Why Do We Always Blame The Storage Stack?

Hao Luo*, Hong Jiang, Myra B. Cohen
University of Nebraska-Lincoln

Abstract

Much research effort has been devoted to improving the performance of the I/O stack in mobile devices, but limited time has been spent evaluating mobile application performance from the user's perspective. In this paper, we try to understand how applications running on the newest devices behave with respect to this metric. We develop a methodology for quantifying user-perceived latency and use it to evaluate four common application benchmarks with I/O stack optimization on two of the latest smartphones. Contrary to our expectation, we find that (i) these applications respond reasonably fast and (ii) the user-perceived latency does not drastically (at most 11.8%) benefit from I/O stack optimizations.

1 Introduction

Over the last decade, mobile devices, including smartphones and tablets, have become ubiquitous. Prevailing wisdom [4, 5] suggests that the performance of the storage subsystem in such devices plays an important role in application performance. One recent study [5] finds that storage performance does indeed affect the performance of several common applications. Another study [4] identifies the Journaling of Journal (JOJ) phenomenon (when the file system journals the database journal activities), as the root cause of inefficiency in the storage I/O stack in smartphones.

This has motivated work on a range of SQLite database optimizations, including tuning the database journaling mode [4], eliminating SQLite journaling I/Os through multi-version B-tree [6], and minimizing the synchronization overhead with WALDIO [7]. Other work focuses on reducing the overhead of the Ext4 file system journaling by ensuring a single I/O operation on the synchronous commit path [12] or trading off data staleness for better application responsiveness [11]. Another recent study points out that boosting the priority of part of the asynchronous I/Os will improve the responsiveness [2]. Moreover, two studies leverage the battery in mobile devices to improve the efficiency of the storage subsystem [3, 8].

Much of the aforementioned research [2, 4, 6–8, 12] is devoted to improving the performance of the SQLite database and the Ext4 file system in smartphones. In comparison, limited effort has been spent on evaluating the application performance from the *user's perspective*. Nitin et al. [5] evaluate this dimension of application performance on a suite of Android application benchmarks. However, their experiments were conducted on a relatively old Android smartphone. Given the advances in recent hardware, these results may not longer be accurate. In another study, Jinglei et al. [11] evaluate app responselatency using the time required to finish a predefined user interaction path on the device. However, the lack of synchronizations between the simulated interactions and the application under test means that the execution time may not accurately measure the latency perceived by the user. For instance, the test harness may return before the web page is 100% rendered, reporting a shorter latency than the user would actually see.

In this paper, we revisit the impact of storage on application performance, and look at this from a fresh perspective. We make two main contributions. First, we develop a methodology for evaluating application performance that is based on user perception. We define the notion of a stable GUI state and then use the transition time between two consecutive stable GUI states as the metric for user-perceived latency. Second, we use our methodology to evaluate the impact of storage I/O stack optimizations on the application performance on two of the latest smartphones (Samsung Galaxy S4 and Google Nexus 5X). We evaluate four application benchmarks from past work with the SQLite optimization that disables `fsync()` and the Ext4 file system optimization that turns off the journaling in the Ext4 file system.

In our evaluation, the four representative applications that we test respond reasonably fast in a consistent manner, with an average user-perceived response time ranging from 0.9 second to 1.9 seconds. Moreover, our results show that the SQLite optimization can only reduce the user-perceived latency by up to 7.2% on the Galaxy S4 and 7.8% on the Nexus 5X; whereas, eliminating Ext4 journaling can only reduce the user-perceived latency by up to 6.7% on the Galaxy S4 and 3.6% on the Nexus 5X. With both optimizations the user-perceived latency can be reduced by up to 11.8% on the Galaxy S4 and 11.5% on the Nexus 5X. Our results seem to suggest that *the performance of the storage I/O stack, while still important, may no longer be a dominating factor for user-perceived application performance in the latest Android smartphones*.

2 Methodology

Smartphone applications by and large are Graphical User Interface (GUI) based interactive applications.

*Currently working at Nimble Storage in San Jose, CA.

These have a front-end that accepts user generated and system generated input events (e.g. clicks or swipes) which produces deterministic graphical output [9]. The GUI of an Android application contains a set W of GUI objects (e.g., Buttons, Frames). Each GUI object $w \in W$ is represented by a set P_w of properties, where each property $p \in P_w$ has a set of values V_p. The state of GUI at a particular time t can be defined as a set of triples $S_{GUI} = \{(w, p, v) | w \in W, p \in P_w, v \in V_p\}$.

A typical smartphone user interaction usually involves three steps: (1) find a GUI object (e.g., button, textbox), (2) perform some operations (e.g., click, type, swipe) on the GUI object, and (3) wait until a certain state is reached (e.g., the content is shown on screen). This is repeated for the next operation. The user-perceived latency of such operations greatly impacts the user's quality of experience and satisfaction [10]. The HCI community has defined three time limits for user-perceived latency based on human perceptual ability [10]:

- **0.1 second** limit for the user feeling that the system is reacting instantaneously.
- **1.0 second** limit for the user to keep an uninterrupted flow on the task at hand.
- **10 second** limit for keeping the user's attention focused on the dialogue.

In general operations which take around 1 second are considered relatively fast [10]. Given the GUI based and interactive nature of mobile applications, we believe that the user-perceived latency is the best metric for measuring mobile application performance and will use these numbers as baselines in our evaluation. If we deviate much beyond the times given, then we assume that the user is troubled by an applications performance.

The exact way to measure this latency while running an application is subjective. In previous studies [5, 11], MonkeyRunner (a GUI replay tool) is used to simulate user interactions, and the execution time is used to represent the user-perceived latency. However, due to a lack of synchronization between the simulated interaction and the application under test, the measured latency may not accurately reflect the latency a user actually perceives. For instance, the application may still be waiting to fetch data through the network while MonkeyRunner has already returned from performing the user action. In this case the content may not be fully displayed on the screen and when the user checks the screen after performing the action, he or she will think the interaction is still under processing and incomplete.

To make sense of this, we define a *stable GUI state* and use the transition time between two consecutive stable GUI states as the metric for user-perceived latency. When the application receives an input from the user, the application will change its GUI state, and eventually reach a stable state where the user thinks the interaction is completed (e.g., a web page fully loaded). A stable GUI state, must satisfy two conditions:

- the GUI state (S_{GUI}) remains unchanged without further user input.
- background jobs directly related to the operation are completed.

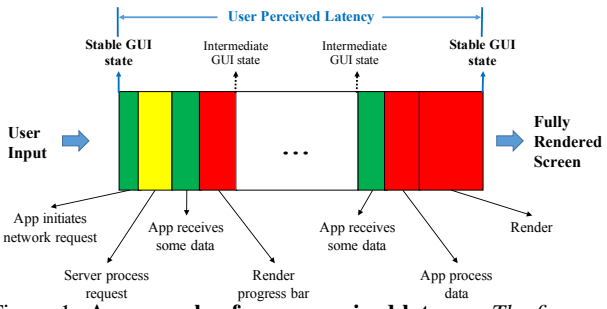


Figure 1: **An example of user-perceived latency.** *The figure shows the breakdown view of the user-perceived latency loading a single web page.*

Figure 1 shows an example of the user-perceived latency and the GUI state transition throughout the loading of a web page. Applications that take longer to respond usually employ a percent-done indicator and incrementally update the screen to make the response feel immediate to the user. When the user hits the button to start loading a web page, which marks the start of the user-perceived latency, the browser will send the request through the network and wait to receive the data needed for rendering the web page asynchronously. While receiving such data, the browser will periodically update the progress bar and change the GUI state into a series of unstable intermediate states which include when the web page is not fully rendered. Eventually when all the data is received, the fully rendered web page will be presented to the user, and the app enters a stable GUI state, which marks the end of the user-perceived latency.

Thus, user-perceived latency is measured as the time taken to transition from the current stable GUI state at the point an event is triggered by a user input, to the next stable GUI state when operation is complete (e.g., a web page is fully loaded and displayed on the screen). We believe that the user-perceived latency, which is quantified by the interval time between two consecutive stable GUI states, is an appropriate metric for measuring mobile application performance.

With this metric, we build our benchmark suite using the Espresso [1] testing framework that provides APIs for simulating user interactions (e.g., click, type) within a single app. The Espresso framework automatically synchronizes between the user interaction and application under test. One user operation is considered finished in Espresso tests when there are no tasks in the default AsyncTask thread pool and the UI thread is idle. In most

cases, the application will enter the next stable GUI state when these two conditions are met. However, in some cases the user interaction results in a series of GUI state changes, which makes it impossible for Espresso to know when the GUI enters a stable state. To address this issue, Espresso allows test writers to register their own customized idling resources with Espresso so that Espresso will wait until the background asynchronous jobs are done. We implement our customized `IdlingResource` for the WebView widget so that it will notify Espresso when the web page is fully loaded. Our benchmark suite simulates a set of predefined user interactions and reports the GUI state transition time between two consecutive GUI stable states, which we consider the best representation of the user-perceived latency.

3 Experiment Setup

3.1 Device Setup

We performed experiments on two smartphones, the Samsung Galaxy S4 Google Play Edition and the Google Nexus 5X. The Samsung Galaxy S4 was released in June 2013, and is equipped with a 1.9 GHz quad-core CPU, 2GB RAM and 16GB NAND flash internal storage, running Cyanogenmod 11.0 (Android 4.4 equivalent). The Google Nexus 5X was released in October 2015, employing a 1.8 GHz hexa-core CPU, 2GB RAM and 32GB internal NAND flash storage, running stock Android 6.0 Marshmallow. By default both phones store application data in the Ext4 file system with journaling (in ordered mode).

To estimate the performance benefits of storage stack optimizations reported in the latest literature [2, 6–8, 12], we adopt two specific performance optimizations in our evaluation, that are each designed for the SQLite database and Ext4 file systems:

- **SQLITE_NO_SYNC**: disable `fsync()` in SQLite.
- **EXT4_NO_JOURNAL**: turn off Ext4 file system journaling.

These optimizations eliminate the journaling overhead in the SQLite database and Ext4 file systems completely. Although these optimizations clearly compromise the persistency guarantees from SQLite and Ext4, they yield close to the optimal performance we can expect from mitigating the journaling overhead in SQLite and Ext4. Therefore, they represent the performance upper bounds achievable by the latest optimizations aiming to minimize the storage stack overhead. **SQLITE_NO_SYNC** removes I/Os to both database table files and journal files from the transaction commit path, and **EXT4_NO_JOURNAL** eliminates all I/Os to Ext4 journals.

We use the default/stock configuration of the devices as the baseline, which has neither of the two optimizations turned on. To apply the **SQLITE_NO_SYNC** optimization, we rebuild the `libsqlite.so` shared library with `SQLITE_NO_SYNC` defined and replace the original library. To apply the **EXT4_NO_JOURNAL** optimization, we disable the `has_journal` option in Ext4 using `tune2fs`. By turning on and off these optimizations, we test a total of four different configurations.

App	Workload
Web	Loading top 50 websites in U.S one by one
Facebook	Swipe up the screen 50 times to load news feed
Messenger	Send 50 messages
Twitter	Post 50 tweets

Table 1: **Summary of application benchmarks.**

3.2 Application Benchmarks

We evaluate four popular Android applications, summarized in Table 1. These applications have been studied extensively in recent literature [4, 5, 11], much of which has been in the context of providing evidence that the storage performance greatly impacts the application performance. Therefore, we keep the workload the same across the configurations and evaluate the application performance benefits from the different storage stack optimizations. For the Web benchmark we implement a web browsing app consisting of a text box for entering a website url and a WebView widget to display the web page. We also implement a customized WebView idling resources for Espresso that will notify Espresso when the web page loading progress reaches 100%. For the Facebook, Messenger (a.k.a. Facebook Messenger) and Twitter benchmarks, we use a re-signed version of the apps downloaded from Google Play so that the Espresso instrumentation tests run on real devices.

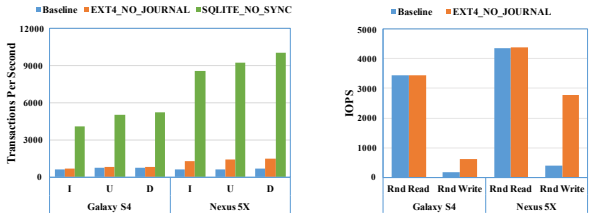
The application benchmarks are implemented as Android instrumentation tests to simulate the user actions to perform the workloads specified in Table 1. Active accounts are used in the Facebook, Messenger and Twitter benchmark runs, and application data are cleaned up before each benchmark is run to force the application to write data to the local storage.

4 Evaluation

The evaluation, based on the experimental platform and workload described above, seeks to answer the following three key questions in our attempt to shed more light on the application performance and the performance impact of storage in Android smartphones.

4.1 How much do the database and file system benefit from storage stack optimizations?

The performance of the SQLite database and Ext4 file system are shown in Figure 2. The **SQLITE_NO_SYNC** optimization speeds up the insert, update and delete



(a) **SQLite performance.** (b) **Ext4 performance.**

Figure 2: **Database and file system performance with optimizations.** *Figure (a) shows the transaction throughput of single op transaction in SQLite; I: Insert, U: Update, D: Delete. Figure(b) illustrates the 4KB random read / write IOPS of the Ext4 file system.*

transaction throughput by respectively 15.0×, 17.6× and 16.8× on Samsung Galaxy S4 and 14.4×, 15.0× and 15.1× on Nexus 5X. The **EXT4_NO_JOURNAL** optimization speeds up the 4KB random write IOPS on Galaxy S4 and Nexus 5X by 3.4× and 7.1× respectively, and does not affect the 4KB read IOPS. The boost on random write IOPS leads to a SQLite transaction throughput speedup of 1.5×, 1.2× and 1.2× on Samsung Galaxy S4 and 2.15×, 2.32× and 2.22× on Nexus 5X for the insert, update and delete transactions respectively.

4.2 How much does the application performance benefit from storage stack optimization?

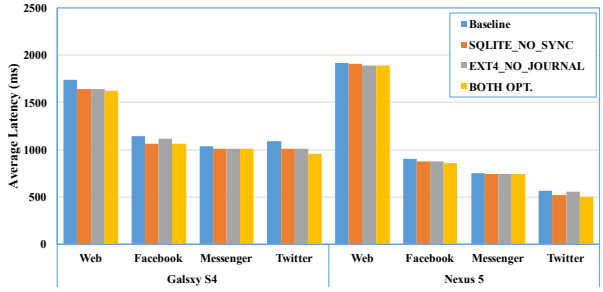


Figure 3: **Average response time of the four application benchmarks.** *Each latency value is averaged over 10 runs. The standard deviation between runs is less than 5%.*

To better assess the importance of storage performance, we filter out the low-latency user actions (which poses less user-perceived latency) that are unlikely to trigger storage I/Os (e.g. type url), so that we can focus on specific operations in each application. These are

- Web: click Go button on soft keyboard to start loading a web page.
- Facebook: swipe up on the screen.
- Messenger: click the Send button.
- Twitter: click the Post button.

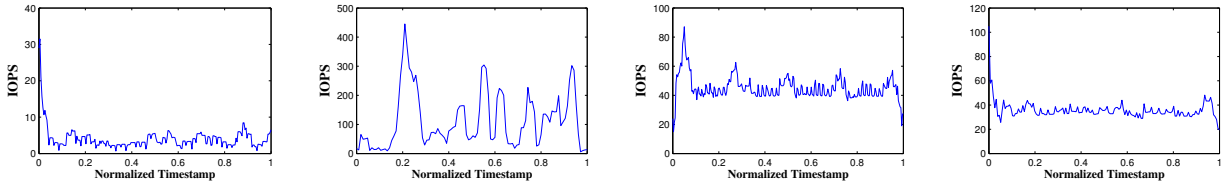
The average latency of the aforementioned operations is shown in Figure 3. In this graph we show the average of 10 runs of the experiments (the standard deviation is less than 5%). Under the default configuration, the Samsung Galaxy S4 takes about 1.7 seconds to load a web page, and around 1.1 seconds to finish the measured user interaction; Nexus 5X takes about 1.9 seconds to load a web page, 0.9 second to swipe down the Facebook feed, 0.7 second to send a message and 0.5 second to post a tweet. According to the response time guidelines we presented [10], these user interactions are reasonably fast and thus provides a satisfactory user experience.

With only **SQLITE_NO_SYNC** turned on, the average user-perceived latency of the measured user interactions in the Web, Facebook, Messenger and Twitter benchmarks is reduced by 5.5%, 7.2%, 2.7% and 6.7% from the baseline respectively on Samsung Galaxy S4, and 0.6%, 2.8%, 0.5% and 7.8% from the baseline respectively on Nexus 5X. With only **EXT4_NO_JOURNAL** turned on, the application benchmarks exhibit 5.3%, 2.1%, 3.0% and 6.7% improvements in user-perceived latency over the baseline on Samsung Galaxy S4, and 1.4%, 3.6%, 0.4% and 3.0% on Nexus 5X. Turning on both optimizations can reduce the user-perceived latency of the four applications under test by 6.3%, 7.2%, 2.6% and 11.8% from the baseline on Samsung Galaxy S4, and 1.2%, 5.0%, 1.1% and 11.5% from the baseline on Nexus 5X.

4.3 Why doesn't the application benefit from better storage performance?

The storage performance benefits seen by the applications largely depend on how they manage their data. Some applications manage all their data in the database, some may leave all the application data unstructured in raw files, while others have both the database and raw files in their storage schema. We collect the I/O trace to */data* partition using the `blktrace` tool in a benchmark run under the default configuration on Galaxy S4. As shown in Figure 4, the IO intensity in the benchmark test of Web, Messenger and Twitter is relatively low compare to the maximum IOPS available, while in the Facebook benchmark test the bursty IOPS reaches 450. Further analysis of the I/O traces demonstrates that the average response time of these I/Os (mostly writes) is only 28 milliseconds, which is relatively small compared to the user-perceived latency of 1.1 seconds. Nevertheless, considering that the benchmark test excludes all human reaction time, the IO intensity is likely to be much smaller in real life scenarios.

We were surprised at the result of the Web benchmark for which the database optimizaiton has almost no effect. (see Figure 3) given that it is basically the same as the WebBench [5] in a previous study, where the WebBench served as the evidence that storage performance significantly affects the performance of the application. The previous work shows that the storage schema used by the browser application consists of a set of blobs storing



(a) **Web** (b) **Facebook** (c) **Twitter** (d) **Messenger**

Figure 4: **IO activity of four benchmarks.**

cached image and media files, and two SQLite database files storing the index to manage the web cache. They also identified that the synchronous writes to the SQLite database files cause a significant delay. However, we find no SQLite database files for the web cache index in both smartphones we tested, instead the index is stored in a flat file. By comparing the WebView cache directory before and after the benchmark run we find that the app generates 740 new files (including cache blobs and index), for a total of approximately 18MB data.

To gain a better understanding of the correlation between web page loading and I/O activities, we re-run the Web benchmark with a 4-second interval inserted between each individual page loading. We collect I/O traces using blktrace during the benchmark run and plot the activity in Figure 5. The user-perceived web page loading time is denoted as red (shaded) in the figure. As seen, the I/O bursts and the web page loading periods are not aligned, suggesting that the writes to the index files and cache blobs are no longer on the critical path of web page loading. This is exactly the solution previous studies have suggested. Therefore, the inefficient design and unnecessary delay in web browser may have been resolved as the Android software has evolved in the last four years.

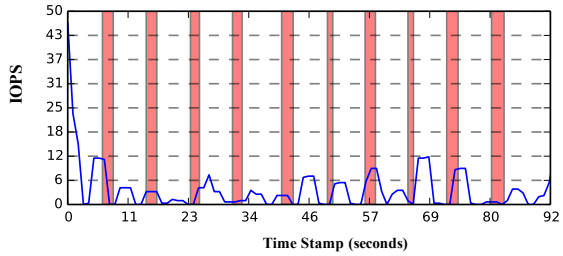


Figure 5: **I/O activities for loading 10 web pages.** *The figure shows the IOPS when loading 10 web pages in the Web benchmark. A four-second interval is inserted between two consecutive web page loadings. The red area in the figure denotes the loading time of each web page.*

5 Conclusion

Our results question the prevailing wisdom about the impact of storage on the smartphone application performance. A threat to validity of our study is that it does not look a vast range of applications nor a wide variety of smartphones. As a result, we cannot claim that our results are broadly representative. However, the fact that on the latest smartphones with powerful CPUs, large DRAM and fast storage devices, the user-perceived latency of four common applications does not benefit much from storage system optimizations suggests that more work is needed to identify the bottleneck in modern mobile systems. With the latest hardware and software in modern smartphones, the storage system seems to no longer be the performance bottleneck whereas applications used to suffer from wimpy storage systems.

Acknowledgments

We are grateful to anonymous reviewers for their feedback. This work is supported in part by the US NSF under Grant Nos. CNS-1116606, CNS-1016609, IIS-0916859, CCF-0937993, CCF-1161767 and CNS-1205472.

References

[1] Espresso testing platform. `https://google.github.io/android-testing-support-library/docs/espresso/`.

[2] DAEHO, AND ET AL. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *USENIX FAST* (2015).

[3] HUANG, AND ET AL. Weardrive: fast and energy-efficient storage for wearables. In *USENIX ATC 15* (2015).

[4] JEONG, AND ET AL. I/O stack optimization for smartphones. In *USENIX ATC* (2013).

[5] KIM, AND ET AL. Revisiting storage for smartphones. *ACM TOS* (2012).

[6] KIM, AND ET AL. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *USENIX FAST* (2014).

[7] LEE, AND ET AL. Waldio: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *USENIX ATC 15* (2015), pp. 235–247.

[8] LUO, AND ET AL. qnvram: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *USENIX HotStorage* (2014).

[9] MEMON, AND ET AL. GUI ripping: Reverse engineering of graphical user interfaces for testing.

[10] NIELSEN, J. *Usability engineering*. Elsevier, 1994.

[11] REN, AND ET AL. Memory-centric data storage for mobile systems. In *USENIX ATC 15* (2015).

[12] SHEN, AND ET AL. Journaling of journal is (almost) free. In *USENIX FAST* (2014).