BatTrace: Android battery performance testing via system call tracing

Yeayeun Park
ypark2@swarthmore.edu

Mark Serrano
mserran2@swarthmore.edu

Craig Pentrack
cpentra1@swarthmore.edu

Abstract

The increasing number of tasks we can perform on our mobile devices feeds positively into the demand for devices with longer battery power. Since mobile devices have become integral parts of our daily lives with the number of tasks we can accomplish far outpacing battery performance improvements, consumers have increasingly encountered the issue of efficient device usage and battery life management. In this paper, we examine Android devices in particular and present BatTrace, an Android analysis tool that evaluates battery performance on the android platform by tracing system calls. BatTrace will execute different types of popular system calls, and extract the correlation between a particular system call and its influence on the battery. Subsequently, it will trace system calls made by individual Android applications and use system call performance data to profile each application. Finally, the analysis on the correlation between system calls and their battery usage, as well as the correlation between each application and system calls they initiate, will be combined to estimate battery usage of individual Android applications.

1 Introduction

Our project is motivated by an issue that we face daily: limited battery power on our mobile devices. The vast power available at our fingertips in mobile devices is tamed by the amount of battery physically available. Given that dynamic analysis executes data in real-time to evaluate and test programs, we utilized *strace* in combination with C programs and Android system data to perform dynamic analysis on Android devices, uncovering low-level explanations as to what is really draining the battery.

Hypothesizing system calls to be the key to understanding battery usage at a low-level, we used *strace* to profile applications system call usage. Such testing gave us aggregate statistics on which system calls were used and how often. In the process of tracing popular 3rd party applications (e.g. Facebook, Gmail, etc.), we also recorded battery usage by utilizing built in Android system data pre and post traces.

Once aware of the most used system calls, we created C programs to repeatedly run those calls and collect battery statistics. These programs gave us a measure of how much battery each system call consumes on average. From the aggregate trace data and average system call consumption data, we were able to make predictions on battery usage of 3rd party applications. In collecting high level data on battery consumption, we hope to better inform Android users which applications drain their battery level most, providing a good set of guidelines for mobile users to follow when low battery crisises hit. With our fine-grained system call data, we hope to better

inform software developers and hardware manufacturers which system calls consume the most power.

2 Background and related work

Historically much power consumption research has focused on using utilization-based methods (on individual components, e.g. CPU). However, modern smartphones employ complex power strategies in device drivers and OS-level power management, sometimes rendering utilization as a poor model for representing power states and deducing battery usage (Pathak et al., 2011). While sometimes strong correlation exists between utilization and power consumption, often applications have constant power consumption while in certain states (while utilization fluctuates) or have high power consumption while low utilization (Pathak et al., 2011; Google, 2013). Additionally, measuring utilization via performance counters results in accuracy loss (Pathak et al., 2011). Instead of modeling power with utilization, system calls, the only way of interacting with hardware and performing I/O, serve as a much more precise indicator of power consumption (Pathak et al., 2011). Past work and tools, such as eProf, have shown system calls to be an effective way of modeling power (Pathak et al., 2011; Yoon et al., 2012; Pathtak et al., 2012; Ding et al., 2013). Using the findings of eProf and other studies as justification, we measured and classified system calls on Android smartphones in terms of their effect on battery life.

While eProf foregrounded system calls as an effective indicator of changes in power state, eProf used system calls as a means toward profiling applications' power consumption on a sub-routine level (Pathak et al., 2011; Pathtak et al., 2012). Developing models based off of system calls supplied a powerful tool, however the eProf research did not study battery drain as a result of particular system calls themselves and the frequency with which applications rely on certain system calls. Other work in smartphone battery research, including detecting energy-related bugs, correlating wireless signal strength with battery consumption, and generating battery usage information on the process or application level, has relied on system calls (Yoon et al., 2012; Pathak et al., 2012; Ding et al., 2013). We plan to supplement the research area by focusing our study on the system calls themselves, rather than using them as a means in tracking changes in power state, detecting bugs, measuring signal strength effects, or producing higher-level profilings as explored previously.

3 Our Idea

While dynamic analysis on traditional devices involves the most efficient use of finite computing resources, mobile devices introduce a new problem; finite power. The issue we immediately encounter when trying to analyze mobile software applications is that we ~~almost~~ never have access to the source code of the applications. This is especially true given the fact that most mobile software is proprietary in nature, leaving open source software to the relics that are desktop computers.

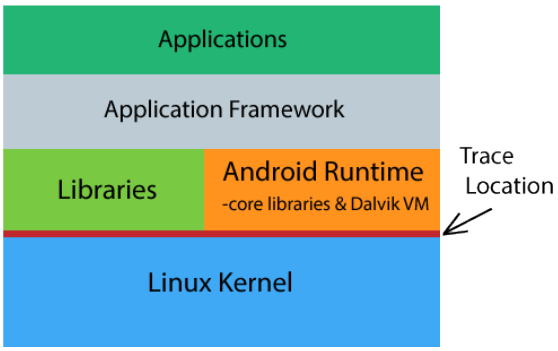

Figure 1: The Android environment stack. Trace location marks where we will be intercepting system calls

With this in mind we set out find a way of measuring mobile battery usage at very low level (software wise). We decided a good approach would involve monitoring activity at the system call level using a tool like *strace*. Ideally, we want to profile a variety of system calls based on how much battery is used while they are running. We intend to establish a baseline battery consumption level so we know how much battery is used by just the OS. Then, using simple programs that repeatedly make the same system call X times, we can determine how much battery was used as a result of initiating a particular system call X times.

Once system calls were profiled, we proceeded to the last phase of the analysis. Our goal was to iden-

tify the system calls initiated by the Dalvik VM as a result of running an individual app. By identifying the types of system calls, as well as the number of calls made to an individual system call, we were able to predict the app's impact on the battery based on what we learned about battery usage for individual system calls. While this approach may not be the most accurate, we believe it is the broadest approach that will allow us to profile any application regardless of the author or the nature of the software's license.

4 Evaluation

In the process of analyzing battery usage through system call tracing we successfully found most used system calls by popular Android applications, wrote C programs to repeatedly execute those system calls, recorded battery usage as a result of particular system calls, gathered system call data of popular apps dynamically, and made predictions based on the aforementioned data. This section is split into subsections based on the above categories.

4.1 Popular System Calls

In order calculate accurate estimates of battery usage, we first traced 10 popular Android applications to acquire a list of the most significant system calls (i.e. ones that were called with the highest frequency). We traced Google Maps, Facebook, Angry Birds, Youtube, Google Play, Gmail, Twitter, Skype, Pandora, and Instagram using *strace*, collecting the ten most used system calls by each. Applications were run for at least 2 minutes simulating normal usage patters. The results appear in the Table 4.1.1.

Distilling the results, we found that there were 14 most frequent and unique system calls. They included: clock_gettime, ioctl, getpid, epoll_wait, getuid32, futex, read, mprotect, gettid, write, gettimeofday, cacheflush, sigprocmask, mmap2, and munmap. While munmap never made the top 10 most frequent system calls for the above applications, it often fell just outside the top 10, so we decided to include it. With this list, we knew which system calls to recreate in order to gather battery usage data.

System Call	# Appearances
clock_gettime	10
ioctl	10
getpid	10
epoll_wait	10
getuid32	10
futex	10
mprotect	10
cacheflush	9
read	7
write	6
sigprocmask	4
gettid	2
gettimeofday	1
mmap2	1
munmap	0

Table 4.1.1 Frequent System Calls in Popular Android Applications. Appearances denote how many times a certain system call was in a top 10 most frequent system call list for each of 10 different applications tested.

4.2 System Call C programs

In developing averages for battery usage per system call, we created C programs for each of the 14 unique system calls. Once these programs were created, we ran a bash script running directly on the device that took a battery reading before and after each program completed. An example system call C program can be seen below in Figure 4.2.1.

```
/* getclocktime.c */
#include <sys/syscall.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/time.h>

int main() {
  struct timespec ts;
  unsigned long long i;

  //750,000,000 calls
  for(i=0; i<750000000; i++) {
    syscall(SYS_clock_gettime,
        CLOCK_REALTIME, &ts);
  }
  return EXIT_SUCCESS;
}
```

Figure 4.2.1 Example C Program for get_clocktime system call.

While the execution time and number of calls made varied among programs, they all ran for at least ten minutes, giving us a significant change in battery. After running the script for each program, we recorded an average of battery level consumption per system call for each of the 14 unique most frequent. See results in Table 4.2.1.

System Call	# Battery Consumption
clock_gettime	XX
ioctl	XX
getpid	XX
epoll_wait	XX
getuid32	XX
futex	XX
mprotect	XX
cacheflush	XX
read	XX
write	XX
sigprocmask	XX
gettid	XX
gettimeofday	XX
mmap2	XX
munmap	XX

Table 4.2.1 Battery Consumption of System Calls. For readability consumption is display per every 1,000,000 calls.

These take a really long time to run. Still in process of getting statistics. Will add a bit more here and fill in values once collected.

4.3 Prediction Data

To make our predictions on battery usage of popular Android applications based off of battery consumption per call statistics, we needed a complete list of system calls made by application including the number of times each system call was made. Using *strace*, we attached to application-specific processes and recorded the desired data. In addition, we recorded the battery level before and after to enable comparison later between predicted consumption to observed battery consumption. The Android applications we used for predictions included Google Maps, Facebook, Youtube, and Pandora. Observed consumption data can be found in Table 4.4.1.

4.4 Predictions

With prediction data (which systems calls are made by a certain application and their frequencies) and average consumption per system call recorded, predicting battery usage was a matter of summing the estimated consumption of each system call. With this summation, we subtracted our power reading for a baseline trace and arrived at a prediction for applications' overall battery usage.

When calculating these predictions, we only had the ability to factor consumption of system calls we had average consumption statistics for. While this may seem limiting, our most popular system calls covered the top 14 used system calls for each application tested. Comparison between predicted consumption and observed consumpted can be seen below in Table 4.4.1.

Application	Predicted	Observed
Google Maps	XX	6
Facebook	XX	4
Youtube	XX	5
Pandora	XX	4

Table 4.4.1 Battery Consumption Predictions and Observations.

Waiting on completion of Table 4.2.1 to complete this section.

4.5 Analysis of Error & Future Work

Once we have Table 4.2.1 we can perform analysis of our error and outline future work. Anticipating that the discussion will include:

- didn't factor in varying system call parameters
- only captured 14 most used system calls for each application
- only tested on one device
- battery levels were imprecise (x/100)

5 Conclusions

CONCLUSION... Waiting on completion of Table 4.2.1 to complete this section.

References

Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. 2013. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In SIGMETRICS '13 Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems.

Inc. Google. 2013. Power profiles for android. Developer Guides.

Abhinav Pathak, Paramvir Bahl, Y. Charlie Hu, Ming Zhang, and Yi-Min Wang. 2011. Fine-grained power modeling for smartphones using system call tracing. In EuroSys '11 Proceedings of the sixth conference on Computer systems.

Abinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In MobiSys '12 Proceedings of the 10th international conference on Mobile systems, applications, and services.

Abhinav Pathtak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In EuroSys '12 Proceedings of the 7th ACM european conference on Computer Systems.

Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference.