Reproducible Load Tests for Android Systems with Trace-based Benchmarks

Alexander Lochmann
alexander.lochmann@tu-dortmund.de

Fabian Bruckner
fabian.bruckner@tu-dortmund.de

Olaf Spinczyk
olaf.spinczyk@tu-dortmund.de

Technische Universität Dortmund
Department of Computer Science 12
44227 Dortmund

ABSTRACT

The development of system software and hardware components for Android devices is strongly influenced by the necessity to save energy. However, there is no methodology that provides developers with reproducible and comparable benchmarks for testing the device under a representative load. Such a benchmark would have to stimulate all relevant parts of the system and must neither depend on the current state of external servers in the Internet nor on any interactive user.

This paper describes the first steps towards such a benchmark. The approach is based on recorded workload traces of prominent Android applications. From these traces we can "mix a cocktail" that yields a representative workload profile. By replaying the recorded and mixed loads with a workload generator combined with an external environment for dealing with communication workloads we obtain benchmarks that fulfill the requirements.

Keywords

Android, Benchmark Generator, Benchmark, Load Testing, Application Tracing

1. INTRODUCTION

Many researchers worldwide develop ideas to improve Android systems. Their motivation is to reduce a mobile device's overall power consumption, because that influences its usability in daily life the most.

As an example, Athivarapu et al. [1] report energy savings of 20--40 % by optimizing the use of mobile networks. However, their evaluation on real hardware is very limited and difficult to reproduce. Ideally such improvements would be evaluated using off-the-shelf benchmarks, but standard Android benchmarks such as AnTuTu[1] and benchmarks from the Linux desktop and server world, e.g., sysbench[2], cannot be applied, since they were designed to measure the maximum performance of a particular subsystem and not a representative load. Providing such an Android benchmark is a research problem on its own, because (1) there is no such thing as *the* representative workload and (2) Android offers such a rich set of resources that would all have to be covered, for example the Location Manager, the Power Manager, the Application Service, etc. Furthermore, since an Android device rarely runs at maximum speed and uses various kinds of resources, a new benchmark would have to use the whole Android software stack in a more moderate way. Some researchers already came across this issue, but for now, they have only come up with individual solutions. For example, they use a trace-based replay where they only replay the resource they need [7].

This paper presents the first steps towards a general approach for benchmarking Android systems to fill this gap. The best way to stimulate the Android software stack in a natural way would actually be to use real applications such as popular "apps" from the *Google Play Store*. However, this may lead to significant installation efforts, bad reproducibility due to version updates, and the need for user input, which would render the benchmark non-deterministic. Therefore, our approach is to record every resource that is allocated by an application, and replay the trace later on. We explicitly do not cover user input. That would induce a sophisticated replay logic, since a user's behavior might change due to modifications in the Android system. Moreover, an input has to be mapped to the subsequent resource allocations. Mapping of user input to events in the Android system has already been explored by Zhang et al. [9].

The advantages of this approach over the current state-of-the-art are as follows: First of all, it separates concerns as shown in Figure 1. The component developer only focuses on improving a system component. Based on a profile that describes the "apps" that the component developer regards as "representative", the benchmark developer provides a customized benchmark. Generating a benchmark means to mix pre-recorded app traces. Thus, this approach does not provide a single benchmark, it provides a benchmark generator. Second, the "benchmark" is an application that replays resource-usage patterns and not the exact behavior of the traced applications. It thus runs on any Android-powered smartphone without further modifications or installations.

[1]http://www.antutu.com/

[2]https://github.com/akopytov/sysbench

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4899-7/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3053600.3053615

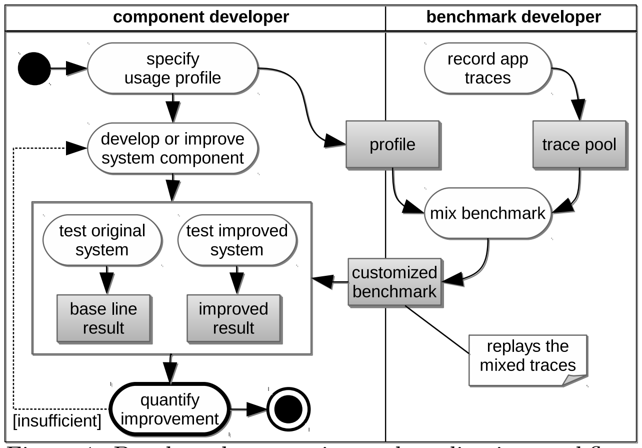


Figure 1: Benchmark generation and application workflow

To sum up, the contributions of this paper are as follows:

- We present a general approach for recording the resource usage of any Android application,
- and show a way of replaying the recorded trace on any unmodified Android device.

2. RELATED WORK

Benchmarks have been an active area of research for many years, and many implementations are available. The various approaches can be categorized into micro-benchmarks, trace-based evaluation, or a mix of real applications with trace-based input.

Micro-benchmarks aim at assessing subsystems or special function units, for instance, file systems or GPUs. Besides the famous SPEC CPU benchmark suite[3], a few Android-specific benchmarks have emerged such as *AnTuTu*, *CF-Bench*[4], or *GFX-Bench*[5]. As the names suggest, they only evaluate parts of a system. Furthermore, they emphasize performance and not representativeness.

Trace-based simulation or replay [7, 6] is another popular approach. The latter resembles ours. Yet, without any trace being available to the community, no one can reproduce the results, or compare results across different works. Trace-based simulation, in contrast, is deterministic, but provides only an estimation on how much one has improved the system. It heavily depends on the precision of the model.

Other work uses a mix of real applications with trace-based input [5]. A set of publicly available input data is used to stimulate the applications. However, the approach is not robust against updates of the applications, because updates would often require the input data to be changed as well.

Besides the aforementioned approach, there is a commercial application-based benchmark called *BAPCoSYSmark 2004* [2]. It provides various profiles with representative applications. Those programs are remote controlled, and have to be pre-installed, or are shipped with the benchmark. Besides the difficulties of remote controlling real-world applications, the main drawback of this approach is the cost and effort for installing the required application suite.

[3]https://www.spec.org/cpu2006/
[4]http://bench.chainfire.eu/
[5]https://gfxbench.com/result.jsp

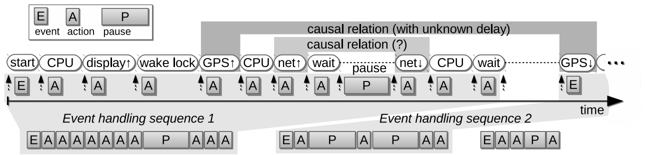


Figure 2: Trace recording and playback: Abstraction of application behavior by event handling sequences

Finally, there has been work conducted on recording resource allocations. Yoon et al. [8] developed *AppScope*. They modified Android including the Linux kernel to do precise recording of nearly every resource, especially the radio usage. They further use that information to estimate the energy consumption of Android applications. It is related to our way of recording resources, but they do not use their traces to generate benchmarks. In order to reproduce results with *AppScope* the traced applications would have to be re-run manually.

3. APPROACH

As shown in the previous section, we strictly focus on an app's resource allocations, which we record as a series of actions ordered by an absolute timestamp. Such a series, for example, consists of CPU bursts, sending network data, or sleep phases, as depicted in Figure 2. The recorded entities can be generalized to *actions* and *pauses*, as shown in the lower part of the figure.

The set of relevant resources on Android is larger than on other Linux systems. Besides common actions such as CPU usage, file I/O, and network traffic, Android provides an application with access to the power management, the GPU, and special peripherals, for instance, GPS, NFC, or Bluetooth. All of them need to be recorded. The power management is of interest, because an application may prevent a device from going to suspend using wake locks[6]. GPS, for example, is a very energy-intensive service. In addition to that, the display state is needed as well, since it has a huge impact on the overall power consumption [3]. Moreover, an application's lifecycle state (for example *resumed*, *suspended*, or *started*), is also required for an accurate replay.

Compared to the recording of actions, the replay is even more challenging: A simple way of replaying a trace would be to "cut" the trace into fixed time intervals and allocate every resource at the beginning of each slot. However, this would not replay the resource usage accurately due to the bursts at the beginning of each slot. A better way is to respect the recorded timestamps and replay actions at the particular points in time. However, this does not allow the benchmark to adapt to improvements in the Android system or faster hardware. Hence, we only respect the delta between two adjacent actions. A better system may therefore process the trace, i.e. benchmark, faster and a user's response time is still replayed correctly. Nevertheless, our approach, for now, has a drawback: It does not respect external *events*, for example, a user explicitly starts the observed application, or a network message arrives. Those events have to be replayed at their respective timestamps, and the benchmark is forbidden to fast-forward them. Otherwise, this would artificially increase

[6]Wake locks are an Android-specific mechanism to prevent the OS from suspending: https://developer.android.com/training/scheduling/wakelock.html

the load. External events mark the beginning of a new event handling sequence, as shown in Figure 2. This means that application behavior is modeled as a set of event handling sequences, which are triggered at fixed times from the start. The actions and pauses that follow an event are replayed with relative time offsets. The main challenge is to distinguish input actions from external events. A simple heuristic could be to mark every incoming packet as an event while the app is not in foreground, and there has not been an outgoing packet for N seconds. The latter condition ensures, to a certain degree, that the incoming packet is not a response to a previously sent one. Periodic actions are not covered by heuristic, since there is no unique identifier. The verification of that heuristic and further research on detecting those events is subject to future work.

In order to implement the trace replay application, there also arise some technical challenges: First, all accessed files have to be extracted from the trace, grouped by file system, and recreated as dummy file operations in a dedicated subdirectory on the respective file system. A grouping is essential, because an Android application may access different file systems such as an app's private data folder or the SD card. Second, we have to take care of the network communication. A remote host is needed for benchmark execution to perform dummy network traffic. Although an Android device may establish many connections to many different servers, our approach simplifies this by always connecting to the same host. However, this simplification has a drawback: Two connections to different servers on the same port are not possible during the replay. An analysis on how often this happens in real apps is regarded as future work.

4. PROTOTYPE

Based on the approach described in Section 3, we have implemented an early prototype. The recording part has been developed for CyanogenMod 12.1[7], which corresponds to Android 5.1. Since information from inside several Android services is needed, we have modified the Android framework. As a result, root access is necessary to flash the modified version of CyanogenMod.

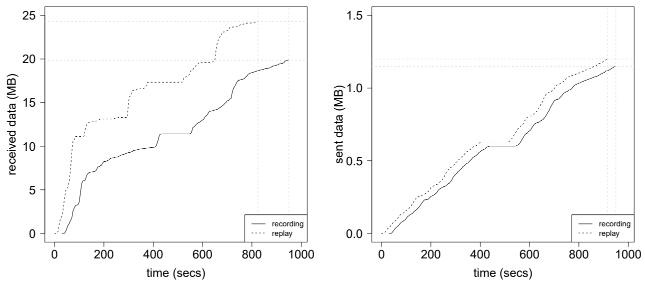
The system basically records all of the aforementioned information. The prototype does not yet record GPU activity, because more work needs to be done to instrument the respective subsystem, and we gave other resources a higher priority. For the same reasons, we do not cover some devices, such as Bluetooth and NFC.

The recording itself works as follows: A background thread periodically stores information about an app's CPU usage. In addition to that, Android services have been instrumented to gather the information, namely the Power Manager and the Location Manager. To record file and network I/O operations, we use a SystemTap script, which uses an application's package name as a filter. SystemTap allows the user to instrument arbitrary Linux kernel functions [4]. For our purpose, we have instrumented all I/O-related systemcalls. Thus, the script automatically logs every I/O performed by the traced app. Since special services, such as the Mediaplayer or the Download Manager, run in process context of the app of interest, their I/O operations are logged as well without further instrumentation.

[7]Former CyanogenMod is now known as Lineage OS: http://lineageos.org

Duration	Actions
6.5 min	View different cities, and browse arbitrary streets
2 min	Inactivity: Closed app, and display turned off
6.5 min	View different cities, and browse arbitrary streets

Table 1: List of actions performed during a 15 minutes record of *Google Maps*



(a) Received network data (b) Sent network data

Figure 3: Comparison of the recorded and replayed network traffic performed by *Google Maps*

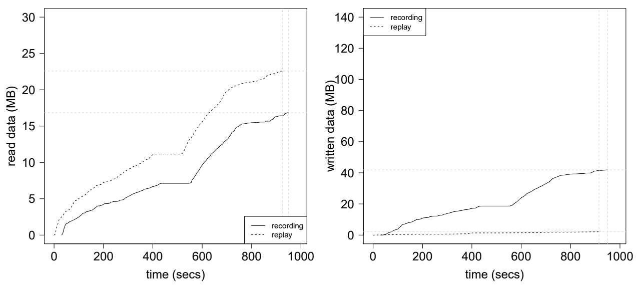
A first attempt towards a replay application is part of our prototype as well. For now, the replay follows a non-adaptive approach. Thus, all actions and pauses are replayed at their respective timestamps. The replay application is implemented as an ordinary Android application. No special permissions are required. This makes it easy for anyone to use it on any device. Before replaying a trace, a certain degree of preprocessing is necessary. The first part is an off-line processing. Since we have multiple sources, multiple output files need to be merged. Moreover, due to SystemTap some events might not be in the correct order. Hence, this step merges and sorts the events into a single file. The rest of the preprocessing is done online right before the replay: The app extracts and creates the accessed files and the upcoming network connections as mentioned above. At the beginning of the replay and at fixed intervals, the replay app informs the remote host about the next packets. It tells the host when to send a packet with a particular size to the app. The remaining tasks of the replay application are straightforward: access files on the different file systems, send and receive network data, and use the various Android services, e.g., request location updates via GPS or acquire a wake lock.

5. EVALUATION

Since our approach aims at realistic benchmarks for Android systems, it is essential to check whether the trace-based replay behaves similar to the original application. Thus, we recorded *Google Maps* for 15 minutes, and performed the actions shown in table 1.

Afterwards, we replayed the trace, and recorded our own replay application. Using both traces, we compare the recording with the replay.

The inbound and outbound network traffic is shown in Figure 3a and Figure 3b, respectively. The time in seconds since start of the recording is shown on the x-axis, and the amount of data read and written is shown on the y-axis. In both figures both curves are aligned, indicating that the approach closely imitates the recorded app's behavior. The



(a) Read file data (b) Written file data

Figure 4: Comparison of the recorded and replayed file I/O performed by *Google Maps*

period of inactivity is also visible at around 400 seconds. However, the replayed trace in Figure 3a shows that our benchmark received too much data, and received that data too early. The latter can be seen by the dashed curve ending too early. Both errors show where the prototype replay app needs improvements: Currently, the app scans the trace for incoming traffic and tells the remote host when to send how much data. For an unknown reason, the application requests the incoming traffic too early and too often. That produces the observed behavior.

The difference between the amount of recorded and replayed sent data is rather a conceptual problem. Since the app needs to tell the remote host the timestamps and the amount of data, there are additional outgoing packets, which contribute to the amount of sent data compared to the original trace.

Figure 4 shows the data transferred during file operations. The axis are labeled in the same manner as in Figure 3. The amount of data read by the replay app is shown in Figure 4a. Both curves basically have the same shape. Due to the necessity to read the trace itself, more data has been read during the replay. The comparison of the written data, shown in Figure 4b, teaches us a another lesson about preprocessing. The replayed trace is significantly different from the recorded one. As a result of our preprocessing, some systemcalls are filtered-out. If those systemcalls contribute to the written data, that information will be lost. Therefore, if the trace contains accesses to many small files, the impact of that filtering is negligible. Otherwise, if only a few large files are accessed, as can be seen in Figure 4b, the impact is large.

To summarize, our prototype shows that the approach looks promising. However, a lot of work needs to be done towards a suitable replay application. From our point of view, the above-mentioned problems are all technical in nature.

6. FUTURE WORK

Although our approach as well as the prototype look promising, some work needs to be done. First of all, an adaptive version of the replay application is necessary. As already mentioned in Section 3, the replay must allow a better Android system or faster hardware to speed up the benchmark execution. As a part of the adaptive replay, the described heuristic to detect external events needs to be implemented and verified.

Second, a case study is required to get representative traces that can provide the community with our benchmark. In a next step, a mix of those traces can be provided. Hence, the benchmark really corresponds to the behavior of a real user.

Finally, a method to analyze several traces of a particular application needs to be developed. Using such a method would allow us to generate more general application profiles. Those can be fed into the replay app.

7. ACKNOWLEDGMENTS

This work has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 “Providing Information by Resource-Constrained Data Analysis”, project A1.

8. CONCLUSION

In this paper, we presented a general approach to generate application-based benchmarks for Android systems. The approach yields comparable, trace-based load tests that can be executed on any Android hardware. We sketched how a proper replay has to be designed, and which challenges arise from that, for example how to determine external events in a stream of actions.

9. REFERENCES

[1] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese. RadioJockey: Mining program execution to optimize cellular radio usage. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, 2012.

[2] BAPCo. Sysmark 2014 - white paper, Apr. 2014.

[3] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference*, 2010.

[4] F. C. Egiler, V. Prasad, W. Cohen, H. Nguyen, M. Hunter, Keniston, and B. Chen. Architecture of systemtap: a linux trace/probe tool, 2005.

[5] Y. Huang, Z. Zha, M. Chen, and L. Zhang. Moby: A mobile benchmark suite for architectural simulators. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, 2014.

[6] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[7] E. J. Vergara, J. Sanjuan, and S. Nadjm-Tehrani. Kernel level energy-efficient 3G background traffic shaper for Android smartphones. In *Wireless Communications and Mobile Computing Conference, 2013 9th International*, July 2013.

[8] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference*, 2012.

[9] 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 *2013 International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2013.