even further. We identify the reasons for the strong correlation between storage and application performance to be a combination of poor flash device performance, random I/O from application databases, and heavy-handed use of synchronous writes. Based on our findings, we implement and evaluate a set of pilot solutions to address the storage performance deficiencies in smartphones.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms: Performance, Measurement

Additional Key Words and Phrases: Storage systems, mobile, smartphones, mobile storage, Android

ACM Reference Format:
Kim, H., Agrawal, N., and Ungureanu, C. 2012. Revisiting storage for smartphones. ACM Trans. Storage 8, 4, Article 14 (November 2012), 25 pages.
DOI = 10.1145/2385603.2385607 http://doi.acm.org/10.1145/2385603.2385607

1. INTRODUCTION

Mobile phones, tablets, and ultra-portable laptops are no longer viewed as the wimpy siblings of the personal computer. For many users, they have become the dominant computing device for a wide variety of applications. According to a recent Gartner [2011] report, within the next 3 years, mobile devices will surpass the PC as the most common Web access device worldwide. By 2013, more than 40% of the enhanced phone installed base will be equipped with advanced browsers [Pentin 2010].

Research pertaining to mobile devices can be broadly split into applications and services, device architecture, and operating systems. From a systems perspective, research has tackled many important aspects: understanding and improving energy management [Flinn and Satyanarayanan 1999; Roy et al. 2011; Carroll and Heiser 2010], network middleware [Meroni et al. 2010], application execution models [Cuervo et al. 2010; Chun et al. 2011], security and privacy [Bickford et al. 2011; Dietz et al. 2011; Enck et al. 2010; Geambasu et al. 2011], and usability [Castellucci and

An earlier version of this article appeared in the *Proceedings of the 10th File and Storage Technologies Conference (FAST)*, San Jose, CA, 2012 [Kim et al. 2012].
Hyojun Kim was an intern at NEC Labs during the course of the work and is now at Georgia Institute of Technology.
Author's address: Nitin Agrawal, NEC Laboratories America; email: nitin@nec-labs.com.

DOI 10.1145/2385603.2385607 http://doi.acm.org/10.1145/2385603.2385607

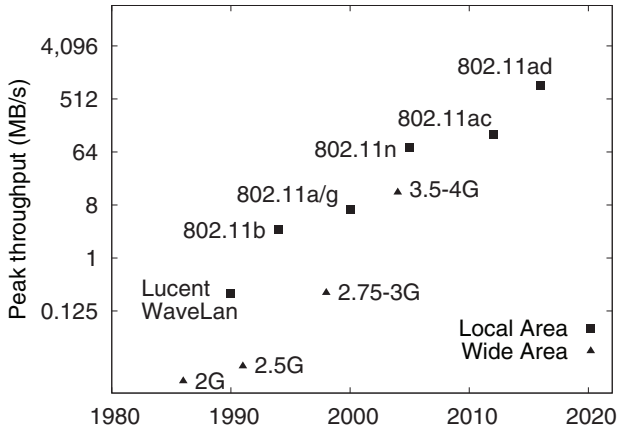

Fig. 1. Peak throughput of wireless networks. Trends for local and wide area wireless networks over past 3 decades; y-axis is log base 2.

MacKenzie 2011]. Prior research has also addressed several important issues centered around mobile functionality [Noble et al. 1997; Tolia et al. 2004], data management [Veeraraghavan et al. 2010], and disconnected access [Kistler and Satyanarayanan 1992; Flinn et al. 2003]. However, one important component is conspicuously missing from the mobile research landscape—storage performance.

Storage has traditionally not been viewed as a critical component of phones, tablets, and PDAs—at least in terms of the expected performance. Despite the impetus to provide faster mobile access to content locally [Gundotra and Barra 2011] and through cloud services [Satyanarayanan 2010], performance of the underlying storage subsystem on mobile devices is not well understood. Our work started with a simple motivating question: Does storage affect the performance of popular mobile applications? Conventional wisdom suggests the answer to be no, as long as storage performance exceeds that of the network subsystem. We find evidence to the contrary—even interactive applications like Web browsing slow down with slower storage.

Storage performance on mobile devices is important for end-user experience today, and its impact is expected to grow due to several reasons. First, emerging wireless technologies such as 802.11n (600Mbps peak throughput) [IEEE WG802.11 2009] and 802.11ad (or "60GHz," 7Gbps peak throughput) offer the potential for significantly higher network throughput to mobile devices [Halperin et al. 2011]. Figure 1 presents the trends for network performance over the last several decades; local area networks are not necessarily the de facto bottleneck on modern mobile devices. Second, while network throughput is increasing phenomenally, latency is not [Satyanarayanan et al. 2009]. As a result, access to several cloud services benefits from a split of functionality between the cloud and the device [Chun et al. 2011], placing a greater burden on local resources including storage [Koukoumidis et al. 2011]. Third, mobile devices are increasingly being used as the primary computing device, running more performance intensive tasks than previously imagined. Smartphone usage is on the rise; smartphones and tablet computers are becoming a popular replacement for laptops [Motorola 2011]. In developing economies, a mobile/enhanced phone is often the only computing device available to a user for a variety of needs.

In this article, we present a detailed analysis of the I/O behavior of mobile applications on Android-based smartphones and flash storage drives. We particularly focus on popular applications used by the majority of mobile users, such as Web browsing, app install, Google Maps, Facebook, and email. Not only are these activities available on

almost all smartphones, but they are also done frequently enough that performance problems with them negatively impacts user experience. Furthermore, we provide pilot solutions to overcome existing limitations.

To perform our analysis, we build a measurement infrastructure for Android consisting of generic firmware changes and a custom Linux kernel modified to provide resource usage information. We also develop novel techniques to enable detailed, automated, and repeatable measurements on the internal and external smartphone flash storage, and with different network configurations that are otherwise not possible with the stock set-up. For automated testing with GUI-based applications, we develop a benchmark harness using MonkeyRunner [2012].

In our initial efforts, we propose and develop a set of pilot solutions that improve the performance of the storage subsystem and consequently mobile applications. Within the context of our Android environment, we investigate the benefits of employing a small amount of phase-change memory to store performance critical data, a RAID driver encompassing the internal flash and external SD card, using a log-structured file system for storing the SQLite databases, and changes to the SQLite `fsync` code path. We find that changes to the storage subsystem can significantly improve user experience; our pilot solutions demonstrate possible benefits and serve as references for deployable solutions in the future.

As the popularity of Android-based devices surges, the set-up we have examined reflects an increasingly relevant software and hardware stack used by hundreds of millions of users worldwide; understanding and improving the experience of mobile users is thus a relevant research thrust for the storage community. Through our analysis and design, we make several observations:

Storage affects application performance. Often in unanticipated ways, storage affects performance of applications that are traditionally thought of as CPU or network bound. For example, we found Web browsing to be severely affected by the choice of the underlying storage; just by varying the underlying flash storage, performance of Web browsing over WiFi varied by 187% and over a faster network (set up over USB) by 220%. In the case of a particularly poor flash device, the variation exceeded 2000% for WiFi and 2450% for USB.

Speed class considered irrelevant. Our benchmarking reveals that the "speed class" marking on SD cards is not necessarily indicative of application performance. Although the class rating is meant for sequential performance, we find several cases in which higher-grade SD cards performed worse than lower-grade ones overall.

Slower storage consumes more CPU. We observe higher total CPU consumption for the same application when using slower cards; the reason can be attributed to deficiencies in either the network subsystem, the storage subsystem, or both. Unless resolved, lower-performing storage not only makes the application run slower, but it also increases the energy consumption of the device.

Application knowledge ensues efficient solutions. Leveraging a small amount of domain or application knowledge provides efficiency, such as in the case of our pilot solutions. Hardware and software solutions can both benefit from a better understanding of how applications are using the underlying storage.

The contributions of this article are threefold. First, we describe our measurement infrastructure that enables custom set-up of the firmware and software stack on Android-devices to perform in-depth I/O analysis; along with the systems software, we contribute a set of benchmarks that automate several popular GUI-based applications. Second, we present a detailed analysis of storage performance on real Android smartphones and flash devices. To the best of our knowledge, no such study currently exists in the research literature. We find a strong correlation between storage and performance

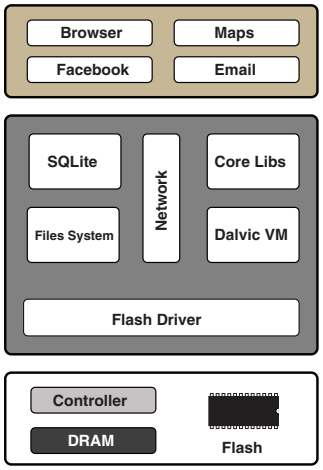

Fig. 2. Android architecture. Overview of the Android stack.

of common applications and contribute all our research findings. Third, we propose and evaluate pilot solutions to address the performance issues on mobile devices.

Based on our experimental findings and observations, we believe improvements in the mobile storage stack can be made along multiple dimensions to keep up with the increasing demands placed on mobile devices. Storage device improvements alone can account for significant improvements to application performance. Device manufacturers are actively looking to bring faster devices to the mobile market; Samsung announced the launch of a PCM-based multichip package for mobile handsets [Samsung Corp 2011]. Mobile I/O and memory bus technology needs to evolve as well to sustain higher throughput to the devices. Limitations in the systems software stack can, however, prevent applications from realizing the full potential of hardware improvements; we believe changes are also warranted in the mobile software stack to complement the hardware.

2. MOBILE DEVICE OVERVIEW

2.1. Android Overview

We present a brief overview of Android as it pertains to our storage analysis and development. Figure 2 shows a simplified Android stack consisting of flash storage, operating system (OS) and Java middleware, and applications; the OS itself is based on Linux and contains low-level drivers (e.g., flash memory, network, and power management), Dalvik virtual machine (VM) for application isolation and memory management, several libraries (e.g., SQLite, libc), and an application framework for development of new applications using system services and hardware.

The Dalvik VM is a fast register-based VM providing a small memory footprint; each application runs as its own process, with its own instance of the Dalvik VM. Android also supports "true" multitasking and several applications run as background processes; processes continue running in the background when user leaves an application (e.g., a browser downloading Web pages). Android's Web browser is based on the open-source Webkit [2012] engine; details on Android architecture and development can be found on the developer Web site [Android Developer 2011].

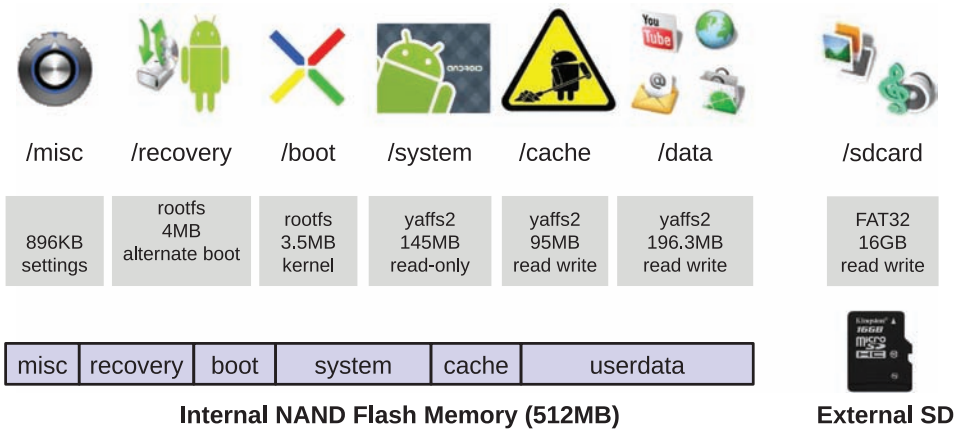

Fig. 3. Overview of Android's Storage Schema.

Table I. Data Storage Partitions for Android
Partitions on internal flash and external SD card for Nexus One phone.

Partition	Function	Size and Type
misc	Miscellaneous system settings (e.g., Carrier ID, USB config, hardware settings, IMEI number); persistent shared space for OS and bootloader to communicate	896KB
recovery	Alternative boot-into-recovery partition for advanced recovery and maintenance ops	4MB, rootfs
boot	Enables the phone to boot, includes the bootloader and kernel/initrd	3.5MB, rootfs
system	Contains remaining OS, pre-installed system apps, and user interface; typically read-only	145MB, yaffs2
cache	Android can use it to stage and apply "over the air" updates; holds system images	95MB, yaffs2
data	Stores user data (e.g., contacts, messages, settings) and installed applications; SQLite database containing app data also stored here. Factory reset wipes this partition	196MB, yaffs2
sdcard	External SD card partition to store media, documents, back-up files, etc.	multi-GB, FAT32
sd-ext	Additional partition on SD card that can act as `data` partition, set-up is possible through a custom ROM and `data2SD` software; nonstandard Android partition	Varies

2.2. Android Storage Subsystem

Most mobile devices are provisioned with an internal flash storage, an external SD card slot, and a limited amount of RAM. In addition, some devices (e.g., LG G2X phone) also have a nonremovable SD partition inside the phone; such storage is still treated as external.

Figure 3 shows the internal raw NAND and external flash storage on the Google Nexus One phone. The internal flash storage contains all the important system partitions, including partitions for the bootloader and kernel, recovery, system settings, preinstalled system applications, and user-installed application data. The external storage is primarily used for storing user content such as media files (i.e., songs, movies, photographs), documents, and back-up images. Table I presents the functionality of the partitions in detail; this storage set-up is fairly typical across Android devices.

Applications can store configuration and data on the device's internal storage as well as on the external SD card. Android uses SQLite [2012] database as the primary

means for storage of structured data. SQLite is a transactional database engine that is lightweight, occupying a small amount of disk storage and memory; it is thus popular on embedded and mobile OSs. Applications are provided a well-defined interface to create, query, and manage their databases; one or more SQLite databases are stored per application on `/data`.

The YAFFS2 [Manning 2004] file system managing raw NAND flash was traditionally the file system of choice for the various internal partitions including `/system` and `/data`; it is lightweight and optimized for flash storage. Recently, Android transitioned to Ext4 as the default file system for these partitions [Tso 2010]. Android provides a file system–like interface to access the external storage as well, with FAT32 as the commonly used file system on SD cards for compatibility reasons.

We believe the storage architecture described in this section is similar for other mobile OSs as well; for example, Apple's iOS also uses SQLite to store application data. iOS Core Data is a data model framework built on top of SQLite; it provides applications access to common functionality such as save, restore, undo, and redo. iOS 4 does not have a central file storage architecture, rather every file is stored within the context of an application. We focus on Android, since it allows systems-level development.

3. ANDROID MEASUREMENT SET-UP

Since setting up smartphones for systems analysis and development is nontrivial, we describe our process here in detail; we believe this set-up can be useful for someone conducting storage research on Android devices.

3.1. Mobile Device Set-up

In this article, we present results for experiments on the Google Nexus One phone [Nexus One 2011]. We also performed the same or a subset of experiments on the HTC Desire [HTC 2011a], LG G2X [LG 2011], and HTC EVO [HTC 2011b]; the results were similar and are omitted to save space.

The Nexus One is a GSM phone with a 1GHz Qualcomm QSD8250 Snapdragon processor, 512MB RAM, and 512MB internal flash storage; the phone is running Android Gingerbread 2.3.4, the CyanogenMod 7.1.0 firmware [CyanogenMod 2012] or the Android Open Source Project (AOSP) [2011] distribution (as needed), and a Linux kernel 2.6.35.7 modified to provide resource usage information. We present a brief description of the generic OS customizations, which are fairly typical, and then explain the storage-specific customization later in this section.

In order to prepare the phones for our experiments, we set up the Android Debug Bridge (ADB) [2011] on a Linux machine running Ubuntu 10.10. ADB is a command-line tool provided as part of Android developer platform tools that lets a host computer communicate with an Android device; the target device needs to be connected to the host via USB (in the USB debugging mode) or via TCP/IP. We subsequently root the device with unrevoked3 [Unrevoked 2012] to flash a custom recovery image (ClockworkMod [Datta 2010]).

For our experiments, we needed to bypass some of the constraints of the stock firmware; in particular, we needed support for reverse tethering the mobile device via USB, the ability to custom partition the storage, and access to a wider range of system tools and Linux utilities for development. For example, BusyBox [2008] is a software application that provides many of the standard Linux tools within a single executable, ideal for an embedded device. CyanogenMod [2012] is a custom firmware that provides these capabilities and is supported on a variety of smartphones. The (AOSP) [2011] distribution provides capabilities similar to CyanogenMod but is supported only on a handful of Google smartphones, including the Google Nexus One.

Table II. Network Performance
Transfer rates for WiFi and USB reverse tether link with iperf (MB/s).

N/W	Rx	Tx
USB	8.04	7.14
WiFi	1.10	0.53

We used the CyanogenMod distribution for all experiments on non-Nexus phones, and for experiments that require comparison between a non-Nexus and the Nexus One phone (not shown in this article). All Google Nexus One results presented in this article exclusively use AOSP; we equipped both CyanogenMod and AOSP distributions with our measurement-centric customizations.

An important requirement, specific to our storage experiments, is to be able to compare and contrast application performance on different storage devices. Some of these applications heavily use the internal nonremovable storage. In order to observe and measure all I/O activity, we change Android's `init` process to mount the different internal partitions on the external storage. Our approach is similar to the one taken by Data2SD [Starburst 2012]. in addition, we were able to also migrate to the SD card the `/system` and `/cache` partitions.

In order to adhere to Android's boot-time compatibility tests, we provided a 256MB FAT32 partition at the beginning of the SD card, mounted as `/sdcard`. The `/system`, `/cache`, and `/data` partitions were formatted as Ext3; at the time we conducted our experiments, YAFFS2 and Ext3 were the preinstalled file systems on our test phones. We performed a preliminary comparison between Ext3 and Ext4, since Android announced the switch to Ext4 [Tso 2010], but found the performance differences to be minor; a detailed comparison across several file systems can provide more useful data in the future.

Note that this set-up is not normally used by end-users, but it allows us to run what-if scenarios with storage devices of different performance characteristics; the internal flash represents only a single data point in this set.

As part of our experiments, we want to understand the impact of storage on application performance under current WiFi networks, as well as under faster network connectivity (likely to be available in the future). For WiFi, we set up a dedicated wireless access point (IEEE 802.11 b/g) on a Dell laptop having 2GB RAM and an Intel Core2 processor. Since we do not have a faster wireless network on the phone, we emulate one by reverse tethering [Carbou 2010] it over the mini-USB cable connection with the same laptop (allowing the device to access the Internet connection of the host); Table II shows the measured performance of our WiFi and USB RT link using *iperf* [SourceForge 2012].

To minimize variability due to network connections and dynamic content, we set up a local Web server running Apache on the laptop. The Web server downloads the Web pages that are to be visited during an experiment and caches them in memory; where available, we download the mobile-friendly version of a Web site.

We conducted all experiments on the internal nonremovable flash storage and eight removable microSDHC cards, two each from the different SD speed classes [SD Association 2012]. Table III lists the SD cards along with their specifications and a baseline performance measurement done on a Transcend TS-RDP8K card reader[1] using the CrystalDiskMark benchmark V3.0.1 [Crystalmark 2012] (shown on the left side). The total amount of data written is 100MB, random I/O size is 4KB, and we report average performance over three runs. The observed standard deviation is low, and we omit it

[1]Note that internal flash could not be measured this way.

Table III. Raw Device Performance and Cost
Measurements on desktop with card reader (left) and on actual phone (right). "Sq" is sequential and "Rn" is random performance.

SD Card	Speed	Cost	Performance on desktop (MB/s)				Performance on phone (MB/s)			
(16GB)	Class	US$	Sq W	Sq R	Rn W	Rn R	Sq W	Sq R	Rn W	Rn R
Transcend	2	26	4.16	18.03	1.18	2.57	4.35	13.52	1.38	2.92
RiData	2	27	**7.93**	16.29	**0.02**	2.15	5.86	11.51	0.03	2.76
Sandisk	4	23	5.48	12.94	0.68	1.06	4.93	8.44	0.67	0.73
Kingston	4	25	**4.92**	16.93	**0.01**	1.68	4.56	9.84	0.01	1.94
Wintec	6	25	15.05	16.34	0.01	3.15	9.91	13.38	0.01	3.82
A-Data	6	30	10.78	17.77	0.01	2.97	8.93	13.49	0.01	3.64
Patriot	10	29	10.54	17.67	0.01	2.96	8.83	13.38	0.01	3.72
PNY	10	29	15.31	17.90	0.01	3.56	10.28	14.02	0.01	3.95

from the table. Prices shown are as ordered from Amazon.com and its resellers, and Buy.com (to be treated as approximate). We also performed similar benchmarking experiments for the eight cards on the Nexus One phone itself, using our own benchmark program. Testing configuration is as before with 4KB random I/O size and 128MB of sequential I/O; results in Table III (shown on the right side) exhibit a similar trend albeit lower performance than for desktop.

To summarize, read performance of the different cards is not a crucial differentiating factor and much better overall than the write performance. Sequential reads clearly show little or no correlation with the speed class; sequential write performance roughly improves with speed class, but with enough exceptions to not qualify as monotonic. Random read performance is not significantly different across the cards. The most surprising finding is for random writes: Most if not all exhibit abysmal performance (0.02MB/s or less!). Even when sequential write performance quadruples (e.g., Transcend vs. Wintec), random writes perform several orders of magnitude worse.

In terms of overall write performance including random and sequential, Kingston consistently performs the worst and tends to considerably skew the results; we try not to rely on Kingston results alone when making a claim about storage performance. In practice, we find that application performance varies even with the other better cards. Transcend performs the best for random writes, by as much as a factor of 100 compared to many cards, but performs the worst for sequential writes; Sandisk shows a similar trend. A-Data, Patriot, Wintec, and PNY perform poorly for random but give very good sequential performance. Kingston and RiData suffer on both counts, as they not only have poor random write performance but also mediocre sequential write performance (shown in bold in Table III); appliation-level measurements in Section 4 reflect the consequences of the poor microbenchmark results.

3.2. Measurement Software

We first explain our measurement environment and the changes introduced to collect performance statistics. First, we made small changes to the microSD card driver to allow us to check "busyness" of the storage device by polling the status of the `/proc/storage_usage` file. Second, We wrote a background monitoring tool (*Monitor*) to periodically read the proc file system and store summary information to a log file; the log file is written to the internal `/cache` partition to avoid influencing the SD card performance. CPU, memory, storage, and network utilization information is obtained from `/proc/stat`, `/proc/meminfo`, `/proc/storage_usage` (busyness) and `/proc/diskstats`, and `/proc/net/dev`, respectively. Third, we used `blktrace` [Blktrace 2006] to collect block-level traces for device I/O.

Table IV. Apps for Install and Launch from Android Market
Install: top apps in August 2011, total size 55.58 MB, average size 5.56MB;
Launch: 10 apps launched individually.

App Name (Install)	Size (MB)	App Name (Launch)	Size (MB)
YouTube	1.95	Angry Birds	18.65
Google Maps	6.65	SnowBoard	23.54
Facebook	2.96	Weather	2.60
Pandora	1.22	Imdb	1.38
Google Sky Map	2.16	Books	1.05
Angry Birds	18.65	Gallery	0.58
Music Download	0.70	Gmail	2.14
Angry Birds Rio	17.44	GasBuddy	1.88
Words With Friends	3.75	Twitter	1.36
Advanced Task Killer	0.10	YouTube	0.80

In order to ascertain the overheads of our instrumentation, we conducted experiments with and without the measurement environment. We found that our changes introduce an overhead of less than 2% in total runtime.

Since many popular mobile applications are interactive, we needed a technique to execute these applications in a representative and reproducible manner. For this purpose, we used the MonkeyRunner [2012] tool to automate the execution of interactive applications. Our MonkeyRunner set-up consists of a number of small programs put together to facilitate benchmarking with the necessary application; we illustrate the methodology next.

First, we start the Monitor tool to collect resource utilization information and note its PID. Second, we start the application under test using MonkeyRunner, which defines "button actions" to emulate pressing of various keys on the device's touchscreen, for example, browsing forward and backward, zooming in and out with the touchscreen pinch, and clicking on screen to change display options. Third, while the various button actions are being performed, CPU usage is tracked in order to automatically determine the end of an interactive action. A class function `UntilIdle()` that we wrote is called from the MonkeyRunner script to detect the execution status of an app; it determines idle status using a specified low CPU threshold and the minimum time the app needs to stay below the threshold to qualify as idle. Fourth, once the sequence of actions is completed, we perform necessary clean-up actions and return to the default home screen. Fifth, the Monitor tool is stopped and the resource usage data is dumped to the host computer. Similar scripts are used to reset the phone to a known state in order to repeat the experiment (to compute mean and deviation).

3.3. Application Benchmarks

We now describe the Android apps that we use to assess the impact of storage on application performance; we automate a variety of popular and frequently used mobile apps to serve as benchmarks.

—WebBench is a custom benchmark program we wrote to measure Web browsing performance in a noninteractive manner; it is based on the standard WebView Java Class provided by Android. WebBench visits a preconfigured set of Web sites one after the other and reports the total elapsed time for loading the Web pages. To accurately measure the completion time, we made use of the public method of WebView class named `onProgressChanged()`; when a Web page is fully loaded,

WebBench starts loading the next Web page on the list. We ran WebBench to visit the top 50 Web sites according to a recent ranking [Compete 2011].

—AppInstall installs a set of top 10 Android apps on Google Android Market (listed in Table IV on the left) successively, using the `adb install` command. App installation is an important and frequently performed activity on smartphones. Once installed, each application on the phone is typically updated several times during subsequent usage. In addition, a user often needs to perform the install "on the go" based on location or situational requirements (e.g., installing the IKEA app while shopping for furniture or installing the GasBuddy app when looking to refuel).

—AppLaunch launches a set of 10 Android apps using MonkeyRunner listed in Table IV on the right; the apps are chosen to cover a variety of usage scenarios: games (Angry Birds and SnowBoard) take relatively longer to load, read traffic to storage dominates. Weather and GasBuddy apps download and show real-time information from remote servers, (i.e., network traffic is high). Gmail and Twitter apps download and store data to a local database (i.e., both network and storage traffic is high). Books and Gallery apps scan the local storage and display the list of contents (i.e., read to storage dominates. IMDb has no storage or network traffic due to Web cache hits, whereas YouTube launch is network intensive.

—Facebook uses the Facebook for Android application; each run constitutes the following steps: (i) sign into the author's Facebook account, (ii) load the news feed displayed initially on the phone screen, (iii) "drag" the screen five times to load more feed data, and (iv) sign out.

—Google Maps uses the Google Maps for Android application; each run constitutes the following steps: (i) open the Maps application, (ii) enter origin and destination addresses, and get directions, (iii) zoom into the map nine times successively, (iv) switch from "map" mode to "satellite " mode, and (v) close application.

—Email uses the native email app in Android; each run constitutes the following steps: (i) open the app, (ii) input account information, (iii) wait until a list of received emails appears, and (iv) close the application.

—RLBench [RedLicense Labs 2012] is a synthetic benchmark app that generates a predefined number of various SQL queries to test SQLite performance on Android.

—Pulse News [Alphonso Labs 2012] is a popular reader app that fetches news articles from a number of Web sites and stores them locally. Our benchmark consists of the following steps: (i) open Pulse app, (ii) wait until news-fetching process completes, and (iii) close the app.

—Background is another popular usage scenario is concurrent execution of two or more applications (Android and iOS are both multithreaded); several apps run in the background to periodically "sync" data with a remote service or to provide proactive notifications. Our benchmark consists of the following set of apps in auto sync mode: Twitter, Books, Contacts, Gmail, Picasa, and Calendar, and a set of active widgets: Pulse, News, Weather, YouTube, Calendar, Facebook, Market, and Twitter.

For many of the benchmarks (e.g., Facebook, Email, Pulse, Background), the actual contents and amount of data can vary across runs; we measure the total amount of data transferred and normalize the results per megabyte. We also repeat the experiment several times to measure variations; for multiple iterations, the local application cache is deleted following each run.

4. PERFORMANCE EVALUATION

In this section, we present detailed measurement results for application runtime performance, application launch times, concurrent app execution, and CPU consumption.

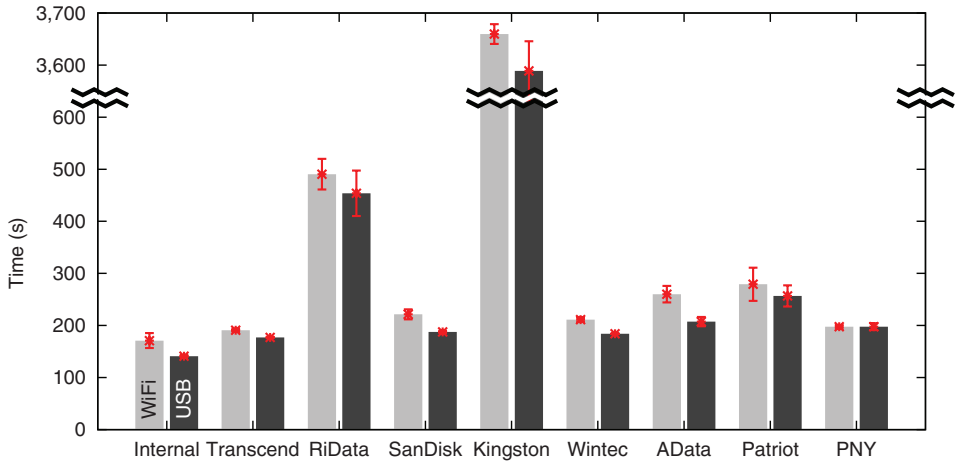

Fig. 4. Runtimes for WebBench on Google Nexus One. Runtime for WebBench for SD cards and internal flash. Each bar represents average over three trials with standard deviation; lighter bar is over WiFi, darker one for USB RT.

4.1. Application Runtime Performance

The first set of experiments compare the performance of WebBench on internal flash and the eight SD cards described earlier. Figure 4 shows the runtime of WebBench for WiFi and USB reverse tethering.

Surprisingly, even with WiFi, we notice a 187% performance difference between the internal flash and RiData; for Kingston, the difference was a whooping 2040%. To ensure that the Kingston results were not due to a defective device, we repeated the experiments with two more new Kingston cards from two different speed classes; we found the results to be similarly poor. In the remainder of the text, so as to not rely on Kingston alone when making a claim about application performance, we mention the difference both with the second-worst and worst performing card for any given experiment.

As expected, the faster the network (USB RT), the higher the impact of storage: 222% difference between internal and RiData, 2450% for Kingston. We find a similar trend for several popular apps. Figure 5 shows the results over WiFi for AppInstall, Email, Google Maps, Facebook, RLBench, and Pulse. Since the phenomenon of storage and application performance correlation is clearly identifiable with existing WiFi networks, we hereafter omit results for the USB network. The difference between the best- and worst-case performance varies from 195% (225%) for AppInstall, 80% (1670%) for Email, 60% (660%) for Maps, 80% (575%) for Facebook, 130% (2210%) for RLBench, and 97% (168%) for Pulse; Kingston numbers are shown in parentheses.

To better understand why storage affects application performance, in Figure 6 we present breakdown of the I/O activity during the WebBench workload. First, there are few reads, the majority of which are issued in the beginning of the experiment. There are significantly more writes, with about 1.3 times more data being written sequentially than randomly. Since the difference between sequential and random performance is at least a factor of 3 for all SD cards (see Table III), the time to complete the random writes dominates. Although not shown in the figure, the `/data` partition receives most of the I/O, with only a few reads going to the `/system` partition. Table V presents a breakdown of the I/O activity for the other popular applications; the ratio of sequential and random writes summarized in the table illustrates the effects of random I/O on performance.

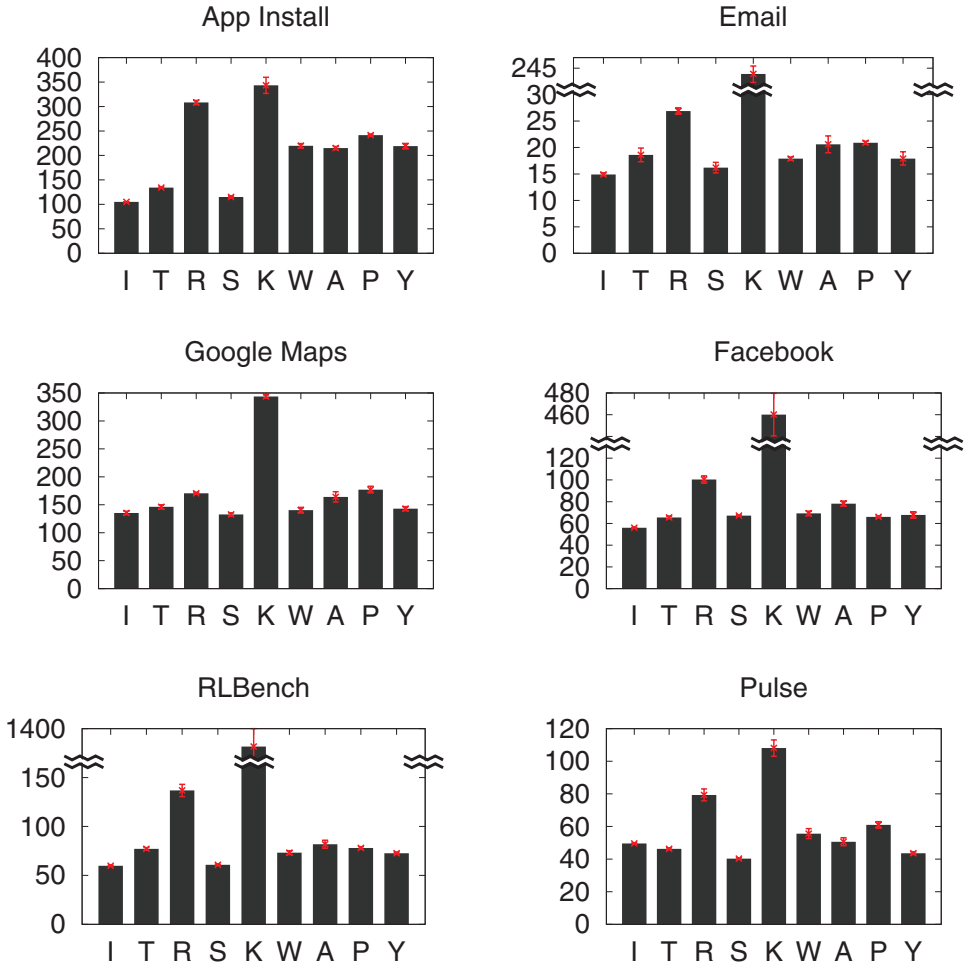

Fig. 5. Runtimes for popular applications. Similar to Figure 4 but for several other apps on WiFi only; I: Internal, T: Transcend, R: RiData, S: Sandisk, K: Kingston, W: Wintec, A: AData, P: Patriot, Y: PNY. Some graphs are plotted with a discontinuous y-axis to preserve clarity of the figure in presence of outliers like Kingston.

The disparity between sequential and random write performance is inherent with flash-based storage; our evaluation results suggest this to be one of the primary reasons behind the slower performance. However, this still does not explain the presence of the random writes and overwrites even for seemingly sequential application needs. In order to understand this, we take a closer look at the applications and their usage of Android storage.

Figure 7 shows the storage schema used by the browser application consisting of several components. The cache component contains webviewCache, the unstructured Web cache storing image and media files. The databases component contains two SQLite database files; `webview.db` is a database for application settings and preferences and `webviewCache.db` stores an index to manage the Web cache. The database files are much smaller in size compared to the cache; in our set-up, the cache consisted of 315 files totaling 6MB, whereas the database files were 34KB and 137KB for `webview.db`, and `webviewCache.db`, respectively.

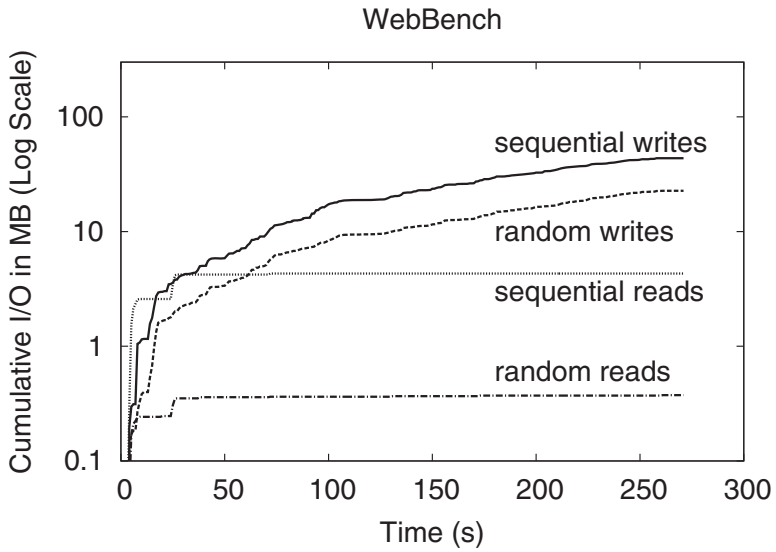

Fig. 6. WebBench sequential and random I/O activity on Android. The graph presents the breakdown of I/O operations issued by WebBench; y-axis is the cumulative I/O issued for that operation type on a log scale, and x-axis is progress time for the experiment.

Table V. I/O Activity Breakdown
Aggregate seq. and random, writes and reads during benchmark; note moderate to high rand:seq write ratios for WebBench, Email, Maps, Facebook, and low for AppInstall. Zero value means no activity during run.

	Write (MB)		Read (MB)	
Activity	Sq	Rn	Sq	Rn
WebBench	41.3	32.2	6.8	0.5
AppInstall	123.1	5.6	0.7	0.1
Email	1.0	2.2	1.1	0.1
Maps	0.2	0.3	0	0
Facebook	2.0	3.1	0	0
RLBench	25.6	16.8	0	0
Pulse	2.6	1.0	0	0

Figure 8 shows the write pattern to the Web cache directory and the SQLite database files. Web cache writes are mostly sequential with reuse of the same address space over time; SQLite exhibits a high degree of random writes and updates to the same block addresses. Since the database writes are synchronous by default, each write causes a (often unnecessary) delay.

4.2. Application Launch

Application launch is an important performance metric [Joo et al. 2011], especially for mobile users. Figure 9 shows the time taken to launch a number of Android applications on the various flash storage devices. Table VI lists those apps along with a summary of disk I/O reads and writes, and data transferred over the network during the launch. Most apps take a few seconds to launch, with games taking upwards of 10 seconds. Larger apps (e.g., games) tend to take a noticeable amount of time to launch, contrary to the target of "significantly less than 1 second to launch a new app" [Hackborn 2010].

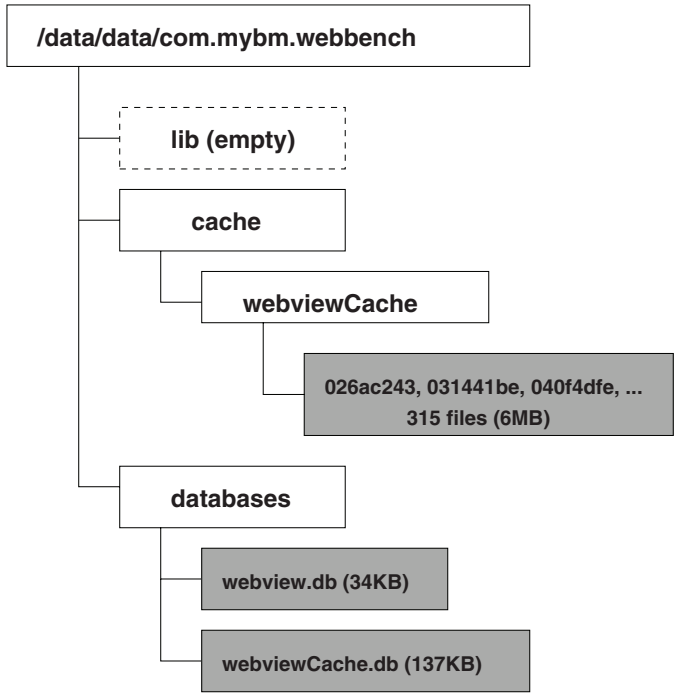

Fig. 7. Storage schema for WebBench. Storage data structures are cache of objects on FS and index of cache in SQLite.

As shown in Figure 9, barring a few exceptions, the launch time varies between about 10% (for the Snowboard game) to 40% (for the Weather app); Twitter (120%) and Gmail (250%) showed the most variation.

In order to ascertain the upper bound of launch time improvement through storage, we placed all application data on a RAMdisk; the test is conducted with the PNY card storing the `/system`, `/sdcard`, `/cache` partitions, and the `/data` partition mounted with `tmpfs`. To remove the effects of reading from `/system` and `/sdcard`, we warm the buffer cache; we verify the same by tracking all I/O to the flash storage. Launch times do not significantly change even when all data is being read from memory. Storage is likely not a significant contributor to app launch performance; research to speed up launch times will perhaps benefit by focusing on other sources of delay such as application think time.

4.3. Concurrent Applications

Figure 10 shows I/O activity for a 7,200-second run of the Background workload. During the period, the phone received about 1.6MB of data over the network. Interestingly, the amount of data written to storage in the same period is 30MB (a factor of roughly 20); the majority of writes are for updating application-specific data and indices to the SQLite databases. Although the storage throughput requirement is quite low, the additional random writes can cause latency spikes for foreground applications (not shown). With the Android development team's desire to minimize application switch time and provide the appearance of "all applications running all of the time" [Hackborn 2010] (see section: "When does an application 'stop'?") for mobile devices, handling concurrent applications and their I/O demands can be an increasingly important challenge in the future.

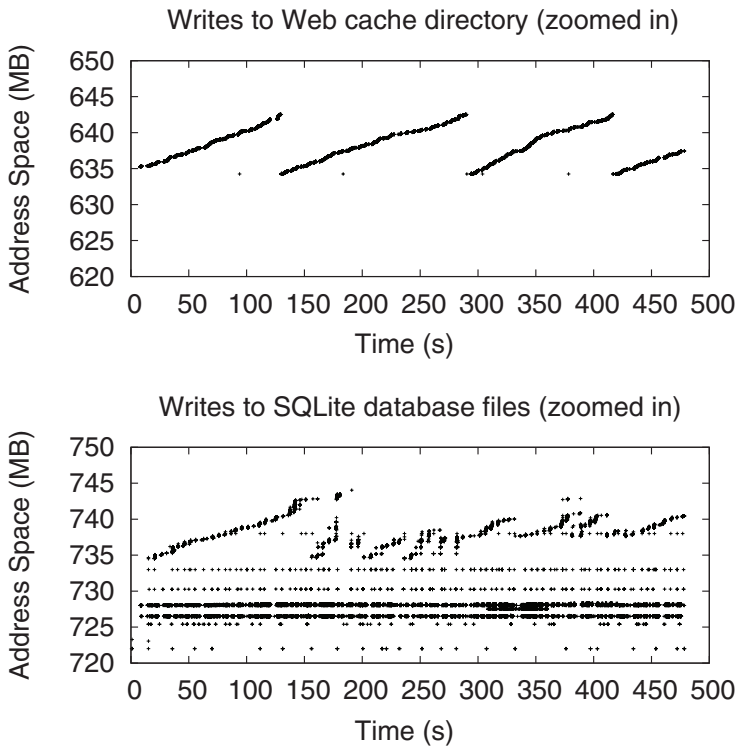

Fig. 8. SQLite I/O pattern. The left graph shows write I/O to the Web cache directory contents on /data, on right are writes to SQLite database files; reads are comparatively less and omitted from presentation.

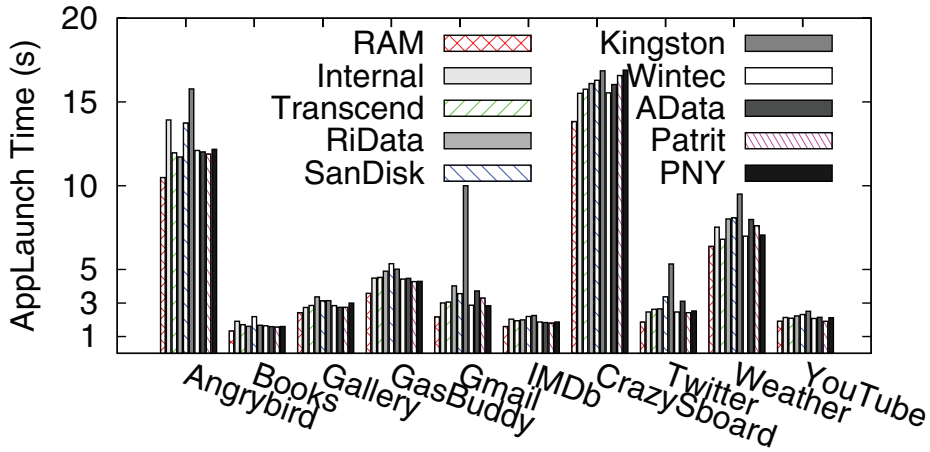

Fig. 9. Application launch. Launch times (s) for several popular apps on eight SD cards, internal flash, and a memory-backed RAMdisk.

4.4. CPU Consumption

Figure 11 shows the breakdown of CPU utilization for WebBench; the stacked bar chart shows the CPU tick counts during active, idle, and ioWait periods (a "tick" corresponds to 10ms on our phone). Figure 12 shows the CPU utilization and I/O busyness for

Table VI. App Launch Summary
Total data (MB) read and written to storage and transferred over the network for the set of apps launched.

App	R	W	Rx	Tx
AngryBird	20.69	0.04	4.09	4.44
SnowBoard	20.92	0.02	1.87	0.53
Weather	8.72	0.07	16.11	2.56
Imdb	2.71	0.00	0.08	0.00
Books	2.98	0.00	0.00	0.00
Gallery	1.88	0.00	0.00	0.00
Gmail	3.20	0.05	3.00	0.93
GasBuddy	7.47	0.00	2.28	0.80
Twitter	4.62	0.06	5.63	1.61
YouTube	2.06	0.00	65.47	4.83

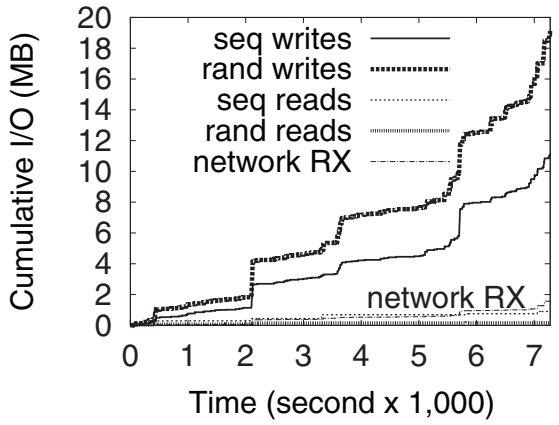

Fig. 10. Background I/O pattern. Breakdown of I/O issued by Background apps in 2 hours; y-axis is the cumulative I/O issued for the operation type, x-axis is time.

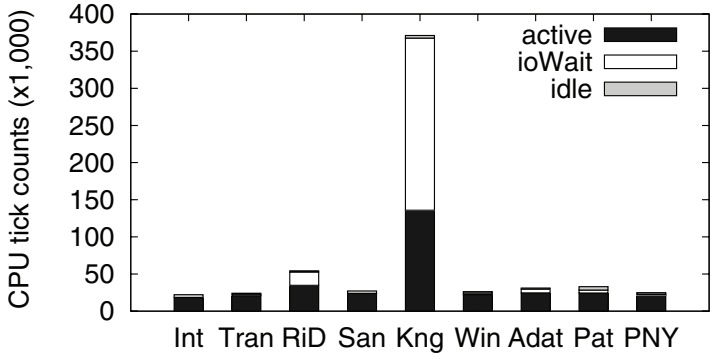

Fig. 11. Aggregate CPU for WebBench. Stacked bar shows active, idle, and ioWait times on Nexus One; ioWait correlates with runtimes (Figure 4). Even active times vary across devices, showing that some devices burn more CPU for the same work!

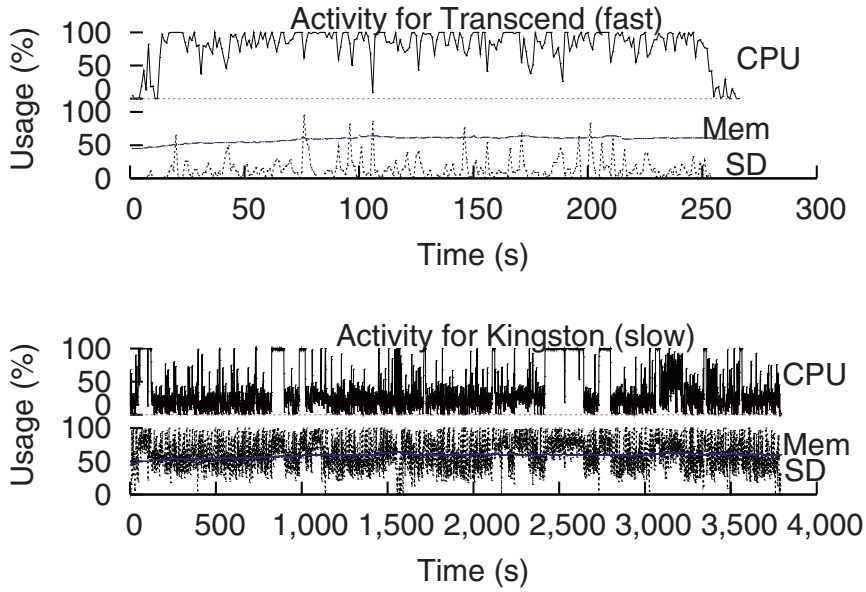

Fig. 12. Storage and CPU activity for WebBench on fast and slow SD cards. The to graph shows instantaneous CPU utilization, memory consumption, and storage busyness during the course of a WebBench run on the fast Transcend card. The lower graph repeats the same experiment for the slow Kingston, taking considerably longer to finish. The table summarizes the aggregate CPU ticks (in thousands) used for WebBench; compare the active counts for fast and slow.

the same experiment for two SD cards: a fast Transcend and a slow Kingston. Since the nonidle, non-ioWait CPU consumption includes not only the contribution of the benchmark but also all background activities, we also measured CPU consumption for background activities alone (to subtract from the total). Note that this is unlike the set of background activities discussed in Section 3.3, as we turned off automatic syncing and active widgets; we find that the share of CPU consumption due to background tasks is less than 1% of the total.

The graphs reveal the interesting phenomenon that aggregate CPU consumed for the same benchmark increases with a slower storage device (by just looking at the "active" component). This points to the fact that storage has an indirect impact on energy consumption by burning more CPU. Ideally, one would expect a fixed amount of CPU to be consumed for the same amount of work. Since the results show CPU consumption to be disproportional to the amount of work, we hypothesize it being due to deficiencies in either the network subsystem, the storage subsystem, or both. We need to investigate this matter further to identify the root causes.

Slower storage also increases energy consumption in other indirect ways; for example, keeping the LCD screen turned on longer while performing interactive tasks, keeping the WiFi radio busy longer, and preventing the phone from going to a low-power mode sooner.

5. PILOT SOLUTIONS

We present potential improvements in application performance through storage system modifications. We start with a what-if analysis to provide the envelope of performance gains and then present a set of pilot solutions.

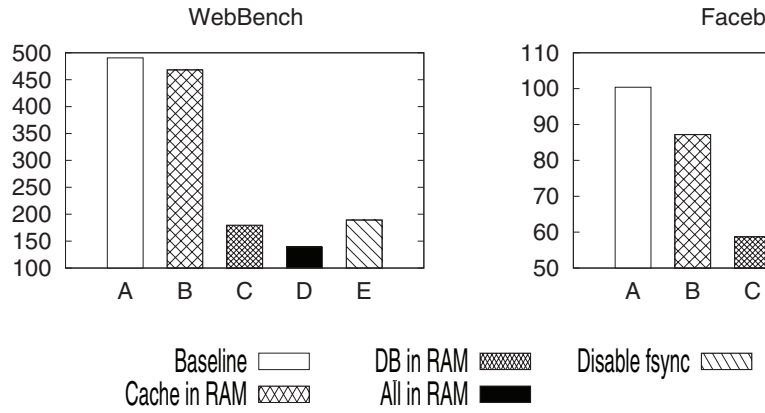

Fig. 13. What-if performance analysis. Experiments were conducted for WebBench (left) and Facebook (right); data stored in memory using a RAMdisk and RiData card as the flash backing store where needed (e.g., for baseline). The *y*-axis is the time in seconds; Solutions A: Baseline, B: Cache in RAM, C: DB in RAM, D: All in RAM, E: Disable Sync.

5.1. What-If Analysis

The detailed analysis of storage performance gave insights into the performance problems faced by applications, but before proposing actual solutions, we wanted to understand the scope for potential improvements. We performed a set of what-if analyses to obtain the upper bounds on performance gains that could be achieved, for example, by storing all data in memory. For comparison sake, we performed experiments, with both memory as the backing store (using RAMdisk) and SD cards as the backing store; in the different analysis experiments we placed different kinds of data on the RAMdisk, for example, the cache, or the database files. Figure 13 compares the relative benefits of the various approaches, as measured for the WebBench and Facebook workloads for the RiData card and a RAMdisk; the trends for the other SD cards were similar, although the actual gains were of course different with every card.

Placing the entire "cache" folder on RAM (bars B) does improve performance, but not by much (i.e., 5% for WebBench and 15% for Facebook). Placing the SQLite database on RAM (bars C), however, improves performance by factors of 3 and 2 for WebBench and Facebook, respectively. Placing both the cache and the database on RAM (bars D) does not provide significant additional benefit. Transforming the cache and database writes to be asynchronous (bars E) recoups most of the performance and performs comparably to the SQLite on RAM solution.

The performance evaluation in the previous section and the what-if analysis lead to the following conclusions: First, the key bottleneck is the "wimpy" storage prevalent today on mobile devices. Even while the internal flash and the SD cards are increasingly being used for desktop-like workloads, their performance is significantly worse than storage media on laptops and desktops. Second, the Android OS exacerbates the poor storage performance through its choice of interfaces; the synchronous SQLite interface primarily geared for ease of application development is being used by applications that are perhaps better off with more lightweight consistency solutions. Third, the SQLite write traffic itself is quite random with plenty of synchronous overwrites to the flash storage causing further slowdown. Finally, apps use the Android interfaces oblivious to performance. A particularly striking example is the heavy-handed management of application caches through SQLite; the Web browser writes a cache map to SQLite significantly slowing down the cache writes.

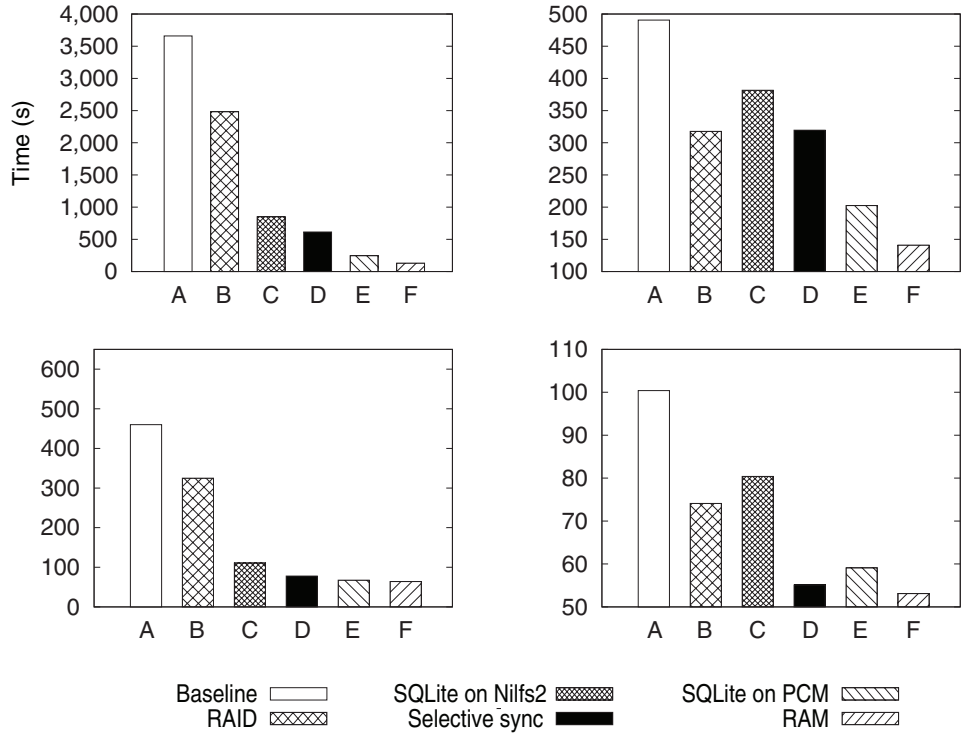

Fig. 14. Pilot solutions. Runtime results for WebBench (top row) and Facebook (bottom row) for the Kingston and RiData cards. The *y*-axis is time in seconds. Solutions A: Baseline, B: RAID over SD, C: SQLite on Nilfs2, D: Selective Sync, E: SQLite on PCM, F: All in RAM.

We implement and evaluate a set of pilot solutions to show the potential for improving user experience through improvements in the Android storage system. While not rigorous enough to serve as deployable solutions, these can evolve into robust and detailed solutions in the future. We classify the solution space into four categories.

—Better storage media for mobile devices to provide baseline improvements
—Firmware and device drivers to effectively utilize existing and upcoming storage devices
—Enhancements to mobile OS to avoid the storage bottlenecks and provide new functionality
—Application-level changes to judiciously use the supplied storage interfaces

Figure 14 shows the improvements through the pilot solutions for WebBench and Facebook using Kingston and RiData; as with the what-if analysis, trends for other SD cards were similar but actual gains varied. Bars A in Figure 14 represent the baseline performance, whereas bars F are meant to represent an upper bound on performance with all data stored in RAM.

5.2. Storage Devices Not Wimpy Anymore

An obvious solution is to improve the performance of the storage device, that is, using better flash storage or a faster nonvolatile memory such as PCM. Indeed, flash fabrication technology itself is improving at a fair pace; scaling trends project flash to double in capacity every 2 years until the year 2016 [ITRS 2009]. However, when it comes

to performance, cost pressures in the consumer market are driving manufacturers to move away from the more reliable, higher-performing SLC flash to the less reliable, lower-performing MLC or TLC flash; this makes it harder to rely solely on improvements due to flash scaling. Our findings reveal that the performance of a relatively small fraction of I/O traffic is responsible for a large fraction of overall application performance. A more efficient solution is thus to use the faster storage media as a persistent write buffer for the performance-sensitive I/O traffic: A small amount of PCM to buffer writes issued by the SQLite database can improve the performance.

We built a simple PCM emulator for Android to evaluate our solution. The emulator is implemented as a pseudoblock device based on the timing specifications from recent work [Chen et al. 2011], using memory as the backing store. The PCM buffer can be used as staging area for all writes or as the final location for the SQLite databases; our emulator can be configured with a small number of device-specific parameters. Figure 14 (bars E) show the performance improvements by using a small amount (16-MB) of PCM; in this experiment, PCM is used as the final location for only the database files.

An alternative approach, as envisioned by Pocket Cloudlets [Koukoumidis et al. 2011], is to rely on substantial augmentation of existing flash storage capabilities on mobile devices and/or full replacement of flash with PCM or STT-MRAM [Huai 2008]. In reality, storage-class memory may be placed in different forms on the mobile system, for example, on the CPU-memory bus, or as backing store for the virtual memory. Our intent here was twofold: (i) understand the approximate benefits of using such a persistent buffer and (ii) demonstrate that even with a relatively small amount of PCM, significant gains can be made by judiciously storing performance-critical data; a deployed solution can certainly incorporate PCM in the storage hierarchy in better ways.

5.3. RAID over SD

Another solution is to leverage the I/O parallelism already existent on most phones: an internal flash drive and an external SD card. We built a simple software RAID driver for Android with I/O striped to the two devices (RAID-0) in 4KB blocks. Note that a deployable solution will require more effort: It would need to handle storage devices of potentially differing speeds and handle accidental removal of the external SD card.

While for some SD cards we obtained the expected improvements as in Figure 14 (bars B) (i.e., greater than 1X and less than 2X), for others we obtained a speed-up greater than 2X (not shown); we suspected that this could be due to the idiosyncrasies of the FTL on the card. As many consumer flash devices employ the log-block wear-leveling scheme [Kim et al. 2002], their performance is sensitive to the write footprint; a reduction in the amount of random writes reduces the overhead of the garbage collection, improving the performance.

To verify our hypothesis, we performed another experiment. Figure 15 shows the throughput obtained for an increasing address range with random writes; the I/O size is 4KB and number of requests is 2,048, totaling 8MB of writes. In order to minimize the effects of FTL state being carried forward from the previous experiment, we sequentially write 1GB of data before every run.

For Kingston, Wintec, A-Data, Patriot, and PNY, as the address range increases, the throughput drops significantly and then stabilizes at the low level. For RiData, throughput drops but not as sharply, whereas for Transcend the throughput remains consistently high (we do not have an explanation for the slight increase, multiple measurements gave similar results). Sandisk exhibits more than one regime change, dropping first around the 32MB mark and then around the 1,024MB mark.

To explain our surprising performance improvements, in a log-block FTL, a small number of physical blocks are available for use as log blocks to stage an updated block; a one-to-one correspondence exists between logical and physical blocks. Since the

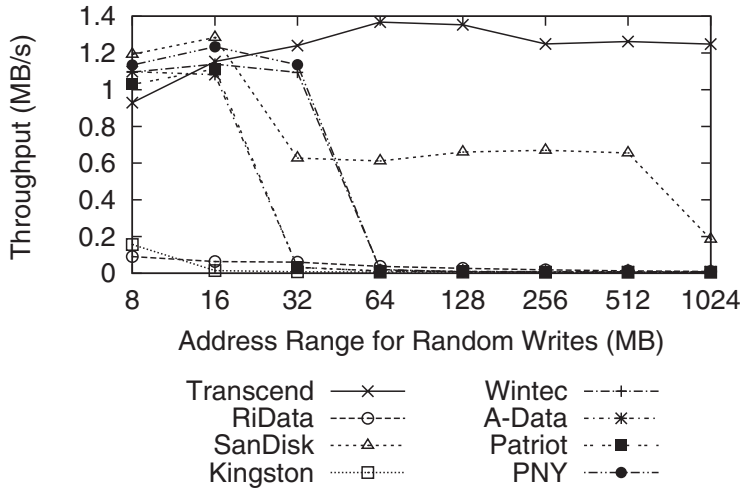

Fig. 15. Explanation of RAID speed-up. Variation in throughput for SD cards with increasing write address range.

amount of data written to one disk in a 2-disk RAID-0 array is roughly half of the total, the disk write footprint reduces and block address range shrinks; the RAID scheme simply pushes the operating regime of an SD card toward the left, and depending on the actual footprint, provides super-linear speed-up! While we came across this performance variation in course of our RAID experiments, the implications are more generic; one can design other solutions centered around the compaction of the write address range.

5.4. Using a Log-structured File System

Log-structured file systems provide good performance for random writes [Rosenblum and Ousterhout 1992]; another solution to alleviate the effects of the random writes is thus to place the database files on a log-structured file system. We used the Nilfs2 [Konishi et al. 2006] file system on Android because it works with block devices, and we created a separate partition on the phone's flash storage to store the entire SQLite database. Figure 14 (bars C) show the benefits of log-structuring; SQLite on Nilfs2 improves the performance of WebBench and Facebook by more than a factor of 4 for Kingston, and more than 20% for RiData.

5.5. Application Modifications

Finally, several solutions are possible if one is able to modify either the SQLite interface or the applications themselves. We demonstrate the benefits of such techniques with a simple modification to SQLite: providing the capability to perform selective `sync` operations based on application-specific requirements. In our current implementation, we simply turn off sync for the database files that are deemed asynchronous as per our analysis (e.g., the WebView database file serving as the index for the Web cache). Figure 14 (bars D) compare the benefits of the selective sync operation with other previously proposed solutions, providing noteworthy benefits especially for Facebook.

Another potential technique to improve performance at the application level is through the use of larger transactions, amortizing the overhead of the SQLite sync interface. A careful restructuring of the application programming interface can perhaps lead to significant gains for future apps but is beyond the scope for this article, the interface discussion is a classic chicken-and-egg problem in the context of storage

systems [Sivathanu et al. 2003; Schlosser and Ganger 2004]. Recently, a new backend for SQLite has been proposed that uses write-ahead logging [SQLite 2011]; such techniques have the potential to ameliorate the random write bottleneck without requiring changes to the API.

5.6. Summary of Solutions

Through our investigation of the solution space we notice several avenues for further performance improvements in the storage subsystem on mobile devices, and consequently the end-user experience. Our analysis reveals that a small amount of domain or application knowledge can improve performance in a more efficient way. Through our pilot solutions, we demonstrate the potential benefits of explicit and implicit storage improvements.

Programmers tend to heavily use the general-purpose "all-synchronous" SQLite interface for its ease of use but end up suffering from performance shortcomings. We posit that a data-oriented I/O interface would be one that enables the programmer to specify the I/O requirements in terms of its reliability, consistency, and the property of the data (i.e., temporary, permanent, or cache data), without worrying about how its stored underneath. For example, a key-value store specifically for cache data), does not need to provide ultra-reliability; a Web browser can use the cache key-value store as its Web cache in a more performance-efficient manner than SQLite.

6. RELATED WORK

We found little published literature on storage performance for mobile devices. One of the earliest works on storage for mobile computers [Douglis et al. 1994] compares the performance of hard disks and flash storage on an HP OmniBook; remarkably, many of their general observations are still valid. Datalight [2011], provider of data management technologies for mobile and embedded devices to OEMs, make an observation similar to ours with reference to their proprietary Reliance Nitro file system. According to their Web site, lack of device performance and responsiveness is one of the important shortcomings of the [Windows] Mobile platforms; OEMs using an optimized software stack can improve performance. Our results also reaffirm some of the recent findings for desktop applications on the Mac OS X [Harter et al. 2011]: lack of pure sequential access for seemingly sequential application requests, heavy-handed use of synchronization primitives, and the influence of underlying libraries on application I/O.

A recent study of Web browsers on smartphones [Wang et al. 2011] examined the reasons behind slow Web browsing performance and found that optimizations centering around compute-intensive operations provide only marginal improvements; instead, "resource loading" (e.g., files of various types being fetched from the Web server) contributes most to browser delay. While this work focuses more specifically on the browser and the network, it reaffirms the observation that improvements in the OS and hardware are needed to improve application performance.

Other related work has focused on the implications of network performance on smartphone applications [Huang et al. 2010] and on the diversity of smartphone usage [Falaki et al. 2010]. Finally, there is extensive work in developing smarter, richer, and more powerful applications for mobile devices, far too much to cite here. We believe the needs of these applications are in turn going to drive the performance requirements expected of hardware devices, including storage, as well as the OS software.

7. CONCLUSIONS

Contrary to conventional wisdom, we find evidence that storage is a significant contributor to application performance on mobile devices; our experiments provide insight into the Android storage stack and reveal its correlation with application performance.

Surprisingly, we find that even for an interactive application such as Web browsing, storage can affect the performance in nontrivial ways; for I/O intensive applications, the effects can get much more pronounced. With the advent of faster networks and I/O interconnects on the one hand, and a more diverse, powerful set of mobile apps on the other, the performance required from storage is going to increase in the future. We believe the storage system on mobile devices needs a fresh look and we have taken the first steps in this direction.

ACKNOWLEDGMENTS

We thank Akshat Aranya for his assistance in setting up the Android test environment and experimental data analysis. We thank Kishore Ramachandran for providing several useful discussions and detailed comments on the article. We also thank our FAST reviewers and shepherd, Raju Rangaswami, for their valuable feedback that improved the presentation of this article.

REFERENCES

ALPHONSO LABS. Pulse News Reader. https://market.android.com/details?id=com.alphonso.pulse&hl=en.

ANDROID DEBUG BRIDGE (ADB). 2011. Homepage. http://developer.android.com/guide/developing/tools/adb.html.

ANDROID DEVELOPER. 2011. Android Developers Website. http://developer.android.com/index.html.

ANDROID OPEN SOURCE PROJECT. 2011. HomePage http://source.android.com/index.html.

BICKFORD, J., LAGAR-CAVILLA, H. A., VARSHAVSKY, A., GANAPATHY, V., AND IFTODE, L. 2011. Security versus energy tradeoffs in host-based mobile malware detection. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. ACM, New Yorm.

BLKTRACE. 2006. Block I/O Layer Tracing: blktrace. http://linux.die.net/man/8/blktrace.

BUSYBOX. 2008. Busybox Unix utilities. http://www.busybox.net/about.html.

CARBOU, M. 2010. USB reverse tethering setup for Android 2.2. http://blog.mycila.com/2010/06/reverse-usb-tethering-with-android-22.html.

CARROLL, A. AND HEISER, G. 2010. An analysis of power consumption in a smartphone. In *Proceedings of the USENIX conference on USENIX Annual Technical Conference (USENIX ATC'10)*. Berkeley, CA, 21.

CASTELLUCCI, S. J. AND MACKENZIE, I. S. 2011. Gathering text entry metrics on android devices. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'11)*. ACM, New York, 1507–1512.

CHEN, S., GIBBONS, P. B., AND NATH, S. 2011. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Sysytems Research (CIDR'11)*, 21–31.

CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the 6th Conference on Computer Systems (EuroSys '11)*. ACM, New York, 301–314.

COMPETE. 2011. Compete ranking of top 50 Web sites for february 2011 reveals familiar dip. http://tinyurl.com/3ubxzbl.

CRYSTALMARK. 2012. CrystalDiskMark Benchmark V3.0.1. http://crystalmark.info/software/CrystalDiskMark/index-e.html.

CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. 2010. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, 49–62.

CYANOGENMOD. 2012. Homepage. http://wiki.cyanogenmod.com/index.php?title=What_is_CyanogenMod.

DATALIGHT. 2011. Datalight: software for risk-free mobile data. http://www.datalight.com/solutions/linux-flash-file-system/performance-hardware-managed-media.

DATTA, K. 2010. Clockworkmod rom manager and recovery image. http://www.koushikdutta.com/2010/02/clockwork-recovery-image.html.

DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. 2011. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*.

DOUGLIS, F., CÁCERES, R., KAASHOEK, M. F., LI, K., MARSH, B., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI '94)*. 25–37.

ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. 2010. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX, Berkeley, CA, 1–6.

FALAKI, H., MAHAJAN, R., KANDULA, S., LYMBEROPOULOS, D., GOVINDAN, R., AND ESTRIN, D. 2010. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, 179–194.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, New York, 48–63.

FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. 2003. Data staging on untrusted surrogates. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*.

GARTNER. 2011. Gartner highlights key predictions for it organizations and users in 2010 and beyond. http://www.gartner.com/it/page.jsp?id=1278413.

GEAMBASU, R., JOHN, J. P., GRIBBLE, S. D., KOHNO, T., AND LEVY, H. M. 2011. Keypad: an auditing file system for theft-prone devices. In *Proceedings of the 6th Conference on Computer Systems (EuroSys '11)*. ACM, New York, 1–16.

GUNDOTRA, V. AND BARRA, H. 2011. Android: momentum, mobile and more at Google I/O. http://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html.

HACKBORN, D. 2010. Multitasking the Android way. http://android-developers.blogspot.com/2010/04/multitasking-android-way.html.

HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. 2011. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, 38–49.

HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2011. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

HTC. 2011a. HTC Desire. http://www.htc.com/www/product/desire/specification.html.

HTC. 2011b. HTC EVO 4G. http://www.htc.com/us/products/evo-sprint#tech-specs.

HUAI, Y. 2008. Spin-transfer torque MRAM (STT-MRAM): challenges and prospects. *AAPPS Bull. 18,* 6 (Dec.), 33–40.

HUANG, J., XU, Q., TIWANA, B., MAO, Z. M., ZHANG, M., AND BAHL, P. 2010. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*. ACM, New York, 165–178.

IEEE WG802.11. IEEE STANDARD 802.11n-2009 Wireless LAN Working Group. http://standards.ieee.org/findstds/standard/802.11n-2009.html.

ITRS. 2009. ITRS 2009 Edition. Tech. rep., International Technology Roadmap for Semiconductors.

JOO, Y., RYU, J., PARK, S., AND SHIN, K. G. 2011. Fast: quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*.

KIM, H., AGRAWAL, N., AND UNGUREANU, C. 2012. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

KIM, J., KIM, J. M., NOH, S., MIN, S. L., AND CHO, Y. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. Consum. Electron. 48,* 2, 366–375.

KISTLER, J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10,* 1, 3–25.

KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. 2006. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev. 40,* 3, 102–107.

KOUKOUMIDIS, E., LYMBEROPOULOS, D., STRAUSS, K., LIU, J., AND BURGER, D. 2011. Pocket cloudlets. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, 171–184.

LG. 2011. LG G2X datasheet. http://www.lg.com/us/products/documents/LG-G2x-Datasheet.pdf.

MANNING, C. 2004. YAFFS: Yet another flash file system. http://www.aleph1.co.uk/yaffs.

MERONI, P., PAGANI, E., ROSSI, G. P., AND VALERIO, L. 2010. An opportunistic platform for android-based mobile devices. In *Proceedings of the 2nd International Workshop on Mobile Opportunistic Networking (MobiOpp'10)*. ACM, New York, 191–193.

MONKEYRUNNER. 2012. MonkeyRunner for Android developers. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

MOTOROLA. 2011. Motorola Webtop: release your smartphone's true potential. http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/WEBTOP/Meet-WEBTOP.

NEXUS ONE. 2011. Google Nexus One. http://en.wikipedia.org/wiki/Nexus_One.

NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. 1997. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, 276–287.

PENTIN. 2010. Gartner's mobile predictions. http://ifonlyblog.wordpress.com/2010/01/14/gartners-mobile-predictions/.

REDLICENSE LABS. 2012. RL benchmark: SQLite. https://market.android.com/details?id=com.redlicense.benchmark.sqlite.

ROSENBLUM, M. AND OUSTERHOUT, J. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10,* 1 (Feb.), 26–52.

ROY, A., RUMBLE, S. M., STUTSMAN, R., LEVIS, P., MAZIÈRES, D., AND ZELDOVICH, N. 2011. Energy management in mobile devices with the cinder operating system. In *Proceedings of the 6th Conference on Computer Systems (EuroSys '11)*. ACM, New York, 139–152.

SAMSUNG CORP. 2011. Samsung ships industrys first multi-chip package with a PRAM chip for handsets. http://tinyurl.com/4y9bsds.

SATYANARAYANAN, M. 2010. Mobile computing: the next decade. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing and Services: Social Networks and Beyond (MCS '10)*. ACM, New York, 5:1–5:6.

SATYANARAYANAN, M., BAHL, P., CACERES, R., AND DAVIES, N. 2009. The case for VM based cloudlets in mobile computing. *IEEE Pervasive Comput. 8*, 14–23.

SCHLOSSER, S. W. AND GANGER, G. R. 2004. MEMS-based storage devices and standard disk interfaces: a square peg in a round hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*. 87–100.

SD ASSOCIATION. 2012. SD speed class/UHS speed class. https://www.sdcard.org/consumers/speed_class/.

SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*. 73–88.

SOURCEFORGE. 2012. Iperf. iperf network performance tool. http://sourceforge.net/projects/iperf.

SQLITE. 2012. Using databases. http://developer.android.com/guide/topics/data/data-storage.html#db.

SQLITE. 2011. SQLite backend with write-ahead logging. http://www.sqlite.org/draft/releaselog/3_7_0.html.

STARBURST. 2012. Starburst data2sd. http://starburst.droidzone.in/.

TOLIA, N., HARKES, J., KOZUCH, M., AND SATYANARAYANAN, M. 2004. Integrating portable and distributed storage. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*. 227–238.

TSO, T. 2010. Android will be using ext4 starting with Gingerbread. http://www.linuxfoundation.org/news-media/blogs/browse/2010/12/android-will-be-using-ext4-starting-gingerbread.

UNREVOKED. 2012. Unrevoked 3: Set your phone free. http://unrevoked.com/recovery/.

VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. 2010. quFiles the right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA.

WANG, Z., LIN, F. X., AND ZHONG, L. 2011. Why are Web browsers slow on smartphones? In *Proceedings of the ACM International Workshop on Mobile Computing Systems and Applications (HotMobile '11)*.

WEBKIT. 2012. Android WebKit package. http://developer.android.com/reference/android/webkit/package-summary.html.

Received April 2012; accepted April 2012