ShuffleDog: Characterizing and Adapting User-Perceived Latency of Android Apps

Gang Huang *Member, IEEE*, Mengwei Xu, Felix Xiaozhu Lin, Yunxin Liu *Senior Member, IEEE*,
Yun Ma *Student Member, IEEE*, Saumay Pushp, Xuanzhe Liu *Member, IEEE*,

Abstract—Numerous complains have been made by Android users who severely suffer from the sluggish response when interacting with their devices. However, very few studies have been conducted to understand the *user-perceived latency* or mitigate the UI-lagging problem. In this paper, we conduct the first systematic measurement study to quantify the user-perceived latency using typical interaction-intensive Android apps in running with and without background workloads. We reveal the insufficiency of Android system in ensuring the performance of foreground apps and therefore design a new system to address the insufficiency accordingly. We develop a lightweight tracker to accurately identify all *delay-critical* threads that contribute to the slow response of user interactions. We then build a resource manager that can efficiently schedule various system resources including CPU, I/O, and GPU, for optimizing the performance of these threads. We implement the proposed system on commercial smartphones and conduct comprehensive experiments to evaluate our implementation. Evaluation results show that our system is able to significantly reduce the user-perceived latency of foreground apps in running with aggressive background workloads, up to 10x, while incurring negligible system overhead of less than 3.1% CPU and 7 MB memory.

Index Terms—Measurements; Systems; Cross-Layer Design; Scheduling

1 Introduction

Fast, responsive, and smooth interaction on mobile apps has become a key requirement of mobile users' experience. In order to enable various UI metaphors, apps often adopt continuous user interactions. Examples include touch gestures such as swipe and pinch, and finger painting on a touch screen. To continuous user inputs, apps respond with continuous outputs, e.g., animated screen updates.

Unfortunately, today's mobile users are often frustrated by slow system responses and "janky" outputs when interacting with their devices. A recent study on smartphone performance anomalies [1] reveals a high percentage (76%) of UI lagging issues. Looking closer, user experience in interaction is decided by *user-perceived latency*, i.e., the system latency perceived by users in how a user's stimuli to a mobile system is propagated, processed, and finally presented as the system's response to the user. To the best of our knowledge, despite the significance of user-perceived latency in ensuring user experience, very few studies have been conducted to quantify the user-perceived latency of various mobile apps, identify the root causes of UI-lagging, and improve the existing mobile systems to reduce user-perceived latency.

In this paper, we first conduct a quantitative study to measure the imperfect user-perceived latency of various typical interactive Android apps such as games. We examine the performance difference of these apps in running with and without background workloads. We find that all the apps significantly suffer from background workloads, with a up to 12x increase of user-perceived latency against the normal latency without workloads. The reasons are because: 1) multiple threads both in user mode and kernel mode contribute to the user-perceived latency, and 2) those threads compete with other threads on various system resources, but 3) the Android OS currently cannot adequately ensure the performance of those threads in managing the system resources including CPU, I/O, and GPU.

Motivated by these findings, we propose and design a new system, as called *ShuffleDog*[1] , to reduce user-perceived latency of Android apps. It consists of two key components. The first one takes charge of precisely identifying all the *delay-critical* threads which contribute to the user-perceived latency. These *delay-critical* threads include not only the UI thread of an app, but also other necessary threads of the app as well as the background services which process user interactions and can affect screen updates accordingly. Such an identification task is non-trivial as the threads are found to interact indirectly in various ways. We develop an API-instrument-based approach to identify proper threads accurately at a low cost. That approach requires no modification from apps and thus is able to support all existing apps. The second key component of *ShuffleDog* is managing system resources to provide better services to all the identified threads. We apply a higher priority to those threads in allocating and scheduling various resources. Specifically, we improve the existing CPU, I/O, and GPU schedulers of Android OS to ensure the performance of these "need-to-be-prior" threads. We design our new schedulers carefully so that we are able to reduce the user-perceived latency without compromising the overall system performance.

We implement *ShuffleDog* on commercial Android smartphones and conduct comprehensive evaluations. Experi-

- *Gang Huang, Mengwei Xu, Yun Ma, and Xuanzhe Liu are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871. Email: {hg, xumengwei, mayun, xzl}@pku.edu.cn.*
- *Felix Xiaozhu Lin is with Purdue University. Email: xzl@purdue.edu.*
- *Yunxin Liu is with Microsoft Research. Email: yunxin.liu@microsoft.com.*
- *Saumay Pushp is with Korea Advanced Institute of Science and Technology. Email: saumay@nclab.kaist.ac.kr.*

[1] To request the source code of *ShuffleDog*, please contact the corresponding author Xuanzhe Liu via xzl@pku.edu.cn.

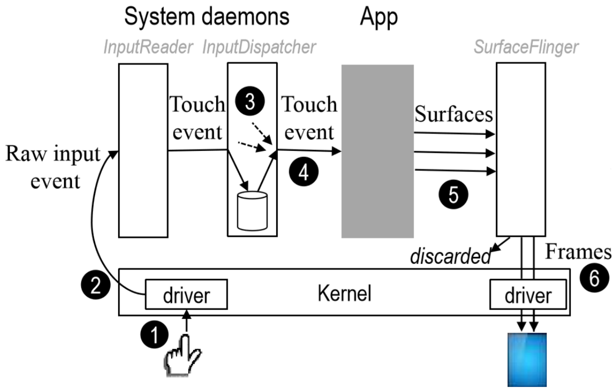


Fig. 1: Processing of user inputs in the Android OS.

mental results show that *ShuffleDog* is able to significantly reduce the user-perceived latency of various apps, up to 10 times, even under very aggressive and heavy background workloads. The impact on overall system performance is quite minor, as no more than 7% decrease on the performance of background workloads. *ShuffleDog* is able to accurately identify necessary threads without any false positives and has only one type of false negatives. The system overhead of our implementation is small, with less than 3.1% CPU usage and 7MB memory usage.

This paper makes the following main contributions:

- The first measurement study to quantify user-perceived latency and identify the root causes of imperfect user-perceived latency with background workloads. (Section 2)
- A thread tracker that is able to locate all relevant threads that contribute to user-perceived latency accurately with very low system overhead. (Section 4)
- A system resource manager which ensures the performance of the necessary threads through better CPU, I/O, and GPU scheduling. (Section 5)
- A prototype implementation on commercial smartphones (Section 6) and comprehensive evaluations to demonstrate the effectiveness of our approach (Section 7).

The rest of the paper is organized as follows. We describe the architecture of *ShuffleDog* in Section 3. We discuss the limitations in Section 8. We survey the related work in Section 9 and conclude in Section 10.

2 Quantifying User-Perceived Latency

In this section, we first describe how user-perceived latency is determined by the user-input processing in Android OS. Then we present a quantitative study to measure the user-perceived latency of typical interactive apps in running with and without background workloads, respectively. We demonstrate that Android fails to guarantee a low user-perceived latency and we discuss the root causes.

2.1 User-perceived Latency in Android System

Figure 1 describes a general work flow on how a user interaction is processed in Android OS. A touch event from a touch stimulus is generated by the touch-screen hardware as the user's finger moves on the touch screen. Such a touch event, in the form of a hardware event, is subsequently read by a kernel driver by means of interrupts(❶). The kernel formats the raw event and then sends the event to the user space through the device file interface(❷). In the user space, the *Input Reader* takes charge of pre-processing the input event (during which the event can get deferred or batched with others(❸)) and *Input Dispatcher* determines the destination app to which the event is sent in the form of an OS touch event through Inter-Process Communication (**IPC**)(❹) [2]. The app gets the touch input (possibly together with other inputs), processes the input, and generates one or more graphic surfaces(❺). The app sends the graphic surface(s) to *SurfaceFlinger*[3] that is responsible for surface composition, who ultimately submits the composited frame(s)(❻) to the display hardware for presentation.

From the preceding work flow, it can be observed that user-perceived latency is determined by multiple complicated factors. It consists of multiple stages and involves multiple threads in both kernel and user spaces. Furthermore, the app is treated as a black box in Figure 1. However, the app itself may have a complex structure and handles the interaction events in its own way. In addition, the app may also need support from system services through asynchronous IPCs, e.g., Location Service, to update the screen UI, which further complicates the user-interaction processing.

Not surprisingly, background threads can compete with the foreground apps for various system resources and thus affect the foreground apps and leads to a longer user-perceived latency. To quantify such impacts, we conduct a measurement study as follows.

2.2 Measurement Setup

We measure the user-perceived latency of 10 typical apps as listed in Table 1, including popular apps (*Google Map, WeChat*) with millions of Google Play downloads and famous open source apps (*OpenFlappyBird, MuPDF*). The former indicates the problem identified in this section is universal, and the latter enables us to look deeper into the root causes of this problem. Also, these apps are representative in terms of diverse system-resource usage. For example, *MuPDF* is CPU-intensive in computing the pixels, *RAR* is I/O-intensive in accessing disk to compress files, and games are GPU-intensive for graphic rendering. For each of them, we select a typical interactive behavior, and use a proper way to measure the user-perceived latency. For example, we can simply extract activity start latency from ActivityManager[4] for Google Map and Gmail. For the open-source *OpenFlappyBird* (it is based on *AndEngine*[5]) and *MuPDF*, we instrument these apps to trace the key timestamps of each user action and calculate the user-perceived latency. For games like *Furious Racing*, we use the Adreno Profiler [2] to get the drawing time of each frame.

We measure the performance of the apps in running with and without background workloads, respectively. Table 2

[2] Input Reader and Input Dispatcher are system daemons in Android OS and responsible for reading, filtering, pre-processing user events and dispatching them from kernel to apps.

[3] SurfaceFlinger is a vital system service in Android system, responsible for composing all the graphic layers to screen.

[4] A framework module who interacts with the overall activities running in the system.

[5] A popular open-source game engine in Android platform.

App	App Description	Selected Performance Metric
Google Map	Map	Time to start the application
Messenger	Pre-installed text message app	Time to start the application and list the recent text messages
Gmail	Mail client	Time to start the application and list recent incoming emails
SystemUI	System utility	Time to display recent apps by clicking the "Recents" button
WeChat	Instant message app	Time to log in and display the messages page
RAR	File Compressor	Time to compress an image file (10MB) in SD card
OpenFlappyBird	2D Game	Update interval between two frames inside AndEngine
Furious Racing	3D Game	Frame time (the reciprocal of frame rate) of screen
MuPDF	PDF Viewer	Latency of zooming in/out actions, from when user releases fingers to when the resized page is clearly displayed on screen
ImageViewFlipper	Image gallery	Time to display 15 image thumbnails from SD card.

TABLE 1: A list of foreground interactive applications used in our measurement study.

Workload	Workload Description
CPUGenerator	Float computations in 5 threads
IOGenerator	Using fio to randomly read a 128MB file from SD card, job number=2
GPUGenerator	Drawing triangles via OpenGL ES in a single thread, but not displayed on screen
CPUFilter	Keep doing image processing in 5 threads (filter = intrinsic blend, image size = 1MB)
GPUFilter	Keep doing image processing in 5 threads (filter = levels vec3 relaxed, image size = 1MB)
AppInstall	Installing an apk file (20MB) in SD card

TABLE 2: A list of background workloads used in our measurement study.

shows the background workloads involved in the study. Some of these workloads are implemented by us (i.e., CPUGenerator, IOGenerator, and GPUGenerator), and others are real-world workloads (i.e., CPUFilter, GPUFilter, and AppInstall). CPUFilter and GPUFilter are both based on a test app inside Android framework [3], which is implemented in Renderscript. We have made some modifications so that they can run as a background service. For IOGenerator, we leverage the Android-version of fio [4], which was ported from Linux by D.Nguyen et al [5].

Our experiments are conducted on a Nexus 6 smartphone[6] running Android 5.0.1 (Lollipop). For each app, we first run it without any background workload and collect the performance data as baseline. Then we run the same app again with a background workload to compare the difference. We repeat each experiment for 20 times and present the median numbers. During the experiments, we turn off unnecessary hardware components such as Wi-Fi, cellular, Bluetooth, and sensors, and stop unnecessary apps and background services. To make the I/O requests actually access the disk, we also clear the file-system cache before each testing.

Fore App + Back Workloads	Latency	Ratio
Google Map (Baseline)	1,266ms	1.0x
Google Map + CPUGenerator	4,088ms	3.2x
Google Map + CPUFilter	4,824ms	3.8x
Messenger (Baseline)	1,673ms	1.0x
Messenger + CPUGenerator	2,368ms	1.4x
Messenger + CPUFilter	2,631ms	1.6x
Gmail (Baseline)	2,217ms	1.0x
Gmail + CPUGenerator	4,525ms	2.0x
Gmail + CPUFilter	4,933ms	2.2x
MuPDF (Baseline)	696ms	1.0x
MuPDF + CPUGenerator	7,325ms	10.5x
MuPDF + CPUFilter	8,733ms	12.5x
WeChat (Baseline)	3.8s	1.0x
WeChat + IOGenerator	7.9s	2.1x
WeChat + AppInstall	7.1s	1.9x
RAR (Baseline)	3.9s	1.0x
RAR + IOGenerator	5.9s	1.5x
RAR + AppInstall	5.1s	1.3x
ImageViewFlipper (Baseline)	878ms	1.0x
ImageViewFlipper + IOGenerator	1,365ms	1.6x
ImageViewFlipper + AppInstall	1,252ms	1.4x
SystemUI (Baseline)	516ms	1.0x
SystemUI + GPUGenerator	2,218ms	4.3x
SystemUI + GPUFilter	1,346ms	2.6x
Furious Racing (Baseline)	16.7ms	1.0x
Furious Racing + GPUGenerator	35.2ms	2.1x
Furious Racing + GPUFilter	27.4ms	1.6x
OpenFlappyBird (Baseline)	16.7ms	1.0x
OpenFlappyBird + GPUGenerator	74.6ms	4.5x
OpenFlappyBird + GPUFilter	32.7ms	2.0x

TABLE 3: Experiment results. Here "Ratio" means how much latency is lengthened by background workloads compared to the baseline without any background workloads.

2.3 Measurement Results

Table 3 shows the experiment results. We can observe that the user-perceived latency of the apps is significantly affected by the background workloads, with an increased latency of 1.3x - 12.5x. Taking *MuPDF* as an example. When users try to zoom in a page, it takes less than 1s to display the resized page in the baseline case. However, adding an image processing workload (CPUFilter) in background, the latency increases sharply beyond 8s, which is 12 times compared to the baseline. For file compression in *RAR*, the baseline latency is 3.9s without any background workload. Running IOGenerator in background adds about 2.0s delay, which can be obviously perceived by users. Similarly, if an app is installed to SD card at the same time, the latency goes to 5.1s, taking 1.2s more time compared to the baseline. Such a delay is undesirable as users usually continue to use their phones when installing an app in background. Another case is checking recent apps by clicking the "Recents" button, a physical or virtual button in most Android devices. Nowadays, it is very common that users can launch multiple apps at the same time and frequently switch among these apps. Usually, this action can be completed in around 0.5s. However, the background GPUFilter makes the delay to 1s, and our GPUGenerator even makes it to 2s, 3 times longer

[6] We have also carried out our measurement experiments on the Nexus 5, and the results are quite similar. Due to the space limit, we only present the results of the Nexus 6 that was a relatively newer and more powerful smartphone when this paper was written.

Price($)	Often	Sometimes	Never	# of Users
<300	33.6%	62.0%	4.4%	113
300 - 750	19.0%	71.2%	9.8%	163
>750	14.3%	50.0%	35.7%	42
Overall	23.6%	65.1%	11.3%	318

TABLE 4: Results of our user survey. The survey question: "How often do you encounter UI-lagging issue?".

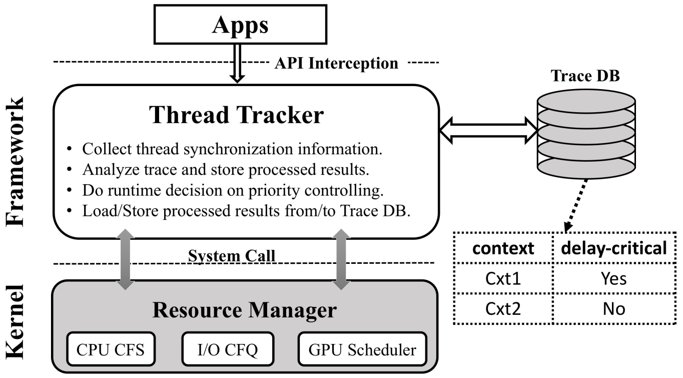
compared to the baseline. Such an increased latency can cause severe frustration to users, especially when they are frequently switching among multiple apps. The similar observation also exists for game playing. Without background GPU workloads, *Furious Racing* runs at 60fps (frames per second), i.e., it takes about 16.7ms to draw one frame. However, GPUGenerator can drag down the frame rate to 28fps, taking 35.2ms to render a frame on average. During the experiments, we can obviously feel that the game becomes sluggish. Similar to *Furious Racing*, the screen-update interval of *OpenFlappyBird* that we get from *AndEngine* is dramatically affected by background GPU workloads. Our GPUGenerator makes the interval almost 4 times longer compared to the baseline, leading to unacceptable user experience.

2.4 UI-lagging Issue and Its Root Cause

Our measurement results show that current Android OS fails to guarantee the performance of foreground apps in running with background workloads. The user-perceived latency can be significantly increased, leading to undesirable user experience. Although our measurement study is conducted using controlled experiments and aggressive background workloads, this UI-lagging problem is real. For example, a recent study on smartphone performance anomalies [1] reveals a high percentage (76%) of UI-lagging issues. Also, many UI-lagging issues in Android have been reported online [6], [7], [8], [9], [10], [11], [12], [13].

To further explore the sluggish and unsatisfying user experiences on Android devices, we have conducted a simple user survey with 318 Android student volunteers in China via an online Web sheet. The survey includes only two questions: i) *What's the price of your phone*, and ii) *How often do you encounter UI-lagging issues in using your phone*. The results are reported in Table 4. Overall, 23.6% of the participants complain that they often encounter UI-lagging issues, while 65.1% report as sometimes. Only 11.3% of the participants claim they are never bothered by this problem. Considering the price difference, despite that expensive devices provide a better user experience than the cheaper ones, they cannot fully mitigate the UI-lagging issue. Only 4.4% of the users with cheap devices (<300$) never encountered UI-lagging issues, but 35.7% of the users with expensive devices (>750$) never encountered UI-lagging issues. However, 14.3% users of expensive phones (>750$) still often encountered UI-lagging issues. Such results evidence that the UI-lagging issue is a real and widely encountered problem among Android users.

As shown in our measurement results, we believe that the UI-lagging problem is caused by the insufficient resource management of Android OS. We then investigated how Android manages various system resources and found that Android does not provide enough support to guarantee the performance of foreground apps. For example, in terms of CPU scheduling, the UI threads and its child threads are assigned a high scheduling priority but not all the threads contributing to the user-perceived latency are assigned with high priority. In terms of I/O scheduling, despite that the scheduler can support different priorities, all I/O requests to SD card are treated as the same priority and thus served equally. Furthermore, to the best of our knowledge, there has been no kernel scheduler for GPU so far. Consequently, due to the multi-threading nature of Android OS, the threads that contribute to user-perceived latency have to compete with various threads. If they cannot be served with enough resources and assigned with a proper priority, users can perceive a long interaction latency. Such an insufficient resource management in Android OS motivates us to conduct the work of this paper. Next, we present how we optimize the existing system to reduce user-perceived latency.


Apps
API Interception
Framework
Thread Tracker
• Collect thread synchronization information.
• Analyze trace and store processed results.
• Do runtime decision on priority controlling.
• Load/Store processed results from/to Trace DB.
Trace DB
System Call
Kernel
Resource Manager
CPU CFS
I/O CFQ
GPU Scheduler

context	delay-critical
Cxt1	Yes
Cxt2	No

Fig. 2: The software architecture of *ShuffleDog*.

3 System Design Overview

To reduce user-perceived latency in complex runtime environments such as Android, we need to achieve two goals: i) *accurately identify all the threads*[7] *that process user input and affect user-perceived latency* — we call those threads *delay-critical threads*, and ii) *properly manage system resources and guarantee the performance of delay-critical threads*. Accordingly, as shown in Figure 2, we design our system, as called *ShuffleDog* with two main components, *Thread Tracker* and *Resource Manager*, to realize the two goals, respectively.

In *ShuffleDog*, we propose two key design principles. The first one is *application-transparent*. *ShuffleDog* need to work with existing apps without requiring any modifications or instruments of them. *ShuffleDog* should automatically identify the *delay-critical* threads and provide optimized supports to guarantee the user-perceived latency. The second design principle is *low overhead*. *ShuffleDog* should be lightweight and cannot sacrifice the performance of background apps and services too much. To this end, *ShuffleDog* is designed to reduce user-perceived latency without requiring any efforts from developers and users, or compromising the overall system performance.

Thread Tracker. The Thread Tracker module runs as a system service in background. It keeps monitoring the foreground app. It continuously tracks the threads of the foreground app and their interactions both among themselves and with other threads of system services. By analyzing

[7] We follow the Linux design to treat thread as the resource principal in *ShuffleDog*.

the behaviors of those threads, the Thread Tracker decides which threads are *delay-critical* threads and which ones are not. To this end, it intercepts the Android framework and thus can support existing apps (see more details in Section 4).

To reduce runtime overhead, the Thread Tracker tracks only the threads of the foreground app. When the foreground app is switched to background, the Thread Tracker stops tracking the app. When a user launches a new app or switches another app from background to foreground, the Thread Tracker starts to track this new foreground app.

The Thread Tracker employs a *continuous-learning* based approach to further reduce the runtime overhead. For an app, when the Thread Tracker can learn and identify that some threads are *delay-critical* threads, it stores the information of the threads into a database at local storage. When the same app is launched again, the Thread Tracker loads the information of the previously-learned *delay-critical* threads from the database. Consequently, the Thread Tracker can leverage the information to quickly identify the threads from the new launched app without further analyzing the app's behavior. Such a fashion can reduce the runtime overhead.

After the Thread Tracker decides that a thread is a *delay-critical* thread, it sends the thread ID to the Resource Manager that manages the system resources required by the thread.

Resource Manager. The Resource Manager is a kernel module which is designed to manage and allocate all the system resources. The design principle of the Resource Manager is to provide differential services to various types of threads. For *delay-critical* threads, a high priority is applied to ensure that the threads can occupy necessary resources to finish their work quickly and thus the user-perceived latency can be reduced. For other threads that do not affect user-perceived latency, a low priority is applied to avoid resource contention with the *delay-critical* threads. Given the thread information obtained from the Thread Tracker, the Resource Manager schedules various system resources accordingly.

In this paper, we develop three improved resource schedulers that controls the usage of CPU, I/O, and GPU, respectively. Specifically, for CPU scheduling, we leverage the existing priority-based CPU scheduler and ensure that all the *delay-critical* threads can get a high priority. For I/O scheduling, we improve the current I/O scheduler in Android OS to allow that the I/O requests to SD card from *delay-critical* threads are more preferentially served than the ones from normal threads. For GPU scheduling, we design a new priority-based GPU scheduler to guarantee the performance of *delay-critical* threads in using GPU. One challenge in resource scheduling is providing differential services without compromising the overall system performance. In Section 5, we present how our Resource Manger can provide better graphics processing to *delay-critical* threads without sacrificing background workloads.

4 Identifying Delay-Critical Threads

In this paper, a thread is regarded as a *delay-critical* thread if and only if it meets two conditions simultaneously: *i) it processes user input directly or indirectly*, and *ii) it changes screen output directly or indirectly*. Obviously, if a thread meets the preceding two conditions (e.g., displaying an image upon a button click), its execution time can be perceivable to users and thus affect user-perceived latency. That is, if the thread spends more time on processing user input, the user-perceived latency will be larger and vice versa. If a thread processes user input but does not affect screen output (e.g., logging user inputs to a database), or it does change screen output but that screen update is not driven by user input (e.g., showing a pop-up notification for an incoming short message), users cannot be aware of how much time the thread costs and thus the thread is irrelevant to user-perceived latency.

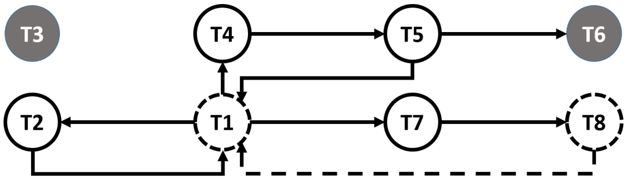


Fig. 3: Delay-critical threads vs. delay-noncritical threads. Each circle represents a thread, and arrows represent interactions between threads. White circles are *delay-critical* threads, while grey circles are *delay-noncritical* threads. T1 is UI thread. T8 is GL thread.

Figure 3 describes examples to illustrate how to decide whether a thread is *delay-critical* or not. These examples are abstractions of real-world cases observed from some open-sourced apps such as MuPDF, K-9 Mail, and so on. **T1** is the UI thread which receives user inputs and changes screen outputs and thus is a *delay-critical* thread. A UI thread is usually the main thread of Android apps and often creates other child threads to process user inputs. For example, in Figure 3, **T1** creates the thread **T2** that processes user inputs and updates the UI through **T1** (e.g., in MuPDF when a user uses fingers to zoom in, an `AsyncTask` is created to compute the resized page). Thus, **T2** is also a *delay-critical* thread. A thread may not process user inputs or change the UI directly. For example, **T4** is created by **T1** but it also creates another thread **T5** that then updates the UI through **T1**. As a result, both **T4** and **T5** are *delay-critical* threads.

Both **T2** and **T5** can change the screen outputs through **T1** that calls the `invalidate()` method of the Android UI Framework to refresh the UI. Besides the basic widgets extended from `View`[8], Android also provides another set of APIs for developers so that they can render directly via *OpenGL ES* [14] libraries. This case is common for games, which require continual animation at a fixed interval. For example, OpenFlappyBird creates a dedicated thread called GL thread (**T8**) by leveraging Android API `GLSurfaceView` to directly draw the screen without switching back to the UI thread. If the GL thread becomes slow, users can perceive the delay. Thus, the GL thread should be treated as a *delay-critical* thread. In Figure 3, we add a dot line between **T8** and **T1** to indicate the **T8** indeeds changes the screen output. Consequently, both **T7** and **T8** are *delay-critical* threads.

In Figure 3, **T6** is a *delay-noncritical* thread because it does not change screen output. Such a case can happen in many apps. As an example, in WeChat[9], a thread called *RWCache_timeourChecker* is created when users send a text message, but it never interacts with the UI thread to change the UI of WeChat and thus does not affect user-perceived

[8] A Java class that represents the basic building block for user interface components.

[9] One of the most popular social networking apps in China.

latency. **T3** in Figure 3 is also a *delay-noncritical* thread as it satisfies none of the two conditions. Examples of **T3** are background services of apps created by the Android system.

The Thread Tracker decides whether a thread of an app is a *delay-critical* thread or not by observing the interactions among all the threads of the app. Without loss of generality, if we observe an interaction circle of $\{T_{ui}, T_1, T_2, ...T_n, T_{ui}\}$, where T_{ui} is the UI thread, then we decide that the UI thread and threads $\{T_1, T_2, ...T_n\}$ are all *delay-critical* threads. The aforementioned GL thread is also a *delay-critical* thread. All other threads are *delay-noncritical* threads.

Identifying *delay-critical* threads is indeed a non-trivial task. It is challenging because *i)* threads interact and communicate with each other in various ways, directly or indirectly; *ii)* whether a thread is *delay-critical* is context-dependent, i.e., two threads created by the same function call (`Thread.create()`) from the same thread can do different types of tasks and thus cannot be judged as *delay-critical* or not; and *iii)* we have to make decisions very fast at runtime as well as with low overhead.

To address the challenges, we instrument all Android APIs that related to inter-thread interactions and communications. These APIs can be divided into three categories: 1) creating a new thread; 2) sending a message from one thread to another; 3) sharing memory among multiple threads. We inject our code into these APIs (see Section 6 for the APIs we instrumented). As a result, when these APIs are called, we can collect the information of calling thread and send it to the Thread Tracker. For sharing memory, developers can use a `lock` or `atomic object` for synchronization, rather than using explicit function calls. Therefore, we can annotate each `lock`, `atomic object`, and `message` with a unique ID so that the Thread Tracker can trace which two threads are acquiring the same lock or atomic object, or which thread sending the `message`. With such information, we are then able to construct a thread-interaction graph as shown in Figure 3 and identify *delay-critical* threads.

One limitation of the preceding approach is that it may take time to construct the desired graph and thus we cannot make a quick-enough decision. For example, after the UI thread creates a new thread to process a user input, it may take a long time for the new thread to finish the task (e.g., loading a large image file and doing image processing) before asking the UI thread to update the screen. We can determine that the new thread is *delay-critical* only after it calls back to the UI thread. However, it may be too late as the new thread has already finished the task.

To address the limitation, the Thread Tracker keeps recording the decision on whether a thread is *delay-critical* or not, together with the execution context in form of a $< context, decision >$ pair. In this way, it can immediately react to the same execution context with an accurate decision afterwards. The execution $context$ is the call stack (i.e., a series of function calls of $\{f_1, f_2, ...f_n\}$) of the thread when the decision is made, and the $decision$ is simply $ture$ or $false$. The Thread Tracker stores the $< context, decision >$ pairs in memory and thus it can make a quick decision when the same execution context happens again. Additionally, the Thread Tracker keeps building and analyzing the thread-interaction graph. When a new decision is made, it updates an existing $< context, decision >$ pair or adds a new pair. Such online learning approach distinguishes our Thread Tracker from other prior efforts [15], [16] based on monitoring IPC, which can be applied for only offline analysis rather than making decision at runtime.

Furthermore, the Thread Tracker stores all the recorded $< context, decision >$ pairs into a database at local storage. In the new launch of the same app, the Thread Tracker loads the previously-learnt decisions from local storage and thus records only incremental information. With this continuous-learning approach, we not only improve the system performance by identifying the *delay-critical* threads early, but also reduce the system overhead.

To reduce the system overhead of the Thread Tracker, we track only the threads of foreground app and stop tracking them when the app is switched into background. In Section 7, we demonstrate that our system introduces a quite minor overhead.

5 Reducing User-Perceived Latency

To reduce user-perceived latency, the Resource Manager manages various system resources to provide prioritized services to *delay-critical* threads. In this paper, we focus on three critical resources, CPU, I/O, and GPU. We next analyze why current scheduling mechanisms are not adequate to promise user-perceived latency, and then describe how we schedule these resources to improve the performance of *delay-critical* threads.

5.1 CPU Scheduling

Current CPU Scheduling. After the release of the Linux 2.6 kernel, the *Completely Fair Scheduler*(**CFS**) [17] is employed as the default process scheduler. The priority of a thread is called `nice` [18], ranging from -20 (highest priority) to 19 (lowest priority). Android inherits such a priority policy, and pays special attention to foreground apps: i) UI thread belonging to the foreground apps is assigned with a relatively high priority (*nice* = 0), whereas background threads (such as a thread executing an `AsyncTask`) are typically given a background priority (*nice* = 10). ii) Android enforces an even stricter scheduling policy using Linux control groups (cgroups [19]). Threads with background priorities are implicitly moved into a background cgroup, where they are limited to only a small percentage of the available CPU if threads in other groups are busy. iii) Besides the default strategy, Android also provides developers alternatives to manually modify the priority via the API `android.os.Process.setThreadPriority`. As a consequence, through careful programming efforts, the developers can make their apps run with high priorities.

Inadequacies. In practice, however, our measurement study in Section 2 demonstrates that current policies are not satisfying enough. The reason is that developers often fail to choose proper APIs or set proper priority, and the system fails to identify the *delay-critical* threads. For example, MuPDF dispatches *delay-critical* tasks to a *AsyncTask*, which runs in background priority, and therefore gets no advantage over background workloads.

Our Approach. We leverage the CFS priority, but apply a high priority to *all* the *delay-critical* threads identified by the Thread Tracker, rather than only the UI thread. Specifically, we set the `nice` value of *delay-critical* threads to 3 (*nice* = 3) that is slightly higher than UI threads but lower than background threads. As a result, we can ensure that the *delay-critical* threads can get more CPU time than background threads to reduce the user-perceived latency, and still retain good UI response as the UI thread still has the highest priority. This priority can also release the *delay-critical* threads from background cgroups.

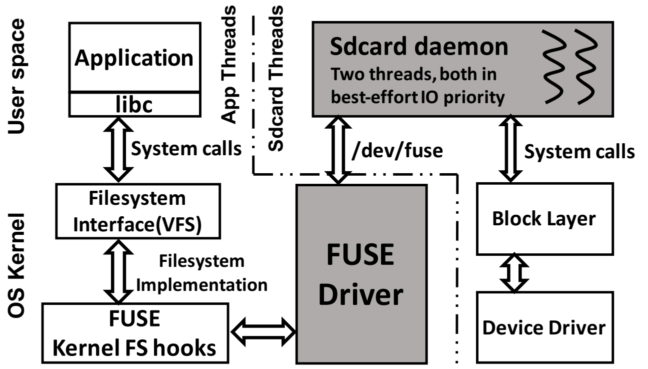


Fig. 4: The file system in userspace (FUSE) architecture in Android. We modify the shaded parts for I/O scheduling.

5.2 I/O Scheduling

Current I/O Scheduling. In current Linux, the *Complete Fair Queuing* (**CFQ**) [20] scheduler is the default I/O scheduler. The **CFQ** maintains a request queue of outstanding I/O requests for each thread that requests synchronous I/O operations. For the asynchronous requests, all the requests from all threads are batched together according to their thread's I/O priority.

At the system level, there are two priority levels: one is at the *class-level* (real-time, best-effort, idle), and the other is the priority *within the class* (8 queues in the real-time class, 8 queues in the best-effort class, and 1 queue in the idle class). Disk access requests from best-effort class are granted only when there is no real-time request left. If not set specifically, the default I/O class will be best-effort and the additional priority number will be related to its CPU-scheduling priority.

Inadequacies. Although Android inherits the CFQ algorithm, it does not expose Java APIs for developers to control the I/O priority. What's even worse, even if the developers can set the threads with high I/O priority as they wish, such modification cannot work because Android uses a separate system daemon to emulate Linux file permissions atop a FAT filesystem (FUSE) on SD cards [21]. As a result, the kernel-side I/O scheduler knows only the priorities of the daemon rather than those of apps where these I/O requests originate.

Figure 4 shows the Android FUSE architecture. When an app tries to read/write a file on a SD card, it will trap into the kernel via a system call. After going through VFS, FS hooks and FUSE Driver, it will send I/O requests back to user-space via a char device: `/dev/fuse`. Then, Sdcard, the system daemon, will read these I/O requests and check the permissions of that app. If permitted, it will go down to other kernel parts, like Block Layer and Device Driver, and finally access the SD card. There are two threads inside Sdcard daemon, both of which are with the same I/O priority (best-effort). Since Sdcard daemon runs in separate threads out of the app, the CFQ scheduler in Block Layer can see only the I/O priority of the daemon. It means that all apps will be treated equally, whatever the I/O priority they own.

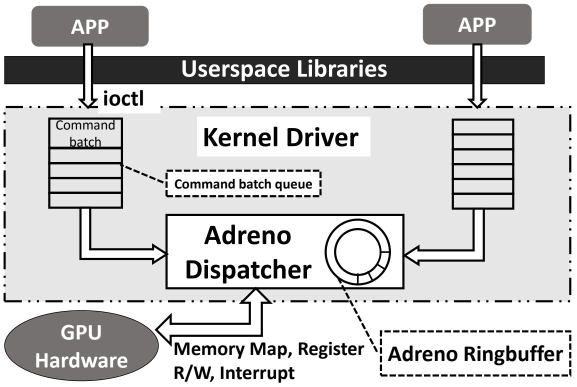


Fig. 5: The existing KGSL architecture for mobile GPU.

Our Approach. To address the above problem, we modify the Sdcard daemon and FUSE driver in two aspects. First, we annotate all I/O requests with the I/O priority of the thread that issues these requests. Such an annotation is done inside the FUSE kernel driver as the I/O priority cannot be accessible in user space. Second, in the Sdcard daemon, we check the annotated I/O information and realize different I/O priorities.

More specifically, in the Sdcard daemon, we use a *dispatcher thread* to read all I/O requests from kernel. However, this thread itself doesn't handle these requests. Instead, it will dispatch each request to a *handler thread* based on the I/O priority of the request. We create two *handler threads* to serve best-effort I/O requests, and another two *handler threads* to serve real-time I/O requests. Hence, when a request from a real-time thread is received by Sdcard daemon, the *dispatcher thread* will check the I/O priority and dispatch it to a *handler thread* with real-time priority.

With this modification, we can identify I/O requests of *delay-critical* threads out of those belonging to other threads, and leverage the CFQ scheduling to provide better I/O service for *delay-critical* threads. Inside the Resource Manager, we set the I/O priority of *delay-critical* threads as real-time. We make this extension available to developers, by providing APIs for developers to set the I/O priority of threads.

5.3 GPU Scheduling

Current GPU Scheduling. GPU has become a powerful and shareable resource in modern mobile systems. With various user-space libraries such as OpenGL ES [14], RenderScript [22], and OpenCL [23], developers can use GPU for not only graphics rendering but also general-purpose programming such as image processing, face recognition, and so on.

Unlike CPU and I/O, the current Android OS does not provide a kernel scheduler for GPU. Instead, all the GPU requests from different apps are directly sent to a GPU driver in kernel via `ioctl()`, and the GPU driver dispatches the requests to GPU hardware. For example, the Nexus 6 has a Adreno[10] [24] GPU that uses the KGSL driver [25] developed by Qualcomm.

Figure 5 illustrates the architecture of the KGSL driver. The GPU kernel driver is shared among multiple threads

[10] Adreno is a popular GPU widely used in many devices like Samsung Galaxy Note, ZTE, XiaoMi, etc.

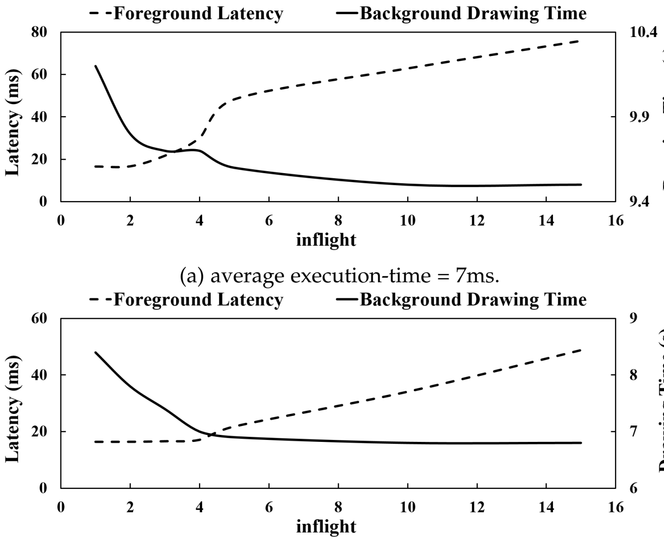
that could belong to different applications (denoted as "*APP*" in this figure). Each thread using GPU has a private GPU context, and the KGSL driver maintains a command-batch[11] queue for each context. The Adreno Dispatcher is the core module in KGSL. It runs in a separate kernel thread and keeps reading command batches from command-batch queues and dispatching them to GPU. The Adreno Dispatcher may submit multiple command batches to GPU simultaneously and it uses a ring buffer, called Adreno Ringbuffer, to remember how many command batches are running in GPU currently. The length of Adreno Ringbuffer determines the max number of command batches allowed to run in GPU at the same time. This max number is called `inflight` and its default value is 15. When one or more command batches are completed, the GPU notifies the KGSL driver via interrupts. Then the Adreno Dispatcher removes those completed batches from Adreno Ringbuffer and fetches new batches if available.

The lifecycle of a command batch in KGSL can be divided into two main stages: i) *Issued stage:* from the time when it arrives in KGSL to the time when it is dispatched to GPU. Usually, the "issue-time" at this stage can be very short (e.g., less than 3ms) when GPU's load is low, since a new command batch will immediately be issued to GPU. However, if GPU load is high and thus the Adreno Ringbuffer is full, a new command batch needs to wait for the completion of a previously-issued command batch. ii) *Execution stage:* the actual time a command batch runs on GPU hardware, from being dispatched into GPU to when GPU notifies KGSL the completion of this command batch. This stage, as we call "execution-time", can vary in a large range (e.g., from 1ms to more than 100ms), depending on the complexity of the command batch.

Inadequacies. Obviously, the support on GPU scheduling in current Android system is far from adequate. It does not have a GPU scheduler in the kernel at all. The Adreno Dispatcher in the KGSL driver applies a scheduling policy of First-In-First-Out (FIFO), providing no priority to different GPU requests. Furthermore, GPU is non-preemptive and thus a new GPU request may have to wait for up to 15 previously-issues requests to be finished, as defined by the large Adreno Ringbuffer. Consequently, as we have shown in Section 2, users may perceive a very long latency when there are background threads fighting for GPU resource.

Our Approach. We propose to design a new GPU scheduler to address the above inadequacies. Specifically, we design a priority-based round-robin scheduler by modifying the KGSL driver. Like CFQ, we define two kinds of GPU priorities: *real-time* and *best-effort*, and annotate each command-batch queue with a proper priority. The command batches in best-effort queues will not be dispatched unless there are no real-time queue or all real-time queues are empty. Unlike CFQ, we don't use a second-level priority. Instead, if there are multiple queues with the same priority, we take a round-robin approach among the queues. We set *delay-critical* threads with real-time priority, and *delay-noncritical* threads as best-effort. Similar to our I/O scheduler, we also provide APIs for developers to change the GPU priorities of threads manually.

One critical design in our GPU scheduling is how to balance the latency and efficiency in using GPU hardware. As mobile GPUs do not support preemption, we cannot stop a command batch that have been already sent to GPU, even if it is a low-priority one. As a result, `inflight` is a critical parameter determining how long a new command batch may wait in the worst case before it is sent to GPU. The larger `inflight`, the longer waiting time, and vice versa. Ideally, to minimize the waiting time and the user-perceived latency, `inflight` should be assigned as 1. However, due to the pipeline-processing nature of GPU, the overall GPU utilization can be low. Therefore, there is a trade-off between the latency of high-priority requests and the total throughputs of GPU.



(a) average execution-time = 7ms.

(b) average execution-time = 4ms.

Fig. 6: Experiments on how `inflight` affects the game performance with different background workloads.

To explore such a trade-off in depth, we conduct experiments to study how different values of `inflight` can affect the latency of a foreground app and the performance of a background workload. Besides changing the value of `inflight`, we also change the execution-time of command batches of the background workload. Figure 6 shows the results in running OpenFlappyBird in foreground and GPUGenerator in background, with the value of `inflight` ranging from 1 and 15. The controlled average execution-time of command batches of GPUGenerator is 7ms and 4ms, respectively.

We make the following observations from Figure 6. First, the value of `inflight` significantly affects the foreground latency and the background performance. A larger `inflight` improves the background performance (i.e., reduces the total drawing time of GPUGenerator) but increases the foreground latency. Second, there is an optimal `inflight` value that achieves the best trade-off between foreground latency and background performance. However, such an optimal point is different when the execution-time of command batches of the background workloads is different. To minimize the foreground latency, the optimal `inflight` value is 2 in Figure 6a and 4 in Figure 6b, respectively.

Motivated by these observations, we employ an adaptive design in controlling the value of `inflight`. We make `inflight` adaptive to the execution-time of recent best-effort command batches as follows:

$$\text{inflight}(x)=\lfloor 16.7ms/x \rfloor \tag{1}$$

[11] A batch of GPU drawing commands, bundled in user-space library. It is the atomic unit of drawing commands.

Module	LoC
Thread Tracker Service	5,731
Android APIs Interception	845
GPU Scheduler	1,237
Sdcard daemon + FUSE Driver	762
Total	*8,575*

TABLE 5: LoC (Lines of Code) of *ShuffleDog*.

where x is the median execution-time of the last 5 best-effort command batches (in ms). 16.7ms is the frame time with the default Android frame rate of 60fps. The rationale of this design is to ensure that a real-time command batch can be likely finished within a frame time to maintain smooth user experience, and multiple best-effort command batches can still be submitted to GPU simultaneously. Indeed, if a best-effort command batch takes a time longer than 16.7ms to finish, we cannot ensure that a real-time command batch can be finished within a frame time. However, based on our experiments, command batches are usually small, and our system is effective in reducing foreground latency. We will demonstrate the results in Section 7.

6 Implementation

We have implemented *ShuffleDog* on Android 5.0.1 (Lollipop) with Linux 3.10 kernel, running upon Nexus 6. As a cross-layer design, we modify both Android Framework (Input Dispatcher, Sdcard daemon, core APIs) and Linux Kernel (GPU Driver, FUSE Driver). Overall, our implemented system contains 8,575 lines of code as illustrated in Table 5.

User-space modification. To collect the propagation information among multiple threads, we instrument Android APIs as illustrated in Table 6.

To help filter useless traces and reduce the overhead, we only turn on this collecting feature when there are user interactions triggered. To get the interaction information, we instrument Input Dispatcher so that it can notify Thread Tracker to start collecting. The collecting function will be automatically turned off after the method `invalidate()` is called or the app is switched to background. The `ActivityManager` provides APIs to check foreground app.

Analyzed results of propagation traces is stored as $< context, decision >$ pairs in a SQLite[12] table, and constructed as a Hashmap in memory. We reserve a configurable XML file inside Thread Tracker, where we can specify the max size of in-memory data size (default is 2MB) and replacement policy of in-memory items (MRU or LRU[13]).

Kernel-space modification. We add a boolean parameter called `ui_critical` in `task_struct`[14] to record whether a thread is *delay-critical* or not. It allows the Resource Manager to access thread priority information all through the kernel. To expose this information to user-space so that our Thread Tracker can update the `ui_critical` value, we register two more system calls in OS kernel, i.e., the `set_ui_critical`, and the `get_ui_critical`. In addition, the callers of these two system calls must own "*root*" permission, indicating that non-rooted apps can not casually change their priority by themselves.

For CPU and I/O scheduling, we use some kernel functions such as `setpriority`, `ioprio_set` to directly leverage the current priority mechanism. We set the `nice` value of *delay-critical* threads to 3, which means these threads will not be limited by the background cgroup with 5% CPU usage limitation. For GPU, we implement our scheduler from scratch on Nexus 6, equipped with Adreno 420 [24]. As explained in Section 5, our GPU scheduler makes completely-fair scheduling on command batches from *delay-critical* threads (with 5 commands being dispatched at most for one time). Command batches from *delay-noncritical* threads can not be scheduled until there are high-priority commands. To monitor the execution time of previous command batches, we record the timestamps of dispatching events generated by Adreno Dispatcher, and the specific interrupts from hardware which will be triggered when a command is consumed. The average execution time of GPU command batches will also be stored in the `task_struct` and updated every one second. Our implementation can work with some other Adreno drivers, as we have experimented on Nexus 5 equipped with 330 Adreno. We plan to evaluate our implementation to work on more Android smartphones in the future.

7 Evaluation

In this section, we evaluate *ShuffleDog* from three aspects: i) *performance improvement of user-perceived latency*; ii) *accuracy in identifying delay-critical threads*; and iii) *system overhead*. We use the 10 apps described in Section 2 and more other apps. We first run these apps with background workloads without enabling *ShuffleDog*, and then run them again by enabling *ShuffleDog*. All experiments are conducted on a Nexus 6 phone running Android 5.0.1. Each experiment is repeated for 20 times and we take median as the metric.

7.1 Performance Improvement

Our experimental results show that *ShuffleDog* is able to significantly reduce the user-perceived latency in running the apps with the aggressive background workloads, as shown in Table 7. Here, "default" means the default Android system.

For CPU scheduling (Table 7a), as afore-mentioned, the background workloads can severely interfere the interactive experience in *MuPDF*, i.e., the latency increases from 696ms to more than 7s with CPUGenerator enabled, and even more than 8s with CPUFilter enabled. In contrast, in *ShuffleDog*, *MuPDF* can perform more smoothly with the same background workloads running, and the latency is less than 800ms with both CPUGenerator and CPUFilter workloads enabled.

The similar results can be found in I/O scheduling. In Table 7b, compared to the baseline (3.8s), the latency of WeChat is extended by 4.1s with IOGenerator and 3.3s with AppInstall, respectively. With our I/O Scheduler enabled, the latency is maintained within 4.0s, which almost equals to the baseline.

The performance improvement of GPU scheduling is also significant, as shown in Table 7c. For example, with GPUGenerator workloads in background, the latency of *OpenFlappyBird* in default system increases by almost 4 times compared to baseline, i.e., from 16.7ms to 74.6ms. In *ShuffleDog*, the latency is still 16.7ms with the same workloads. In other words, *ShuffleDog* can make the apps perform as smooth as if no workloads run in background.

[12] A lightweight relational database built-in in Android.

[13] MRU: Most Recently Used. LRU: Least Recently Used.

[14] Processor descriptor in Linux Kernel.

Category	Instrumented APIs	Collected data
create threads	`Thread, AsyncTask`	call_stack, parent_tid, child_tid
ITC	`Message, Handler, ThreadPoolExecutor, Runnable, etc`	call_stack, msg_id, task_id, current_tid
lock	`Semaphore, ReentrantLock, ReentrantReadWriteLock, etc`	call_stack, lock_id, current_tid
atomic	`BlockingQueue, AtomicIntegerArray, ConcurrentMap, etc`	call_stack, object_id, current_tid
IPC	binder driver in kernel	from_tid, to_tid, to_tname, transaction_id

TABLE 6: A list of instrumented APIs.

ShuffleDog also imposes a small impact on the performance of background workloads. For example, the processing time of CPUGenerator is 6,825ms in *ShuffleDog*, only 6% higher than 6,420ms in default Android OS. Similarly, for CPUFilter, the image processing time is only 3% higher than the default Android OS. For I/O, such a loss is also marginal, as the installation time increases by 0.3s (from 8.4s to 8.7s) and 0.1s (from 8.4s to 8.5s) for *RAR* and *ImageViewFlipper*, respectively. For GPU, the impact on background performance is a little bit higher, but still no more than 7% and only 3ms (from 44ms to 47ms). The reason is that *OpenFlappyBird* requests for GPU resource continually as long as it runs in foreground, while other apps burst for resource only when user interactions occur.

More evaluations on games. Given that games are mostly latency-sensitive and resource-intensive and we design and implement a priority-based GPU scheduler from scratch, we conduct more experiments on games. We choose a typical interaction situation - playing game in foreground while doing GPU-intensive workloads in background. We select 10 popular games from different categories, including Action (*FlappyBird*, *Temple Run*), Racing (*Furious Racing*, *City Driving 3D*), Sports (*3D-Tennis*, *Dream League Soccer*), Role Playing (*Into the Dead*), etc. For each game, we measure the frame rate of running the game in five cases: on default Android system without any background workload (baseline), on default Android system with one of two background workloads (GPUGenerator and GPUFilter), and on *ShuffleDog* with one of the two background workloads.

Experimental results shown in Figure 7 demonstrate that *ShuffleDog* is able to significantly improve the performance of the games in running with a background workload. For example, on the default Android system, the background image processing workload can reduce the frame rate by 46% (from 59fps to 32fps) in *City Driving 3D*, and 63% (from 59fps to 22fps) in *Subway Surfers*. The GPUGenerator, can even more seriously reduce the flame rate, from 59fps to 13fps (i.e., a 78% reduction) in *City Driving 3D*. To compare, the frame rate in *ShuffleDog* is very close to the baseline. All except one of the games have a frame-rate reduction of within 3 frames in *ShuffleDog*. The improvement of *City Driving 3D* can reach up to 23 times compared to default Android case, i.e., a 2-frame reduction against a 46-frame reduction. The exception is *Implosion*, in which *ShuffleDog* can help reach a frame rate of only 42fps with GPUGenerator workload, a 7-frame reduction compared to baseline. The reason could be that *Implosion* has more complicated and gorgeous graphic elements to render than other games.

Multi-resource case study. To study how multiple-resource contention affects the user-perceived latency collaboratively, we conduct a case study on *RAR*. Instead of studying how I/O resource alone can affect the compression performance, we run both I/O and CPU workloads in background. The results are reported in Table 8. It shows that the compression latency affected by I/O and CPU collaboratively can reach as high as 43s, almost a 10-times increase compared to the baseline (3.9s). By enabling our CPU scheduler alone, the latency is reduced to 7.1s. By enabling our I/O scheduler alone, the latency can be reduced to 37.1s. However, when enabling both I/O and CPU schedulers, the latency can be significantly reduced to 4.3s, which is only 10% higher compared to the baseline.

7.2 Accuracy of Thread Tracker

We then evaluate how the Thread Tracker can identify *delay-critical* thread accurately. Besides the 5 apps (*MuPDF, SystemUI, IO File Manager, Image ViewFlipper*, and *OpenFlappyBird*) used in our measurement study, we add another 5 more apps (*Conversations, AdAway, KeePassDroid, OsmAnd*, and *K9 Mail*). We choose these apps because they are open-sourced and thus we can inspect the source code to get the ground truth on whether a thread should be *delay-critical* or not.

The results are shown in Table 9. Overall, 8 out of the 10 apps have wrongly-treated threads in default Android system; and for other 2 apps, all their threads are with high priority and *delay-critical*. Specifically, False Positive (**FP**) comes from incorrectly treating *delay-noncritical* threads as high priority. Such cases happen when developers use APIs such as `Thread` and `Runnable` to deal with tasks that are not related to user input actions. Take the app *Conversations* as an example. When a user tries to send or fetch messages, this app also writes some tags into local database. Such a task is assigned with in a high-priority thread, but has nothing to do with user interaction. Only one app has this **FP** issue. False Negative (**FN**) means assigning *delay-critical* threads with low-priority, e.g., using `AsyncTask` to process user action related tasks. **FN** is a widely-happened issue. 7 out of the 10 apps suffers from **FN** issues in default Android system.

ShuffleDog does not have any **FP** issue and is able to mitigate **FN** issues in 9 apps. The only exception is *OpenFlappyBird*. The reason is that *OpenFlappyBird* has two threads sharing two integers without locking or using atomic objects. As there is no explicit synchronization mechanism used, our Thread Tracker cannot capture this case. We discuss more on this limitation in Section 8.

7.3 System Overhead

We also evaluate the runtime overhead introduced by *ShuffleDog*. As shown in Table 10, the extra CPU usage is less than 3.1%, mainly caused by the Thread Tracker service running in background. Although the Thread Tracker needs to collect traces, make run-time analysis, and carry data between memory and database, these operations happen only when there are user interactions triggered. The memory overhead is about 7MB, coming from the storage of the $< context, decision >$ pairs, additional threads in Sdcard daemon, and the parameters in `task_struct`. Such small overhead of CPU and memory are minor on modern smartphones and well worth given the significant reduction of user-perceived latency achieved by *ShuffleDog*.

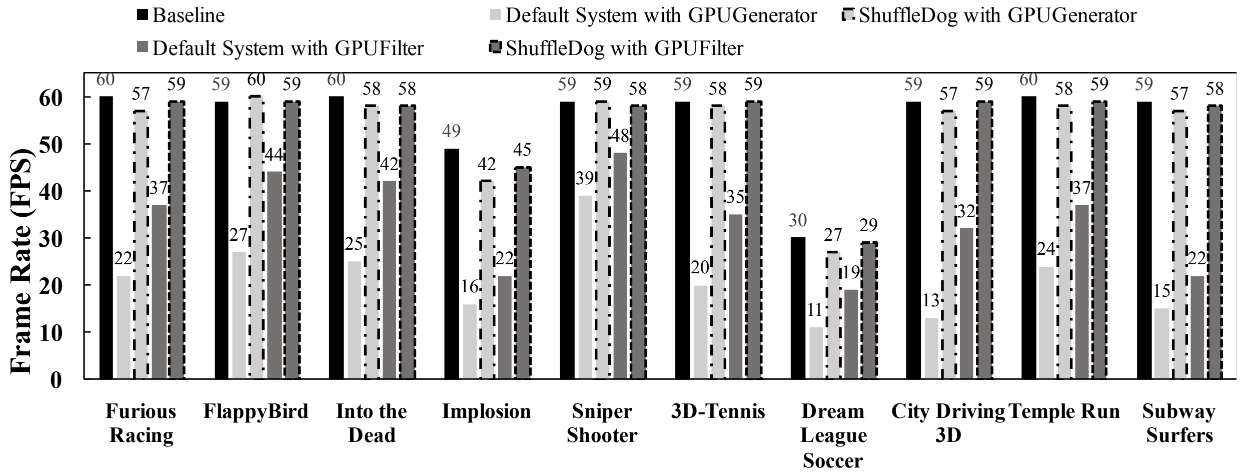


Fig. 7: Frame rates of 10 games with and without *ShuffleDog*.

8 Discussion

Our current system has some limitations that call for more future work. First, the Thread Tracker can handle most cases but is limited in three aspects. First, it cannot work if two threads share memory without using locks or atomic objects (which is not a good programming practice indeed) as such implicit interactions cannot be captured. Second, it cannot work if developers use NDK (Android Native Development Kit) to handle user action as we make instruments at Java API level. Third, it cannot identify the threads communication via `synchronized` and `wait` as instruments in Runtime VM (i.e., Dalvik and ART [26][15]) are needed. The latter two limitations are implementation-level issues and can be addressed by NDK and VM level instruments.

Another limitation is our GPU scheduler cannot guarantee foreground performance in "large" background GPU calls. Due to the non-preemptive nature of GPU, a high-priory request may have to wait for a large but low-priority request to complete, leading to undesirable user-perceived latency. Possible solutions include enabling GPU preemption and dividing a large call into small ones.

It's worth mentioning that, if developers' implementation is improper (e.g., intensive calculation more than 16ms during each frame time), *ShuffleDog* can not help achieve smooth user experience any longer. The reason is that *ShuffleDog* focuses on resource scheduling policies, which can help only when there are multiple threads competing for the same resource. Improving the performance of only one single thread is not the focus of this work. For these cases, some previously measurement analysis and developed tools such as QoE Doctor [27], [28] can be leveraged.

Our current implementation can support CPU, disk I/O, and GPU. There are also other system resources such as memory and network delay that can affect user-perceived latency as well. Although they are already explored in some extend [29] [30] [31], it is desirable to explore how those resources impact user-perceived latency and how to further reduce the latency. The main reason why we don't consider network scheduling in this paper is that long network requests are usually caused by poor network condition (e.g., long round-trip time (RTT) and low bandwidth) and/or large computation workloads at server-side (I cannot find good references here. Yunxin said it's okay we leave it as common sense.). The bottlenecks often reside in the network and the server rather than resource contention at client-side.

Our implementation may not work on all Android devices due to the heavy fragmentation of Android devices [32]. In addition, we focus on Android platform only in this paper. Although modern OSes such as Windows and iOS share the similar high-level OS abstractions of Android, it is still interesting to explore how our approach works on other popular mobile platforms. We should also take into account the compatibility of *ShuffleDog* when the version of Android OS evolves [33].

9 Related Work

In this section, we discuss existing literature studies that relate to our work in this paper.

RSIO [34] applies a similar approach to trace propagation among "latency-sensitive" processes. However, their strategy depends on the observation that latency-sensitive activities typically need to respond quickly to I/O involving user interactions, which may not be true for many mobile applications. Also, RSIO needs manual configuration while our approach does not. TIPME [35] allows users to trigger trace capturing post-analysis when experiencing long latencies, a method too disruptive to mobile users today. Magpie [36] asks developer-provided event semantics to characterize each individual user action handled by Windows-based servers. These studies are motivational to our work. AppInsight [16] instruments app binaries to log user transaction events, therefore identifying critical execution paths in user transactions and performance issues. Panappticon [15] adopts a similar approach by instrumenting the Android OS; in particular, it extends the approach to the OS kernel. However, such work focused only on offline analysis and targeted revealing app inefficiencies to their developers. For example, Panappticon can not decide a thread to be delay-critical or not when it's created by UI thread, since we do not know if it will affect UI elements

[15] Although ART has replaced Dalvik since Android 4.4, we believe that the instrumentation is similar because the main features introduced by ART (e.g., ahead-of-time compilation and improved garbage collection) have little relationship with *ShuffleDog*.

App/Workloads	System	Fore Perf	/ Ratio	Back Perf
Google Map	default	1,266ms	1.00x	/
Google Map + CPUGenerator	default	4,088ms	3.23x	6,420ms
	ShuffleDog	1,297ms	1.02x	6,515ms
Google Map + CPUFilter	default	4,824ms	3.81x	65ms
	ShuffleDog	1,318ms	1.04x	68ms
Messenger	default	1,673ms	1.00x	/
Messenger + CPUGenerator	default	2,368ms	1.42x	6,420ms
	ShuffleDog	1,702ms	1.02x	6,782ms
Messenger + CPUFilter	default	2,631ms	1.57x	65ms
	ShuffleDog	1,682ms	1.01x	69ms
Gmail	default	2,217ms	1.00x	/
Gmail + CPUGenerator	default	4,525ms	2.04x	6,420ms
	ShuffleDog	2,301ms	1.04x	6,685ms
Gmail + CPUFilter	default	4,933ms	2.23x	65ms
	ShuffleDog	2,293ms	1.03x	67ms
MuPDF	default	696ms	1.00x	/
MuPDF + CPUGenerator	default	7,325ms	10.52x	6,420ms
	ShuffleDog	714ms	1.03x	6,825ms
MuPDF + CPUFilter	default	8,733ms	12.55x	65ms
	ShuffleDog	721ms	1.04x	67ms

(a) The Back Perf of CPUGenerator is the time to do 10^9 float computations, and for CPUFilter is the time to process an 10MB image.

App/Workloads	System	Fore Perf	/ Ratio	Back Perf
WeChat	default	3.8s	1.00x	/
WeChat + IOGenerator	default	7.9s	2.08x	9.8s
	ShuffleDog	4.0s	1.05x	10.0s
WeChat + AppInstall	default	7.1s	1.87x	8.4s
	ShuffleDog	3.9s	1.03x	8.6s
RAR	default	3.9s	1.00x	/
RAR + IOGenerator	default	5.9s	1.51x	9.8s
	ShuffleDog	3.9s	1.00x	10.1s
RAR + AppInstall	default	5.1s	1.31x	8.4s
	ShuffleDog	4.0s	1.03x	8.7s
ImageViewFlipper	default	878ms	1.00x	/
ImageViewFlipper + IOGenerator	default	1,365ms	1.55x	9.8s
	ShuffleDog	897ms	1.02x	10.2s
ImageViewFlipper + AppInstall	default	1,252ms	1.43x	8.4s
	ShuffleDog	882ms	1.00x	8.5s

(b) The Back Perf of IOGenerator is the time to read a 256MB file randomly via fio, and for AppInstall is the time to complete installation.

App/Workloads	System	Fore Perf	/ Ratio	Back Perf
SystemUI	default	516ms	1.00x	/
SystemUI + GPUGenerator	default	2,218ms	4.30x	39ms
	ShuffleDog	527ms	1.02x	40ms
SystemUI + GPUFilter	default	1,346ms	2.61x	62ms
	ShuffleDog	522ms	1.01x	64ms
OpenFlappyBird	default	16.7ms	1.00x	/
OpenFlappyBird + GPUGenerator	default	74.6ms	4.47x	44ms
	ShuffleDog	16.7ms	1.00x	47ms
OpenFlappyBird + GPUFilter	default	32.7ms	1.96x	65ms
	ShuffleDog	16.7ms	1.00x	67ms

(c) The Back Perf of GPUGenerator is the time to draw 5000 triangles, and for GPUFilter is the time to process an 10MB image.

TABLE 7: Performance evaluation. Here "Fore Perf" indicates performance of foreground apps, and "Back Perf" indicates performance of background workloads.

App/Workloads	Enabled Scheduler	Latency	Ratio
RAR (Baseline)	None	3.9s	1.0x
RAR + IOGenerator + CPUGenerator	None	43.0s	11.0x
	CPU	7.1s	1.8x
	I/O	37.1s	9.5x
	CPU + I/O	4.3s	1.1x

TABLE 8: The latency impact of contenting on both GPU and I/O resources.

App	Category	default		*ShuffleDog*	
		FP	FN	FP	FN
MuPDF	PDF reader	✗	✓	✗	✗
SystemUI	System utility	✗	✓	✗	✗
OI File Manager	File manager	✗	✗	✗	✗
ImageViewFlipper	Image Viewer	✗	✓	✗	✗
OpenFlappyBird	2D game	✗	✗	✗	✓
Conversations	Instant message	✓	✗	✗	✗
AdAway	Advertisement block	✗	✓	✗	✗
KeePassDroid	Security tool	✗	✓	✗	✗
OsmAnd	Navigation	✗	✓	✗	✗
K9 Mail	Email client	✗	✓	✗	✗

TABLE 9: Accuracy of the Thread Tracker. "FP" means incorrectly setting *delay-noncritical* threads as high-priority, and "FN" means incorrectly setting *delay-critical* threads as low-priority.

afterwards. In contrast, our Thread Tracker can identify *delay-critical* threads at runtime with very small overhead, by learning from prior executions.

Redline [37] provides OS support that allows apps to statically reserve resources for their tasks. Compared to it, our latency-based QoS interface doesn't need efforts from developers. It identifies *delay-critical* tasks and properly allocates resource for them automatically. Timecard [38] tracks user requests in a mobile-cloud system, providing latency information to the cloud server so that the latter can adapt its behavior in order to meet deadlines. However, it can only be applied on certain kinds of apps, and does not address resource management for continuous interaction in a single mobile OS. For mobile devices, ura [39] proposes to identify app execution epochs that have no direct impact on user-perceived latencies and thus reduce CPU frequency. However, it is unable to differentiate threads that do impact user experiences and provision for them accordingly.

The system proposed by Huh *et al.* [40] identifies interactive tasks and provisions them by modifying CPU scheduling policies for them. They focus on CFS itself, and treat all foreground threads as high-priority. In contrast, we leverage the current CFS policy, but propose a more accurate Tracker module to identify *delay-critical* tasks. For mobile I/O scheduling, SmartIO [5] reduces Android I/O delay by adding a third priority class to CFQ, isolating read and write operations. Jeong *et al.* [41] proposed a novel scheme to improve the responsiveness of file system operations by detecting and boosting QASIOs at runtime in I/O scheduler. Also, many other researchers have proposed improvements to I/O schedulers for flash memory based Solid State Drives (SSD) [42], [43], [44], but they mainly focus on overall throughput and fairness, and didn't fix the gap between Android and Linux as we have mentioned.

To manage GPUs, some scheduling design and implementations have been proposed for desktops and servers. GPU scheduling can be realized by replacing black-box driver with open-source library [45], [46] or interposing the memory protection mechanism [47]. PTask [48] proposes a new set of OS abstractions to support GPUs as first class

App	Memory	CPU Usage
Google Map	7,152KB	2.2%
Messenger	7,018KB	1.8%
Gmail	7,125KB	2.6%
WeChat	7,391KB	3.1%
MuPDF	6,557KB	1.6%
RAR	6,623KB	1.8%
ImageViewFlipper	6,538KB	1.3%
SystemUI	6,541KB	1.3%
Furious Racing	6,562KB	1.9%
OpenFlappyBird	6,538KB	1.7%

TABLE 10: Measured system overhead.

computing resources, and provides system-wide guarantees like fairness and performance isolation. Instead of desktops and servers, we implement our GPU scheduler by revising the open-source KGSL driver and enable priority-based scheduling without compromising the performance of background workloads on mobile devices. To further address the non-preemptive issue in GPUs, some approaches have been proposed such as splitting longer requests [49] and context switch [50], [51], [52]. However, these approaches depend on open-sourced software stacks or need hardware modification. We leave the non-preemptive issues to our future work.

10 Conclusion

In this paper, we have conducted a set of measurement studies to quantify user-perceived latency of Android apps and revealed that current resource management mechanism in Android is not sufficient to promise satisfying user experience, i.e., the user-perceived latency of foreground apps can be dramatically increased by background resource-intensive workloads. We have designed and implemented a novel system to accurately identify *delay-critical* threads at runtime and improve the resource management for these threads. Evaluation results show that our system can significantly reduce user-perceived latency of foreground apps in running with aggressive background workloads, with minimal system overhead and without compromising the system throughput.

Acknowledgment

This work was supported by the National Basic Research Program (973) of China under Grant No. 2014CB347701, the Natural Science Foundation of China (Grant No. 61370020, 61421091, 61528201), and the Microsoft-PKU Joint Research Program. Felix Xiaozhu Lin's work was supported by NSF Award #1464357 and a Google Faculty Award. The three authors, Gang Huang, Mengwei Xu, and Felix Xiaozhu Lin, contributed equally to this work. Xuanzhe Liu acts as the corresponding author of this work.

References

[1] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, 2014, pp. 1013–1024.

[2] "Adreno GPU Profiler," https://developer.qualcomm.com/software/adreno-gpu-profiler, 2016.

[3] "Image Processing Test," https://android.googlesource.com/platform/frameworks/rs/+/master/java/tests/ImageProcessing, 2016.

[4] J.Axboe, "FIO: Flexible io tester," http://linux.die.net/man/1/fio, 2016.

[5] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, Q. Yang, and M.-S. Hall, "Reducing smartphone application delay through read/write isolation." in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015*, 2015, pp. 287–300.

[6] "Touch events lost during garbage collection," https://code.google.com/p/android/issues/detail?id=1742, 2009.

[7] "SQLite optimization and async I/O," https://groups.google.com/forum/#!topic/android-ndk/veutXL9oO4w, 2011.

[8] "Nexus 7 doesn't respond to touch," https://code.google.com/p/android/issues/detail?id=35663, 2012.

[9] "Android spends 7.4% of its time unresponsive," https://groups.google.com/forum/#!msg/android-platform/7O_ClQkuGSE/nKbeFOPoXNkJ, 2009.

[10] "Game unresponsive in Android," http://androidqueries.com/games-unresponsive-android-3164.html, 2012.

[11] "Why my android running so slowly," http://phonetipz.com/why-is-my-android-running-so-slow/, 2014.

[12] "UI lagging problem in MIUI," http://en.miui.com/thread-62958-1-1.html, 2014.

[13] "When will we be done with UI lagging," https://www.reddit.com/r/Android/comments/2cxadc/when_will_we_be_done_with_ui_stutterlag/, 2015.

[14] Khronos Group, "OpenGL ES - The Standard for Embedded Accelerated 3D Graphics," https://www.khronos.org/opengles/, 2016.

[15] L. Zhang, D. Bild, R. Dick, Z. Mao, and P. Dinda, "Panappticon: Event-based tracing to measure mobile application and platform performance," in *Proceedings of Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*, 2013, pp. 1–10.

[16] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI 2012*, 2012, pp. 107–120.

[17] I. Molnar, "Linux CFS scheduler," 2007.

[18] "Linux sched nice design," https://www.kernel.org/doc/Documentation/scheduler/sched-nice-design.txt, 2016.

[19] "Linux cgroups," https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, 2004.

[20] J. Axboe, "Linux block I/O-present and future," in *Ottawa Linux Symp 2004*, 2004.

[21] "Android FAT-on-sdcard," https://github.com/android/platform_system_core/blob/master/sdcard/sdcard.c, 2010.

[22] "Android Renderscript," http://developer.android.com/guide/topics/renderscript/compute.html, 2016.

[23] Khronos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/, 2016.

[24] "Qualcomm Adreno," https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu, 2016.

[25] "Qualcomm 2D/3D graphics driver," https://lwn.net/Articles/394665/, 2010.

[26] ART and Dalvik, "https://source.android.com/devices/tech/dalvik/," 2013.

[27] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "QoE Doctor: Diagnosing mobile app QoE with automated UI control and cross-layer analysis," in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014*, 2014, pp. 151–164.

[28] Y. Feng, Q. Liu, M. Dou, J. Liu, and Z. Chen, "Mubug: a mobile service for rapid bug tracking," *SCIENCE CHINA Information Sciences*, vol. 59, no. 1, pp. 1–5, 2016.

[29] M. Mohandespour, M. Govindarasu, and Z. Wang, "Rate, energy, and delay tradeoffs in wireless multicast: Network coding versus routing," *IEEE Transactions on Mobile Computing, TMC 2016*, pp. 952–963, 2016.

[30] H. Al-Tous and I. Barhumi, "Resource allocation for multiple-sources single-relay cooperative communication OFDMA systems," *IEEE Transactions on Mobile Computing, TMC 2016*, pp. 964–981, 2016.

[31] K. Lee, J. Jeong, Y. Yi, H. Won, I. Rhee, and S. Chong, "Max contribution: An online approximation of optimal resource allocation in delay tolerant networks," *IEEE Transactions on Mobile Computing, TMC 2016*, pp. 592–605, 2015.

[32] L. Wei, Y. Liu, and S. Cheung, "Taming android fragmentation: characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 226–237.

[33] H. Wang and B. Ding, "Growing construction and adaptive evolution of complex software systems," *SCIENCE CHINA Information Sciences*, vol. 59, no. 5, pp. 050 101:1–050 101:3, 2016.

[34] H. Zheng and J. Nieh, "RSIO: automatic user interaction detection and scheduling," in *Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2010*, 2010, pp. 263–274.

[35] Y. Endo and M. Seltzer, "Improving interactive performance using TIPME," in *Proceedings of the 2000 ACM International Conference on Measurement and Modeling of Computer Systems, SIGEMETRICS 2000*, 2000, pp. 240–251.

[36] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation, OSDI 2004*. USENIX Association, 2004, pp. 18–18.

[37] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: First class support for interactivity in commodity operating systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008*, 2008, pp. 73–86.

[38] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP 2013*, 2013, pp. 85–100.

[39] W. Song, N. Sung, B.-G. Chun, and J. Kim, "Reducing energy consumption of smartphones using user-perceived response time analysis," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications, HotMobile 2014*, 2014, pp. 20:1–20:6.

[40] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI 1999*, 1999, pp. 45–58.

[41] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015*, 2015, pp. 191–202.

[42] M. P. DUNN, "A new I/O scheduler for solid state devices," Ph.D. dissertation, 2009.

[43] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *Proceedings of the 7th ACM international conference on Embedded software, EMSOFT 2009*, 2009, pp. 295–304.

[44] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware i/o management for solid state disks (ssds)," *IEEE Trans. Computers*, vol. 61, no. 5, pp. 636–649, 2012.

[45] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proceedings of 2011 USENIX Annual Technical Conference, ATC 2011*, 2011, p. 17.

[46] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proceedings of 2012 USENIX Annual Technical Conference, ATC 2012*, 2012, pp. 401–412.

[47] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proceedings of Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014*, 2014, pp. 301–316.

[48] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage gpus as compute devices," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011*, 2011, pp. 233–248.

[49] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Proceedings of the 2011 IEEE Real-Time Systems Symposium, RTSS 2011*, 2011, pp. 57–66.

[50] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015*, 2015, pp. 593–606.

[51] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proceedings of the 41st International Symposium on Computer Architecture, ISCA 2014*, 2014, pp. 193–204.

[52] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: exception support and speculative execution on GPUs," in *Proceedings of the 39th International Symposium on Computer Architecture, ISCA 2012*, 2012, pp. 72–83.

Gang Huang is now a full professor in Institute of Software, Peking University. His research interests are in the area of middleware of cloud computing and mobile computing. He is a member of IEEE.

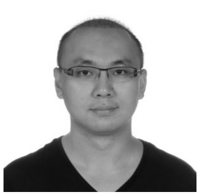
Mengwei Xu is a Ph.D candidate student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include mobile computing and operating system.

Felix Xiaozhu Lin is an assistant professor of Purdue University. He received his PhD in Computer Science at Rice (2014), M.S. in Computer Science (2008) and B.S. in Automation (2006) both from Tsinghua University, respectively. His research interests include operating system and runtime for programmability, energy efficiency, and performance.

Yunxin Liu is a lead researcher in Microsoft Research Asia. He received his Ph.D. in computer science from Shanghai Jiao Tong University in 2011 (through the SJTU-MSRA joint PhD program). His current research interests include mobile systems and networking.

Yun Ma is a Ph.D student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include services computing and web engineering.

Saumay Pushp is a Ph.D candidate in the School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, South Korea. His research interest include Mobile Systems and Networking.



Xuanzhe Liu is an associate professor in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests are in the area of services computing, mobile computing, web-based systems, and big data analytic. He is the corresponding author of this work.