Towards Measuring Real-World Performance of Android Devices

Pascal Bissig, Gino Brunner, Florian Gubler, Roger Wattenhofer, Andreas Zingg
Department of Electrical Engineering and Information Technology
ETH Zurich, Switzerland
bissigp,brunnegi,flgubler,wattenhofer,zinggand@ethz.ch

***Abstract*—In this paper we investigate how to measure real world performance of Android devices using app start durations. To this end we collect ground truth app start times using an automated mechanical setup. The ground truth start times are highly correlated with the outputs from Android's ActivityManager, which we then use to obtain app start times during normal use on a range of rooted devices. We then predict app start times with supervised learning to detect if device performance has changed over time. We show that training data can be gathered on a small set of rooted devices and then applied to other, non rooted devices. We also present an unsupervised method that can track the evolution of the system performance without requiring root access at all.**

***Index Terms*—Android, mobile, phone, benchmarking, real-world, performance, measurement**

I. Introduction

A responsive and fast mobile user experience is high up on everyone's priority list. Various synthetic and application benchmarks measure the performance of smartphones, but they usually only report numerical scores which are difficult to interpret. While such benchmarks can be very useful to compare phones in a controlled fashion, it is not clear what a score of "91157" actually means for everyday usage. Furthermore, a phone's performance can degrade over time, e.g., due to an increasing number of installed apps or badly optimized firmware updates. Different phones from different manufacturers are not equally affected, and traditional benchmarking applications cannot reflect this.

We launch many apps during daily use. Fast app launch times are therefore important for a good user experience. In this paper we thus focus on app launch times as our real-world performance metric. We also show that when artificial load is added, which noticeably slows down the phone, the app launch times become longer as well. Thus, we will use app launch times as proxy for overall device performance. Tracking how launch times of apps develop over time can also help identify which changes to the system are responsible for slowdowns.

To this end we first gather ground truth app start times. We do this by mechanically activating the touchscreen of phones, while a high speed video recording of the screen allows us to measure the start times of apps precisely albeit manually. To increase the number of starts included in our results, we use a computer vision method that automates the process of measuring app start times.

Next, we show that the ground truth app start times are highly correlated with start times reported by the Android `ActivityManager` (`AM`) [1] on rooted devices. This allows us to perform experiments without requiring a video recording of the screen, thus making it easy to collect data using multiple rooted devices that we use as our daily phones for ten weeks in total. While recording start times provided by the `AM`, we also collect data about CPU and RAM usage as well as `AccessibilityEvents` (`AEs`) [2]. Note that `AEs`, CPU, and RAM usage can be gathered without root access.

Finally, this data is used to predict the start times gathered with the `AM`. We show both supervised and unsupervised methods that can detect changes in device performance based on the types and time stamps of `AccessibilityEvents`. The supervised methods can be trained on a few rooted devices and applied to non-rooted devices with little performance degradation, thus alleviating the need for root privileges on every device that is to be benchmarked. We also investigate using RAM and CPU utilization to measure app launch times.

A recurring theme in our paper is that we need to differentiate between *types* of app launches. According to Google [3] there exist three types of app starts: *cold*, *warm* and *lukewarm*. Since the Android system does not provide any information at runtime about the type of a start, there is no easy way to distinguish these three types. Even more so as *lukewarm* starts are a mixture between *cold* and *warm* starts that is not clearly defined; the Android Memory Management [4] is complex and apps can even be partially cached. Thus, we take a slightly different approach and group app launches into *cold* and *hot starts*. The `AM` only reports app launch times for starts that go through the `onCreate` Android lifecycle method; we call such starts *cold starts*. All other app starts that, e.g., only go through `onResume`, and consequently do not have a `AM` launch time, are considered as *hot start*.

II. Related Work

The need to measure the performance of computer systems is as old as the computer systems themselves. How else could we determine whether a new generation of CPUs, GPUs, SSDs, complete System on Chips (SoCs) or even entire platforms consisting of many individual components, are worth the upgrade and the accompanying monetary costs. Thus, there are many tools to measure the performance of computing systems. A quick Google search reveals more than 40 commonly used

benchmark tools [5]. Some of these benchmarks such as the SPEC benchmarking suites [6]–[8] for CPU benchmarking, or Mobile-/Sys-/Tabletmark [9] are industry standards and maintained by entire (non-profit) corporations. For an in-depth coverage of various topics in performance benchmarking we refer the interested reader to [10]–[13].

Computer benchmarks can be roughly categorized into *synthetic* and *application* benchmarks [14]. Synthetic benchmarks measure the performance of individual components. These benchmarks generally include artificial workloads that try to match an "average" execution profile. This limits their usefulness when trying to understand real-world performance. For example, two GPUs might reach similar benchmark scores, but perform differently in real games due to software optimizations. Application benchmarks on the other hand generally measure the performance of a system as a whole. Such benchmarks represent real-world workloads more accurately. For example an application GPU benchmark could run real games and measure the framerates. The main drawback of application benchmarks is that the benchmark developers need to make assumptions about what workloads best represent realistic usage scenarios. We extend this standard definition by a term we call *real-world* benchmarking. When we observe a performance metric such as app launch times while a user is operating her phone normally, we call this a *real-world* benchmark, since we are not making any assumptions about the users' usage patterns. Also, we are not actively executing a benchmark program, but rather passively observing. Note that no single benchmark type is superior in general. Synthetic benchmarks allow for more detailed analyses at the cost of generality. Application and real-world benchmarks on the other hand offer more realistic results, but it is more difficult to pinpoint the reasons for either good or bad performance. Each type of benchmark gives valuable feedback about different aspects of system performance.

The areas of application are also different. For example, in systems that do not experience much change and where the future workload is well known, synthetic or application benchmarks are well suited. In systems such as mobile phones that experience many updates and where the user can download a myriad of different applications, all with different effects on performance, constant monitoring through a real-world benchmark makes sense, since we are mostly interested in *changes* in performance.

There are many benchmarking apps for Android [15], most of which rely on synthetic performance measures. Phone manufacturers regularly try to cheat such benchmarks as uncovered by *AnandTech* [16] and *XDA* [17]. Gustafson [18] advocates the use of *purpose based benchmarks*, where the purpose of the benchmark is in alignment with the goals of the user. If manufacturers were to optimize their hard- and software to launch apps faster and thus excel in our benchmark, real-world user experience would also improve, and it could arguably not be called cheating.

Pandiyan et al. [19] investigate which factors govern the performance of the Android platform in order to guide future platform design. Yoon [20] studies the performance of the Android platform using existing benchmarks and profiler software. While we are also interested in overall system performance, we do not aim at guiding system design, but instead at measuring the real-world performance of phones during daily usage. Furthermore, our benchmarks run without modifications to the operating system or external hard- and software.

Guo et al. [21] examine the current state of benchmarking on mobile platforms. They mostly focus on finding ways to reduce the variance of existing synthetic benchmarks. Kim et al. [22] show how the storage performance of Android affects user experience, and how to benchmark it through mixed workloads of SQLite transactions. Although such methods can accurately capture storage performance for given workloads, they do not necessarily reflect the system's performance in everyday use. Joshi et al. [23] recognize that synthetic benchmarks generally do not capture the performance that a specific application will exhibit on a given hardware platform. Their *BenchMaker* framework facilitates the construction of synthetic benchmarks that help capturing specific application behaviors. *MobileBench* presented by Kim et al. [24] is a benchmarking tool that focuses on user experience and realistic workloads such as web browsing. These approaches have in common that they focus on either synthetic or application workloads. Our goal is to measure real-world performance of Android phones during normal usage, without the need to specify any "average" workload or component-specific synthetic tests.

III. Video Benchmarking

In order to accurately measure the start duration of apps as perceived by the user, we build a mechanical system that simulates user input to launch apps. We show that the app start times output by the Android Activity Manager (`AM`) correlate well with the actual start times for apps as observed by users, and we will thus use them as a metric for overall device performance.

A. ActivityManager Launch Time

Since Android 4.4 there is a built-in tool [3] for profiling app launch time performance on rooted phones or through the Android Debug Bridge (ADB). Developers can use it to identify performance bottlenecks that cause their applications to start slowly. When an app is launched for the first time, the following line is written to the log: *ActivityManager: Displayed myApp/.myAct: +3s12ms*. The *displayed* time does not necessarily capture the entire app launching process, as resources can be asynchronously loaded after the initial drawing of the UI. This process is called *lazy loading*. Developers have the possibility to manually call *reportFullyDrawn* to measure an app's complete launch time including lazy loading. However, we have not observed any such apps. Note that the `AM` only provides start times for *cold starts*, since in the case of *hot starts*, the *onCreate* lifecycle method is never called. Remarkably, we have observed that around half of all app

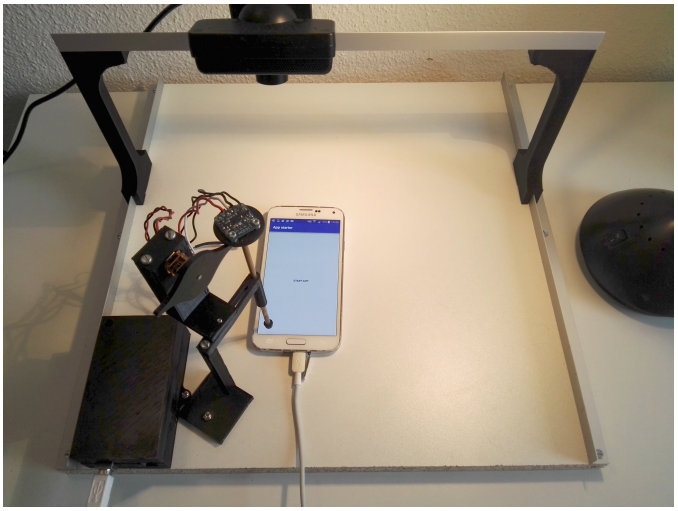

Fig. 1. Our setup showing the Arduino, the stylus with the accelerometer, the motor with the 3D printed arms, the camera and the smartphone under test.

starts in our real world experiments (varies depending on phone and app) are *cold starts*.

Figure 1 shows our hardware setup. The black box contains an *Arduino* board that is connected to a PC via USB. The Arduino controls the setup and receives all instructions from the PC. Connected to the Arduino is a motor that has a two-armed wheel attached to it. When this wheel is turning, it lifts and drops a touch stylus. The stylus is guided such that it can only move vertically. On top of the stylus there is an accelerometer that is connected to the Arduino. We use the data from the accelerometer to control the rotation of the motor, i.e., we detect when the stylus is picked up by the wheel, at which point we can halt the wheel. When we want to trigger the next app launch, we turn the motor back on, which will cause the stylus to drop onto the screen. Half a rotation later it will be lifted from the screen again, which will be registered as a button press by the app.

Below the stylus we place the phone to be tested. We created an app that starts a predefined set of real apps by pressing the fullscreen button. Therefore, the exact phone placement is not important. Above the phone is a *PlayStation 3 Eye* [25] camera. The camera has a frame rate of 120fps, which is higher than the refresh rate of typical smartphone screens. The video data from the camera will be used to measure the app launch times, as described in Section III-B. The Arduino notifies the PC when it detects that the stylus is lifted, which indicates the beginning of the app start.

B. Automated App Launch Time Measurements

We use the mechanical setup introduced in the previous section to collect data about app launches. We then automatically analyze the video recorded by the camera to measure the app launch times. We assume an app has finished starting when the video frames do not change anymore. However, there are apps, such as *Shazam*, that contain animations that do not stop after the app has finished starting. To avoid such animations from causing issues for our system, we group pixels together, essentially downsampling the video. In the final setup we group the images into squares of 5×5 pixels. For each square we calculate the frame-wise differences and report a change if that difference exceeds a threshold. We used a threshold of $\frac{1}{10} \cdot I_{max}$, where $I_{max} = 255$ is the maximum possible pixel intensity. These parameters were tuned such that apps like *Shazam* can be measured accurately. Note that the grouping happens on the pixels of the video, i.e., the parameters depend on the resolution of the camera, and not on the resolution of the smartphone screen. This way of detecting changes works well for slow continuous movements, as on the splash screen of *Shazam*. To evaluate the performance of our automated computer vision method, we launch 16 different apps ten times each. To obtain ground truth launch times, we look at the video and manually annotate the app launch times. Figure 2 shows how well our automated computer vision approach works compared to manually finding the first frame in the video after which the app has finished launching. The resulting average absolute error is 0.09s. We consider this error negligible and thus conclude that our automated method works.

In Figure 3 we show the relation between the launch times measured by our automated system and the app launch times as reported by the `AM`. For apps like *Calculator* that do not perform lazy loading, the `AM` times are very close to the start times as they are observed by the user. For apps that perform a lot of lazy loading, such as *Spotify*, the absolute difference is higher. While for many apps the `AM` times cannot be used to accurately predict the absolute app launch times, the `AM` launch times are highly correlated with the user-percieved app launch durations. That is, when an app start is slower on average than usual as measured by our system, then the `AM` launch times will also be higher on average. Thus the `AM` launch times indeed capture the loading of basic app components and provide a good *lower bound* for app launch times as perceived by users. We will thus use them as proxy for ground truth launch times for the rest of this paper. Note again that in order to get the `AM` launch times the phone needs to be rooted.

IV. Resource Usage Benchmarking

We expect that the RAM utilization changes when apps are started from flash storage as data is being loaded into RAM. Similarly, the CPU should do more work during an app launch than during an idle period. Hence we look at the changes in CPU load as well as RAM usage during an app start to find out whether we can infer the startup time of apps. We only need to detect the end of an app start, as the beginning can be easily found using *View Clicked* events of the `AccessibilityService` (`AS`), as described in Section V. The data presented in the following has been recorded with an app we call *DataNiffler*. The *DataNiffler* app contains a root module which allows it to store the app startup times reported by the Android `AM`.

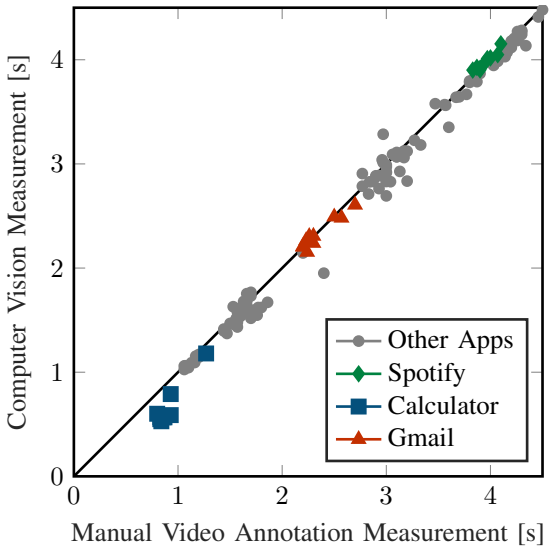

Fig. 2. Our video measurements show high correlations with the ground truth annotated start times. Hence, we argue that we can use app launches automatically annotated from video recordings to evaluate how well the `AM`-reported start times can capture the actual start times.

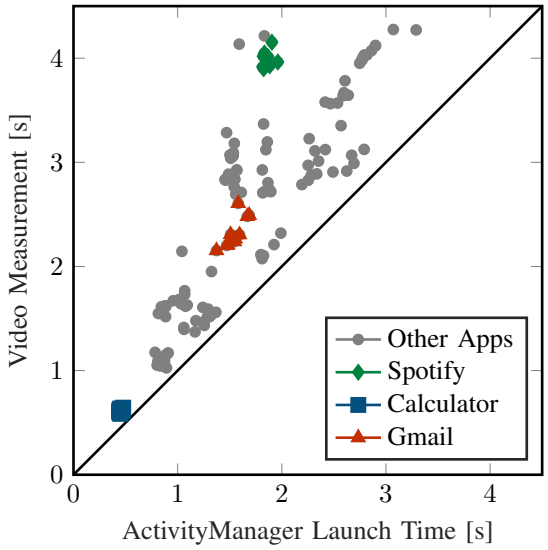

Fig. 3. The start times as reported by the Android ActivityManager are correlated with the start times as found by our video measurement. The larger offsets for Spotify and *Gmail* are caused by lazy loading.

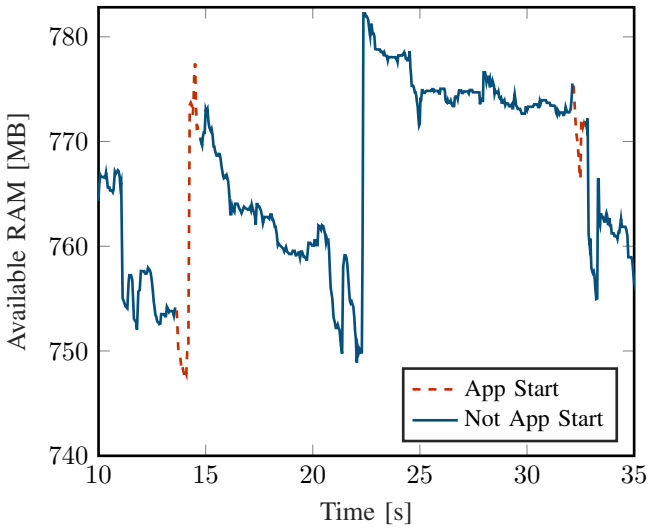

Fig. 4. Example of available RAM during two consecutive app starts.

A. RAM Usage

Our assumption is that the amount of available RAM should decrease rapidly during a *cold start* and then level out when all the app contents are loaded. Consequently, we interpret the point in time after which the available memory is stable as the end of a start. This effect should be less pronounced in the case of *hot starts*, since app data is already cached in RAM. The amount of total available RAM can be retrieved using the `AM` API [1]. Figure 4 shows how the amount of available RAM behaves during two *cold starts*. Clearly, the amount of available total RAM cannot simply be used to measure app launch times. During the first start, the available total system memory increases, whereas during the second start, the available memory first decreases and then increases again. The first problem is that Linux keeps closed applications in memory until it needs that memory for something else. Consequently, a new app start can cause an increase in the amount of free RAM because another app gets removed, and the new app might need less RAM than the removed app. Another difficulty is due to the memory management of both Java and Android. Data which is no longer used by any app will remain in memory until the garbage collector deletes it. As the behavior of the garbage collector can neither be influenced nor observed without root, this can also have an impact on our measurements. Furthermore, apps can share memory, which makes it even harder to find out what happens during an app launch. In our experiments, there were only few cases where RAM usage could be used to accurately estimate app start durations.

B. CPU Load

Similar to the RAM usage idea in the last section, we now look at the CPU usage before, during, and after an app start. We expect the CPU load to be higher during an app start. As of Android 7, the CPU load can only be computed for the overall system and not for individual apps. There is no Android API providing CPU load information, and it thus needs to be computed from values read from the files `/proc/stat`, `scaling_cur_freq` and `cpuinfo_max_freq`.

Figure 5 shows that there are indeed instances of app starts during which the CPU load behaves as expected, and the app launch time can be deduced. However, Figure 6 shows another instance of an app launch during which the CPU load behaves more erratically, and causes our prediction to fail.

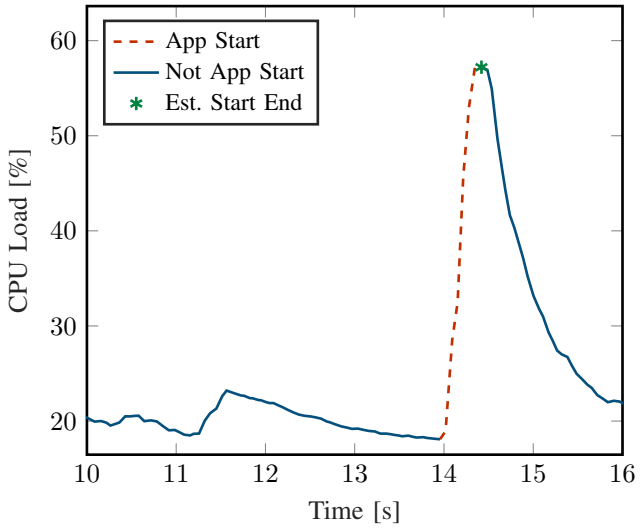

Fig. 5. There are app launches that clearly show increased CPU load during an app launch which could be used to determine the app start times. However, the CPU load measurements also consume a lot of resources.

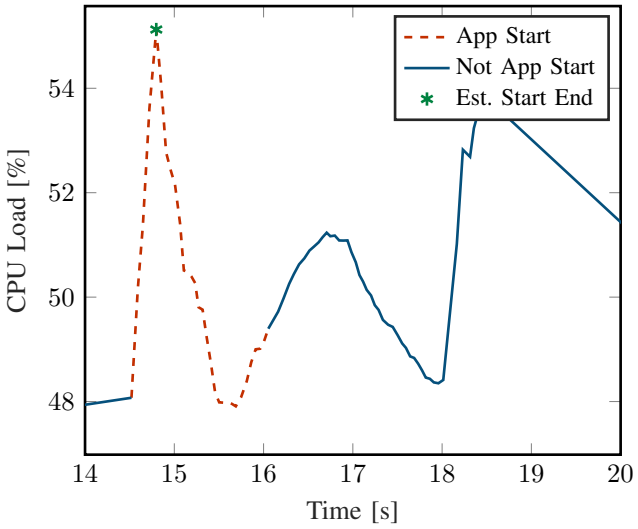

Fig. 6. Unfortunately, there are many app launches, we cannot predict, especially when there is high overall system load (partially induced by our CPU polling), since we do not get enough data points and the data is too noisy.

As expected, the CPU load does usually increase when an app is started and decrease after the start is finished. However, the end of a start is hard to detect because in many cases the app continues to produce load after the end of the start (e.g., due to lazy loading, as discussed in Section III-A). In other cases, the CPU load might already start to decrease before the app start is finished, e.g., because some parts of the starting process are more resource consuming than others.

On top of that, background processes may create additional CPU load, e.g., the garbage collector starts freeing up memory from the previous app which generates CPU load as well. In addition, we face problems with the performance overhead introduced by the measurements themselves. To gather enough data points we need a high polling rate of at least 20 measurements per second. However, this generates load, and the Android scheduler often waits much longer than requested before reawakening the corresponding process. Consequently we get only few data points for most starts, which makes the predictions unreliable and generally worse than illustrated in Figure 6.

Due to the results presented in this section, we conclude that CPU and RAM usage cannot be used to efficiently and accurately measure app launch times on Android.

V. Accessibility Service Benchmarking

Since the CPU and RAM usage does not allow for accurate start time measurements, we show how to use AccessibilityEvents (`AE`) instead. These events can be received by any app which registers as a listener through the AccessibilityService (`AS`) [26] API. Most of the `AEs` describe a state transition of the user interface [2]. Associated with each `AE` is a timestamp and the package/class name of the activity that triggered it. For us, the most important `AEs` are the following:

View Clicked (VCL)
: VCL events are triggered whenever the user clicks on a UI element such as a button or app icon. When a user starts an app by clicking on its app icon, the VCL event provides us with the exact timestamp of the app start.

Window State Changed (WSC)
: WSC events are triggered every time a new window is opened. Every app start at least produces one such event.

Window Content Changed (WCC)
: WCC events occur when new content is loaded in a window. Usually, many of these events are triggered during a start and the exact number varies strongly, as shown in Figure 7.

In the following we first describe how we estimate the durations of *cold starts*. We then train a classifier to distinguish between *hot starts* and *cold starts*, such that the start time estimation is not falsely applied to *hot starts*. We can then use the estimated app start durations to detect performance regressions. Finally, we investigate an unsupervised method to detect changes in system performance that does not require root access. Whenever there is an accuracy metric for a classifier or predictor, we state the performance on the test set. If the train and test data come from the same phone, we use a 90/10 train/test split. If we train on data from one device and test on data from another device, we use all the data available from each device. We did not perform a hyper parameter search or cross-validation over different learning algorithms. We were most interested in the general feasibility of our approach, and not in achieving the absolute lowest errors. Thus, it is likely that the results could be further improved in the future.

A. Dataset

Our dataset consists of two parts: (1) Data collected during controlled lab experiments and (2) data collected during daily use. Table I shows how many apps were included in our lab experiment, how many app starts we performed in total, and which phones we used. The experiments were conducted under controlled conditions while being connected to the Internet through WiFi. We used sets of 17 and 24 popular apps, including Google apps such as *Calendar*, as well as third party apps such as *Facebook*, *Instagram* and *Spotify*. All devices are running the manufacturer provided versions of Android 7.

Table I also shows the properties of our real-world dataset. This data was collected on three different phones over the course of ten weeks of normal use. As we have no control over the test users' behavior, some of these apps have been started only a few times while others have hundreds of starts. For the subsequent analysis we only include apps with a lot of starts. The results presented in this section are based on the data from the following apps: *WhatsApp*, *GTasks*, *Gmail*, and *Google Calendar*. It is interesting to note that the fraction of *cold starts* in our real-world dataset is 44%, which is higher than we expected. This means that focusing only on *cold start* launch times, for which we get an `AM` launch time, is justified. The Nexus 5X and 6P were running stock Android 7, while the Nexus 6 was using a custom ROM called *Lineage OS* [27] based on Android 7.

B. Supervised App Launch Duration Estimation

In this section we show how well we can predict the `AM` launch times. We train a Random Forest with 100 trees, using the launch times reported by the `AM` API as ground truth. We use the number of `AEs`, the `AE` timestamps and the app names as features. The app names are used as features in order to condition the Random Forest on specific apps, since starts from different apps have different `AE` sequences.

The results are shown in the first two columns of Table II. *Internal* denotes the case when training and test data were gathered on the same device, and *external* is the case when the Random Forest was trained on data from one device, and tested on data from another device. The absolute errors are relatively large, especially for the real-world dataset. However, the estimated launch times always have high correlation with the `AM` launch times. A correlation value of, e.g, 80% means that if the predicted launch time increases, the `AM` launch time increases as well with 80% probability. Furthermore, the start durations of certain apps, e.g., *Youtube*, can be predicted very accurately, as shown in Figure 8. Table II also shows that we can train on data from one device and predict on other devices (*external*) and still retain high correlations. Due to the high correlations between the predictions and the `AM` launch times, we can detect qualitative differences in phone performance, as will be discussed in Section V-D.

TABLE I
DATASET SIZE IN THE REAL-WORLD AND LAB EXPERIMENTS.

	Device	#Apps	#Starts Total	#Cold Starts
Lab	Nexus 5X	24	3484	1765
	Xperia Z5 Compact	17	3219	1447
Real	Nexus 6P	45	1959	921
	Nexus 6	38	959	389
	Nexus 5X	37	669	269

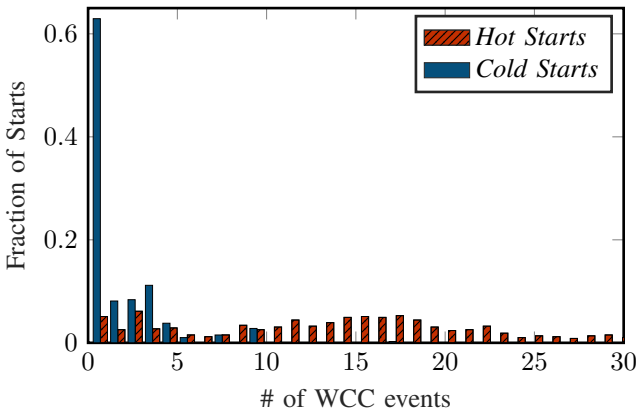

Fig. 7. The number of `AEs` depends on the start type. This is an example for 394 *cold starts* and 587 *hot starts* of *WhatsApp*. *Hot starts* have a much higher variance in the number of WCC events. Starts with only one WCC event are most likely *cold starts*.

TABLE II
COMPARISON OF THE INTERNAL (TRAINED AND TESTED ON THE SAME DEVICES) AND EXTERNAL (TRAINED AND TESTED ON DIFFERENT DEVICES) PERFORMANCE OF THE SUPERVISED MODELS FOR LAB DATA AND REAL-WORLD MEASUREMENTS.

	Duration Error	Duration Correlation	Type Error
Lab internal	25.2%	86.9%	5.3%
Lab external	30.0%	83.2%	17.7%
Real internal	57.0%	81.5%	13.9%
Real external	66.6%	74.1%	16.9%

C. Supervised Start Type Detection

As explained in Section I, the `AM` only reports launch times for *cold starts*. Thus we should only apply the app launch predictions to *cold starts*, which requires a way to distinguish *cold starts* from *hot starts*. Figure 7 shows that *hot starts* and *cold starts* exhibit different `AE` patterns, and a classifier should thus be able to perform well. We again train a Random Forest with 100 trees. As features we use the number of WSC and WCC events as well as the app name. The `AM` provides us with the label; when the `AM` reports a launch time we know it was a *cold start*, and a *hot start* otherwise.

The classification performance of our start type detector is summarized in the last column of Table II. As expected, prediction performance is best when training and predicting on the same device (*internal*). Again, the lab data allows for higher accuracy than the real-world data. When taking the training and test data from different devices (*external*), we observe an increase of the classification error in both the lab setting, and the real-world setting. For our lab experiments we used two completely different phones, which explains the

relatively large performance difference between *internal* and *external*. For our real-world experiments, we only used Google Nexus phones, albeit one of them running *LineageOS*, which could explain why the classification accuracy for the internal and external cases is almost the same. Being able to predict what kind of start just occurred, we can apply our Random Forest start time prediction from Section V-B to all starts we classify as *cold starts*.

D. Supervised Detection of Performance Changes

We have shown that we can predict app launch times with high correlation to the AM launch times, and that we can classify app launches into hot- and cold starts with accuracies of 80-95%. We now combine these predictions to detect a degradation in device performance. In our real-world datasets, there are periods where *DataNiffler* was operating in *heavy mode*, i.e., additional background tasks to artificially slow down the phone were running, and periods where *DataNiffler* was operating in *lite mode* with minimal impact on overall system performance. Running *DataNiffler* in the heavy mode slows down app starts by roughly 25% (based on AM launch times). The slowdown does not just affect app launches, but can be clearly felt by the user during normal operation, e.g., while scrolling or navigating apps. Since we cause the performance decrease by running *DataNiffler* in *heavy mode*, we know exactly when the phone's overall performance changes. Thus, we can compare periods of normal performance with periods of decreased performance. Since we know that a decrease in system performance corresponds to an increase in app launch times, we can simply use our predicted app launch times to detect the changes in performance. In the remainder of this section we evaluate whether it is possible to distinguish periods of good performance from periods of bad performance using the app launch time and type predictors.

First, we classify app starts into *cold starts* and *hot starts*, and then apply the launch time predictions to all starts we classified as *cold starts*. We then compute the average difference between starts from the *heavy mode* and the *lite mode*. If our method works this difference should be positive, i.e., we should predict that app starts from the *heavy mode* are slower on average. If on the other hand the system performance has not changed, we should not observe a difference in predicted launch times. Table III shows the results. For the four most used apps in the real-world dataset, the average estimated increase of app launch times is indicated. All the differences

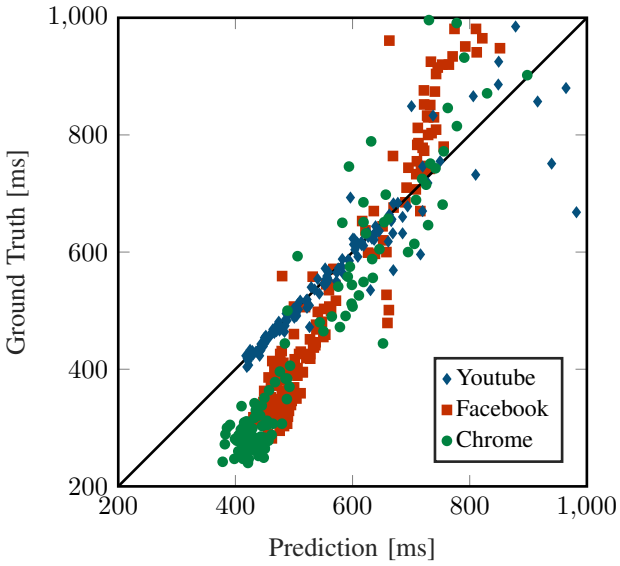

Fig. 8. For some apps, e.g., *YouTube*, the prediction has low error (8%) *and* high correlation (94%). For apps like *Facebook* we have a higher error (28%), but retain high correlation (95%). There are also apps like Chrome, where we have a higher error (51%) and lower correlation (45%).

are positive, i.e., we correctly predict increased launch times. Thus, we can successfully detect the decrease in system performance induced by *DataNiffler* running in *heavy mode*. We verified that launch time predictions from app launches within the same mode are not significantly different. For example, when comparing *WhatsApp* launches from two *heavy mode* periods, the mean and median differences in predicted launch times are -2ms and -0.3ms.

E. Unsupervised Detection of Performance Changes

We showed that we are able to detect a change in smartphone performance using simple supervised Machine Learning methods. Training the models requires phones with root access to obtain the training data. To make the method applicable on a large scale, we also discussed how well a predictor can be trained on one set of devices and applied on another. Collecting more training data should generally improve the results. However, it might be difficult to gather enough data. Also, the performance of our system will be generally worse for phones that are unpopular, and thus generate less training data. Furthermore, users might not be willing to install an app that monitors (albeit anonymously) their system usage and

TABLE III
AVERAGE INCREASE OF APP LAUNCH TIMES DUE TO RESOURCE CONSUMING BACKGROUND PROCESSES.

	Increase of App Start Times Prediction [ms] Mean / Median
WhatsApp	44 / 29
GTasks	159 / 146
Gmail	142 / 108
Calendar	165 / 137

TABLE IV
THE TIME BETWEEN CONSECUTIVE AEs GETS LONGER AS DEVICE PERFORMANCE DECREASES DUE TO RESOURCE CONSUMING BACKGROUND PROCESSES.

	Mean Event Differences [ms]	Median Event Differences [ms]
WhatsApp	54	21
GTasks	94	16
Gmail	12	3
Calendar	64	3

sends the data back to a server, where we use it to train our models.

Therefore we also propose an unsupervised method to detect changes in phone performance based on the interval length between consecutive AEs. We hypothesize that when the phone is slowed down, the average distance in time between AEs should increase. The intuition is that the system needs to allocate CPU time to the processes that send and recieve AEs, and as the overall system load increases, it will take longer on average for the scheduler to serve these processes. Thus for each app launch L_i we calculate the average temporal distance D_i between AEs. To compare two app launches L_1 and L_2 we calculate the difference between D_1 and D_2. We use this to compare app starts from periods when *DataNiffler* was running in *heavy* mode with starts from the *light* mode. Table IV lists the resulting average AE interval differences for the four most launched apps in our real-world dataset. Clearly, the average temporal distance between consecutive AEs has increased. Thus, we conclude that the slowdown induced by *DataNiffler* running in heavy mode can be detected by only observing the intervals between consecutive AEs, since the mean and median event time differences all increase when we artificially slow down the phone.

VI. Conclusion and Future Work

In this paper we take step a towards real-world benchmarking of the Android system, using app launch times as performance metric. We investigate the feasibility of different measurement methods, and show that we are able to detect system performance changes across different devices, even without root access. Currently, we are only able to detect an increase or decrease in system performance, but we cannot yet accurately specify by *how much* and *why* the performance has changed. We also plan to investigate other real-world metrics such as frames per second during UI rendering. As discussed, we have come to the conclusion that RAM and CPU utilization cannot be used to accurately measure app launch times. However, an increased overall CPU load or a consistently lower amount of free RAM could serve as an additional indicator of decreased system performance.

References

[1] Google. Android activitymanager. https://developer.android.com/reference/android/app/ActivityManager.html. Accessed: 2017-10-09.

[2] ——. Android accessibilityevent. https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html. Accessed: 2017-10-09.

[3] Google developers: Launch-time performance. https://developer.android.com/topic/performance/launch-time.html. Accessed: 2017-10-09.

[4] Google. Overview of android memory management. https://developer.android.com/topic/performance/memory-overview.html. Accessed: 2017-10-09.

[5] Benchmark (computing). https://en.wikipedia.org/wiki/Benchmark_(computing). Accessed: 2017-10-09.

[6] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000. [Online]. Available: https://doi.org/10.1109/2.869367

[7] ——, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[8] Spec cpu 2017. https://www.spec.org/cpu2017/. Accessed: 2017-10-09.

[9] Bapco: Application based benchmarking for windows, android and ios. https://bapco.com/. Accessed: 2017-10-09.

[10] R. O. Nambiar and M. Poess, Eds., *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5895. Springer, 2009. [Online]. Available: https://doi.org/10.1007/978-3-642-10424-4

[11] ——, *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7755. Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-36727-4

[12] R. Nambiar and M. Poess, Eds., *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things - 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 9508. Springer, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-31409-9

[13] ——, *Performance Evaluation and Benchmarking. Traditional - Big Data - Interest of Things - 8th TPC Technology Conference, TPCTC 2016, New Delhi, India, September 5-9, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 10080. Springer, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-54334-5

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[15] Google play: Benchmark apps. https://play.google.com/store/search?q=benchmark. Accessed: 2017-10-09.

[16] B. K. A. Anand Lal Shimpi. (2013) They're (almost) all dirty: The state of cheating in android benchmarks. http://www.anandtech.com/show/7384/state-of-cheating-in-android-benchmarks. Accessed: 2017-10-09.

[17] S. Z. X. Developers. (2017) Benchmark cheating strikes back: How oneplus and others got caught red-handed, and what they've done about it. https://www.xda-developers.com/benchmark-cheating-strikes-back-how-oneplus-and-others-got-caught-red-handed-and-what-theyve-done-about-it/. Accessed:2017-10-09.

[18] J. Gustafson, "Purpose-based benchmarks," *IJHPCA*, vol. 18, no. 4, pp. 475–487, 2004. [Online]. Available: https://doi.org/10.1177/1094342004048540

[19] D. Pandiyan, S. Lee, and C. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench," in *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-24, 2013*, 2013, pp. 133–142. [Online]. Available: https://doi.org/10.1109/IISWC.2013.6704679

[20] H.-J. Yoon, "A study on the performance of android platform," *International Journal on Computer Science and Engineering*, vol. 4, no. 4, p. 532, 2012.

[21] Y. Guo, Y. Xu, and X. Chen, "Freeze it if you can: Challenges and future directions in benchmarking smartphone performance," in *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications, HotMobile 2017, Sonoma, CA, USA, February 21 - 22, 2017*, 2017, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/3032970.3032979

[22] J. Kim and J. Kim, "Androbench: Benchmarking the storage performance of android-based mobile devices," in *Frontiers in Computer Education [International Conference on Frontiers in Computer Education, ICFCE 2011, Macao, China, December 1-2, 2011]*, 2011, pp. 667–674. [Online]. Available: https://doi.org/10.1007/978-3-642-27552-4_89

[23] A. Joshi, L. Eeckhout, and L. John, "The return of synthetic benchmarks," in *2008 SPEC Benchmark Workshop*, 2008, pp. 1–11.

[24] C. Kim, J.-h. Jung, T.-K. Ko, S. W. Lim, S. Kim, K. Lee, and W. Lee, "Mobilebench: A thorough performance evaluation framework for mobile systems," in *The First International Workshop on Parallelism in Mobile Platforms (PRISM-1), in conjunction with HPCA-19*, 2013.

[25] Wikipedia: Playstation eye. https://en.wikipedia.org/wiki/PlayStation_Eye. Accessed: 2017-10-09.

[26] Google. Android accessibilityservice api. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html. Accessed: 2017-10-09.

[27] L. Project. Lineage os. http://www.lineageos.org. Accessed: 2017-10-09.