Analysis of the Effective Use of Thread-Level Parallelism in Mobile Applications

A preliminary study on iOS and Android devices

Elana Bogdan
Swarthmore College
ebogdan1

Hongin Yun
Swarthmore College
hyun1

Abstract

Following the model of earlier studies of desktop software, we perform a survey of thread-level parallelism in common mobile applications. In particular, we study the Android and iOS platforms, through a representative sample of 3rd-party software in their respective app stores. Ultimately, we conclude that multiple cores may not be necessary for the majority of mobile experiences, observing that iOS apps tend to have slightly greater average parallelism, with slightly lower variance, than Android apps.

Keywords multi-core, thread-level parallelism, mobile apps

1. Introduction

In chip engineering, there are two broad strategies for increasing computational speed: design processors that can sustain higher clock rates, and design systems that incorporate more processing cores. The former strategy has more or less hit a plateau, as overheating and the limitations of modern cooling systems have come to present an inherent physical barrier. Therefore, in recent years, the majority of time and energy has instead gone into developing more parallel hardware: multiprocessors containing several cores each, and computer systems that contain one or more of these multprocessing chips. However, while this frontier continues to advance, its computational benefits remain limited by the structure of the software that it serves: to take advantage of parallel hardware, developers must practice good parallel design patterns.

We suspect that many systems do not fully utilize their parallel capacities, and, in this paper, we seek to extend the existing body of research in this area into the realm of mobile devices.

2. Background and related work

As a field of computer science, parallel design has a relatively long history, dating back to the early days of servers and networked computers. We draw our inspiration from a couple of more recent studies.

2.1 Parallelism on desktop workstations

In 2010, Flautner et al. studied a range of desktop applications running on Microsoft Windows 7 and Apple's OS X Snow Leopard, analyzing for parallelism. [10] Using the metric of Thread Level Parallelism, or TLP (see section 3.1), they concluded that 2-3 cores were more than sufficient for most applications, and that current desktop applications were not fully utilizing multi-core architectures.

Other studies in a similar vein date back to 2000, when Flautner et al. first investigated the thread-level parallelism and interactive response time of desktop applications. [9] This study was done when multiprocessing was prevalent mostly in servers and had only just begun to enter into desktop machines. While servers were considered to be a natural fit for multiprocessing, due to the parallel nature of serving multiple clients, the benefits of multiprocessing for desktop applications were not obvious. Now, as multiprocessor systems are entering the smartphone market, we believe it is a natural extension of these studies to ask whether the benefits of multiprocessing are fully realized on mobile devices.

2.2 Multi-processing in Android phones

Since its initial release in May 2007, Android and the devices that run Android OS have evolved rapidly. The first commercially available phone to run Android was the HTC Dream, released on October 22, 2008. [3] HTC Dream had a Qualcomm MSM7201A chipset, including an ARM11 application processor, ARM9 modem, and high-performance digital signal processors. [12] In 2010, Google and several handset manufacturers launched a line of smartphones and tablets as their flagship Android devices under the name Nexus. The Nexus One, manufactured by HTC in January 2010, was released with Android 2.1 and had a Qualcomm QSD 8250 with a single core Qualcomm Scorpion CPU [5]. The world's first Android device with a multi-core processor was the LG Optimus 2X, which was equipped with NVIDIA Tegra 2 system-on-a-chip and a 1 GHz dual-core processor. [4]

The latest Samsung Galaxy S4 and Galaxy Note 3, released respectively in March and September 2013, are

equipped with multiple-core processors. All versions of Galaxy S4 and Note 3 are equipped with quad-core processors, and flagship models shipped to certain markets are even equipped with octa-core processors. Both Galaxy S4 and Galaxy Note 3 can run the latest Android 4.3 Jelly Bean. [1] [2]

2.3 Multi-processing in iPhones

Historically, Apple has been very tight-lipped about the internal components of its products – iPhone processors included. However, with a little bit of investigative work, various third parties have determined that the first two iPhone models, released in 2007 and 2008 respectively, both had single-core processors clocked at around 412 MHz. In 2009, with the release of the purportedly fast 3GS model (the "S" stood for "speed"), the iPhone got a boost to 600 MHz. Empirical data shows that subsequent iPhone models have had variable speed processors, ranging from 750 MHz at the low end (of the iPhone 4) to 1.3 GHz at the high end (of the iPhone 5S). Speculation has it that these models use their variable clock rates to conserve energy whenever possible, an important consideration on mobile devices. [8]

It wasn't until the iPhone 4S that Apple first began using dual-core processors, with the introduction of their A5 chip in 2011. They threw in an improved version of this chip, along with a three-core graphical processor, in the iPhone 5.

Interestingly, it had been a year earlier, in 2010, that Apple released the first version of iOS to support multi-tasking, iOS 4. This release was further notable for including Grand Central Dispatch, a new technology that offered developers a simple, high-level interface for efficient thread management. The operating system first shipped on the iPhone 4, which, with its single-core A4 processor, could not support genuine parallelism. Nevertheless, the A4 was sufficiently fast to achieve an illusion of concurrency, and the new operating system and multi-threading technology helped pave the way for the advent of the 4S the following year.

3. Methodology

In the interests of consistency, we have tried to model our methodology on the aforementioned work by Flautner et al. In this section, we will summarize that approach, then go on to discuss the particularities of working with Android and iOS systems. It is important to note that we carried out all of our research using actual, physical phones. This hardware allowed a far higher degree of accuracy and transparency in our data collection than any kind of emulation would have offered.

3.1 Metrics

There are several different common metrics for quantifying parallelism in computer systems.[9][10] One simple and intuitive metric is Machine Utilization, which is a measure of the percentage of total processing resources that gets used during execution. The formula for Machine Utilization is shown in Equation 1:

$$\text{Machine Utilization} = \frac{\sum_{i=1}^{n} c_i i}{n} \tag{1}$$

In this equation, n is the number of thread contexts in the subject machine, and c_i is the fraction of time that $i = 0, ..., n$ number of threads were executed concurrently. If all processors in the machine were fully utilized during the execution of the benchmark, Machine Utilization would be 1. This intuitive metric is, however, not suitiable for the type of study we conducted. Applications on mobile devices tend to incur a significant amount of idle time, when no threads are being executed in any of the processors, due to a high degree of user interactivity and I/O activity.

We therefore decided to use Thread Level Parallelism (TLP), the same metric that Flautner et al. used in their pioneering research. TLP is a variation of Machine Utilization that factors out idle time. The formula for TLP is given in Equation 2:

$$\text{TLP} = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \tag{2}$$

As in Equation 1, n is the number of thread contexts in the subject machine, and c_i is the fraction of time that i threads were executed concurrently. TLP output values will fall between 1 and n.

One last caveat we had to consider was how exactly to measure c_0, the idle time of the system. While this measurement might be straightforward on a relatively simple operating system, both Android and iOS are constantly running a host of background processes. Some of these processes are indispensible to the active app, while others may serve entirely unrelated functions of the system; (for example, they might monitor cellular connectivity). Our goal was to measure each app in as much isolation as possible, but, since we had no sure way of determining which background processes were a part of the app and which weren't, we ran all of our TLP calculations twice. The first time, we assumed that all background processes constituted "idle" time for the foreground app, while the second time we assumed that they were "active." The reality is most likely somewhere in the middle (see section 5 for further discussion).

3.2 Benchmarks

We conducted two separate sets of experiments. The first was intended to encompass a broad and representative cross-section of applications currently available on the Android Market and iTunes app stores. Our sampling policy involved selecting three applications from each of seven popular categories:

1. Business
2. Entertainment
3. Games (Action)
4. Games (Puzzle)
5. Media
6. News
7. Social Networking

To ensure that these applications accounted for a substantial amount of user experience, we pseudorandomly selected them out of the 50 most popular free apps for each category.[1]

For our second set, we handpicked 10 apps that had cross-platform success on both Android and iOS. We found two such apps in each of five popular categories:

1. Entertainment
2. Games
3. Media
4. Productivity
5. Social Networking

The goal here was to conduct a more focused study of how each mobile operating system handles parallelism differently, by controlling for the particular software being run.

We collected data on each of these benchmarks in a three-pass process. The first pass involved exploring basic functionality without our debugging software running. This pass enabled us to gain familiarity with each app; observe its baseline behavior; and take care of setting up any accounts, granting any permissions, completing any tutorials, etc. that would be required on an initial run. During our second and third passes, we recorded data while manually engaging the central features of each app, as we had previously determined them. The purpose of recording in two passes was to help control for noise associated with background processes that we did not have control over. As much as possible, we tried to reproduce the behavior of the second pass during the third pass, based on careful notes of the input we had provided.

On Android, each of the second and third passes spanned a duration of 90 seconds, while they spanned 45 seconds on iOS (see section 3.4.1). This duration always included the opening and start-up time for each app.

3.3 Android setup

3.3.1 Systrace

Systrace helps analyze the performance of an application by capturing that application and other Android system processes, then representing them in a graphical format.[6] The tool comes with the Android Software Development Kit (SDK), available for free at the official Android developer website. In order for an application's activity to be traced, the application must be run on a physical device connected to a developing system via USB. Systrace, which runs as a Python script on the developing machine, then establishes a debugging connection via the Android Debugging Bridge (ADB). Systrace calls ATrace, a native Android binary, via ADB, and then ATrace in turn uses FTrace to capture kernel events. FTrace is a Linux kernel tool for tracing function execution in the Linux kernel. FTrace operates through instrumenting kernel functions; when the kernel is configured to support function tracing, the compiler adds code to the prologue of each function.[7] This routine does cause some overhead to the application that is being traced, but quantifying the exact amount of overhead is outside the scope of this paper.

Systrace outputs combined data from the Android kernel and generates an HTML report that gives an overview of every activity that was processed on the device for a given period of time. The output trace file shows a detailed overview of CPU activity, including process name, start time, process and thread id, and the CPU on which the process was executed. We built a Python script of our own to parse the string output data and compute the duration and TLP for all cores in the subject system.

The version of Systrace that we used came included in the Android SDK 4.2 (API 17).

3.3.2 Hardware

Our Android device was a Samsung Galaxy S3 I747, which has a Qualcomm MSM8960 Snapdragon chipset with 1.5 GHz Advanced Dual-core. The operating system on the device was upgraded to version 4.1.2.

3.4 iOS setup

3.4.1 Instruments

Apple restricts most low-level access to system information on iOS, but it does provide Instruments – a free, first-party debugging package, bundled with every download of Apple's Xcode IDE. Instruments incorporates a wide variety of tools for measuring anything from backlight brightness and battery usage to zombie processes and memory leaks. These tools, which permit some degree of customization, are essentially a front end to DTrace, a common dynamic instrumentation program for Unix-based systems. DTrace itself relies on making modifications to the system kernel, but Apple does not allow third-party developers that privilege directly.

Through Instruments, we ran the Time Profiler tool, which claims to perform "low-overhead time-based sampling of processes running on the system's CPUs." This tool does not have a visible impact on the operation of most apps, but it does seem to incur a fair amount of overheard on the computer system that runs it. Although our initial goal had been to record over 90-second time windows, trial runs indicated that this duration would consistently cause Time Profiler to hang indefinitely, requiring a force quit. We therefore chose to record over a less taxing 45-second range, which

[1] We used the Python `random` library to generate pseudorandom values. We had to reject a couple of apps on the basis of requiring preexisting accounts or coroporate affiliations.

was still unreliable, but resulted in crashes only around 50% of the time. [2]

We used the latest version of Instruments at the time of our research, v5.0.1, running on Mac OS 10.9.

3.4.2 System preparation

In order to further reduce noise in our data and enhance the isolation of the foreground app, we deactivated several of the automatic features of iOS 7. Among these features were Background App Refresh, (which schedules apps to run time-limited tasks in anticipation of their next use) and the parallax effect, (which acceses accelerometer data to re-render UI elements at high frequency). Both of these features are easily controlled via the Settings menu. Finally, we made sure to open the iOS 7 app switcher and kill all apps before the start of each data collection pass.

3.4.3 Hardware

Our test device was a 16 GB iPhone 4S with a dual-core A5 processor running at around 800 MHz. At the time of data collection, it had just over 1 GB of free space and was running iOS 7.0.4. We connected it via USB 2.0 to a 2011 iMac with a 3.1 GHz Intel Core i5 processor and 8 GB of RAM.

4. Experimental results and analysis

In total, we collected data on 52 distinct apps – 21 for Android, 21 for iOS, and 10 that ran on both platforms. The arithmetic mean of all TLP values we calculated was around 1.28, indicating a low, yet non-negligible degree of parallelism among mobile apps. Generally speaking, TLP values were significantly higher when calculated with background processes counted as a part of the active process than when calculated with background processes as idle. Likewise, TLP values for iOS tended to be higher than those for Android, on the order of 1.33 to 1.22 for TLP w/background and 1.15 to 1.00 for TLP w/o background.

At the extreme ends of our TLP w/background ranges, we had values as low as 1.02 (for Android) and as high as 1.70 (also for Android). In fact, the variance for our Android data overall (0.028) was significantly higher than the variance for our iOS data (0.008).

Beyond these broad comparisons, there were no obvious patterns that emerged in our data. Category or genre of app does not appear to be a strong determinant of TLP, although it is true that, in our direct comparison of 10 apps, both platforms had maximal TLP values for Entertainment and Social Networking apps and minimal TLP values for Games.

For a complete listing of our data, please consult the tables and figures on pages 6-9 of this paper.

[2] We confirmed this crash rate on multiple systems, suggesting that it was not an error in our setup. It remains unclear why Instruments could not handle the full 90 seconds, even despite a large buffer size.

5. Discussion

Trying to suss out the primary culprit behind our low TLP values is a difficult business, since actual parallelism is a product of hardware, application, and operating system combined.

On the hardware side, even a phone equipped with a multi-core processor may redirect its most parallel computations to a designated GPU, for which we have no source of data. (There's a decent chance that this divison of labor is precisely why our Games category had some of the lowest TLP on the CPU.) The hardware may also shut down one of its cores altogether to conserve energy, preempting any potential for parallelism. (Although, given that our devices were receiving continuous USB power throughout data collection, this possibility seems unlikely.)

On the application side, even an app with many threads may be designed to offload the bulk of its processing to a remote server, then retrieve the results in a less parallel fashion. (On iOS, the Pho.to Lab app functioned exactly like this.)

And, lastly, the operating system needs to do the actual scheduling of those threads, and it is anything but predictable how the OS will balance the work of a single application alongside other duties of the system.

Amidst all this uncertainty, we can minimally conclude that counting background processes as non-idle work, associated with the foreground app, is the better metric to be using for TLP. On the Android side, parallelism was practically nonexistent under the other metric, which runs contrary to our intuition that at least a few apps should have good parallel design. Meanwhile, on the iOS side, the primary two background processes we observed were `SpringBoard` and `backboardd`, both of which handle crucial, UI-related services for all apps, (such as processing taps and gestures on the multi-touch display). The activity of these processes is thus inherently tied to the activity of the foreground app, and we would be remiss to count it as idle.

Further discussion of issues specific to one operating system or the other follows below:

5.1 Android

The aforementioned high TLP value of 1.70 was a definite anomaly, produced by an application called Drippler. Drippler is a rather simple news and magazine app that provides daily tips and updates about the mobile devices with which the user accesses it. We did not have access to the source code, but, upon examining the trace data of the app, we found out that it uses the thread pool function, which is a built-in Java technology for generating and managing threads. We suspect that thread pool was what enabled and facilitated Drippler's strongly parallel behavior.

5.2 iOS

If there is one obvious reason why iOS apps would have consistently higher TLP values than Android apps, it's Grand Central Dispatch (GCD). Managing pthreads and mutex locks at a low level can be intimidating for many developers, resulting in a high barrier to entry for parallel design. However, using GCD is a very straightforward process, for which there exists a wealth of official documentation and accessible tutorials. As a testament to the ubiquitousness of this technology, it is an industry best practice for iOS developers to assign all heavy computation to background threads via GCD, in order to avoid UI hangs. Thus it is logical to assume both that GCD would increase the amount of parallelism in an average iOS app and that it would yield fairly consistent TLP values across the board, given that it is the single go-to strategy for managing threads.

6. Future directions

For this paper, our goal was to study apps in isolation as much as possible. This would allow us to determine whether or not mobile developers were building good parallel structure into their own apps, on an individual basis. However, these data alone cannot answer the broader question of whether multiprocessors are right for smartphones. For instance, it could be the case that typical usage of one of these devices – which involves listening to music, checking e-mail, tracking geolocation, and downloading software updates all at once – actually yields far higher TLP, close to 2.00. In this case, multiprocessors would be an invaluable component of mobile systems, even if individual app developers don't know how to make good use of them. Thus, future work might include recording data based on other usage patterns.

Another obvious realm to explore would be the third major mobile operating system: Windows Mobile. While Windows-based phones currently hold less than 5% market share, they are on the rise and already twice as prevalent as the next closest runner-up (Blackberry).[11]

Finally, on all of these platforms, it would be worth collecting additional data, both to increase the rigor of our existing conclusions and to look into related questions. These questions might include: how consistent TLP results are across many runs of the same app; what kinds of threading behavior different apps exhibit; and how much parallelism is handled by the GPU.

7. Conclusion

For the vast majority of mobile applications, having access to two cores on a fast, modern multiprocessor seems to be overkill. Either the current demand for software does not require it, or developers do not currently know how to meet a demand that does. Nonetheless, there is reason to believe that improved tools for handling multi-threading, such as Java thread pools and Apple's Grand Central Dispatch, may already be helping developers bridge the gap between physical capacity and realized potential. We believe that there remains room for further progress on these sorts of high-level technologies.

Acknowledgments

We would like to thank Ben Ylvisaker for helping us direct our line of inquiry and for collaborating with us as we continue this research.

References

[1] Galaxy Note 3 Specification. http://www.samsung.com/us/mobile/cell-phones/SM-N900AZKEATT-specs.

[2] Galaxy S4 specification. http://www.samsung.com/us/mobile/cell-phones/SGH-I337ZRAATT-specs.

[3] T-mobile press release. http://www.t-mobile.com/company/PressReleases_Article.aspx?assetName=Prs_Prs_20080923&title=T-Mobile%20Unveils%20the%20T-Mobile%20G1%20%E2%80%93%20the%20First%20Phone%20Powered%20by%20Android.

[4] LG press release. http://www.lg.com/global/press-release/article/lg-launches-world-first-and-fastest-dual-core-smartphone.jsp.

[5] "Nexus One Owner's Guide," Google, Tech. Rep., 2010.

[6] Systrace. http://developer.android.com/tools/help/systrace.html.

[7] T. Bird, "Measuring function duration with ftrace," 2009.

[8] EveryiPhone.com, "What processor or processors do the iPhone models use?" last accessed 12/2013. [Online]. Available: http://www.everymac.com/systems/apple/iphone/iphone-faq/iphone-processor-types.html

[9] K. Flautner, R. Uhlig, S. Reinhardt, and T.Mudge, "Thread-level parallelism of desktop applications," *Workshop on Multi-threaded Execution, Architecture, and Compilation*, 2000.

[10] K. Flautner, G. Blake, R. G. Dreslinski, and T.Mudge, "Evolution of thread-level parallelism in desktop applications," *SIGARCH Computer Architecture News*, vol. 38, pp. 302–313, 2010.

[11] D. Kerr. (2013, November) Android dominates 81 percent of world smartphone market. [Online]. Available: http://news.cnet.com/8301-1035_3-57612057-94/android-dominates-81-percent-of-world-smartphone-market/

[12] L. Zhang, B. Tiwana, Z. Qian, and Z. Wang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," *CODES/ISSS '10 Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010.

	Android			iOS		
Category	**App**	**TLP (w/back.)**	**TLP (w/o back.)**	**App**	**TLP (w/back.)**	**TLP (w/o back.)**
Business	Backpage	1.28	1.00	Fiverr	1.43	1.31
	Olive Office	1.08	1.00	QuickVoice	1.21	1.06
	Vault-Hide	1.38	1.00	Photo Collage Free	1.18	1.09
	Mean:	**1.25**	**1.00**	**Mean:**	**1.27**	**1.16**
Games - Action	Battle of Zombies	1.04	1.00	Temple Run	1.23	1.07
	Jetpack Joyride	1.03	1.00	Dark District	1.30	1.18
	Skee-Ball	1.21	1.00	Real Steel World Robot Boxing	1.39	1.27
	Mean:	**1.09**	**1.00**	**Mean:**	**1.30**	**1.17**
Games - Puzzle	Fruits & Berries	1.02	1.00	Disco Bees	1.35	1.16
	Heads Up Charades	1.09	1.00	Can You Escape?	1.25	1.06
	Where's My Water 2	1.22	1.02	Bejeweled Blitz	1.26	1.09
	Mean:	**1.11**	**1.01**	**Mean:**	**1.29**	**1.10**
Entertainment	Best Vines	1.19	1.00	TwitchTV	1.31	1.25
	Bitstrips	1.24	1.00	Bell for Christmas	1.15	1.04
	Watch ABC	1.15	1.02	CamWow	1.33	1.08
	Mean:	**1.19**	**1.01**	**Mean:**	**1.26**	**1.12**
Media	BS Player	1.12	1.00	Vimeo	1.33	1.13
	Torrent Video Player	1.05	1.00	InstaSize	1.29	1.14
	Photo Locker	1.39	1.00	Pho.to Lab	1.40	1.16
	Mean:	**1.19**	**1.00**	**Mean:**	**1.34**	**1.14**
News	Drippler	1.70	1.00	BBC News	1.44	1.23
	Lotto Results	1.21	1.00	iCitizen	1.36	1.22
	Yahoo	1.17	1.00	AARP	1.45	1.27
	Mean:	**1.36**	**1.00**	**Mean:**	**1.42**	**1.24**
Social Networking	Emojidom Smileys	1.47	1.00	LinkedIn	1.43	1.13
	Foursquare	1.36	1.00	Kik Messenger	1.38	1.13
	Text Free	1.17	1.00	OKCupid	1.41	1.15
	Mean:	**1.33**	**1.00**	**Mean:**	**1.41**	**1.14**
	Overall mean:	**1.22**	**1.00**	**Overall mean:**	**1.33**	**1.15**
	Variance:	**0.028**	**0.000**	**Variance:**	**0.008**	**0.006**

Table 1: TLP data by category and operating system
(w/back. = counting background as part of the active process)
(w/o back. = counting background as idle)

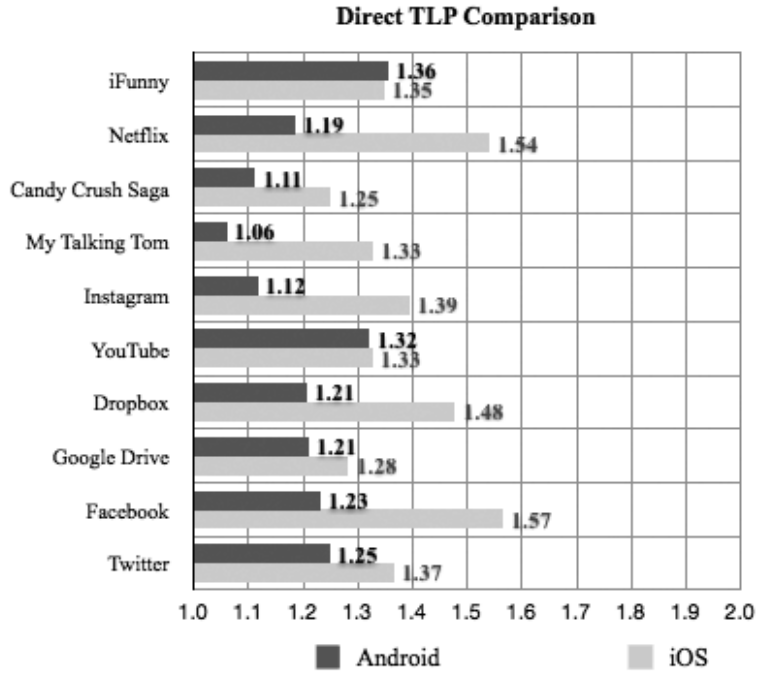

Figure 1

Category	App	Android TLP	iOS TLP
Entertainment	iFunny	1.36	1.35
	Netflix	1.19	1.54
	Mean:	**1.27**	**1.44**
Games	Candy Crush Saga	1.11	1.25
	My Talking Tom	1.06	1.33
	Mean:	**1.09**	**1.29**
Media	Instagram	1.12	1.39
	YouTube	1.32	1.33
	Mean:	**1.22**	**1.36**
Productivity	Dropbox	1.21	1.48
	Google Drive	1.21	1.28
	Mean:	**1.21**	**1.38**
Social Networking	Facebook	1.23	1.57
	Twitter	1.25	1.37
	Mean:	**1.24**	**1.47**
	Overall mean:	**1.20**	**1.39**
	Variance:	**0.008**	**0.011**

Table 2: Direct TLP comparison data
(calculated with background as part of the active process)

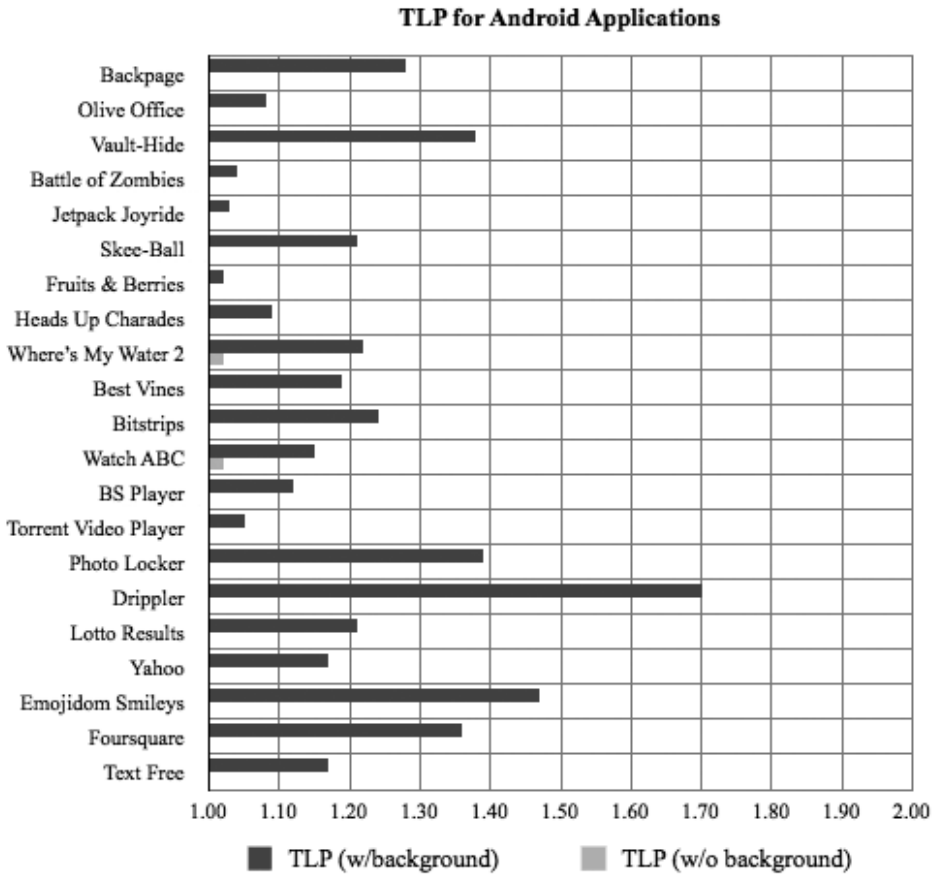
TLP for Android Applications
Backpage
Olive Office
Vault-Hide
Battle of Zombies
Jetpack Joyride
Skee-Ball
Fruits & Berries
Heads Up Charades
Where's My Water 2
Best Vines
Bitstrips
Watch ABC
BS Player
Torrent Video Player
Photo Locker
Drippler
Lotto Results
Yahoo
Emojidom Smileys
Foursquare
Text Free
1.00
1.10
1.20
1.30
1.40
1.50
1.60
1.70
1.80
1.90
2.00
TLP (w/background)
TLP (w/o background)

Figure 2

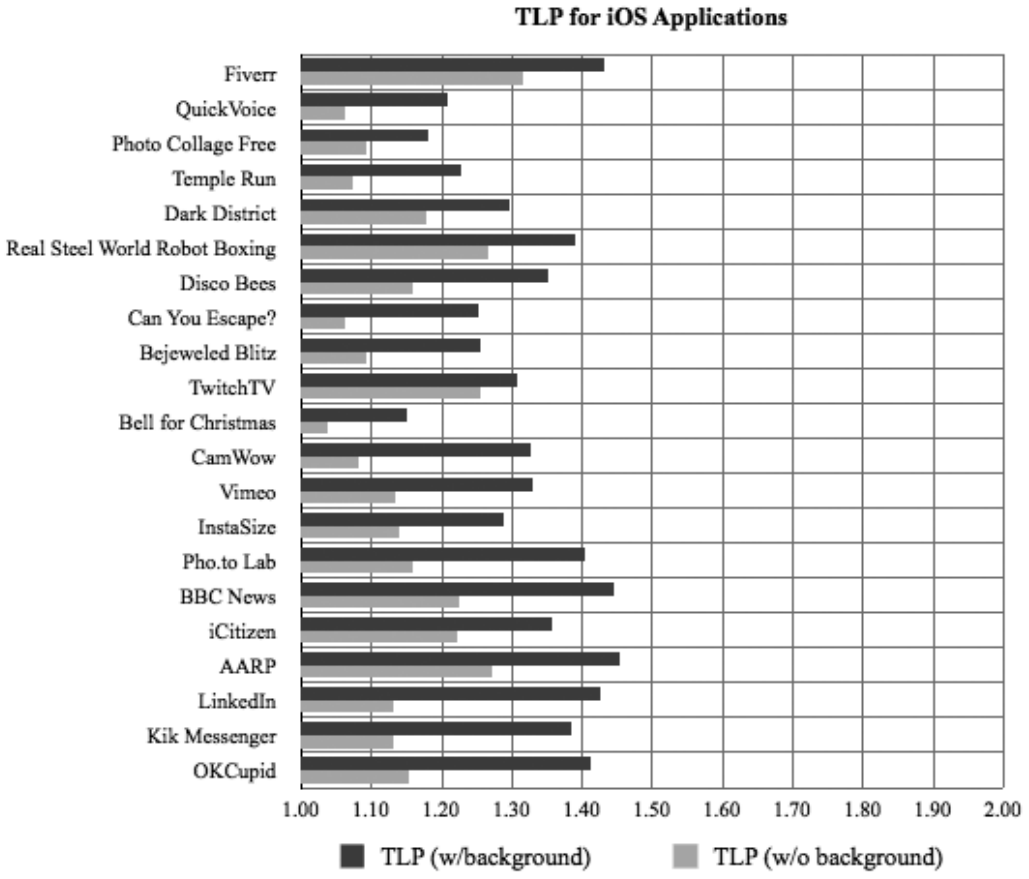
TLP for iOS Applications
Fiverr
QuickVoice
Photo Collage Free
Temple Run
Dark District
Real Steel World Robot Boxing
Disco Bees
Can You Escape?
Bejeweled Blitz
TwitchTV
Bell for Christmas
CamWow
Vimeo
InstaSize
Pho.to Lab
BBC News
iCitizen
AARP
LinkedIn
Kik Messenger
OKCupid
1.00
1.10
1.20
1.30
1.40
1.50
1.60
1.70
1.80
1.90
2.00
TLP (w/background)
TLP (w/o background)

Figure 3