Improving Smartphone Responsiveness through I/O Optimizations

David T. Nguyen
College of William and Mary
McGlothlin-Street Hall 126
Williamsburg, VA 23606, USA
dnguyen@cs.wm.edu

Ubicomp '14, September 13–17, 2014, Seattle, WA, USA.
Copyright © 2014 ACM 978-1-4503-3047-3/14/09$15.00.
http://dx.doi.org/10.1145/2638728.2638841

Abstract

Smartphones suffer various unpredictable delays, e.g., when launching an application. In this work, we investigate the behavior of reads and writes in smartphones. We conduct the first large-scale measurement study on the Android I/O delay using the data collected from our Android application running on 1480 devices within 188 days. Among others, we observe that reads experience up to 626% slowdown when blocked by concurrent writes. We use this obtained knowledge to design a pilot solution that reduces application delays by prioritizing reads over writes. The evaluation shows that our system reduces launch delays by up to 37.8%.

Author Keywords

Smartphone Responsiveness; Application Delay; I/O Optimizations; Application Launch

ACM Classification Keywords

C.4 [Performance of Systems]: Design studies.

Introduction

A recent analysis [13] indicates that most user interactions with smartphones are short. Specifically, 80% of the apps are used for less than two minutes. With such brief interactions, apps should be rapid and responsive. However, the same study reports that many apps incur

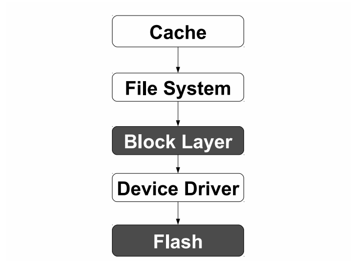

Figure 1: I/O Path.

significant delays during launch and run-time. This work addresses two key research questions towards achieving rapid app response. (1) *How does disk I/O performance affect smartphone app response time?* (2) *How can we improve app performance with I/O optimization techniques?*

In order to understand how disk I/O performance affects smartphone application response time, we plan to conduct a series of measurement studies. First, we want to investigate what portion of the CPU active time Android devices spend in storage waiting for I/Os to complete. When the time the CPUs spend in the storage subsystem is significant, this will negatively affect the smartphone's overall application performance, and result in slow response time. To identify what may be causing such waits, we intend to learn more about I/O activities and their properties. The first property that may be a reason of such waits is I/O slowdown, which quantifies how one I/O type is slowed down due to presence of another. If one I/O activity (e.g., read) is slowed down by another (e.g., write), there will be certain cases in the application life cycle that will suffer from such slowdown (e.g., launch, since reads dominate during launch). Another property to be researched is concurrency. Depending on hardware characteristics, different devices may benefit differently from concurrency. Therefore, we plan to study the speedup of concurrent I/Os over serial ones.

Research Statement
We summarize our research statement as follows:

- Investigate the impact of storage I/O performance on smartphone application delay.
- Explain root reasons of such impact.
- Develop storage-aware solutions with fast application response.

Expected Contributions
If we succeed, we will contribute to better understanding of the limitations of current day and future smartphone devices, i.e., how and why such devices exhibit significantly different application performance when different storage policies are applied in the I/O path and what device and system improvements are necessary. We will also contribute specific innovative storage-aware solutions with fast application response that work well in practice.

Background

First, we introduce the background of our work. In particular, the kernel components on the I/O path are discussed, with the emphasis on the block layer and the flash disk that are directly related to our work. We illustrate the main kernel components affected by a block device operation on the I/O path in Figure 1. The figure is adapted from the literature [5].

Block Layer. At the block layer [1], the main work is scheduling I/O requests from above and sending them down to the device driver. The Linux kernels on recent Android smartphones offer 3 scheduling algorithms: Complete Fair Queuing (CFQ), Deadline, and Noop. CFQ scheduler presented in [4] is the default I/O scheduler in Android smartphones. It attempts to distribute available I/O bandwidth equally among all I/O requests. There are two priority levels: one is the class, and the other is the priority within the class. There are three classes: Real-time, Best Effort, and Idle. Real-time class requests have the highest priority, followed by the Best Effort class whose disk access requests are granted only when there is no real-time request left. The Idle class is given a disk access only when the disk is idle. Within the Real-time and Best Effort classes, there are eight additional priorities

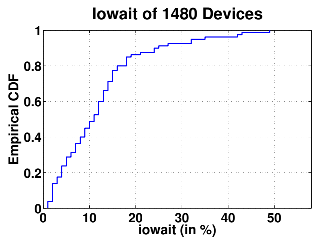

Figure 2: Iowait.

[0(highest) to 7(lowest)]. Requests are placed into queues where each of the queues gets a time slice allocated. There are 8 queues in the Real-time class, 8 queues in the Best Effort class, and 1 queue in the Idle class.

Flash Disk. The last level to be reached by the I/Os is the storage subsystem that contains an internal NAND flash memory, an external SD card (optional), and a limited amount of RAM. The subsystem contains different numbers of partitions, depending on the manufacturer. The partitions can be found in the /dev/block directory. Flash disk differs significantly from the conventional rotating storage. While rotating disks suffer from the seek time bottleneck, flash disks do not. Although providing superior performance compared to conventional storage, flash does have its own limitations. For instance, the erase-before-write limitation requires erase before overwriting a location. This leads to a substantial read/write speed discrepancy, which, among other issues, is discussed in the following subsections as a motivation for our work.

Motivation

In order to understand how disk I/O performance affects smartphone application response time, we conduct a large-scale measurement study using the data collected from our Android app [3] running on 1480 Android devices. The results in Figure 2 reveal that Android devices spend a significant portion of their CPU active time waiting for storage I/Os to complete, also known as *iowait*. Specifically, around 40% of the devices have *iowait* values between 13% and 58%. This negatively affects the smartphone's overall application performance, and results in slow response time. Therefore, in order to improve the application performance, it is essential to investigate possible causes of such waits.

Further investigation identifies that one of the reasons causing such waits is I/O slowdown, which represents the slowdown of one I/O type due to presence of another. In particular, our experimentation reveals a significant slowdown of reads in the presence of writes. Specifically, a sequential read experiences up to 626% slowdown when blocked by a concurrent write. Similarly, a random read experiences up to 293% slowdown when blocked by a concurrent write. This significant read slowdown may negatively impact the application performance during the life cycles when the number of reads dominates. A good example is application launch. Finally, the last property researched is concurrency. Our study suggests that different devices may benefit differently from concurrency. Therefore, we need to be able to adapt to the concurrency characteristics of each device.

Pilot Solution

In order to improve the application delay performance in smartphones, we present our pilot solution called SmartIO [11, 10, 7, 8], a system that reduces the application response time by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters.

I/O Priority Assignment. Our system follows the implications from the previous measurement study. First, since a read suffers a large slowdown in the presence of a concurrent write, the goal is to allow reads to be completed before writes, while avoiding write starvation. In order to achieve this, a third level of I/O priority is added into the current block layer [1], assigning higher priority to reads and lower to writes. This third priority level has a lower priority than the first two priority levels in the existing Linux I/O scheduler (CFQ). Details will be elaborated in the following paragraph. Write starvation is

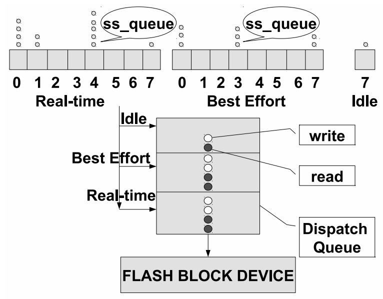

Figure 3: Dispatch Example.

avoided by applying a maximal period of time assigned to a process, which is by default 100ms as used in the CFQ's time slice concept. Beside CFQ, the proposed solution can be easily adapted to other schedulers.

I/O Dispatch. A sample dispatch is illustrated in Figure 3. In the current CFQ implementation, each block device has 17 queues of I/O requests (8 Real-time, 8 Best Effort, and 1 Idle). The existing system selects a queue based on the priorities, takes a request in the queue, and inserts it in the dispatch queue. The queue selection process accounts for two priority levels: the class priority (Real-time, Best Effort, Idle), and the priority within the class (0-7). Our system does not change the above dispatch process but uses a third priority level to organize the dispatch queue in favor of the reads. The dispatch queue is then divided into three sections, from the bottom up real-time, best effort, and idle requests. Each section is organized such that reads precede writes.

Concurrency Profiler. The system uses the knowledge of the device's four concurrency parameters to issue I/Os from the dispatch queue. The parameters include the optimal number of sequential or random reads (writes) that benefit most from concurrency. To achieve this, SmartIO measures the concurrency parameters during installation by issuing reads and writes, and calculates the speedup of concurrent I/Os over serial ones.

Evaluation

To evaluate the performance of our pilot solution, we measure the launch and run-time delay of 20 popular apps (5 sensing, 5 regular, 5 streaming, and 5 games) from Google Play, with and without SmartIO. During the experiment, our Nexus 4 has all radio communication disabled except for WiFi that is necessary to provide stable Internet connections required on most apps. The screen is set to stay-awake mode with constant brightness, and the screen auto-rotation is disabled. Only one app runs at a time, and no other app is in the background. The cache is cleared before each measurement in order to evaluate real performance improvement caused by SmartIO.

Launch Delay. The Android Monkey tool [2] is utilized to trigger the launch process of each app. The application *launch delay* starts when the launch process is triggered, and ends when the process completes. The delays include four components obtained through the Linux *time* command: the time taken by the app in the user mode (*user*), the time taken by the app in the kernel mode (*system*), the time the app spends waiting for the disk I/Os, and the time the app spends waiting for the network I/Os.

The launch delay of the 20 apps with SmartIO *disabled* is illustrated in Figure 4(a). The launch delay of the 20 apps with SmartIO *enabled* is illustrated in Figure 4(b). Both figures are plotted with standard deviations. The reduction in launch delays with SmartIO ranges from 6.3% (Accelerometer Monitor) to 37.8% (The Simpsons) as compared to launch delays without SmartIO. The launch delay with SmartIO enabled for all the 20 apps is on average 20.5% faster than with SmartIO disabled. These results are expected. The app launch is I/O intensive, and includes a lot of read activities. The average number of reads observed for the 20 apps is 5 times higher than writes. Some apps even go to the extremes, for instance, the Temple Run game has reads exceeding writes by 58 times. Therefore, the read-preference nature of SmartIO contributes to reducing disk I/O delay during the launch. Specifically, the disk I/O delay portion itself is reduced on average by 69%. Slight difference in the user and system time of several apps suggests that SmartIO also affects other time components. We reserve further investigation for future work.

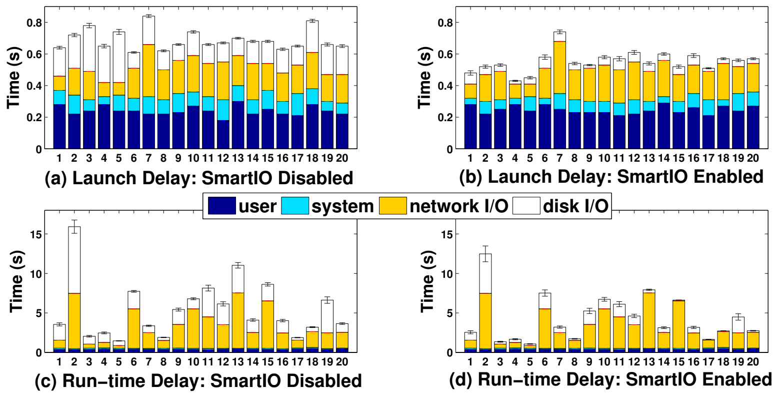

Figure 4: Launch and Run-time Delay. **1:**Angry Birds; **2:**GTA; **3:**NFS; **4:**Temple Run; **5:**The Simpsons; **6:**CNN; **7:**Nightly News; **8:**ABC News; **9:**YouTube; **10:**Pandora; **11:**Facebook; **12:**Twitter; **13:**Gmail; **14:**Maps; **15:**AccuWeather; **16:**Accelerometer; **17:**Gyroscope; **18:**Proximity Sensor; **19:**Compass; **20:**Barometer.

Run-time Delay. In order to test delays of apps running on the phone with SmartIO, we utilize again the Android Monkey tool to generate streams of 500 user events such as clicks, touches, or gestures. The *run-time delay* is defined as the time needed to complete the 500 user events in a running app. We run the experiments with the same 20 Android apps mentioned previously. Each app has a predefined set of user activities triggered through the Monkey tool. The run-time delay for both cases is measured with the *time* command, once with SmartIO enabled, and once with SmartIO disabled. Monkey is a command-line tool that can send a stream of events into the phone's system in a repeatable manner. We apply a constant seed value (10) to generate the same sequence of events. The events are individually adjusted for each app to represent a typical usage, for instance, in Gmail we read and write an email, add a contact, change a label, etc.

The run-time delay of the 20 apps with SmartIO disabled is illustrated in Figure 4(c). The run-time delay of the 20 apps with SmartIO enabled is illustrated in Figure 4(d). Both figures are plotted with standard deviations. The reduction in run-time delay with SmartIO ranges from 2% (Pandora) to 29.6% (Angry Birds) as compared to run-time delay without SmartIO. The run-time delay with SmartIO enabled for all the 20 apps is on average 16.9% smaller than with SmartIO disabled. Clearly, the run-time delays do not benefit from using SmartIO as much as the application launch. This is reasonable, since the application launch is more I/O intensive than the application run-time. For the 20 apps, the average number of I/Os during launch is 2 times higher than during run-time. While the run-time delay of the games with SmartIO is on average 23% smaller, the streaming apps have on average only 4% smaller run-time delay. This is expected, since the games have decent disk I/O activity during the run-time, whereas the streaming apps are mainly network-bounded. Finally, the average gains of the sensing and regular category are 18% and 20%, respectively. The improvement in the disk I/O portion of the time spent during run-time is on average by 54%.

Remaining Steps

We plan to conduct more controlled performance evaluations. In particular, we intend to choose a group of I/O intensive and a group of I/O non-intensive apps, and determine the delay performance gain. Additionally, the impact of our system on writes will be studied. It will be important to have deeper understanding of what kinds of workloads are benefiting from SmartIO, and how prevalent they are in apps.

Although SmartIO is designed to favor reads over writes, it may be desirable to have the ability of lowering this read-preference, or even changing it to write-preference for some workloads. This more accurate priority control will require further understanding of the read and write priority. We will have to investigate how to determine such priorities, and how to dynamically and efficiently enforce them in the system. In particular, we plan to collect I/O patterns from a large number of apps, and train a supervised classification model or an unsupervised clustering model for run-time priority assignment.

Finally, we expect that the energy overhead will fluctuate as we introduce a goal oriented approach to controlling the read/write priority. Our experience from previous work [9, 6] will guide us in balancing the energy and performance trade-off through various I/O optimization techniques.

Related Work

Little work in the research community directly relates to ours. Yan et al. [13] and Parate et al. [12] propose systems predicting application launch to reduce the launch delay. Their systems reduce perceived delay through application prelaunching. However, mispredictions of the proposed approaches will lead to significant memory and energy overhead. We address the problem of slow application launch by analyzing possible reasons of the slowdowns in the granularity of read and write I/Os, and designing a system that has a positive impact on the application performance beyond the launch delay.

Biographical Sketch

David T. Nguyen has been working on his Ph.D. in Computer Science at the College of William and Mary since August 2011, and is expected to graduate in May 2016. He is advised by Dr. Gang Zhou.

References

[1] Block layer. `http://goo.gl/SwdLZ5`, 2014.

[2] Monkey. `http://goo.gl/F14hW`, 2014.

[3] Storebench download. `http://goo.gl/ava9eV`, 2014.

[4] Axboe, J. Linux block io-present and future. In *Proc. of Ottawa Linux Symp* (2004).

[5] Bovet, D., and Cesati, M. *Understanding the Linux Kernel*, 3 ed. O'Reilly & Associates, Inc., 2005.

[6] Nguyen, D. T. Evaluating impact of storage on smartphone energy efficiency. In *Proc. of ACM UbiComp* (2013).

[7] Nguyen, D. T. Smartphone application delay optimizations. In *Proc. of ACM MobiSys* (2014).

[8] Nguyen, D. T. Smartphone application launch with smarter scheduling. In *Proc. of ACM UbiComp* (2014).

[9] Nguyen, D. T., Zhou, G., Qi, X., Peng, G., Zhao, J., Nguyen, T., and Le, D. Storage-aware smartphone energy savings. In *Proc. of ACM UbiComp* (2013).

[10] Nguyen, D. T., Zhou, G., and Xing, G. Poster: Towards reducing smartphone application delay through read/write isolation. In *Proc. of ACM MobiSys* (2014).

[11] Nguyen, D. T., Zhou, G., and Xing, G. Video: Study of storage impact on smartphone application delay. In *Proc. of ACM MobiSys* (2014).

[12] Parate, A., Böhmer, M., Chu, D., Ganesan, D., and Marlin, B. M. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proc. of ACM UbiComp* (2013).

[13] Yan, T., Chu, D., Ganesan, D., Kansal, A., and Liu, J. Fast app launching for mobile devices. In *Proc. of ACM MobiSys* (2012).