BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100003

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Performance optimization opportunities in the Android software stack ☆

Varun Gohil [a,*,1], Nisarg Ujjainkar [b,1], Joycee Mekie [b], Manu Awasthi [a]

[a] Ashoka University, India
[b] Indian Institute of Technology Gandhinagar, India

ARTICLE INFO

Keywords:
CPU utilization
Android smartphone
Workload characterization

ABSTRACT

The smartphone hardware and software ecosystems have evolved very rapidly. Multiple innovations in the system software, including OS, languages, and runtimes have been made in the last decade. Although, performance characterization of microarchitecture has been done, there is little analysis available for application performance bottlenecks of the system software stack, especially for contemporary applications on mobile operating systems.

In this work, we perform system utilization analysis from a software perspective, thereby supplementing the hardware perspective offered by prior work. We focus our analysis on Android powered smartphones, running newer versions of Android. Using 11 representative apps and regions of interest within them, we carry out performance analysis of the entire Android software stack to identify system performance bottlenecks.

We observe that for the majority of apps, the most time-consuming system level thread is a frame rendering thread. However, more surprisingly, our results indicate that *all apps* spend a significant amount of time doing Inter Process Communication (IPC), hinting that the Android IPC stack is a ripe target for performance optimization via software development and a potential target for hardware acceleration.

1. Introduction

Smartphones have become an integral part of our daily lives. People depend on smartphones for many tasks related to business, finance, entertainment, and social interactions. Currently, there are more than 2 billion mobile devices in use worldwide [1]. The Ericsson Mobility Report 2019 states that there are 6.1 billion mobile broadband subscriptions globally and the number of Long-Term-Evolution (LTE) subscriptions have grown to 3.9 billion [2]. This widespread adoption of mobile devices can be largely attributed to increasing device affordability, which has been made possible due to numerous hardware and software innovations. This includes the open-source nature of the Android Operating System [1], which has allowed smartphone vendors to customize the software stack for their hardware. As a result, Android has quickly gained a majority market share for smartphones [3].

Smartphones are very interesting from a system design perspective since they need to provide a number of functionalities that require general purpose as well as special purpose compute. As a result, smartphone SoCs have evolved rapidly to become complex ecosystems incorporating many specialized IP blocks, including DSPs and GPUs in addition to general purpose CPUs [1]. The number and diversity of architectures of such units has also increased over time to accommodate the evolving needs of applications.

Many recent efforts have been made to understand the performance bottlenecks and utilization characteristics of smartphone devices [4–7]. However, most prior studies focus on bottom-up understanding of smartphone utilization from an architectural design perspective. For example, [5] present the distribution of computation amongst ARM's big and little cores. They also study clock frequencies at which one can perform computations on a mobile device in an energy efficient manner. These studies are important since mobile SoC architectures evolve rapidly and characterization of new architectures is important to understand and alleviate performance bottlenecks of new architectures.

The software stack for smartphones has been evolving even faster than hardware. Android has been following a yearly release cycle in recent years, with each iteration adding more functionality and optimizations [18]. As a result, every release causes major changes to the software stack which potentially lead to performance bottlenecks. Knowledge of these bottlenecks is not only useful for optimizing the next generation apps but also for making decisions about future architectural innovations. Despite its importance, there is a lack of understanding of software bottlenecks in both the apps as well as the system software. Understanding and enumerating performance bottlenecks of the software stack remains an important endeavor that has not been taken up in earnest by the systems research community. However,

☆ This work is supported through grants received from Huawei Technology India (DSA2020112121) and Ashoka University (R/IFR/CMS/MAW/18).
* Corresponding author.
E-mail address: varun.gohil@ashoka.edu.in (V. Gohil).
[1] Both authors contributed equally.

https://doi.org/10.1016/j.tbench.2021.100003
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 5 November 2021
2772-4859/© 2021 The Authors. Publishing services by Elsevier B.V. on behalf of KeAi Communications Co. Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Table 1
Applications traced and their Region of Interest.

Category	Application	Regions of Interest (ROI)
PDF Viewer	Adobe Acrobat [8]	Read PDF
Camera	Camera	Take a picture, Record a video
Game	Candy Crush [9]	Play one level of the game
Social Network	Facebook [10]	Scroll through the feed
Mailing app	Gmail [11]	Send mail
Virtual Assistant	Google Assistant [12]	Perform a query
Browsing app	Google Chrome [13]	Search, Scroll through a page
Location app	Google Maps [14]	Search a location, Zoom into a location
Audio Streaming	Spotify [15]	Play a song in background, Play a song in foreground
Messaging app	WhatsApp [16]	Send a message
Video Streaming	YouTube [17]	Play a video

Apart from the regions of interest mentioned above, we also trace the launch of each of the apps.

recent announcements from technology companies [19] indicate that there exists a large room for performance improvement in the Android software stack.

We believe that a top-down analysis of application characteristics will augment our understanding of mobile devices by supplementing prior work. Hence, we study the software subsystem of Android based smartphones by tracing the entire system (application + operating system) stack at runtime, capturing performance bottlenecks. Prior works [4,6,7] have measured CPU utilization using Thread Level Parallelism (TLP) as a metric to identify the amount of parallelism the hardware can exploit. While TLP is a useful metric to decide the number of cores to be placed on the chip, it does not provide information on the computation being performed by the cores and the functionality supported by the computations. Knowledge of the functionality for which the computation is being performed is necessary to optimize software and to design novel hardware accelerators to be used alongside the CPU. Generally, in Android smartphones, a particular thread or a group of threads is responsible for a particular functionality. By identifying the threads having high execution times, one can identify the functionality that consumes higher CPU time and should be optimized. Hence, we focus this paper on trying to answer the following questions.

- Which are the most time-consuming threads per app?
- Are there any common threads across a cross section of apps that end up consuming the most time?
- Which threads take up the most time during app launch?

We believe that this type of analysis will help the process of developing high performance software but and helps identify potential hardware acceleration opportunities for mobile devices. Since many previous studies have pointed out the importance of app launch times for user engagement and experience [20], we also pay special attention to app launches as a region of interest. Overall, the major contributions of this work are as follows:

- We identify and perform system-level tracing of eleven popular mobile applications on actual hardware, running Pie version of Android (Android 9), which helps us analyze time consumed by application and OS threads.
- To better represent performance information, we group threads into bins based on their functionalities. This helps us increase interpretablity of results and analyze the time consumed per functionality.
- We identify that for majority of applications, the most time consuming thread is a system-managed thread named `RenderThread` or another thread involved in frame rendering.
- Using thread bins, we identify that although the most time-consuming thread is almost always a thread related to frame rendering, a larger portion of execution time is consumed by the group of threads responsible for Inter Process Communication (IPC). This insight makes inter process communication a potential target for software optimization and hardware acceleration.

Table 2
Smartphone details.

Technical specifications	
Device model	Nokia 6.1 Plus
Operating System	Android Pie
Architecture	ARM 64-bit
CPU	Qualcomm Snapdragon 636
Cpu Cores	8
GPU	Adreno (TM) 509
RAM	6 GB
Resolution	1080 × 2280
Display PPI	431

2. Methodology

2.1. Applications traced

We choose eleven applications for our study, each of which represents a common use case of a smartphone. For example, we include Google Chrome [13] as a browsing app, Youtube [17] as video-streaming app, WhatsApp [16] as a messaging app, and Gmail [11] as a mailing app. Most of the selected apps come pre-installed in the majority of Android smartphones. We select the remaining apps based on their popularity which we measure using their position on Google Play [21] Store's Top Charts. The selected apps were at the top of the Top Charts when we performed our study .

Prior work [1] suggests that one should divide the applications into *regions-of-interest* (ROI) to gain deeper insight into the applications. A region-of-interest (ROI) is a smaller portion of the application's execution which performs a particular task. For example, Google Chrome has multiple regions-of-interest like performing a search, switching a tab, and scrolling. Each of these ROIs deals with a specific functionality of Google Chrome. The reason for dividing the applications into ROIs is that these individual ROIs can directly influence user-experience and studying them independently of each other reduces the complexity of analysis that needs to be performed. We provide a comprehensive list of all applications we trace and their ROIs in Table 1. Apart from the ROIs mentioned in Table 1, we also trace the app launches for all apps.

2.2. Tools and setup

For system-level (app + operating system) tracing, we use the Systrace [22] tool. Systrace is a tool shipped with Android Studio and is primarily used for analyzing the performance of an Android device. It is a wrapper around Atrace [23] and Ftrace [24]. Atrace performs user space tracing while ftrace traces the Linux kernel. The traces capture not only the threads spawned by the app, but also background threads being executed by the Android operating system. From the traces obtained using Systrace, we find the time for which each thread executes on the processor core.

To trace the ROIs, we start Systrace tracing and perform the task related to the ROI. We immediately stop Systrace tracing when the task

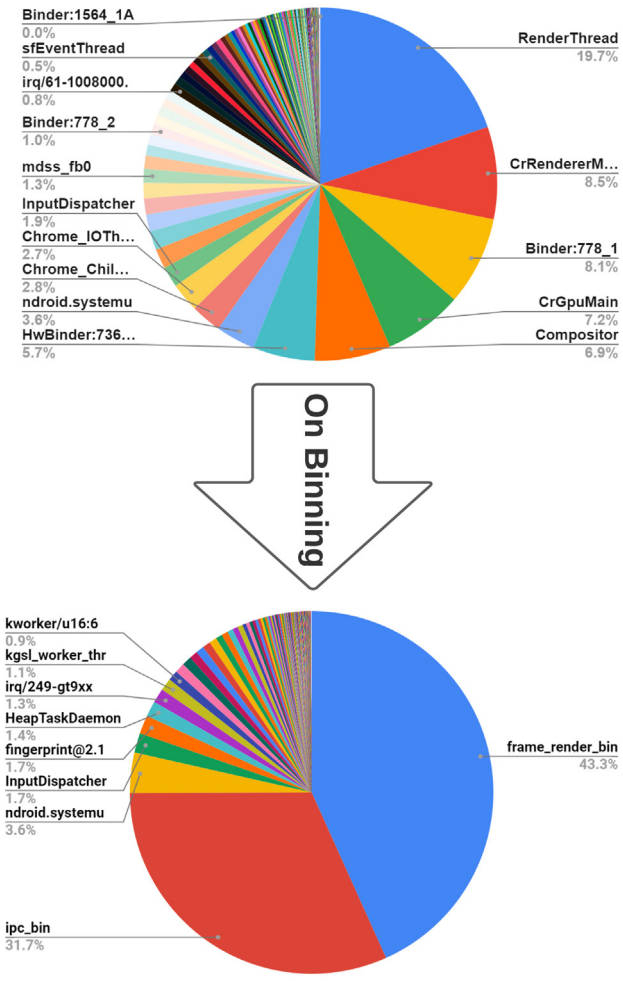

Fig. 1. Effect of Binning. Results for Google Chrome's scrolling ROI.

of the ROI ends. We perform tracing for each region of interest of each app at least five times.

We perform our experiments on Nokia 6.1 Plus [25] smartphone. It runs the stock Android Pie (Android 9) operating system. Further details about the smartphone are presented in Table 2.

2.3. Binning threads

The Android operating system and the apps spawn a large number of threads. Since Systrace performs system level tracing the generated traces have information for a large number of threads. This leads to the resulting plot being cluttered and difficult to interpret. Hence, to reduce clutter and improve interpretability, we group threads working for a common functionality into a single bin. We identify two major bins which aid our analysis. They are:

- Frame Rendering Bin (FR Bin)
- Inter-Process Communication Bin (IPC Bin)

Fig. 1 shows the effect of thread binning. The pie chart on the top in Fig. 1 shows the execution time distribution across individual threads for Google Chrome's scrolling ROI. After thread binning, the pie chart on top is transformed to the one on the bottom. The latter shows execution time distribution among selected bins and the remaining threads. Observing the bottom pie-chart, we can easily infer that the major portion of execution time is spent on frame rendering. We were able to create two classes of thread bins based on the functionality of individual threads. While binning threads, we ensured that threads were mapped to correct bins and that no thread was mapped to more than one bin.

Table 3
List of threads within bins.

Frame rendering bin
RenderThread
surfaceflinger
UiThread
Compositor
CrGpuMain
CrRendererMain
android.display
mdss_fb0
DispSync
android.anim
Above list is not exhaustive
Inter Process Communication Bin
Binder
HwBinder
Chrome_IOThread
Chrome_ChildIOT

Frame Rendering Bin : The Frame Rendering (FR) bin is a group of all threads which are responsible for rendering a frame on the mobile device's screen. Table 3 provides a list of threads within this bin. The major threads within this bin are `RenderThread`, `SurfaceFlinger` and `UiThread`.

Inter Process Communication Bin : The Inter Process Communication (IPC) bin is a group of all threads that are executed to share information between processes. Table 3 provides a comprehensive list of all threads within this bin. The major threads within the IPC bin are `Binder` and `HwBinder`.

3. Results and observations

In this section, we discuss the answers to the questions that we initially set out to answer in Section 1.

3.1. What are the most time-consuming threads/bins per app?

Table 4 shows the most time consuming threads for each region of interest for all eleven applications. We observe that for most ROIs across applications, `RenderThread` is the most time-consuming thread. `RenderThread` is a system-managed thread that is primarily responsible for offloading rendering work to GPU to reduce the burden on `UiThread` [26]. By doing so it ensures the animations are smooth even when the `UiThread` is delayed, which is essential to maintain Quality-of-Service (QoS) for the user [26]. `RenderThread` is the most time-consuming thread in ROIs like scrolling in Facebook and Chrome, messaging using WhatsApp and Gmail, recording a video, or playing a song in foreground on Spotify. All these ROIs involve frequent modifications to the user display which justify most time being consumed by `RenderThread`.

For the game Candy Crush, `GLThread` is the most time-consuming thread. `GLThread` is also a rendering thread and is responsible for performing OpenGL graphics rendering operations [27]. Similarly, for Google Maps' "Zoom into a location" ROI, `GLViewThreadImp` is the most time consuming thread. `GLViewThreadImp` is responsible for managing Views, which are basic building blocks of user-interface components, of the OpenGL graphics library [28,29]. For Google Chrome Search ROI, we observe that `CrRendererMain` is the most time-consuming thread. `CrRendererMain` is the renderer thread for a webpage. As per Chromium's documentation, `CrRendererMain` runs the javascript, html and css code which is displayed on the screen [30].

Table 4
Most time-consuming thread and bin per ROI. Numbers within parenthesis indicate percentage execution time.

Application	Region of Interest	Most time consuming thread	Most time consuming bin
Adobe	Read PDF	om.adobe.reade (13.6%)	FR (26.7%)
Camera	Take a picture	PostProcessingImag (14.5%)	IPC (30.3%)
Camera	Record Video	RenderThread (11.0%)	IPC (22.1%)
Candy Crush	Play 1 level	GLThread (45.2%)	FR (54.2%)
Facebook	Scroll	RenderThread (17.2%)	IPC (23.1%)
Gmail	Send Mail	RenderThread (17.3%)	IPC (29.8%)
Google Assistant	Query	RenderThread (11.2%)	IPC (25.4%)
Google Chrome	Scroll	RenderThread (19.4%)	FR (43.4%)
Google Chrome	Search	CrRendererMain (13.4%)	IPC (33.8%)
Google Maps	Search Location	Jit thread pool (11.6%)	IPC (26.4%)
Google Maps	Zoom into Location	GlViewThreadImp (17.1%)	FR (26.6%)
Spotify	Play Music in Background	AndroidOut_1D (6.2%)	IPC (20.6%)
Spotify	Play Music in Foreground	RenderThread (27.4%)	FR (39.2%)
Whatsapp	Send Message	RenderThread (19.4%)	IPC (35.8%)
YouTube	Play Video	ExoPlayerImplIn (8.8%)	IPC (38.6%)

Overall, the most time-consuming threads for these ROIs are involved in rendering the frame on user display.

For YouTube's "Play a Video" ROI we observe that `ExoPlayerImplIn` is the most time-consuming thread. This thread runs ExoPlayer that is an alternative media player for Android [31,32]. For Camera's "Take a Picture" ROI, `PostProcessingImag` thread is the highest time-consumer. From the thread's name, we hypothesize that this thread might be involved in an image's post-processing which involves tasks like setting the exposure, white balance, and applying selected filters. Unfortunately, we do not find any documentation on `om.adobe.reade` and `AndroidOut_1D` threads and hence cannot comment on their functionality.

Overall, for 7 out of 15 ROIs concerning the eleven applications involved in our study, `RenderThread` is the most time-consuming thread. Further, the highest time-consuming threads in 10 of 15 ROIs are working on the appropriate rendering of the frame. One should also note that the execution time of `RenderThread` is not contiguous. Execution times of multiple instances of `RenderThread` are added together to obtain the total execution time. We find that each individual instance of `RenderThread` is short-lived, on average it takes 0.73 ms to execute, and there exists thousands (1000–2000) of such instances within each region of interest.

The above results may lead one to conclude that frame rendering is the major time consumer for the applications since the most time consuming thread for majority of applications is related to frame-rendering. However, we find that this is not the case when we analyze the results for thread bins. Table 4 shows that the most time consuming bin is the Inter Process Communication (IPC) bin. The IPC bin is the highest time consumer for 10 out of 15 ROIs across the applications. This indicates that even though the major time-consuming thread is related to frame rendering, as a whole, threads used to communicate between processes are the larger time-consumer than threads involved in frame rendering. This observation indicates that inter process communication might be a bigger bottleneck for mobile applications than frame rendering.

3.2. What are the time-consuming threads which are common across apps?

We isolate the common time-consuming threads across applications. We believe optimizing these threads would result in higher performance benefits across applications. We observe the following time-consuming threads to be common across apps:

`RenderThread`: It is the most time consuming thread for 7 out of 15 ROIs under consideration and it is one of the top three most time consuming threads for 11 out of 15 ROIs. It offloads the rendering tasks to GPU from the `UiThread`, to maintain the smoothness of animations by avoiding frame drops [26].

`surfaceflinger`: It is the dominant time-consuming thread after `RenderThread` within the Frame Rendering bin. It is one of the top three most time consuming threads for 5 out of the 15 ROIs. The `surfaceflinger` thread takes in multiple items from various graphics buffers and composes them into a single buffer which is then sent to the user display [33].

`Binder`: The `Binder` threads are a major time consumer for the Inter Process Communication bin. They are used for communication within application processes and within framework and application processes [34]. The framework processes are managed by the Android framework and are device-independent.

`HwBinder`: Similar to `Binder` threads, `HwBinder` threads are also a major time consumer for the Inter Process Communication bin. They are used for communication between framework and vendor processes [34]. The vendor processes are processes spawned by the code that the vendors add to Android framework and are generally device-dependent.

Table 5
Most time-consuming thread and bins on app launch. Numbers within parenthesis indicate the percentage of execution time occupied by the thread/bin.

Application	Most time consuming thread	Most time consuming bin
Adobe	om.adobe.reade (12.6%)	FR (23.5%)
Camera	RenderThread (14.9%)	IPC (35.2%)
Candy Crush	GLThread (44.6%)	FR (57.9%)
Facebook	Jit thread pool (11.6%)	IPC (12.6%)
Gmail	Jit thread pool (10.2%)	IPC (32.1%)
Google Assistant	RenderThread (11.6%)	IPC (39.7%)
Google Chrome	RenderThread (11.6%)	IPC (36.6%)
Google Maps	Jit thread pool (14.0%)	IPC (23.2%)
Spotify	m.spotify.musi (13.9%)	FR (18.0%)
Whatsapp	RenderThread (19.4%)	IPC (36.0%)
YouTube	RenderThread (12.5%)	IPC (25.1%)

3.3. What are the time-consuming threads during an app launch?

App launches are crucial regions of interest in the context of smartphones. One might think that reducing app launch time results in fewer benefits than reducing the app's running time. Although this statement is true and intuitive, app launches are important because of the usage pattern of smartphones. Many users have a large number of short-lived sessions on their smartphones. These short sessions last for less than 10 seconds [20]. During these short sessions, a long app launch time significantly degrades user experience, which is the reason why several efforts have been made to optimize app launch time. For example, Android preserves an apps memory even after it is closed, so the time taken by an app launch in the future can be reduced [35].

We trace the app launches of each of the apps listed in Table 1. Table 5 shows the most time-consuming thread and bin during the launch of the applications. We observe that `RenderThread` consumes a large percentage of execution time for the majority of the

applications. During an app launch, `RenderThread` is the most time-consuming thread for 5 out of the 11 apps, while it is in the top 2 most time-consuming threads for 9 out of the 11 apps. This is expected since when a new application is launched, new views corresponding to the launched application need to be rendered on the screen.

Similar to other ROIs, the Inter Process Communication bin is the highest time consumer during an app launch. This indicates that optimizing Inter process communication would also optimize app launches which would directly improve Quality of Service (QoS).

4. Related work

Several prior publications have focused on evaluating performance and energy of smartphones by characterizing the hardware. For example, Gao et al. [6,7] demonstrated that mobile applications had low Thread-Level Parallelism (TLP) leading to under utilization of allocated cores. A recent work by [5] studied the core utilization in smartphone architectures which have both big and little cores. They report that standalone applications rarely utilize all big cores during execution, however during application launches or updates all big cores are utilized to meet latency targets and avoid degradation in user experience. Most of these works primarily try to answer the question, *"For what percentage of execution time is the core being utilized?"*. While answering the above question is crucial to identify performance inefficiencies, it does not provide insights into the system software stack that may help alleviate these bottlenecks. Our work supplements the prior work by identifying the functionalities (IPC and RenderThread) which have the highest execution time, which on optimization would lead to significant performance benefits.

There have been some research that takes a software-first approach for performance analysis of smartphone applications. [36] use static code analysis to identify frequently occurring performance bug patterns in applications. Further, [37] develop a tool that can automatically detect performance bottlenecks on Android smartphones. However given the nature of the Android ecosystem and the frequent major release cycles require constant performance bottleneck analysis of the system software stack as well. Our work complements such works which perform a software-focused performance analysis. Instead of using any form of static analysis, we identify the time consuming threads of smartphone applications by actually running the applications on a real-world smartphone and provide targets for performance optimization.

5. Limitations and future work

Our current study is limited to Android Version 9. Because of the quick moving nature of the Android ecosystem, owing to yearly release cycles, new versions of Android had been released while we were undertaking this study.

In addition, there is a lack of performance analysis tools for the Android ecosystem, unlike x86/x64, where a large number of open source, well maintained performance analysis tools exist, this is not the case for Android on ARM. Lack of performance analysis tools severely hampers the types of analyses that can be carried out. The analysis done in this paper was carried out using Systrace [22], which is supported for Android version 9. However, more recent Android versions provide a tool called Perfetto [38] for system-level tracing. Further, Perfetto on Android 9 requires the system tracing service to be turned on, which was not possible due to the fact that we performed our experiments on stock android [39]. These factors compelled us to limit out study to Android 9. However, we believe a study similar to this work across Android versions could potentially reveal important performance optimization trends. We also believe future work would be a more comprehensive study by using more smartphone models and different Android versions on each model.

The scope of this work is limited to answering the question *"Which functionality or subsystem of the Android system stack takes up highest portion of execution time?"*. Although extremely important, this work does not reveal what part within the subsystem needs to be optimized and what kind of optimizations would be beneficial. For example, our work indicates that the IPC bin consumes higher portion of execution time but it does not point out which exact components of the IPC subsystem should be optimized to reduce this time. As we have alluded to before, this is primarily due to the of lack of tools which can be used for such analysis. Tools like *Systrace* do not provide such information.

The presented analysis is limited to an Android smartphone. We could not perform similar analysis on smartphones with other operating systems because there do not exist any open-source tools that may act as alternatives/equivalents of *Systrace* for those operating systems.

Finally, the work focuses on Regions of Interest (ROIs) for analyzing the execution time breakdown. The authors have tried to select the most relevant ROIs for each application, which is similar to studies done in the past, which are based on the most common user behavioral patterns, and whose performance determined user engagement [1,20]. However, we acknowledge that the set of ROIs for each application is not necessarily the most representative nor is it necessarily exhaustive. Future work will focus on identifying a much more representative and exhaustive set of regions-of-interest for the application.

6. Conclusion

In this work, we performed a system level performance bottlenecks analysis for an Android smartphone for eleven popular applications. Our results demonstrate that for *all* applications, the highest time consuming thread is either `RenderThread` or another thread related to frame rendering. Further, on grouping threads into bins based on their functionality, we find that the highest time consuming functionality is Inter Process Communication. We find similar distribution in time consumption for both app executions and app launches. Our results identify that software optimization and hardware acceleration should target Inter Process Communication to maximize performance and improve user experience.

References

[1] V.J. Reddi, H. Yoon, A. Knies, Two billion devices and counting, IEEE Micro 38 (1) (2018) 6–21.
[2] Ericsson mobility report Q2 update August 2019, pp. 4, URL: https://www.ericsson.com/4912aa/assets/local/mobility-report/documents/2019/ericsson-mobility-report-q2-2019-update.pdf.
[3] IDC - Smartphone Market Share - OS, IDC: The Premier Global Market Intelligence Company (2021) URL: https://www.idc.com/promo/smartphone-market-share.
[4] M. Halpern, Y. Zhu, V.J. Reddi, Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 64–76, http://dx.doi.org/10.1109/HPCA.2016.7446054.
[5] J. Whitehouse, Q. Wu, S. Song, E. John, A. Gerstlauer, L.K. John, A study of core utilization and residency in heterogeneous smart phone architectures, in: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering ICPE '19, 2019.
[6] C. Gao, A. Gutierrez, R.G. Dreslinski, T. Mudge, K. Flautner, G. Blake, A study of thread level parallelism on mobile devices, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2014, pp. 126–127, http://dx.doi.org/10.1109/ISPASS.2014.6844468.
[7] C. Gao, A. Gutierrez, M. Rajan, R.G. Dreslinski, T. Mudge, C. Wu, A study of mobile device utilization, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2015, pp. 225–234, http://dx.doi.org/10.1109/ISPASS.2015.7095808.
[8] Adobe acrobat reader: PDF viewer, editor & creator - Apps on google play, URL: https://play.google.com/store/apps/details?id=com.adobe.reader&hl=en.
[9] Candy crush saga – Apps on google play, URL: https://play.google.com/store/apps/details?id=com.king.candycrushsaga&hl=en{_}IN.
[10] Facebook, (2021) URL: https://www.facebook.com/.
[11] Gmail, URL: https://www.google.com/gmail/.
[12] Google Assistant | Your own personal google, URL: https://assistant.google.com/intl/en{_}in/.
[13] Google chrome - The new chrome & most secure web browser, URL: https://www.google.com/chrome/.

[14] Google Maps, (2021) URL: https://www.google.com/maps.
[15] Music for everyone - Spotify, URL: https://www.spotify.com/in/.
[16] WhatsApp, , URL: https://www.whatsapp.com/.
[17] YouTube, URL: https://www.youtube.com/.
[18] Can android "O" de-fragment android ?, URL: https://www.counterpointresearch.com/can-android-o-de-fragment-android/.
[19] D. Burke, What's new in Android 12 Beta, 2021 URL: https://android-developers.googleblog.com/2021/05/whats-new-in-android-12-beta.html.
[20] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, D. Estrin, 2021.
[21] Google play, URL: https://play.google.com/store.
[22] Overview of system tracing, Android Developers, (2021) URL: https://developer.android.com/studio/profile/systrace.
[23] Atrace/atrace.c - platform/system/extras - git at google, URL: https://android.googlesource.com/platform/system/extras/+/jb-mr1-dev-plus-aosp/atrace/atrace.c.
[24] FTrace, URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.
[25] Nokia 6.1 plus, URL: https://www.nokia.com/phones/e_in/nokia-6-plus.
[26] E. Marletti, Understanding the RenderThread, Medium (2017) URL: https://medium.com/@workingkills/understanding-the-renderthread-4dc17bcaf979.
[27] GLSurfaceView, Android Developers, (2021) URL: https://developer.android.com/reference/android/opengl/GLSurfaceView.
[28] GLView | Tizen Docs, URL: https://docs.tizen.org/application/native/guides/ui/efl/mobile/component-glview/.
[29] View, Android Developers (2021) URL: https://developer.android.com/reference/android/view/View.
[30] Understanding about:tracing results - The chromium projects, URL: https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/trace-event-reading.
[31] ExoPlayer, Android Developers, (2021) URL: https://developer.android.com/guide/topics/media/exoplayer.
[32] ExoPlayer, URL: https://exoplayer.dev/.
[33] SurfaceFlinger and WindowManager, Android Open Source Project (2021) URL: https://source.android.com/devices/graphics/surfaceflinger-windowmanager.
[34] Using binder IPC | Android open source project, URL: https://source.android.com/devices/architecture/hidl/binder-ipc.
[35] Manage your app's memory | Android developers, URL: https://developer.android.com/topic/performance/memory.
[36] Y. Liu, C. Xu, S.C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 1013–1024.
[37] Y. Gao, W. Dong, H. Huang, J. Bu, C. Chen, M. Xia, X. Liu, Whom to blame? Automatic diagnosis of performance bottlenecks on smartphones, IEEE Trans. Mob. Comput. 16 (2017) 1773–1785.
[38] Perfetto : System profiling, app tracing and trace analysis, URL: https://perfetto.dev/.
[39] Quickstart: Record traces on android, URL: https://perfetto.dev/docs/quickstart/android-tracing.