Research Statement

Abhilash Jindal

Programmers nowadays are able to implement their code logic fairly quickly by standing on "the shoulders of giant" software frameworks. The intricate framework complexities are abstracted away from programmers and are exposed via easy to use software APIs. Such abstractions have made programming throughput skyrocket, but due to the limited understanding of underlying frameworks, programmers often make mistakes in using the APIs. Moreover, this limited understanding of underlying frameworks has made debugging the software systems and improving their performance significantly harder.

My research vision is to build developer tools that can provide *actionable diagnosis* to programmers for fixing bugs and for improving their system's performance. *After looking at a tool's analysis, programmers should be able to immediately answer how exactly do I change* ***my code*** *to fix failures or to improve the performance of my system as a whole.*

During my PhD, I took the first steps towards this vision. I worked on automated energy debugging and energy optimizations in mobile software. Energy debugging and optimizations has risen to importance in the past decade, since despite the far-reaching societal impact of smartphones, the user experience remains severely limited by the limited phone battery life. I first studied several problems caused due to the incorrect use of power control APIs in Android software and built static analysis and runtime tools to expose such problems. I then worked on providing actionable energy optimization opportunities via diffing energy profiler output of similar apps.

1. Exposing incorrect use of power control APIs

The push for maximal energy savings on smartphones quickly drove their OSes, such as Android, to pursue an aggressive system sleeping policy. To save the crucial phone battery, smartphone OSes aggressively suspend the phone SOC after a brief period of user inactivity. The aggressive sleeping policy, however, can severely impact smartphone software, which may be performing critical tasks whose execution needs to be continuous, i.e., not disrupted by the system going to sleep. We term such code sections over which the system should not suspend as *time-critical sections*.

To prevent phone from suddenly going to sleep, the OS provides explicit power control APIs— wakelocks with acquire and release APIs. Developers now have to perform explicit power management in the app layer to ensure the correct operation of the apps. Programming mistakes in using the power control APIs make time-critical sections not being properly protected by the keep-awake mechanisms. A direct consequence of such mistakes is that the CPU can go to sleep when it is supposed to stay awake, and vice versa. We term such programming mistakes that alter program semantics or cause unexpected battery drain due to smartphone sleep behavior as *sleep disorder bugs.*

In [1], we conducted the first characterization study of sleep disorder bugs and developed a taxonomy of sleep disorder bugs into *no-sleep bugs*, *under-sleep bugs*, and *over-sleep bugs*.

(1) No-sleep bugs happen when a wakelock is acquired correctly, e.g., to prevent the CPU from going to sleep, but is not released in some paths of a program execution. The consequence is that the wakelock-protected device cannot go to sleep, draining the phone battery indefinitely.

(2) Under-sleep bugs happen when wakelock APIs are properly matched, i.e., each wakelock acquire is later matched with a wakelock release, and hence the component can eventually go to sleep. However, the code section protected by the wakelock is larger than necessary, and hence the component is kept awake longer than necessary, wasting energy.

(3) Over-Sleep bugs happen when there exists a section of code that needs to be protected by wakelocks, i.e., a time-critical section, but the developer forgot to protect it with wakelocks, and hence system suspend can happen in the middle of the code section. This type of bugs, unlike no-sleep and under-sleep bugs, can affect the correctness of the app.

The taxonomy enabled us to systematically develop treatment for the class of sleep disorder bugs. In [2], we built the first static analysis tool based on reaching-definitions to automatically find no-sleep bugs in Android framework and apps.

In [3], we built a runtime system that found *sleep conflicts,* a type of over-sleep bugs in device drivers. Sleep conflict happens when a device in a high power state is unable to transition back to the base power state because the CPU is asleep and the device driver cannot make progress in its execution to drive the transition.

In [4], we built a static analysis tool that found a subclass of over-sleep bugs— sleep induced time bugs in the Linux kernel. Sleep induced time bugs manifest as logical errors from unexpected SOC suspension during the manipulation of time related values.

We demonstrated the effectiveness of these tools; the tools found new energy bugs across all software layers including popular apps like Facebook, Android framework, Android kernel and its device drivers.

Together, these papers laid the foundation for systematic treatment of the new class of energy and semantics bugs unique to mobile software and inspired a large number of follow-on work by the research community with over 300 citations (Google Scholar).

2. Providing Actionable Energy Optimizations Opportunities via Diffing Similar Apps

Towards the end of my Ph.D., I co-founded Mobile Enerlytics and have served as a PI on an NSF SBIR grant ($750K) to explore commercial applications of my Ph.D. research, where I have been serving as the company's CTO and growing and managing a team of 4 software

engineers. From this undertaking I have gained valuable first-hand experience in technology transfer and R&D in the computer industry. Most importantly, I learned first-hand about formulating research problems that are relevant to the real industry.

From working closely with mobile app developers, I learned that while mobile app energy profilers provide a foundational energy diagnostic tool by identifying energy hotspots in the app source code, they only tackle the first challenge faced by the developers, as, after presented with the energy hotspots, developers typically do not have any guidance on how to proceed with the remaining optimization process: (1) Is there a more energy-efficient implementation for the same app task? (2) How to come up with the more efficient implementation?

To help developers tackle these challenges, we developed a new energy profiling methodology called differential energy profiling [5] that automatically uncovers more efficient implementations of common app tasks by leveraging existing implementations of similar apps which are bountiful in the app marketplace. By performing approximate matching of energy profiles of similar apps by a traditional energy profiler, energy diffing automatically uncovers more efficient implementations of common app tasks and app-unique tasks among similar apps. We showed how our prototype DIFFPROF tool provides developers with actionable diagnosis beyond a traditional energy profiler: it effortlessly reveals 12 inefficient or buggy implementations in 9 apps, and it further allows developers to quickly understand the reasons and develop fixes for the energy difference.

3. Other Research

In addition to my these research, I have also worked on a number of projects towards understanding battery drain on the smartphones in the wild and developing practical techniques for reducing app battery drain in the wild.

In [6], we undertook one of the first efforts in understanding where and how energy drain happens in smartphones running in the wild and drew several implications to the phone vendors, SOC vendors, cellular carriers, and app developers on better system, network, and app design to extend battery life. One of the implications was that the background energy can be significant for apps: accounting for 27% of the total energy on an average. The study was widely covered in the media such as this [8].

In [7], we quantified all such background activities and possible energy saving and developed HUSH, the first app background activity manager inside Android that applies learning to dynamically suppress app background activities learned to be not useful to individual phone users, and was shown to reduce the average daily battery drain by 16% across a 2000-galaxy phone/user trace. The Android power team had a similar idea and concurrently developed what is known as the “app-standby” feature very similar to HUSH in Android Marshmallow, released at Google I/O 2015. HUSH was released in Github and was forked over 50 times by the Android developer community to retrofit into older versions of Android, and widely covered in media such as BBC.

Future Research Directions

The overall goal of my research is to make debugging and performance optimizations straightforward for an average programmer. I'm constantly amazed at how modern compilers and IDEs can pinpoint syntactical errors and even apply quick fixes. I believe there is significant room for similar improvements when doing performance optimizations and debugging failures. Due to the vast variety of systems and a large number of possible faults, debugging failures requires an arsenal of tools, techniques, and methods. Such an endeavour is fundamentally interdisciplinary in nature. Building successful debugging tools and methods require expertise in programming languages, systems in general, and the particular type of software failures being examined. Moreover, the modern API-driven nature of programming has opened many avenues for applying machine learning towards effective debugging and performance optimization.

In the near future, I would like to continue my work on energy debugging and energy optimizations of mobile software.

Automatic translation of UI tests to similar apps

First, I would like to extend DIFFPROF to improve its adoption by app developers. For running DIFFPROF, developers need to write black box tests for their app and their competitors' apps that perform identical user interactions. However, in my experience working with developers in the mobile industry, I learned developers show significant friction working with apps not authored by them. To overcome this hurdle, I would like to investigate automatically translating black-box UI tests written for one app to its competitor apps. With such a system in place, developers can submit their app along with black-box tests for it, and a list of competitor apps; the system will automatically translate test for competitor apps, run them, and return a list of actionable optimizations using DIFFPROF.

ConfigProf: Accounting Energy Consumption to Configuration Parameters

As framework services and libraries mature and cater to a larger number of apps, their APIs allow an ever increasing number of configurations with each parameter impacting the energy consumption. The primary reason developers do not know how to optimize the use, and therefore the energy drain of framework services is that the services often are triggered implicitly by these intricate configuration parameters, e.g., set in an xml file. To tackle this problem, I propose ConfigProf, which will automatically identify the root cause configuration parameters responsible for framework/library energy consumption. A parameter is responsible for a code section being executed when that code section is control-dependent on the parameter, directly or indirectly. The output of ConfigProf will calculate and show the energy influence of individual configuration parameters or combinations of configuration parameters used by an app. App developers can then modify the configuration parameters to reduce the framework service energy footprint.

Reliable test failures in automated battery tests

At our startup, Mobile Enerlytics, we also built a continuous integration battery testing framework. Our tool enables mobile app developers to run automated battery tests after each commit to the app source code and get alerts if the test is consuming higher battery

than usual. Instead of alerts, we would have liked to directly fail the build, but reliably failing battery tests is a challenging problem. To avoid flaky test failures, the system must be confident that the energy consumption increase was indeed caused by a breaking change in the app source code and not due to changing environments such as changing network conditions, changing signal strength, dynamic scheduling decisions especially in big.LITTLE architecture and memory pressure at the time of test which can significantly alter the energy consumption.

We cannot strictly control the test environment to disallow such environmental changes since it places a significant burden on the app developers' DevOps team. We also cannot simply log and enumerate all the environmental factors that can impact the energy consumption of a test due to the risk of having prohibitive logging overhead and the risk of omitting some important factors. I will thus explore data science and machine learning based approaches, such as clustering, classification and hypothesis testing, to train app-agnostic models that given the raw logs, are able to identify whether a higher energy consumption was a result of a breaking change in the app source code or because of changes in environmental factors. Using the trained model, I would like to further understand and enumerate all the environmental factors that impact energy drain. Such enumeration may expose previously unknown factors that impact the power modeling of phone components.

References

1. **Abhilash Jindal**, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. 2013. On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems - HotPower '13*.
2. Abhinav Pathak, **Abhilash Jindal**, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is keeping my phone awake? In *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*.
3. **Abhilash Jindal**, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. 2013. Hypnos. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*.
4. **Abhilash Jindal**, Y. Charlie Hu, Samuel P. Midkiff, and Prahlad Joshi. 2016. Unsafe Time Handling in Smartphones. In *USENIX Annual Technical Conference*, *ATC' 16*.
5. **Abhilash Jindal** and Y. Charlie Hu. 2018. Differential energy profiling: energy optimization via diffing similar apps. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, *OSDI' 18*.
6. Xiaomeng Chen, Ning Ding, **Abhilash Jindal**, Y. Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone Energy Drain in the Wild: Analysis and Implications. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS '15).
7. Xiaomeng Chen, **Abhilash Jindal**, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (MobiCom '15).
8. Christopher Intagliata. 2015. Smartphone Battery Drains a Lot Even with Dark Screen. *Scientific American*. Retrieved November 14, 2018 from https://www.scientificamerican.com/podcast/episode/smartphone-battery-drains-a-lot-even-with-dark-screen/