PerfProbe: A Systematic, Cross-Layer Performance Diagnosis Framework for Mobile Platforms

David Ke Hong†, Ashkan Nikravesh†, Z. Morley Mao†, Mahesh Ketkar‡, Michael Kishinevsky‡
†University of Michigan, ‡Intel Corporation
{kehong, ashnik, zmao}@umich.edu, {mahesh.c.ketkar, michael.kishinevsky}@intel.com

***Abstract*—User-perceived performance slowdown in mobile apps can occur in unpredictable and sophisticated ways, with root cause spanning at different layers (app or system layer). There is a lack of effective approaches to provide cross-layer, holistic insights to diagnose unpredictable performance slowdown on mobile platforms, motivating us to develop PerfProbe as a performance diagnosis framework for mobile platforms. PerfProbe monitors app performance and records app and system-layer runtime information in a lightweight manner on mobile devices, and performs systematic, novel statistical analysis on collected runtime traces at different layers to localize code-level performance variance in the form of *critical functions* and zoom into them to pinpoint system-level root causes in the form of *relevant resource factors* to explain the performance slowdown. PerfProbe effectively diagnoses performance slowdown due to various root causes in 22 popular Android apps from real-world usage monitoring and in-lab testing, by providing holistic, cross-layer insights to help the root cause diagnosis. Diagnosis findings from PerfProbe provide actionable insights for root cause finding and guiding real-world app developers' code fixing or adjustment of platform-level policies to reduce user-perceived latency of 6 real Android apps by 32-86%. PerfProbe incurs small system overhead and impact to app performance at runtime and is suitable for real-world deployment.**

I. Introduction

With the rapid advancement of mobile computing and networking technologies, mobile devices have become ubiquitous and the app market is multiplying. Today's Google Play store stocks over 2 million Android apps with over 50 billions of total downloads [30], [13]. Different from server-based applications, mobile apps are user-facing and highly interactive, and usually running on a resource-constrained and dynamic environment. These unique runtime features make apps more likely to be affected by internal device-specific resource constraints (e.g., CPU, memory, disk) as well as external environment factors, including network quality and server-side delay. Variance of some key factors may lead to large variation in user-perceived latency, degradation of which is an important type of quality-of-experience (QoE) problems for mobile apps. As one critical performance metric for a wide variety of interactive apps, user-perceived latency has recently drawn attention from the research community [56], [73] and app industry [29]. Google's RAIL performance model [46] shows that a user may lose focus on the task they are performing if the system response takes over 1 second. Therefore, it becomes crucial to uncover performance degradation in critical user interactions and diagnose them at the early testing or deployment stage, so as to provide app developer or device vendors with useful hints for implementing effective strategies to ensure app responsiveness and good user experience.

Our empirical study on 100 popular Android apps shows that user-perceived performance for key user interactions in an app can degrade by multiple times in some runs. As later shown by our diagnosis, such unpredictable performance slowdown can be due to specific runtime context (§V-C) or code-level design issues (§V-A). Pinpointing these sophisticated factors requires analyzing app and system-layer information collected at runtime. On the one hand, localizing code-level performance variance with common program abstractions (e.g., function) provides semantically meaningful hints for root cause reasoning and code fixing by human developers. On the other hand, system-wide runtime events record fine-grained details on how an app interacts with system resources over time, which may help app developers quantify the runtime cost of their code to certain types of resource, especially when an invoked third-party library is proprietary or too complex to understand its resource intensity. Such resource-level root cause reasoning is also useful to guide device vendors' refinement of system-level configurations or policies (e.g., buffer size, frequency governor, code offloading [58], [60], [59]) to achieve better mobile performance.

Unfortunately, we see a lack of effective approach to provide such cross-layer, holistic insights for helping the diagnosis of unpredictable performance slowdown on mobile platforms. To fill this gap, we develop PerfProbe as a performance diagnosis framework for mobile platforms. PerfProbe does not require app source code and takes app binary as input. Developers can specify their interested user interactions to be monitored through its configuration interface. PerfProbe then monitors performance of these interactions triggered by real-world usage and records app and system-layer runtime information in a lightweight manner on a mobile device. Once performance slowdown is detected, PerfProbe performs offline diagnosis by associating the collected runtime traces at different layers using a statistical learning approach. PerfProbe provides cross-layer, informative insights as its diagnosis output to facilitate the root cause analysis by app developers or device vendors: 1) *critical function* showing the executed function calls whose slowdown is most correlated to performance degradation; 2) *relevant resource factor* indicating what system resource (e.g., computation, network, disk, etc.) a critical function interacting with is most correlated to the slowdown of the function. Motivated by a real-world app study and subsequent diagnostic evaluation, our cross-layer characterization approach enables resource-level understanding compared to existing app-level profiling approaches [81] (§II-A), and achieves higher accuracy in pinpointing the relevant resource factors causing performance slowdown than existing OS monitoring or resource profiling approaches [85], [11], [32], [66] (§V-D).

To develop PerfProbe, we overcome two major research challenges. First, recording fine-grained app and OS-layer runtime information can incur large overhead to a mobile device and degrades app performance, influencing both user experience and accuracy of problem diagnosis. To address this challenge, we propose a novel model-driven adaptive sampling mechanism to accommodate the different levels of profiling overhead incurred by different apps on different devices and achieve lightweight call stack profiling. It performs real-time monitoring of the performance impact to an app due to profiling its call stack and based on that adjusting the call stack dumping frequency to limit its impact within some configurable threshold (§III-B). Second, a user interaction in real apps usually involves execution across dozens or hundreds of threads and even across process boundary, with different threads bounded by different system resources. To overcome this challenge, we propose a novel statistical analysis approach that first zooms into app-level execution to identify a small set of critical functions and then pinpoints their underlying resource factors relevant to the cause of performance variance (§IV). As one main novelty of our system, this two-step critical function and resource factor characterization imitates human inspection, but involves no manual efforts.

Our work makes the following research contributions:

- We develop a lightweight performance monitoring mechanism with smaller performance impact than state-of-the-art for mobile platforms that collects detailed app and system-layer runtime information to support the diagnosis of unpredictable performance slowdown in mobile apps.
- We design an automated, systematic cross-layer characterization approach that performs two-step statistical characterization on app and OS-layer traces to pinpoint critical functions and their underlying relevant resource factors for explaining the cause of unpredictable performance slowdown in mobile apps. This cross-layer characterization approach by design can be generalized to diagnosing similar performance issues in other software systems.
- Diagnosis findings from PerfProbe on real-world performance issues provide valuable insights for guiding code-level fixing of real-world app developers and adjustment of platform-level policies to reduce user-perceived latency of 6 real Android apps by 32-86%.

In the following sections, we will use the term performance or user-perceived latency interchangeably.

II. Motivation & Approach

In this section, we motivate the need of associating app and OS-layer runtime information for performance analysis using a popular Android app as a motivating example (§II-A) and present a key design challenge to achieve our diagnosis goal (§II-B).

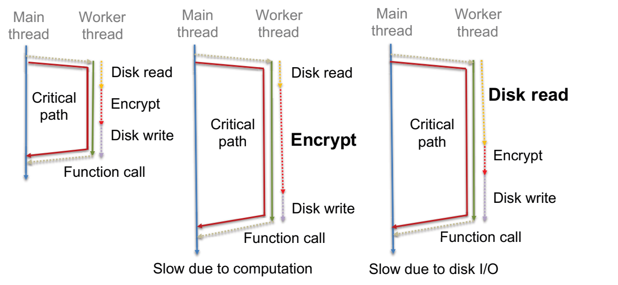


Fig. 1. Performance slowdown due to different root causes (length of the arrows corresponding to execution time) in Run 1, 2, 3 (from left to right). No slowdown in Run 1.

A. Motivating Example

We study *SSE*, a popular Android encryption app, in which users click a UI button to encrypt a file stored in SD card. Perturbing different resources in the system consistently causes severe performance degradation. Figure 1 illustrates the execution workflow for this interaction in different runs. In this interaction, the main/UI thread invokes a worker thread to execute an encryption function that performs three operations: read the file from SD card, encrypt the bits and write the encrypted file to SD card. Performance slowdown that occurs in 2nd and 3rd run, however, are due to different resource bottlenecks – slow disk I/O in loading the file from the local storage in one run and insufficient CPU cycles for performing computation-intensive encrypting operations in the other run.

The benefits of associating app and OS-layer runtime information together for performance diagnosis are two-fold. First, app-level profiling [35], [81] may identify what function calls lead to performance variance (critical functions SCrypt.scryptN or Posix.readBytes in Table II) under different resource perturbations, but is unable to pinpoint underlying resource bottlenecks that are unique to the runtime. Domain knowledge on the script or posix library is required for understanding. In fact, performance slowdowns observed in our deployment study were caused by app's invocation of certain system resources that become a bottleneck (e.g., computation bottleneck due to the CPU frequency cap enforced by DVFS governor policies in §V-C1, disk I/O bottleneck due to the read-ahead buffer limit in §V-C2, etc.), which can hardly be uncovered by analyzing app-layer execution alone. Second, applying traditional resource profiling [11], [32] or tracking system calls [66] or primitive OS events [85] loses track of details of program semantic (e.g., functions) and cannot provide developers with easy-to-reason hints to enable further code-level inspection. Moreover, traditional resource profiling [11], [32] alone, as shown by our evaluation (§V-D), is sometimes too coarse-grained to accurately pinpoint the true resource bottleneck.

PerfProbe's cross-layer diagnosis approach aims to address these limitations. It localizes to function calls with running time correlated to the performance variance (a.k.a., *critical*

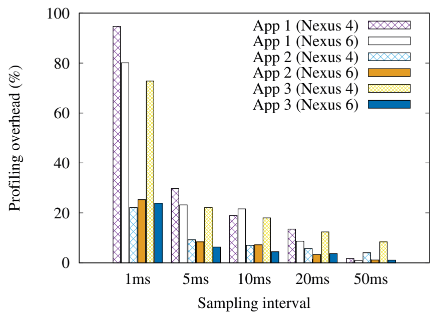


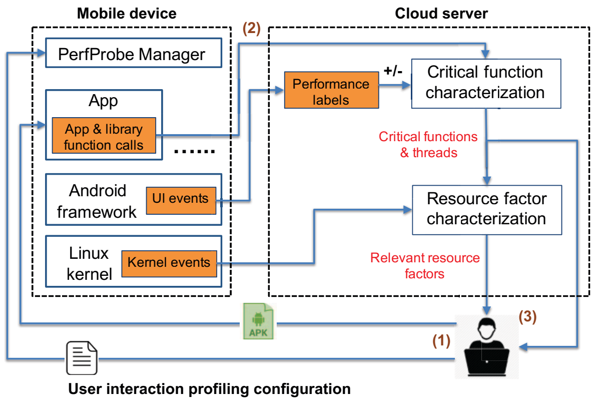
Fig. 2. Profiling in different sampling intervals and hardware platforms

functions), and pinpoints what system resources (e.g., CPU, network, disk, etc.) they interact with cause their running time variance (a.k.a., *relevant resource factors*). In above example, PerfProbe further pinpoints CPU as the resource bottleneck for SCrypt.scryptN and disk I/O for Posix.readBytes (Table II).

B. Profiling Challenge

Android's built-in profiler *Traceview* [35] (integrated in CPU profiler [20] in latest Android) provides runtime visibility of an app's call stack and can be used for characterizing critical functions. While its sampling mode [20], which captures the call stack at fixed *sampling intervals*, is suggested for reducing the performance impact to apps, if it is kept always-on for profiling an actively used app, the app is likely to become unresponsive and throw an ANR error. Thus, we propose to support event-triggered profiling on only developer-configured user interactions (§III-A). Also, our empirical study indicates that small sampling intervals may still introduce high overhead to the runtime execution, especially when it involves CPU or disk I/O intensive workload. Figure 2 shows the profiling overhead (in relative increase of latency due to profiling) under different sampling intervals when the computation-intensive optical character recognition (OCR) is performed to extract texts from images in 3 popular apps. First, the 10-95% overhead incurred by small sampling intervals are unacceptable for real-world deployment. Second, this large overhead may skew the running time of function calls and affect the accuracy in pinpointing app-layer execution slowdown [8]. For example, profiling of App 3 on Nexus 4 device incurs 2-3x increase in running time of file operations, causing corresponding function calls (with small running time in reality) to be wrongly identified as execution hotspots.

As Figure 2 implies, large sampling intervals may lead to smaller overhead, but by design prevent capturing of function calls completed within a sampling interval and thus hinder fine-grained performance inspection. Moreover, though the profiling overhead commonly decreases when the sampling interval scales up, the performance impact of profiling varies across apps with similar workload and across platforms for a same app. One approach to find a proper sampling interval that preserves sufficient profiling granularity with small runtime overhead is profiling in advance, but becomes hard to scale given the large number of apps and high variety of platforms. To address this challenge, we propose to track the performance impact caused by profiling at runtime and based on which adjust the sampling interval to constrain the profiling overhead to the current app execution below some configurable bound. Our approach is agnostic to apps or platforms and requires no extra manual efforts (§III-B).



Mobile device
Cloud server
PerfProbe Manager
App
App & library function calls
Android framework
UI events
Linux kernel
Kernel events
(2)
Performance labels
+/-
Critical function characterization
Critical functions & threads
Resource factor characterization
Relevant resource factors
APK
(3)
(1)
User interaction profiling configuration

Fig. 3. PerfProbe overview ("+/-" represents good/bad performance labels)

C. System Overview

As illustrated in Figure 3, PerfProbe consists of two key modules: an **on-device performance monitoring module** (§III) and a **problem diagnosis module** deployed on a server (§IV). Its workflow has following steps: 1) App binaries are installed and targeted user interactions are configured on a rooted mobile device running PerfProbe; 2) The on-device PerfProbe manager in the performance monitoring module controls the profiling of preconfigured interactions and records multi-layer runtime traces, which are periodically uploaded to a remote server (e.g., once per day); 3) The problem diagnosis module analyzes traces to detect unpredictable performance slowdown in a user interaction and if any slowdown is detected performs further diagnosis to provide app developers or device vendors with cross-layer diagnosis insights.

III. Performance Monitoring

PerfProbe's performance monitoring module measures the latency of user interactions by instrumenting Android's UI framework to intercept common UI input and update events, and records the app-layer runtime execution using Traceview [35] and system-wide OS events using Panappticon [6], which together form the input to the diagnosis module. To mitigate the runtime overhead caused by this cross-layer monitoring, we make two improvements on existing profiling mechanism in Android.

A. Event-Triggered Profiling

We instrument UI event handlers in Android's framework to monitor user's invocation on UI components of an app in the run time and start the profiler when certain UI component (e.g., a touch button on a particular view) is invoked. The PerfProbe manager provides an interface for developers to configure user interactions to be profiled, by providing the resource ID (which is device independent and determined at compile time) of the UI components for denoting input and output of an interaction, app package name of an interaction, and profiling parameters including the profiler's sampling frequency and profiling duration. In the run time, when a pre-configured input UI component is invoked, an intent is broadcasted and intercepted by PerfProbe manager, which then launches the profiler based on the configuration. Auxiliary information (e.g., timestamp, location, network trace, CPU load, system log) when profiling an interaction can also be optionally recorded by PerfProbe manager. This asynchronous messaging, by separating the app execution from the profiling process, aims to prevent any stall on the app due to the launch of profiling. One concern with this design is that the profiler may miss some early phase of the app execution, since the app does not wait once sending the intent. Through our empirical study on a wide range of apps (§V-B), we validate that profiled events consistently cover key execution of an interaction, since intent messages are received promptly by PerfProbe manager and profiling starts immediately after a user input is performed.

B. Adaptation of Sampling Intervals

Android's profiler in sampling mode runs as a background thread spawned from an app and periodically (at sampling interval) records the call stack of each thread in an app process sequentially, during which the whole app process (i.e., all its threads) is paused. This pause is the major source of overhead in profiling an app. Due to this design, the pause time depends on the number of threads in an app process and also the running time of the profiler thread, which can be affected by runtime resources of the platform. Our empirical study on apps of different categories shows that the pause time for one sampling may vary from several to hundreds of milliseconds.

Following the intuition that profiling should be made less frequent to cause shorter pause to an app when the app is performing resource-intensive operations, we propose to adjust the sampling interval at runtime based on the pause duration observed in most recent profiling and the computation intensity of current execution in an app. Based on our observation on the source overhead, we define the *relative profiling overhead* (i.e., the percentage of increase in app latency due to pause for profiling) as $O(n) = \frac{P(n)}{S(n)+P(n)}$, where $P(n)$ denotes the observed app pause duration, $S(n)$ denotes the sampling interval for n^{th} profiling round. To limit the profiling overhead, $O(n+1)$, below some configurable bound during the intervals when the profiled app will be experiencing high load, we determine a new sampling interval $S(n+1)$ using the following equations with a user configurable bound, denoted as T $(0 < T \leq 1)$. Parameter T in our experiments is set to 0.03. Note that we also need to ensure that the new sampling interval is not shorter than the current pause duration.

$$S(n+1) = \begin{cases} max(S(n), P(n), \frac{P(n)}{T} - P(n)), \text{if high load} \\ max(P(n), min(S(n), \frac{P(n)}{T} - P(n))), \text{otherwise} \end{cases}$$

Following this adaptation model, small sampling intervals (e.g., 1ms) are initialized when profiling starts and the sampling interval is updated after each sampling. In our current design, an app is classified as at high load if its total CPU usage time across multiple cores in the most recent sampling round exceeds the sampling interval.

IV. Problem Diagnosis

As show in Figure 3, the input to the diagnosis module includes function call profiles, UI event logs and OS event traces from deployed devices. User-perceived latencies of an interaction can be determined from UI event logs. Performance labels, with the labeling criteria specified by developers, indicates the occurrence of slowdown in one run based on the distribution of all measured latencies of an interaction. In our evaluation, we use a binary indicator for labeling: given runs of an interaction found with long tail latency doubling or multiplying average latency, any run with perceived latency higher than a threshold is associated with a bad performance label and otherwise a good label. If any unpredictable slowdown is detected, a two-step trace-based diagnosis (detailed as follows) is performed to provide human developers with app and OS-layer diagnostic insights and facilitate root cause identification.

A. Approach Overview

Running on a cloud server, the diagnosis module performs trace analysis by first zooming into an app-level program execution and then inspecting its interaction with OS in two sequential steps, in order to gain holistic insights on the source of problem at app program level and the cause of problem at system level. Specifically, as illustrated in Figure 3, the first step takes the performance labels and app and library function call trace for many runs of an interaction as input, and pinpoints a small subset of functions (a.k.a., critical functions) within the function call trace that are most accountable for the performance variance. For each critical function, its executing thread and time intervals are also generated in the output. The second step leverages the output of the first step and OS event traces as input to extract runtime resource usage features relevant to the execution of each critical function, including but not limited to CPU, network, disk resource usage and IPC usage, and associates each critical function to one or several resource usage features based on correlation explain what causes its execution slowdown. Finally, both the critical functions and their relevant resource factors are presented to human developers to guide their further root cause analysis of the performance variance. We select functions as one diagnosis output because they are program semantic-rich and easy-to-reason for developers, and resource factors as the other because they are usually related to the root cause of performance variance. The remaining subsections present the technical details of this two-step analysis.

B. Critical Function Characterization

In critical function characterization, a candidate set of critical functions is first selected. Decision tree based learning, taking each run as one data sample, in which the total execution time of each critical function candidate acts as an input feature and the performance label as an input label, is performed to identify a small set of critical functions with execution time correlated to the performance variance. In the generated tree, each node corresponds to a critical function.

Critical function candidate. A *critical function* satisfies the following requirements:

- A critical function consumes a significant amount of execution time of an interaction, based on the intuition that time-consuming functions tend to cause a stronger impact on the user-perceived latency of an interaction.
- The execution time of a critical function varies significantly between runs with different performance labels, based on the intuition that the extra time spent in that function will contribute to the overall user-perceived latency if it causes the performance slowdown.

To fulfill the first requirement, we compute the total time spent in each function for each run based on the function call trace. Then we pick the top-K functions with longest total execution time in each run and merge them across runs to form a candidate set of critical functions. To satisfy the second requirement, we aim to select a small set of functions from the candidate set such that their total execution time in runs with performance slowdown is consistently longer than that in runs without performance slowdown. In other words, we can apply conjunction on this set of functions to discriminate runs with good performance labels from those with bad labels.

Identifying critical functions. We construct a decision tree to understand what functions are most correlated to the performance variance. We use decision trees for two main reasons. First, a decision tree using a compact combination of features selected from a large feature set naturally determines a linear boundary to separate data samples with different labels. Second, a decision tree well depicts the preconditions for performance slowdown: with each node identifying a critical function, given a path from the root node to some leaf containing performance slowdown instances alone, the conjunction of nodes along this path defines a precondition for the slowdown, and the disjunction of all such paths define a set of preconditions under which performance slowdown occurs.

Decision tree details. We use mutual information gain as the criteria for node selection. Before the decision tree characterization, we first apply function pruning by evaluating the relative difference of the total time feature for each function f in the candidate set: prune f if (m^f_- - m^f_+) $<\alpha$ (p^f_{95} - p^f_5). The relative difference is the absolute difference of means over the central range of a feature's values in groups by performance labels (i.e., m^f_-, m^f_+), where central range is the difference between the two 95-percentile values p^f_{95} and p^f_5. α is set as 0.1 by empirical study. The remaining functions in the candidate set provide the input features, which along with performance labels will be used for feature selection to generate splitting nodes of a decision tree. In feature selection, given a set of features with equally highest mutual information gain, we select one with largest relative difference as the splitting node. Moreover, to reduce the variance and avoid overfitting, we stop generating a splitting node when it reaches certain depth or contains too few samples. Through our empirical study, we find that the depth of a generated decision tree does not usually go beyond 4 when data samples are completely separated. We also exclude a node from being considered as a critical function when it contains too few samples, or when its split gap (i.e., minimum distance between good and bad samples) is not significant, or when the latency of its corresponding function is not significant. We leverage the *scikit-learn* library to implement our decision tree and configure the decision tree to compute *gini* and apply a best split heuristic for feature selection [38].

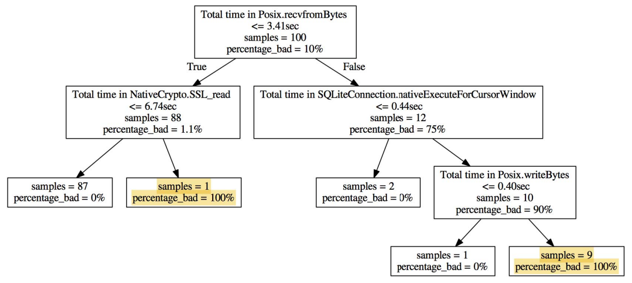


Fig. 4. Decision tree to characterize critical functions in Vine interaction (two slowdown preconditions as conjunctions of nodes from the root to a highlighted leaf, recvfromBytes and SSL_read identified as critical functions, the other two nodes are pruned because the latency of their corresponding functions is insignificant)

An illustrative example. We use the *Vine* interaction to showcase our diagnosis flow. From the deployment study, we observe 10 out of 100 runs of *Vine* interaction (listed in Table VI) show user waiting 2x long as the median. The decision tree generated from critical function characterization is shown in Figure 4 that identifies 2 critical functions, recvfromBytes and SSL_read.

C. Resource Factor Characterization

The resource factor characterization takes the output of the critical function characterization as input and aims to understand what resource usage causes execution slowdown in each critical function. To achieve that, we first profile the resource usage for each critical function by identifying its execution intervals, under which key resource usage features are extracted. Decision tree learning is then applied on each critical function across runs as data samples, with resource usage features as input features and a binary indicator for the execution time of the critical function as input labels, to identify the resource factors relevant to the slowdown of that critical function.

Identifying relevant execution intervals. Given a set of critical functions, we define a thread executing it as a *critical thread* and its duration as a *relevant execution interval*. Thus, a relevant execution interval corresponds to a critical function and a critical thread. The next step of diagnosis is to narrow down to these execution intervals and reason why slowdown happens in each critical function.

To determine the most relevant resource factors, we construct a set of resource features for each critical function. We summarize key resource features in Table I. Note that the resource usage feature set is extensible for the characterization. For each run, we then sum up each type of resource usage under all relevant execution intervals to form one resource feature for a critical function.

Extracting resource usage features. To compute how much time is spent in CPU running state, in interruptible/uninterruptible sleep state or in the ready state waiting for context switch-in, we rely on the context switch events within a relevant execution interval. To extract network or disk blocking time, we analyze the I/O blocking events within a relevant execution interval. To obtain IPC wait time, we compute the waiting time between each binder request and response within a relevant execution interval. We also compute the average CPU frequency when a thread is occupying CPU across its relevant execution intervals based on frequency governor events.

Pinpointing relevant resource factors. The resource factor characterization of each critical function is also achieved through decision tree learning similar to that in the critical function characterization, in which a corresponding node for a critical function contains a subset of runs that becomes input samples to the characterization at this step. The input features of each sample consists of the key resource usage features for a critical function. In the critical function characterization, a threshold on execution time has also been determined for each critical function, which is used for labeling the input data in this step. In other words, the label indicates whether execution slowdown occurs in a critical function. The same feature pruning and node selection technique applied to critical function characterization are followed to construct a decision tree, in which each node identifying a resource usage feature relevant to the slowdown of a critical function. Using the example in Figure 4, all 100 samples (88 positive, 12 negative) are used to characterize the resource factor for recvfromBytes, resulting in a decision tree with the network blocking time as its root (i.e., relevant resource factor). Analysis of 88 samples in the left branch (87 positive, 1 negative) pinpoints interruptible sleep time as the relevant resource factor for SSL_read.

V. Evaluation

We perform controlled experiments on 5 real Android apps with synthetic performance variance introduced by perturbing different system resources or triggering a programming mistake into app source code (summarized in Table II) in some runs. Diagnosis results demonstrate that our diagnosis approach can always localize functions with injected faults or correctly pinpoints the perturbed system resource.

We also conduct a real-world deployment and diagnosis study to answer following questions. How useful is PerfProbe in guiding root cause diagnosis and code fixing of unpredictable performance issues for real-world app developers (§V-A)? How effective is PerfProbe in diagnosing real-world unpredictable performance issues with popular Android apps (§V-B & §V-C)? What benefit can PerfProbe's cross-layer characterization achieve compared to existing diagnosis approaches (e.g., monitoring system calls, resource profiling) (§V-D)? How much overhead can PerfProbe incur to a mobile device and how much performance impact can our adaptive sampling reduce (§V-E)?

A. Android App Developer Study

We apply PerfProbe to diagnose user-reported performance problems in 6 open-source Android apps (user rating above 3.5 and over 50K downloads) and report our findings to their developers for feedbacks. Specifically, we mimic app developers to conduct followup debugging based on the critical functions and relevant resource factors output from PerfProbe. To quantify the extra manual effort for code inspection with hints from PerfProbe, we define a metric *relative extra effort* based on prior works [64], [71], as the ratio of the portion of app source code we manually inspected based on the critical functions from PerfProbe to the portion of app source code invoked in the run time (i.e., baseline).

Table III shows the relative extra effort and summarizes the root cause findings (detailed in app's GitHub issue tracker) for each issue. With PerfProbe, less than 3% of the executed source code needs to be inspected and the pinpointed resource factors give direct explanation to the running time variance in pinpointed critical functions. We reported our findings through GitHub's issue tracker to app developers and obtained acknowledgment from developers of 3 apps (highlighted in Table III). Based on followup analysis using PerfProbe's output, we also suggested feasible optimizing solutions to some issues. The **iNaturalist** and **Riot** developer invited us to submit pull requests for our suggested solutions. Our suggested strategy for iNaturalist [39], implemented in 90 lines of code, has been adopted by its developer (detailed as follows).

Case study. iNaturalist app (over 500K downloads in Google Play) enables users to view or upload plant and animal observations. PerfProbe pinpoints Posix.recvfromBytes (invoked by getAllGuides) as the critical function, with network blocking time as its relevant resource factor. To trace the source of network blocking, we investigate the definition of getAllGuides and discover that a series of HTTP requests are issued sequentially to retrieve many (>1000) JSON objects. As we observe that users cannot view all loaded items from the UI screen, we suggest limiting the number of JSON objects to be retrieved through HTTP requests and adding a "Load more" option in the UI for users to choose whether to continue loading

Resource feature	Description
CPU usage time	Time spent in running state by thread T
CPU wait time	Time spent in the ready state waiting for CPU by thread T
CPU frequency	Average CPU frequency when thread T is running
Interruptible sleep time	Time when thread T is in interruptible sleep
Uninterruptible sleep time	Time when thread T is in uninterruptible sleep
Network blocking time	Time when thread T is blocking for network I/O
Disk blocking time	Time when thread T is blocking for local disk I/O
IPC wait time	Time spent in inter-process communication by thread T

TABLE I
RESOURCE USAGE FEATURES

App	Interaction	Injected problem	Critical function	Relevant resource factor
Download Manager [5]	Download a file	Network delay	`Posix.recvfromBytes`	Network blocking time
H&M [15]	View an item			
CNET [4]	View a post			
Sudoku Solver [25]	Solve a grid	Programming flaw	`SudokuCore.solveMethodOptimised`	CPU usage time
SSE [40]	Encrypt a file from SD card	Background load	`SCrypt.scryptN`	CPU wait time
		Throttled disk access	`Posix.readBytes`	Disk blocking time

TABLE II
DIAGNOSIS RESULTS FOR SYNTHETIC PERFORMANCE ISSUES

App	Interaction	Root cause summary	Relative extra effort
iNaturalist [16]	Click Guides tab	Overloaded sequential web requests [39] $*$	0.45%
Riot [36]	Open chat room	Web requests and computation delay for bitmap decoding [41] $*$	2.43%
K9 Mail [26]	Sync mailbox	Occasional loss and re-establishment of IMAP connection [24] $*$	0.70%
c:geo [3]	Search nearby cache	Delay for sequential web requests [22]	0.33%
GeoHash Droid [10]	Launch app	Location query and computation delay for map rendering [42]	2.78%
Tomahawk Player [47]	Search songs keyword	Web server unavailability [23]	0.81%

TABLE III
SUMMARY OF DIAGNOSIS REPORTING: $*$ INDICATES OUR REPORT IS ACKNOWLEDGED BY APP DEVELOPER

more new items in order to reduce the user waiting time for UI update [39]. We implemented our suggested strategy [39] by adding 90 lines of code and validated that it reduces the user-waiting time for this interaction by 86%. Eventually, the developer adopted our suggestion and changed the app to load only the first page of items for better interactive experience [7].

B. Real-World Deployment

To diagnose unpredictable performance slowdown issues in the real world, we deploy PerfProbe's monitoring module on Nexus 4 and 6 devices to monitor common user interactions for a wide range of popular Android apps. We select top-ranked Android apps [2], [12] from different categories and obtain in total 100 popular apps (summarized in Table IV). For each app, we identified a common interaction based on our domain knowledge of an app and configure PerfProbe manager to monitor it. In each deployment run, a subset of selected apps were installed on a test device and replaced by another subset in the next run. To mimic real-world daily usage, a device was brought to different locations, including an office, campus, and residential environment (all with WiFi access), where UI inputs were automatically replayed using `UIAutomator` [48] to launch an interaction randomly picked from the preconfigured ones. During the deployment, each preconfigured interaction was performed for sufficiently many times (65∼110 runs).

Out of the 100 apps, we discover 11 apps (spanning 6 main categories in Table IV) in which tail latencies are 1.5∼8x as long as the median latency. Figure 5 shows the distribution of waiting time for the user interactions of these apps (listed in Table V). Root cause analysis based on PerfProbe's cross-layer trace diagnosis is presented in §V-C.

Category	# of apps	Downloads
Tool	32	500K∼500M
Shopping	15	10M∼500M
News	10	50K∼1B
Social	8	50M∼5B
Media	6	100M∼500M
Navigation	5	100M∼5B
Other	24	50K∼500M

TABLE IV
ANDROID APPS FOR PERFORMANCE MONITORING IN THE REAL WORLD

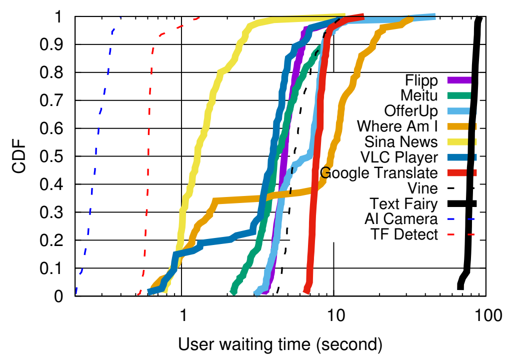


Fig. 5. Unpredictable performance slowdown in popular Android apps

App	Interaction	Version
Vine [49]	Launch app to play a video	5.18.0
Flipp [9]	Launch app to load flyers	5.0.1
OfferUp [31]	Launch app to load deals	2.2.25
Sina News [37]	Click a bookmarked post	4.9.5
Google Translate [14]	Translate texts in an image	5.5.0
VLC Player [50]	Play a video from SD card	2.0.6
Meitu [28]	Enhance a photo in SD card	5.1.9.1
Where Am I [51]	View current address	1.14
Text Fairy [44]	Extract texts in an image	3.0.8
AI Camera [1]	Detect objects in camera	Demo
TF Detect [45]	Detect objects in camera	1.0.0

TABLE V
ANDROID APPS WITH UNPREDICTABLE PERFORMANCE SLOWDOWN

C. Diagnosis of Performance Issues

We apply PerfProbe to diagnose performance variance (listed in Table V) uncovered in our deployment study (§V-B). We label a run as performance slowdown if its waiting time is longer than the median waiting time by one or two standard deviation, depending on the skewness of the distribution of the user waiting time. Table VI summarizes the diagnosis output of PerfProbe for each case, with their diagnosis details documented in an anonymous website [33]. We also perform further analysis and validation based on PerfProbe's diagnosis findings as follows.

- For CPU frequency bound, by reconfiguring existing userspace parameters of the Dynamic Voltage and Frequency Scaling (DVFS) governors [27], the tail latency of 3 CPU-bounded interactions are reduced by 32-40%.
- For disk I/O factor, by applying a common tweak of a system parameter to boost the access speed of on-device SD card, the tail latency of 2 disk-bounded interactions is reduced by near 50%.
- For network or server-side factor, we investigate network trace captured by *tcpdump* for further validation.
- For GPS problem, we trace the destination of the pinpointed inter-process communication to locate GPS-related system process.

1) DVFS governor issue: This case study presents diagnosis and validation on 3 apps where performance is affected by the computation speed controlled by DVFS governors.

Problem diagnosis. When performing offline OCR-based text extraction using *Text Fairy* on a Nexus 4 device (1.5 GHz quad-core Krait), as shown in Figure 5, a user may wait extra 20 seconds (compared to 70 seconds in fast runs) for English texts to be extracted from an image. PerfProbe identifies TessBaseAPI.nativeGetHOCRText, defined in Google's Tesseract OCR API [43] and invoked by a worker thread, as a critical function for all slowdown instances. Furthermore, the average CPU frequency is pinpointed as its relevant resource factor. Based on the split threshold from the resource factor characterization, the average frequency along the execution of the critical fuction reaches above 1.2GHz for all fast runs. Figure 6 shows the time series of the CPU frequency for the core on which the critical function is executed for a randomly picked fast and slow run in traces collected from our real-world deployment. We can clearly see that the frequency scaling

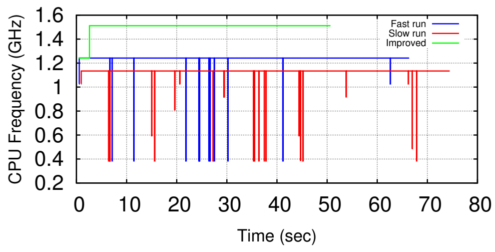


Fig. 6. CPU frquency for executing critical function over time

gets stuck at 1.1GHz in some runs when executing the critical function and thus leads to slowdown of the overall interaction.

Root cause validation. To validate the root cause in frequency governor, we intentionally increase the upper frequency limit for scaling (i.e., *scaling_max_freq*) to 1512MHz (maximum available scaling frequency on Nexus 4) and also change the governor type from *ondemand* (default governor type in Nexus 4) to *performance* right before an interaction is performed. As a result, the tail user waiting time is reduced by 40% (to 53 seconds). Interestingly, even when the *ondemand* or *interactive* governor is used, the tail latency can be reduced to 57 seconds by presetting *scaling_max_freq* to 1512MHz. Figure 6 indicates the execution time for the critical function is significantly reduced when its execution finishes at maximum frequency. Note that this strategy is unrealistic as a long-term strategy given the energy and temperature constraint.

Object detection apps. Our diagnosis on *AI Camera* and *TF Detect* also reveals CPU frequency as the resource bottleneck for running the pinpointed critical functions for detecting objects in a camera frame on a Nexus 6 device (2.7 GHz quad-core Krait 450). Further investigation on the CPU states leads us to the interactive frequency governor policy: *scaling_max_freq* is capped at 1958MHz when either app is running. To improve their object detection performance, we initialize *scaling_max_freq* as 2649MHz (maximum available scaling frequency on Nexus 6) and set *ondemand* governor at app launch. As a result, the per-frame object detection latency is reduced by 32% on **AI Camera** and 40.6% on **TF Detect**.

2) Disk hardware issue: This case study presents diagnosis and validation on 2 apps with performance degradation caused by disk I/O.

Problem diagnosis. As shown in Figure 5, the latency for playing an HD video (of size 35.5MB) from SD card in *VLC Player* or processing a photo (of size 1.93MB) in SD card in *Meitu* can take more than 2.5x of median (both around 4 seconds) on a Nexus 4 device. PerfProbe reveals that 62.5% of slowdown instances are characterized by the critical functionVideoPlayerActivity.onCreate on the main thread to load the video playing activity, while the rest happen in executing the other critical function MediaCodec.start on the VLC object thread. Both are bounded by disk blocking. Similarly, PerfProbe pinpoints the critical function SmartBeautifyActivity.OnCreate and attributes its slowdown to slow disk I/O for

App	Critical functions	Relevant resource factors	Root cause summary
Vine	`Posix.recvfromBytes` `NativeCrypto.SSL_read`	Network blocking time Interruptible sleep time	network or server-side delay
Sina News	`Posix.recvfromBytes`	Network blocking time	network or server-side delay
Flipp OfferUp Google Translate	`NativeCrypto.SSL_read`	Interruptible sleep time	network or server-side delay
VLC Player	`VideoPlayerActivity.onCreate` `MediaCodec.start`	Disk blocking time Disk blocking time	Slow SD card read speed
Meitu	`SmartBeautifyActivity.onCreate`	Disk blocking time	Slow SD card read speed
Where Am I	`MessageQueue.next`	IPC wait time	GPS signal locking delay
Text Fairy	`TessBaseAPI.nativeGetHOCRText`	CPU frequency	Frequency capped by governor policy
AI Camera	`classificationFromCaffe2`	CPU frequency	Frequency capped by governor policy
TF Detect	`org.tensorflow.Senssion.run`	CPU frequency	Frequency capped by governor policy

TABLE VI
DIAGNOSIS OUTPUT FOR REAL-WORLD PERFORMANCE SLOWDOWN

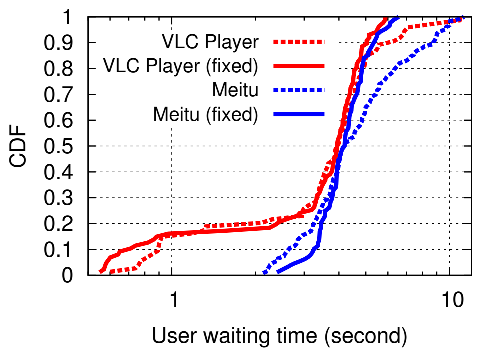


Fig. 7. Performance improvement by mitigating disk I/O bottleneck

the photo enhancing interaction in *Meitu*.
Root cause validation. To validate the disk I/O bottleneck, we increase the SD card access speed for Android devices by tuning the read-ahead buffer [17], [19], [18]. The read-ahead buffer defines the size of a disk block to be loaded into memory for each read. For long sequential file read operation (e.g., copying large file from SD card), having a larger read-ahead buffer will usually speed up the read process. We find its size is set to 128KB by default on Nexus 4 phones and can be configured through the *sysfs* interface. Through empirical tuning, we find the SD card read speed on a Nexus 4 phone is improved significantly as the size of read-ahead buffer increases from 128KB to 2048KB. Therefore, we reconfigure the read-ahead cache size as 2048KB on our Nexus 4 test device, while keeping the other setup unchanged, to perform controlled testing on both interactions. Figure 7 shows the improvement of user waiting time for both apps after increasing the read-ahead cache. Specifically, the tail user waiting time is reduced by 45% (to below 6 seconds) for **VLC Player** and by 42% (to below 7 seconds) for **Meitu**.

D. PerfProbe's Benefit Highlight

One baseline for performance diagnosis is identifying resource bottleneck from the overall resource usage throughout the whole execution of an interaction. This approach leads to misidentification of resource bottlenecks in 8 out of 22 cases, including **Text Fairy**, **AI Camera**, **TF Detect** (true cause is the CPU frequency cap set by the DVFS governor), **VLC Player** (true cause is disk I/O delay), **Where Am I** (true cause is GPS handling delay), **Riot** (true cause is server-side delay and waiting time on CPU resource) and **K9 mail**, **iNaturalist** (true cause for both is server-side delay). Therefore, critical function characterization is indeed necessary for achieving high accuracy in pinpointing relevant resource factors.

Another baseline approach is monitoring system calls. For issues due to the DVFS governor, while system calls can hardly reveal frequency change on different CPU cores, identifying the computation bottleneck caused by frequency governor can be inaccurate even when general resource profiling is used. For issues due to the disk I/O bottleneck, we try profiling system calls using *strace* on both cases to check if the run time of disk-related system calls has significant variance to account for the bottleneck, but find that the total run time spent in system calls for both cases take up less than 5% of the overall latency and that disk-related calls do not show significant variance in the run time.

E. Runtime Impact & System Overhead

App performance impact. To evaluate the benefit of our adaptive sampling mechanism for app profiling, we conduct a controlled experiment to measure profiling impact on app performance (i.e., increase of user-perceived latency in PerfProbe) under adaptive and fixed 10ms/20ms sampling interval (baseline). Compared to fixed sampling intervals, our adaptive sampling mechanism increases the sampling interval of Traceview when resource-intensive operations are ongoing for some apps and converges at a larger interval to maintain low runtime overhead. The sampling interval decreases and converges to 5-50ms once those expensive operations finish. For interactions in Table II, III, VI, adaptive sampling incurs at most 3.5% increase of the median latency of an interaction, while fixed sampling intervals incur 3-22% increase. Note that this 3.5% increase causes negligible effect to detecting and diagnosing performance slowdown with at least 50% increase of the overall latency. Though adaptive sampling may miss function calls with small running time due to reduced sampling frequency for accommodating resource-intensive operations, the output critical functions and relevant resource factors for all studied interactions remain consistent with fixed sampling intervals and adaptive sampling, mainly because only top-K time-consuming functions are taken as input for critical function characterization.

CPU & memory overhead. In our current prototype, PerfProbe manager uses a 10MB memory buffer for logging OS kernel events, a 15MB buffer for Android framework events and Traceview's default 8MB buffer for an app's call stack. PerfProbe showed no noticeable increase in CPU or memory usage in our deployment. We also measure the logging time (averaged over 100K executions): logging a kernel event takes less than 1 microsecond and logging a framework event takes 3.2 microseconds in an instrumented function.

Storage & energy overhead. In our current prototype, function call and OS event traces are stored in the local storage of a device and periodically uploaded to a remote server when certain network (e.g., in WiFi network) and/or battery condition (e.g, charging state) is met. We conduct a controlled measurement on a Nexus 4 device: based on the PhoneLab [34] data we find that 85% of the time real users perform no more than 210 interactions per day, so we replay 210 interactions on the 11 apps with performance slowdown uncovered from the deployment study (§ V-B) using `UIAutomator` [48] for 5 times with or without PerfProbe enabled, and measure the average storage and energy overhead caused by PerfProbe. Measurement results show that each interaction incurs 10KB$\sim$500KB function call trace and on average 2.2MB OS event trace. Given the growing capacity of mobile device storage and high availability of WiFi networks for trace uploading, this storage overhead is acceptable for real-world deployment of PerfProbe. Moreover, PerfProbe incurs only 1.9% energy overhead to a smartphone device.

VI. Related Work

Mobile performance diagnosis: Functionality bugs, crashes or performance issues in mobile apps can be uncovered through automated UI testing [52], [53], [74], [54], [80], [63], [56], [62], [75], [76], [70], runtime analysis [68], [81], [85], [56], [65], [82], [55] or static analysis [69], [73]. First, static analysis is not proper for uncovering runtime cause of performance issues since it does not capture runtime contexts. Second, PerfProbe is complementary to existing app testing and profiling tools. PerfProbe can be used to monitor user interactions with performance issues detected from testing tools and perform further trace-based diagnosis to help understand the root cause of the issues. Moreover, PerfProbe complements existing profiling-based diagnosis systems in providing an automated systematic approach that performs lightweight profiling and holistic analysis of app and OS-layer runtime events to guide the root cause diagnosis of a broad category of real-world performance issues, which is less extensively explored. PerfProbe's cross-layer characterization can provide systematic root cause reasoning and validation on potentially runtime-expensive operations detected by existing tools [73], [84], [82]. PerfProbe leverages Panappticon's instrumentation framework [85] to record OS events and Traceview [35], [20] to capture an app's call stack, but provides a new diagnosis approach by systematically associating such cross-layer runtime information to gain holistic understanding on the cause of performance slowdown. App instrumentation [81], [61], [67] may achieve lower monitoring overhead compared to Traceview, but requires specification from app developers and is thus unscalable for systematic localization of code-level performance variance (i.e., critical function) in arbitrary apps.

Cross-layer analysis: Cross-layer analysis was applied to investigate the performance of wearable systems [72] and smartphone platforms [79], [83], [56]. ARO [79] mainly focuses on radio resource efficiency problems rather than app QoE, while QoE Doctor cannot break down app latencies into more fine-grained operations, such as disk access and inter-process communication that PerfProbe can address, due to the lack of system or application event profiling.

Performance diagnosis using ML: Previous works [57], [77] apply machine learning techniques on system logs to diagnose performance problems in distributed systems. Decision tree is also used for per-application QoS-to-QoE mapping for QoE inference of mobile apps [78]. Though PerfProbe also leverages decision tree learning, the use of decision tree in the two-step analysis is new and different from them.

VII. Discussion

We discuss the scope and limitations of PerfProbe and the threats to the validity of our experiments. First, PerfProbe's approach is general to other mobile platforms, e.g., iOS using Instruments [21]. Second, current PerfProbe targets at performance slowdown occurring occasionally. To diagnose slowdown that consistently occurs, additional mechanism, such as resource amplification [73], [84], [82], can be leveraged to amplify resource-intensive operations for enabling PerfProbe's differential analysis. Third, as the output of PerfProbe, the critical functions and relevant resource factors facilitate the root cause analysis of performance issues and may not be the actual root cause. Forth, the synthetic performance issues (Table II) are mainly for validating the accuracy in pinpointing critical functions and relevant resource factors, but the injected performance causes are ell encountered in our real-world app deployment and effectively pinpointed by PerfProbe.

VIII. Conclusion

PerfProbe is a mobile performance diagnosis framework that associates app and OS-layer runtime information in a lightweight manner to provide holistic, cross-layer insights to the root cause of unpredictable performance slowdown in real-world usage. PerfProbe effectively pinpoints code-level or system resource-layer factors for performance slowdown in 22 Android apps and guides real-world Android developers' code-level fixing to significantly improve app responsiveness.

Acknowledgements

We would like to thank Qi Alfred Chen, Yihua Guo, Yunhan Jia, Yuru Shao, Shichang Xu, Yikai Lin and anonymous reviewers for useful feedbacks. This work is partially funded by NSF under the grants CNS-1629763 and CCF-1628991.

References

[1] AI Camera Demo App. https://caffe2.ai/docs/AI-Camera-demo-android.html.

[2] App Annie - Top Apps on Google Play. https://www.appannie.com/apps/google-play/top/.

[3] c:geo. https://play.google.com/store/apps/details?id=cgeo.geocaching.

[4] CNET. https://play.google.com/store/apps/details?id=com.treemolabs.apps.cnet.

[5] Download Manager. https://play.google.com/store/apps/details?id=com.acr.androiddownloadmanager.

[6] Event-Based Tracing to Measure Android Application and Platform Performance. https://github.com/EmbeddedAtUM/panappticon.

[7] Faster loading for all guides - just show the first page of results. https://github.com/inaturalist/iNaturalistAndroid/commit/ea6d2892d545cc6ae203edcf6823f0200d446fd0.

[8] First Impressions Count: Boost Your App's Start-Up Time. https://developer.amazon.com/blogs/post/Tx1RLL07TPIH1RJ/First-Impressions-Count-Boost-Your-App-s-Start-Up-Time.html.

[9] Flipp - Weekly Ads & Coupons. https://play.google.com/store/apps/details?id=com.wishabi.flipp.

[10] Geohash Droid. https://play.google.com/store/apps/details?id=net.exclaimindustries.geohashdroid.

[11] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[12] Google Play. https://play.google.com/store.

[13] Google play: number of android app downloads 2010-2016. https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

[14] Google Translate. https://play.google.com/store/apps/details?id=com.google.android.apps.translate.

[15] H&M. https://play.google.com/store/apps/details?id=com.hm.

[16] iNaturalist. https://play.google.com/store/apps/details?id=org.inaturalist.android.

[17] Increase Read Cache for better SD Card access. https://forum.xda-developers.com/showthread.php?t=1010807.

[18] Increase the read/write speed of the SD card on your rooted Android tablet. https://goo.gl/PNyYHa.

[19] Increase Your SD Card Read Speeds By 100-200% With A Simple Tweak. https://goo.gl/Hpce69.

[20] Inspect CPU activity with CPU Profiler. https://developer.android.com/studio/profile/cpu-profiler.

[21] Instruments User Guide. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/.

[22] Intermittent slowdown for loading nearby cache. https://github.com/cgeo/cgeo/issues/6632.

[23] Intermittently long waiting time for search result to be displayed. https://github.com/tomahawk-player/tomahawk-android/issues/86.

[24] Intermittently slowness for mailbox refreshing. https://github.com/k9mail/k-9/issues/2575.

[25] JSON Sudoku Solver. https://play.google.com/store/apps/details?id=com.musevisions.android.SudokuSolver.

[26] K-9 Mail. https://play.google.com/store/apps/details?id=com.fsck.k9.

[27] Linux CPUFreq User Guide. https://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt.

[28] Meitu - Beauty Cam, Easy Photo Editor. https://play.google.com/store/apps/details?id=com.mt.mtxx.mtxx.

[29] NimbleDroid Blog. http://blog.nimbledroid.com/.

[30] Number of android applications. http://www.appbrain.com/stats/number-of-android-apps.

[31] OfferUp - Buy. Sell. OfferUp. https://play.google.com/store/apps/details?id=com.offerup.

[32] OProfile. http://oprofile.sourceforge.net/.

[33] PerfProbe Case Study. https://sites.google.com/site/perfprobe/case.

[34] PhoneLab: A Smartphone Platform Testbed. https://www.phone-lab.org/.

[35] Profiling with Traceview and dmtracedump. https://developer.android.com/studio/profile/traceview.html.

[36] Riot.im - open team collaboration. https://play.google.com/store/apps/details?id=im.vector.alpha.

[37] Sina News. https://play.google.com/store/apps/details?id=com.sina.news.

[38] sklearn.tree.decisiontreeclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[39] Slow loading of ALL Guides tab. https://github.com/inaturalist/iNaturalistAndroid/issues/375.

[40] SSE - Universal Encryption App. https://play.google.com/store/apps/details?id=com.paranoiaworks.unicus.android.sse.

[41] Suggestion for improving directory loading performance. https://github.com/vector-im/riot-android/issues/1473.

[42] Suggestion for improving map rendering performance during app launch. https://github.com/CaptainSpam/geohashdroid/issues/67.

[43] Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr/tesseract.

[44] Text Fairy (OCR Text Scanner). https://play.google.com/store/apps/details?id=com.renard.ocr.

[45] TF Detect. https://play.google.com/store/apps/details?id=org.tensorflow.app&hl=en_US.

[46] The RAIL Performance Model. https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail.

[47] Tomahawk Player Beta. https://play.google.com/store/apps/details?id=org.tomahawk.tomahawk_android.

[48] UI Automator. https://developer.android.com/training/testing/ui-automator.

[49] Vine Camera. https://play.google.com/store/apps/details?id=co.vine.android.

[50] VLC for Android. https://play.google.com/store/apps/details?id=org.videolan.vlc.

[51] Where Am I? https://play.google.com/store/apps/details?id=com.ejelta.whereami.

[52] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, 2012.

[53] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[54] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, 2013.

[55] M. Brocanelli and X. Wang. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *Proc. of the 2014 ACM European Conference on Computer Systems*, EuroSys '18, 2018.

[56] 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 *Proc. of IMC*, 2014.

[57] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, 2004.

[58] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, 2010.

[59] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, 2015.

[60] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[61] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, 2013.

[62] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[63] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proc. of the 2014 ACM European Conference on Computer Systems*, EuroSys '14, 2014.

[64] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, 2005.

[65] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu. DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[66] N. Khadke, M. P. Kasick, S. P. Kavulya, J. Tan, and P. Narasimhan. Transparent system call based performance debugging for cloud computing. In *Proceedings of the 2012 Workshop on Managing Systems Automatically and Dynamically*, 2012.

[67] C. H. Kim, J. Rhee, K. H. Lee, X. Zhang, and D. Xu. PerfGuard: Binary-centric Application Performance Monitoring in Production Environments. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, 2016.

[68] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu. IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, 2014.

[69] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic Performance Prediction for Smartphone Applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, 2013.

[70] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[71] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005.

[72] R. Liu and F. X. Lin. Understanding the Characteristics of Android Wear OS. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, 2016.

[73] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[74] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[75] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

[76] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[77] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[78] A. Nikravesh, D. K. Hong, Q. A. Chen, H. V. Madhyastha, and Z. M. Mao. QoE Inference Without Application Control. In *Proceedings of the 2016 Workshop on QoE-based Analysis and Management of Data Communication Networks*, Internet-QoE '16, 2016.

[79] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, 2011.

[80] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[81] 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'12, 2012.

[82] Y. Wang and A. Rountev. Profiling the Responsiveness of Android Applications via Automated Resource Amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, 2016.

[83] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer Profiling of Android Applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, MobiCom '12, 2012.

[84] S. Yang, D. Yan, and A. Rountev. Testing for Poor Responsiveness in Android Applications. In *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems*, MOBS '13, 2013.

[85] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance. In *Proc. of CODES+ISSS*, 2013.