Resource Utilization for Access to Web-Based Services: Browser versus Mobile Application (Extended Version)

Hesham H. Salman and Patrick Seeling

Abstract

The proliferation of mobile device platforms and operating systems has resulted in an increased challenge for developers and mobile users alike. While developers face fragmentation issues in their efforts, mobile users find an increasing number of online services to access in a variety of ways. The increased adoption of HTML5–based service access methods, however, allows to circumvent some of these problems by employing a browser–based service access, either in form of hybrid applications or by directly employing a mobile web browser installed on a user's device. We present and employ a framework that can be readily utilized to perform repeated evaluations of applications on mobile devices with the Android OS to provide statistically meaningful performance measurements. Specifically, we compare dedicated and browser–based access of online social network services with respect to their corresponding CPU and memory resource utilizations. We find a trade–off between increased CPU usage for browser–based service access and increased memory usage when accessing the services with their dedicated mobile applications. Depending on current system resource utilizations, it might hence be advantageous to choose one over the other.

Index Terms

Mobile communication; Applications; Profiling; Optimization

I. Introduction and Related Works

The traditional devices for interacting with networked content and services experience a phase of discontinued growth while being replaced with more portable heterogeneous mobile devices, which increasingly become the primary means for users to connect with content and services [1]. This emergence of mobile devices with different underlying operating systems has given rise to frameworks allowing for (i) rapid application development, (ii) cross–platform development, as well as (iii) integrated testing and performance evaluations. The need to develop for multiple operating systems and frameworks has resulted in a fragmentation challenge that is accompanied by increasing testing woes for developers [2]. To alleviate some of the pressures of current cross–platform development efforts, it is commonplace to employ specific higher level frameworks or to resolve to web–based technologies directly [3]. Indeed, the utilization of web technologies for mobile application development dates back several years to the emergence of a plurality of mobile operating system platforms, which is indicative of its overall practical use [4], [5]. The continuous inclusion of new features makes such an inroad to mobile application development and implementation increasingly appealing [6]. There are, however, some performance drawbacks associated with the utilization of these types of web–centric approaches [7]. Interestingly, not all web application approaches were

Technical Report, Department of Computer Science, Central Michigan University, September 2016.

H. Salman was with the Department of Computer Science, Central Michigan University, Mount Pleasant, MI 48859, USA.

P. Seeling is with the Department of Computer Science, Central Michigan University, Mount Pleasant, MI 48859, USA. Email: pseeling@ieee.org.

found to exhibit a performance decrease, especially when considering networked data [8]. The result is an ongoing debate about the advantages and disadvantages of web–based or hybrid mobile applications in comparison to their dedicated native counterparts [9].

While dedicated applications require initial downloads and installations on a mobile device, hybrid and entirely web–based applications can be executed by employing a typically pre–installed available (dedicated) browser view. These latter approaches generally employ commonplace HTML5 feature sets to access a specific application's underlying web–based service and can employ a regular browser. We note, however, that an evaluation of user experiences is out of scope of this paper. Increasingly, benchmarking of these different applications is desired to provide guidelines for development and adherence to minimum performance requirements, especially for user–facing applications [10]. Without loss of generality, we focus on the Android OS with Chrome as exemplary browser, for which we provide three–fold contributions, namely (i) we describe an overall measurement framework that allows the detailed capture of mobile application activities with respect to the usage of CPU and memory resources, (ii) we describe how this framework can be utilized to perform controlled repeatable experimentation on mobile devices or emulators, and (iii) we employ the framework to perform performance comparison measurements for dedicated native application and browser–based mobile service access. Specifically, we focus on the repeatability of experimentation to allow for more meaningful real–world test results that are not just anecdotally, but also statistically relevant.

The initial evaluation of an Android application is enabled within the default development environment [11], which includes general memory profiling tools. A method previously used by *ProfileDroid* [12] and Microsoft researchers [13] involved compiling an Android native version of the `tcpdump` utility and its libpcap library to evaluate network activity. Other approaches to uncover execution path errors rely on high–level GUI analysis [14]–[18], often including abilities that allow for the recording and subsequent replay of user interactions. For testing purposes, the *Monkey* tool sends random event streams to an application [19] and can be regarded as a first approach to application stability testing. Multiple frameworks allow for scripting of event streams, such as *MonkeyRunner* [20] and *Robotium* [21]. Example solutions that record touch events are *VALERA* [22] and Android *RERAN* [23], which can also be turned into event streams. A more detailed discussion of these different approaches is beyond the scope here due to space constraints. Continuing the trajectories provided by these prior efforts, our framework aims to provide repeatable, automated executions of targeted actions on Android devices in order to be able to reproduce errors and provide a detailed log of an application's performance in general.

In the remainder of this paper, we initially present our framework in greater detail in Section II. We continue by employing our framework in a comparative performance evaluation between native application and browser–based mobile service access for the two popular social networks *Facebook* and *Twitter* before we conclude in Section IV.

II. Android Application Profiling Framework

This section details our Android application profiling framework, with replay capabilities based on a heavily modified version of the original Android *RERAN* [23]. Additional automation of the framework is enabled by scripting its usage, which is comprised of multiple steps for recording, termination, data acquisition, data cleaning, data processing, and data visualization. These individual tasks each have several subtasks associated with them, as illustrated in the overall flow for the developed framework in Figure 1. The framework operates by capturing the output of the `getevent` shell utility, which captures all system events sent to `/dev/input/event*`. The output of the `getevent` tool is redirected to a trace file that is subsequently translated by the library's translation module. The translated events are now able to be sent to the device through the `sendevent` utility. The recorded events

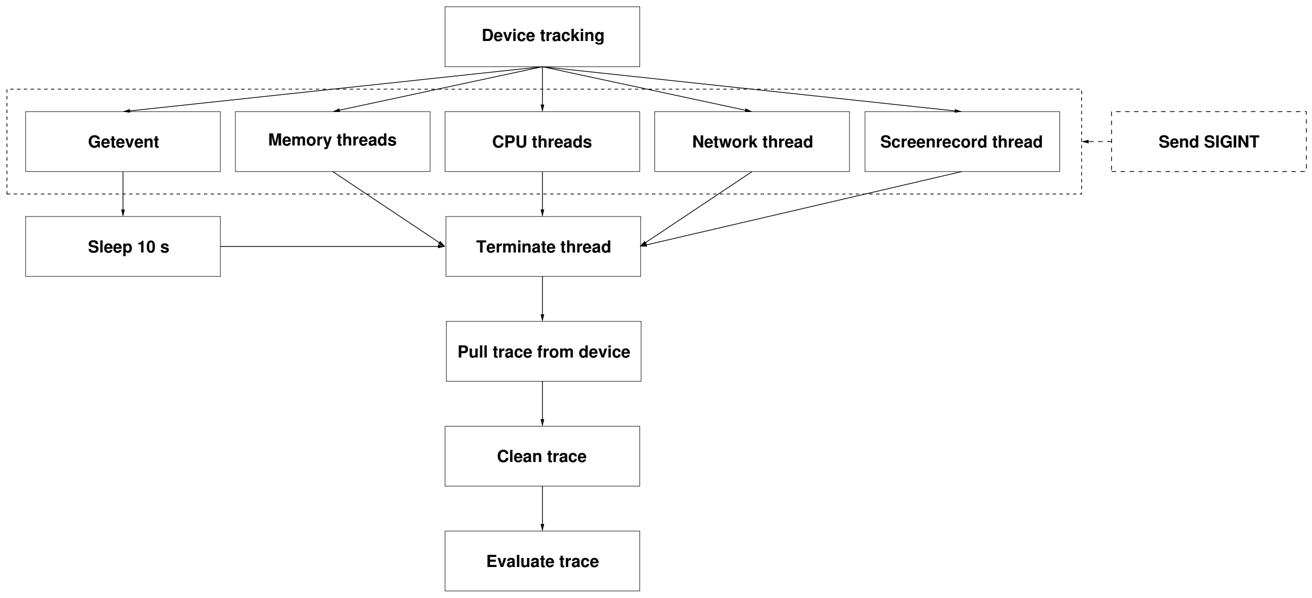


Fig. 1: Flow diagram for generating a single recording trace using the developed framework.

are subsequently executed by the frameworks's replay module on a device. With the ability of repeated executions, several performance measurements are captured to evaluate an application's characteristic resource utilizations in a statistically meaningful manner.

A. *Event and Resource Utilization Captures*

We initially describe the capturing of the detailed resource utilizations, namely tracking (i) interactions, (ii) CPU and memory utilizations, (iii) network packets, and (iv) visual screen content.

1) Tracking (Interaction) Events: The framework's dedicated `getevent` utility reads in data from the system–wide event stream provided by the Android OS. The specific subset of the data to be read is device-specific. On the 2012 Nexus 7 mobile device, for example, the touch screen events are accessible from the `event0` stream, but on an HTC Desire mobile phone, the touchscreen events are available from the `event2` stream. Each device interface is added to the captured event trace file in the order by which they are registered to the actual device. We provide an overview for the output in following example Listing 1. The framework has the capability of tracking all

Listing 1: A sampling of an event log.

```
add device 1: /dev/input/event1
  name:      "lid_input"
add device 2: /dev/input/event0
  name:      "elan-touchscreen"
add device 3: /dev/input/event2
  name:      "gpio-keys"
[    27010.899330] /dev/input/event0: 0003 0039 000000f6
[    27010.899531] /dev/input/event0: 0003 0030 00000009
[    27010.899686] /dev/input/event0: 0003 003a 00000027
[    27010.899839] /dev/input/event0: 0003 0035 000000ab
[    27010.900176] /dev/input/event0: 0003 0036 00000787
[    27010.900384] /dev/input/event0: 0000 0000 00000000
[    27010.967507] /dev/input/event0: 0003 0039 ffffffff
[    27010.967742] /dev/input/event0: 0000 0000 00000000
...
```

publicly accessible event streams, but for the purposes of this contribution, we limit the evaluations to touchscreen event streams. (We note that some event streams are secured, i.e., are not publicly accessible and require higher privileges, such as the GPS event reporting.) An additional challenge is presented by the need for fine–grained timing information that has to be included into the trace together with events. Android reports event times using the monotonic system clock. In order to synchronize these times with real human–readable times, a submodule was introduced to synchronize the monotonic and real–time clocks. These system–level events are all tracked by the `getevent` utility as in Listing 2, which is executed in a single thread spawned by the framework's main thread.

2) Tracking CPU and Memory: Different metrics are available to measure the amount of memory an application uses at any given time: Virtual Set Size (VSS), Proportional Set Size (PSS), Unique Set Size (USS), and Resident Set Size (RSS) are all memory–related metrics that are accessible for performance evaluation purposes [24]–[26]. We illustrate the overall memory space as example in Figure 2. The different types of memory can be differentiated as follows:

VSS
: The Virtual Set Size includes the total accessible address space of a process. This combines heap allocations, occupied memory in RAM, swapped memory, and the allocation of shared libraries into one. The result, however, provides very little insight into the realistic memory usage of a process, see, e.g., [26].

RSS
: The Resident Set Size includes the total amount of memory held in RAM for a process. While this metric does not include swapped memory, it does include shared libraries as long as their pages are in memory. This metric cannot accurately represent memory usage for a single process, but is adequate in representing groups of processes [24], [25].

PSS
: The Proportional Set Size is similar to the RSS, except the amount of memory occupied by shared libraries is split proportionally among all processes that require them. For example, if there are ten processes that use `Library1`, the value of the shared library is split by ten and added to the values of each process that utilize it. This metric is, in turn, well suited for estimating total memory usage of an individual process, as the PSS sum for of all processes makes a fair representation of the total memory usage in the system [26].

USS
: The Unique Set Size details the total private memory for a process, i.e., the memory that is completely unique to that process. Private allocations of a process are included in this measure, which indicates the incremental cost of running a process and is the value of memory that is returned to the system when that

Listing 2: An excerpt from the track_device module showcasing the activities of the `getevent` thread.

```
getevent_thread = fork();
if (getevent_thread == 0) {
  //GetEvent Thread
  int fd = open("recordedEvents", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
  dup2(fd, 1);
  dup2(fd, 2);
  close(fd);
  execlp("getevent", "getevent", "-tt", NULL);

} else {
```

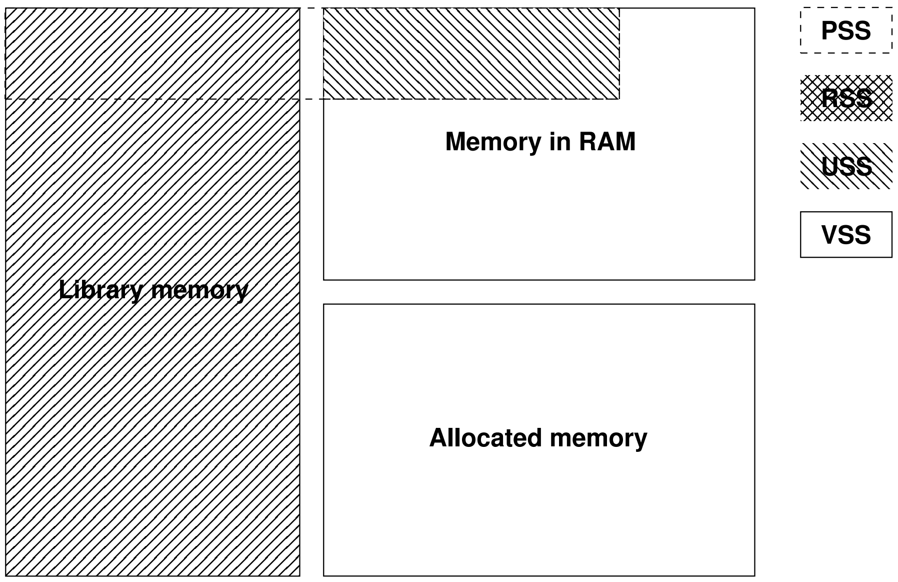


Fig. 2: Exemplary outline of memory space occupied by a process.

process is killed [26].

We chose to rely on the PSS for the reporting of a process' memory utilization, as it represents a fair view on the required memory without skewing the overall system resource utilizations. Tracking the memory and CPU usage of the target application as well as all applications on the system as a whole is performed by analyzing the system `dumpsys` utility's logs on memory and CPU. This utility reports on a number of system services, including sdcard reads/writes and battery usage statistics, which can be used to extend the presented framework further in the future. The utility was employed to continuously report on memory and CPU, outputting detailed usage statistics (including the more realistic PSS), resulting in a plain-text format trace file. The CPU reporting is performed in one dedicated thread, whereas the memory reporting is performed by four threads that are offset by a slightly varying time interval to increase granularity.

3) Tracking Network Information: Tracking network statistics of the device was performed by utilizing the `tcpdump` utility [27] for demonstrative purposes only, as the on–device resource utilization comparison is the focus of this contribution. Nevertheless, the included tcpdump framework module is capable of tracking all network activities on WiFi, Bluetooth, and cellular network (Edge, 3G, HSPA+, LTE, etc.) interfaces. The `tcpdump` utility is executed in its own thread, and its output is redirected to a Packet CAPture (PCAP) file, which is a universally supported file format for packet–level traces. An additionally integrated `pcapfix` utility [28] is employed subsequently to repair any possible damage or incomplete information in the PCAP file. The network tracking is performed in a separate subprocess and due to system privileges, the network monitoring requires root privileges.

4) Recording Video: In order to verify the correct replay of the event traces, a video capture of the playout of an event trace is performed. The video capture serves enable a visual verification for the execution of the replayed events. We note that periodic screenshots could be employed in place of video recording. Periodic screenshots, however, are memory intensive and result in additional gaps in between the user events. (The reason periodic screenshots are more memory intensive is because each call to the `screencapture` utility spawns a new thread that must complete its execution before saving the screenshot.) The `screenrecord` utility is executed in a single thread that saves the video as an mp4 using the system's default encoding parameters.

Listing 3: An excerpt from the track_device module showcasing the activities of the CPU tracking thread.

```
//Main Thread
cpuinfo_thread = fork();
if (cpuinfo_thread == 0) {
  //CPUInfo thread
  int fd = open("cpuinfo.dat", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
  dup2(fd, 1);
  dup2(fd, 2);
  close(fd);
  while (!got_sigterm) {
    /* measure human-readable time */
    clock_gettime(CLOCK_REALTIME, &real_time);
    long sec = mono.tv_sec;
    long nsec = mono.tv_nsec;
    char time_output[255];

    struct tm *nowtm;
    char tmbuf[64], buf[64];
    nowtm = localtime(&real_time.tv_sec);
    strftime(tmbuf, sizeof tmbuf, "%Y-%m-%d %H:%M:%S", nowtm);
    snprintf(buf, sizeof buf, "%s.%06ld %ld", tmbuf, real_time.tv_nsec,
             real_time.tv_sec);
    printf("%s\n", buf);

    char *command = "dumpsys cpuinfo";
    char line[256];
    FILE *fpipe;

    if ( !(fpipe = (FILE*) popen(command, "r")) ) exit (1);

    while (fgets( line, sizeof line, fpipe)) {
      puts(line);
    }
    pclose(fpipe);
    sleep(1);
  }
```

B. Event Replays

While original Android *RERAN* library is the basis of this framework, Android *RERAN* itself has not been updated since the Android OS version 2.3. The Android OS event reporting format since then has changed from its original format and the library no would be usable for devices that employ newer versions of the Android OS. The translation protocol was updated to parse the new event format introduced in Android 4.0, whereas the replay module was updated to repair an open file descriptor leak and introduce time–dilation (i.e., individual actions are spaced out slightly further apart to accommodate for resource fluctuations). The translation protocol manages to parse the new format using a series of regular expressions. An additional improvement to the library was increased automation. Many formerly individual steps were combined into one: recording, cleaning, translation, pushing, and replaying all now are a single procedure within our revised framework. The replay automation script takes an event trace file as an argument and performs all required actions on the file: `sh replay.sh /path/to/events`.

Listing 4: An excerpt from the track_device module showcasing the activities of the tcpdump thread.

```
if (tcpdump_thread == 0) {
  /* TCPDump thread */
  char filename[255]; //255 = max filename size

  long sec = mono.tv_sec;
  long nsec = mono.tv_nsec;
  snprintf(filename, sizeof(filename), "%s_%lu.%lu.pcap", timestr, sec, nsec);
  int fd = open(filename, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
  dup2(fd, 1);
  dup2(fd, 2);
  close(fd);

  execlp("/data/local/tcpdump", "/data/local/tcpdump", "-i",  "any", "-p",
         "-s", "0", "-w", filename, NULL);
}
```

Listing 5: An excerpt from the track_device module showcasing the activities of the screenrecord thread

```
        //Main Thead
        screenrecord_thread = fork();
        if (screenrecord_thread == 0) {
          //Screenrecord thread
          execlp("screenrecord", "screenrecord",
                 "/data/local/videos/capture.mp4", NULL);
```

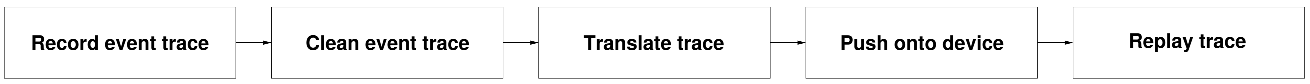


Fig. 3: User flow in the original Android *RERAN* framework.

III. Performance Evaluation of Service Access Through Native Mobile Applications and Mobile Web

In this section, we describe the application of the developed framework to compare the on–device resource utilization performances of two native applications with Chrome browser–based service access based on several repetitions to derive meaningful insights. We note that Chrome is also becoming a common webview supplier within the Android OS and selected to demonstrate ideas here, with results potentially different for other browsers. An in–depth comparison of different browsers, however, is out of scope here.

A. Experimental Setup

For the purpose of our experimental study, the resource effects of the task of launching and logging into two common social network services (*Twitter* and *Facebook*) were measured and evaluated. The tasks for both service access methods, whether via the *Chrome* browser or the services' dedicated mobile application, consist of the following steps:

1) Boot the application (Chrome, Twitter application, or Facebook application)
2) Provide user name and password (login)
3) Wait for assets to load

In addition, an initial neutral state for the device was defined as that of all user–initiated processes being shut–down.

The actions for each of the above steps were to be user–input on a real device while being connected to the host computer. Once the actions were initially recorded, the actions required to reset the respective application back to its neutral state were recorded and stored in the same manner. The experiment then used these recorded events to automate 50 experimental data gathering executions of the actions required for the steps comprising the task of logging into a social network and viewing the result. Similarly, the steps required to return the applications to their neutral state were played out between each experimental data gathering run.

Due to the nature of the underlying framework, actions are replayed without context. Outside factors, such as network speed and variability of loading times, can influence the accuracy of a replayed event. In order to mitigate this, a 10 second sleep operation was placed at the tail end of the recorded trace finishing execution, as well as deliberately slowing input when recording the initial user–trace.

While all details of memory and process resource utilizations can be obtained, a logic grouping with respect to those processes that are non–targeted is required to maintain the ability of general oversight. (We note that we capture the detailed process information on a finer granularity, but employing the individual detailed performance measures for evaluations here is prohibitive for clarity of presentation purposes.) Memory groups are defined through the system default mappings, namely (i) *native* C/C++ executables, (ii) background *services* (including system services), (iii) applications with *cached* data or cached themselves, (iv) applications *visible* to the user, (v) *foreground* processes and subprocesses the user interacts with, and (vi) *persistent* processes. They can generally be described as:

Native

The Native group contains all running C/C++ executables. Big contributors to this group include the mediaserver, surfaceflinger, and tcpdump. Many of the components of the framework operate in this group, such as the screenrecord component and the network monitoring component.

Services

This group contains all background services. The system/system_server thread is grouped here because

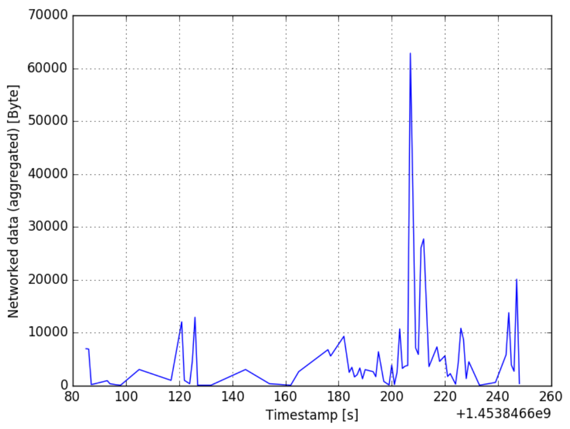


Fig. 4: Network Activity, Experimental Run 1

it includes all native system services. The system/system_server thread is not only used by a number of applications that offload services to the background, but is also used by our framework in accessing the system logs for monitoring CPU and Memory activity.

Cached

Applications with cached data and applications being cached in memory belong into this group. For example, the Google productivity suite software fits into this group and recently closed applications have their states saved in cached memory for quicker relaunch.

Visible

These are applications and processes that are visible to the user, such as the keyboard input, and the Google quick-search box.

Foreground

This group usually contains a listing of the primary process (and its subprocesses) that the user interacts with. It also includes the Android Launcher and essential Google Mobile Services.

Persistent

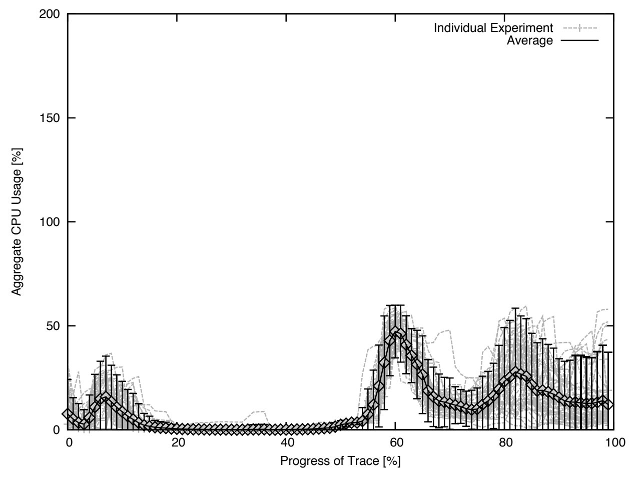
These are processes that are persistently running. These include the Android system user interface, the telephone functionality, and near field communications, for example.

Native services those implemented in the Android OS as C protocols and rolled into the system_server are impossible to associate with an application. The developed framework also includes components that operate on the system_server, which further complicates associating specific off-loaded operations.

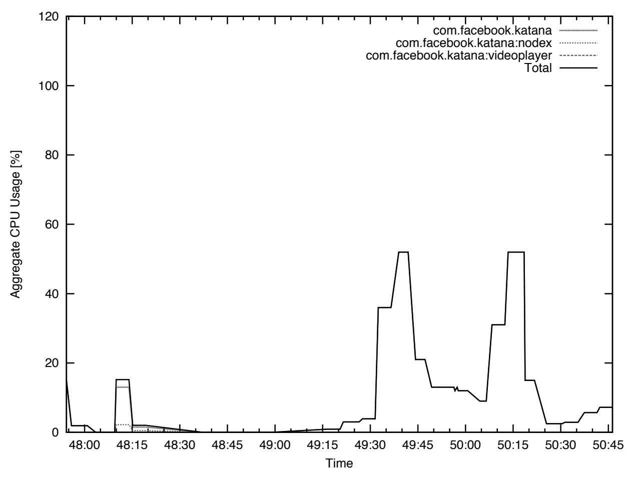
Analysis of the network activity of web and native applications was inconclusive because there was too much noise. The network sensors picked up *all* network activity: bluetooth, https, and http requests from all applications and services. Future work will focus on the filtering of network data, with particular attention to the methods used in previous studies [12]. We employ this specific mapping in the following comparison of resource utilizations between the different service access means.

B. *Facebook*

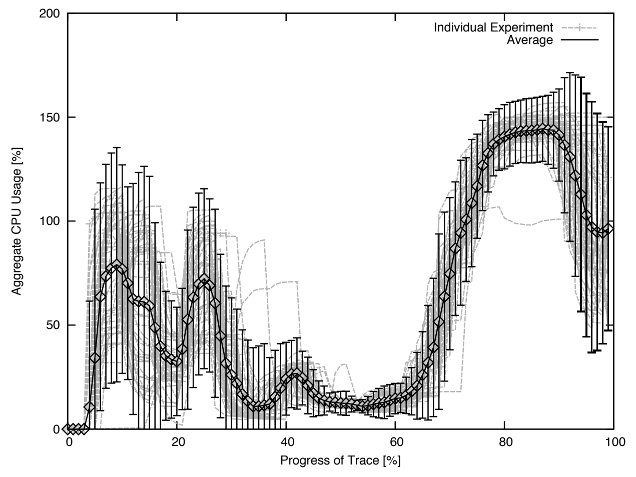
The Facebook mobile application and web–based service access were studied employing 50 replays of logging into the service. Initially, we consider the CPU resource utilization for both access methods, illustrated in Figure 5.



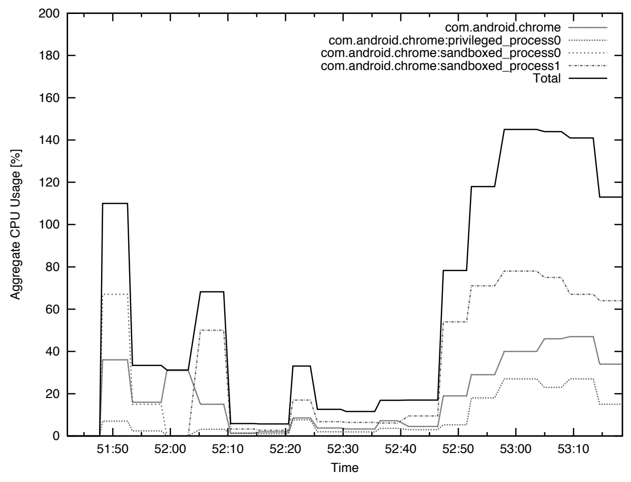
(a) Avg. CPU, App.



(b) Run 16 CPU, App.



(c) Avg. CPU, Web

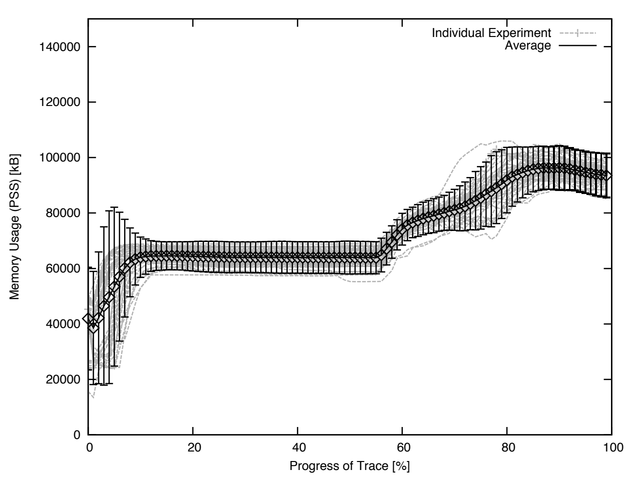


(d) Run 2 CPU, Web

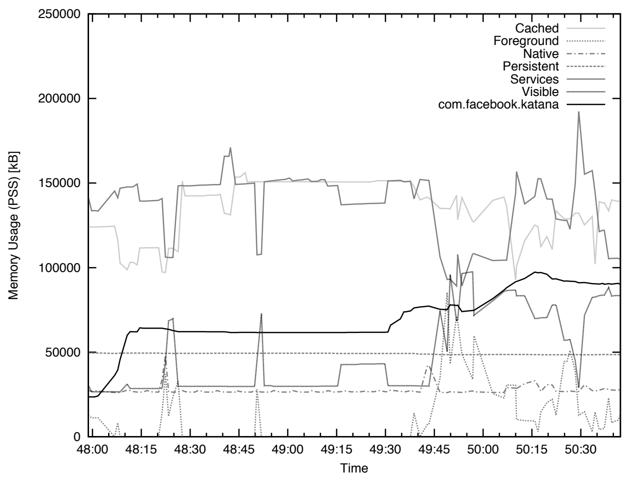
Fig. 5: Overview of CPU usage averages and 95% Confidence Intervals for 50 experimental runs (left) and exemplary individual run with processes involved (right) for mobile application and browser–based service access to Facebook. The device employed had 2 CPU cores for a maximum of 200% CPU utilization.

We note for the mobile application a small increase in the average CPU utilization, followed by a period of low usage, which is trailed by a period of heightened activity. The prolonged period of CPU inactivity for the Facebook application, in the range of 20–45% of the trace, coincides with user input being entered into the system. The activity of the target process can be broken down between its subprocesses and its main thread. By default, the Facebook application operates on a single monolithic thread that delegates small tasks to subprocesses. The browser–accessed Facebook web application consistently exhibited higher CPU usage than the native application as well as much more variability in general. The breakdown of the browser processes also exhibits significantly different characteristics in comparison to the native application. The web application employs more subprocesses than the native application, and offloads more work onto them. Overall, we note that the Facebook application exhibits a much lower CPU usage than the browser application, on average. The application would rarely spike to the 50% usage mark while the browser application even exhibits periods of multi–core utilization.

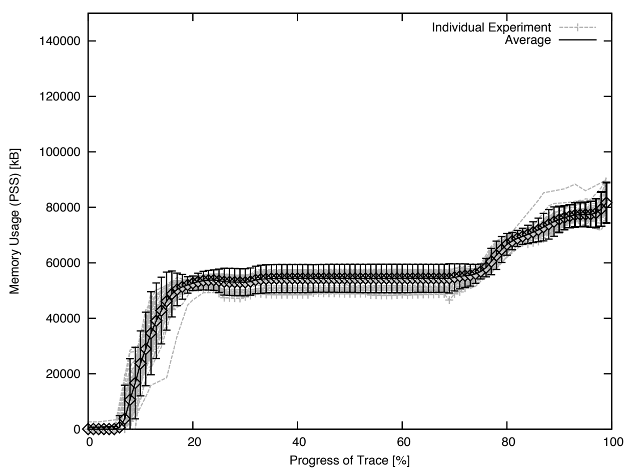
Additionally, we note for the overall stability of these results that in some time periods higher variability can be observer, especially for the browser–based service access. This increased variability in the individual executions of



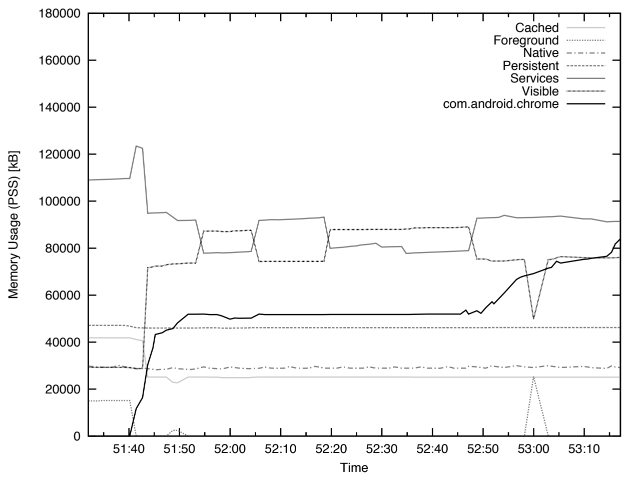
(a) Avg. mem., App.



(b) Run 16 mem., App.



(c) Avg. mem., Web



(d) Run 2 mem., Web

Fig. 6: Overview of memory usage averages and 95% Confidence Intervals for 50 experimental runs (left) and exemplary individual run with processes involved (right) for mobile application and browser–based service access to Facebook.

the playout can readily be attributed to the additional background activities required for the browser–based content presentation. We note, however, that the overall trends are all very similar and our framework is, in turn, able to be utilized to provide a statistically more relevant gathering of experimental results by enabling repeatability of experimentation.

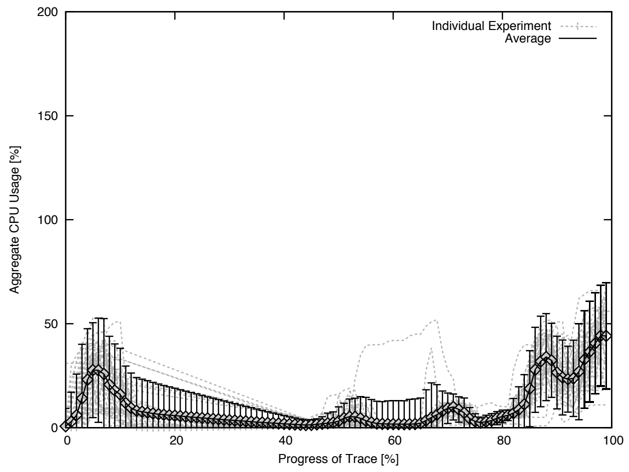
We now shift the view to the memory usage of the Facebook application and browser–based service access and illustrate the averaged memory requirements and an exemplary individual experimental run in Figure 6. Initially, the Facebook application draws in enough memory to launch, about 60 MB PSS. Once the user begins logging in, the Facebook application begins caching resources, such as newsfeed images, text, and video, which is indicated by the increased memory requirements in the last third of the memory trace and additionally reflected in the increased CPU activities during that period (as illustrated in Figure 5). In memory, the Facebook service access through a browser application generally follows the same trend as the application–based accessed. In both cases, the service loads remote resources after successfully completing the user login procedure. Upon closer inspection, however, the browser–based service access requires less memory resources, although its memory usage was rising near the end of *every* experimental run, again due to loading of remote resources.

Each individual trace can also be compared with regards to the overall Android system memory resource utilizations. When compared to the remainder of the system, the Facebook application seems to require an overall average amount of memory resources. In experiment 16's data, for example, the largest contributor to memory is constituted by the Cached process group. Further investigating the detailed contribution within a process group yields that in the case of experiment two, the largest contributors to the Cached group are the Google productivity suite applications: Slides, Docs, and Sheets. Note, however that due to the cached status, these memory utilizations have less impact on overall performance. When examining memory usage for the browser–based service access, the most notable change is that the Cached group is almost consistently exhibiting significantly lower levels of memory utilization. In this scenario, the Chrome browser is a major contributor. In the *Visible* grouping, which makes a consistently strong showing, the Google Mobile Services processes dominate the group.

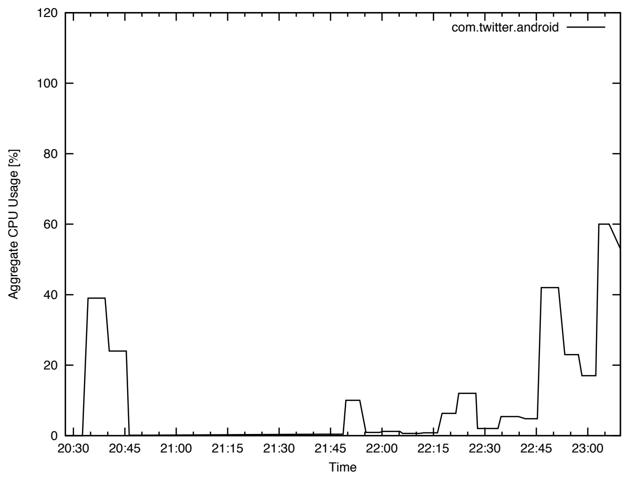
C. Twitter

The second social network provider service access we evaluate is based on the dedicated Twitter mobile application and the browser–based service access, with an approach similar to the Facebook one. We initially illustrate the CPU utilization for the Twitter service access in Figure 7. The Twitter application exhibits a surprisingly low and fairly consistent CPU usage level, with the increase utilization towards the end, similar to the observations made for Facebook in Figure 5. The dedicated application exhibits a consistently lower level of CPU utilization than both the mobile and web versions of Facebook. Unlike the Facebook application or the Chrome browser, the Twitter application operates as a single monolithic process, as indicated by the detailed process view for experimental run 4. The Twitter application exhibits a low and fairly consistent CPU usage level, with a slight increase towards the end. As indicated by a detailed process evaluations, the Twitter application operates a single monolithic process. In other words, the Twitter process handles all matters Twitter, except for those it hands off to the system. Similar to the Facebook browser–based service access, the overall CPU usage observed for Twitter is relatively high in the Chrome web application, but does not reach levels attained for Facebook. Observations made for the stability of results reported apply here as well, as periods of high variability are interspersed with those exhibiting narrow 95% confidence intervals.

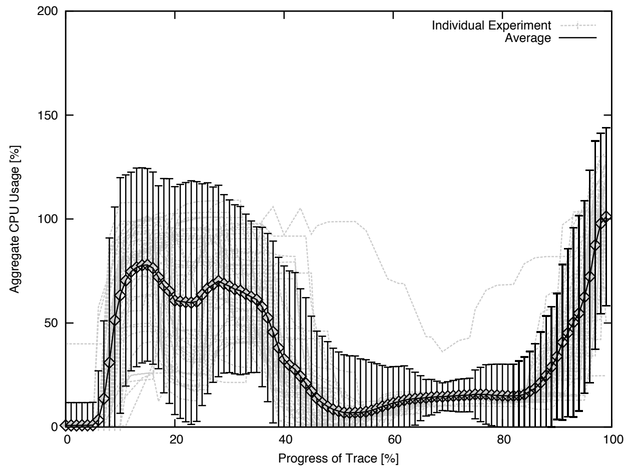
We now shift the view to the memory resource utilization, illustrated in Figure 8. The memory usage of the Twitter application is around 40 MB, lower than for the web–based service access. The mobile application, however, begins automatic media display, which, in turn, is the cause for larger amounts of cached memory for the system's mediaserver. This amount is far lower than observed earlier for Facebook and the web–based Twitter service access. This low memory utilization, however, is at the expense of a huge amount of cached memory, which is primarily constituted by the mediaserver. Twitter, as a mobile application, automatically begins displaying images and playing any videos which may appear in the logged in user's stream and employs the built–in playout mechanisms of the Android OS. The amount of memory used in the browser–based scenario is overall fairly stable and comparable to the observations made for Facebook. Similar to the Facebook browser–based service access, we note that the Chrome browser here splits the workload among many subprocesses and distributes fairly evenly as well. The services group, however, dominated the memory usage in experimental run 4. Little more than half of the contribution to this group was due to the mediaserver, Chrome's privileged process, and surfaceflinger. Mediaserver is the system process that accesses media on-device, which implies that the Twitter application is caching Twitter images and videos, and pulling them from cached memory. Surfaceflinger servers to compose all commonly reusable UI elements in Android.



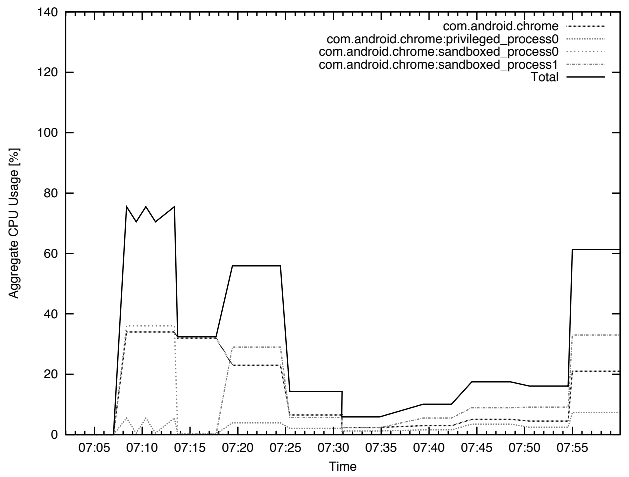
(a) Avg. CPU, App.



(b) Run 4 CPU, App.

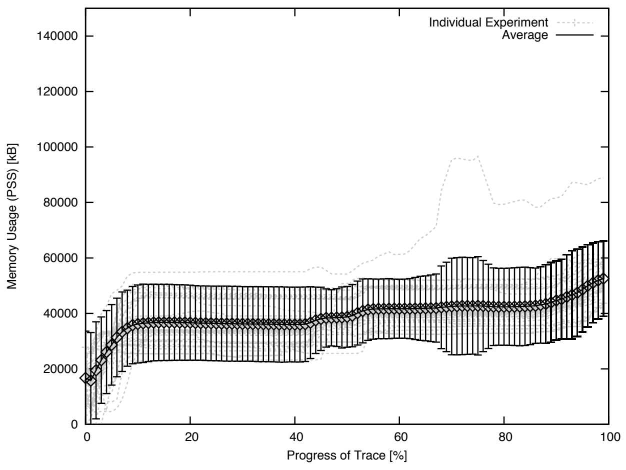


(c) Avg. CPU, Web

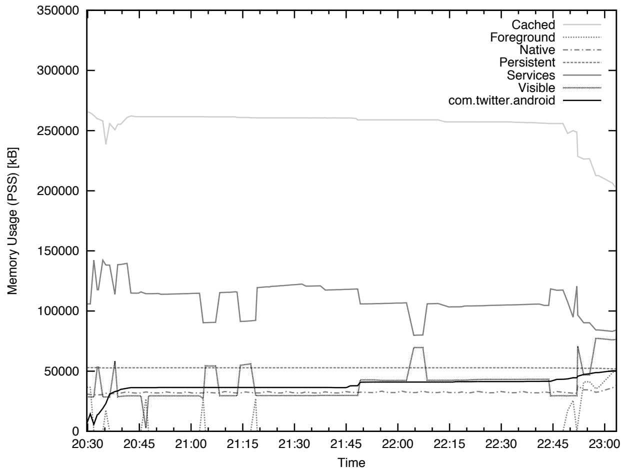


(d) Run 4 CPU, Web

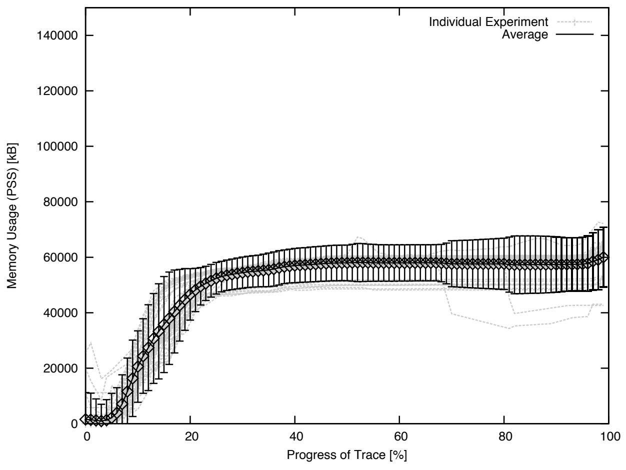
Fig. 7: Overview of CPU usage averages and 95% Confidence Intervals for 50 experimental runs (left) and exemplary individual run with processes involved (right) for mobile application and browser–based service access to Twitter.



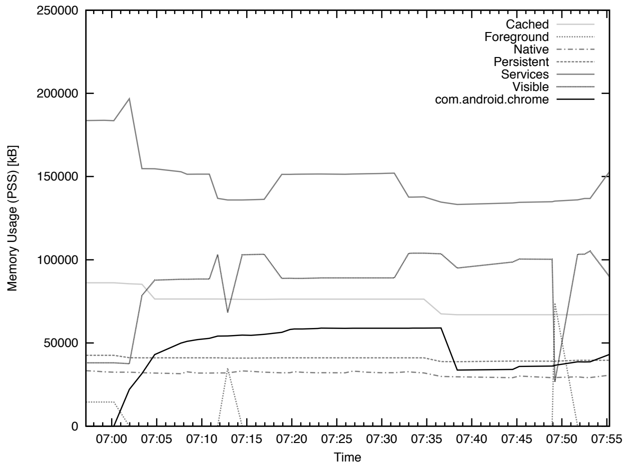
(a) Avg. mem., App.



(b) Run 4 mem., App.



(c) Avg. mem., Web



(d) Run 4 mem., Web

Fig. 8: Overview of memory usage averages and 95% Confidence Intervals for 50 experimental runs (left) and exemplary individual run with processes involved (right) for mobile application and browser–based service access to Twitter.

IV. Conclusion

In this paper, we presented a mobile (web) application profiling framework that enables repeatability of experimentation on different mobile devices and at different times for comparisons of optimization approaches. We applied the framework in a comparison of browser and native applications for Facebook and Twitter service access, whereby we observe significant differences in the usage of underlying system resources. The trade-offs between the service access through natively packaged applications and browser-based can be utilized by application developers to decide which approach is most advantageous. Similarly, optimizations such as loading of resources on-demand could greatly be simplified if a real–time decision can be made with respect to resource availabilities on mobile devices. We are currently investigating the application of our framework in the performance evaluation of mobile application optimization approaches for an increased mobile Quality of Experience (QoE). Furthermore, we are investigating the possibilities for switching between the different modes in real–time. The current framework usage indicates some areas for improvement, which is part of our ongoing works as well. Specifically, an fine–grained evaluation of networking and storage resources is of high interest to provide a holistic view on the access of networked services. The ability to have the replay aware of network loads and UI element loads would also improve its functionality, potentially by incorporating screen comparisons as well. Finally, additional translations enabling replay on multiple devices by compensating for differences could provide additional insights for developers.

References

[1] IDC, "PC Shipments Expected to Shrink Through 2016 as Currency Devaluations and Inventory Constraints Worsens Outlook, According to IDC." [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS25866615

[2] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, USA, Oct. 2013, pp. 15–24.

[3] S. Dhillon and Q. H. Mahmoud, "An evaluation framework for cross-platform mobile application development tools," *Software: Practice and Experience*, vol. 45, no. 10, pp. 1331–1357, 2015.

[4] A. Charland and B. LeRoux, "Mobile application development: Web vs. native," *ACM Queue*, vol. 9, no. 4, pp. 20:20–20:28, Apr. 2011.

[5] D. Sin, E. Lawson, and K. Kannoorpatti, "Mobile web apps – the non-programmer's alternative to native applications," in *Proc. of 5th International Conference on Human System Interactions (HSI)*, Perth, West Australia, Jun. 2012, pp. 8–15.

[6] R. Karthik, D. R. Patlolla, A. Sorokine, D. A. White, and A. T. Myers, "Building a secure and feature-rich mobile mapping service app using html5: challenges and best practices," in *Proc. of 12th ACM International Symposium on Mobility management and wireless access*, Montreal, QC, Canada, Sep. 2014, pp. 115–118.

[7] S.-H. Lim, "Experimental comparison of hybrid and native applications for mobile systems," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 10, no. 3, pp. 1–12, Mar. 2015.

[8] Y. Liu, X. Liu, Y. Ma, Y. Liu, Z. Zheng, G. Huang, and M. B. Blake, "Characterizing restful web services usage on smartphones: A tale of native apps and web apps," in *Proc. of IEEE International Conference on Web Services (ICWS)*, New York, NY, USA, Jun. 2015, pp. 337–344.

[9] V. G. Cerf, "Apps and the web," *Communications of the ACM*, vol. 59, no. 2, pp. 7–7, Jan. 2016.

[10] F. Rösler, A. Nitze, and A. Schmietendorf, "Towards a mobile application performance benchmark," in *Proc. of International Conference on Internet and Web Applications and Services (ICIW)*, Paris, France, Jul. 2014, pp. 55–59.

[11] JetBrains, "Androidstudio memory profilers," Online. [Online]. Available: http://developer.android.com/tools/performance/comparison.html

[12] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of android applications," in *Proc. of 18th Annual International Conference on Mobile Computing and Networking*, 2012, pp. 137–148.

[13] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A first look at traffic on smartphones," in *Proc. of 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 281–287.

[14] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Proc. of IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Mar. 2011, pp. 252–261.

[15] A. M. Memon, "AndroidGUITAR," Online. [Online]. Available: https://sourceforge.net/projects/guitar/files/android-guitar/

[16] ——, "An event-flow model of gui-based applications for testing: Research articles," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, Sep. 2007.

[17] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2012, pp. 258–261.

[18] "Abbot framework for automated testing of java gui components and programs," Online. [Online]. Available: http://abbot.sourceforge.net

[19] Google, "Monkey," Online. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[20] ——, "Monkeyrunner," Online. [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html

[21] R. Reda and et.al., "Robotium," Online. [Online]. Available: https://github.com/RobotiumTech/robotium

[22] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," *SIGPLAN Not.*, vol. 50, no. 10, pp. 349–366, Oct. 2015.

[23] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Touch-Sensitive Record and Replay for Android," in *Proc. of International Conference on Software Engineering*, May 2013, pp. 72–81.

[24] "ELC: How much memory are applications really using?" Online. [Online]. Available: https://lwn.net/Articles/230975/

[25] "Understanding and Optimizing Memory Utilization," Online. [Online]. Available: http://careers.directi.com/display/tu/Understanding+and+optimizing+Memory+utilization

[26] "smem(8) - Linux man page," Online. [Online]. Available: http://linux.die.net/man/8/smem

[27] "TCPDUMP/LIBPCAP," http://www.tcpdump.org/index.html, 2015.

[28] R. Krause, "pcapfix," Online. [Online]. Available: https://f00l.de/pcapfix/