Anatomizing System Activities on Interactive Wearable Devices

Renju Liu, Lintong Jiang, Ningzhe Jiang, and Felix Xiaozhu Lin

Purdue ECE

Abstract

This paper presents a detailed, first-of-its-kind anatomy of a commodity interactive wearable system. We asked two questions: (1) do interactive wearables deliver "close-to-metal" energy efficiency and interactive performance, and if not (2) what are the root causes preventing them from doing so? Recognizing that the usage of a wearable device is dominated by simple, short use scenarios, we profile a core set of the scenarios on two cutting-edge Android Wear devices. Following a drill down approach, we capture system behaviors at a wide spectrum of granularities, from system power and user-perceived latencies, to OS activities, to function calls happened in individual processes. To make such a profiling possible, we have extensively customized profilers, analyzers, and kernel facilities.

The profiling results suggest that the current Android Wear devices are far from efficient and responsive: simply updating a displayed time keeps a device intermittently busy for 400 ms; touching to show a notification takes more than 1 second. Our results further suggest that the Android Wear OS, which inherits much of its architecture from handheld, be responsible. For example, the OS's activity and window managers often dominate CPU usage; a simple UI task, which should finish in a snap, is often scheduled to be interleaved with numerous CPU idle periods and other unrelated tasks. Our findings urge a rethink of the OS towards directly addressing wearable's unique usage.

1 Introduction

A major class of wearable devices, as exemplified by Android Wear and Apple Watch, are emerging as interactive personal computing platforms, bringing us one step closer to the vision of "invisible computing" by Mark Weiser.[1] With unique form factors and proximity to the user, these wearables excel in specific tasks, most notably "glanceable" notifications, sensing, and numerous short interactions.

The tiny battery and heat dissipation area, together with the non-decreasing user expectation, challenge wearable system design. Despite this challenge, there lacks an understanding of the wearable workloads, the OS behaviors, and the resultant design implications. Specifically speaking, two compelling questions should be answered:

- Are current wearable devices delivering "close-to-metal" energy efficiency and interactive performance?

- If not, what are the root causes in the system?

In this study, we seek to answer the two questions through systematic profiling. Our key observation is that the use of wearable devices is dominated by a set of *core scenarios*. Analogous to "compute kernels" in high-performance computing, these scenarios decide user experience and device battery life. Fortunately, the number of core scnenarios is fairly small, giving us the opportunity to thoroughly investigate the system activities in these scenarios.

We have identified a set of such scenarios and measured latency and power on cutting-edge Android Wear devices. We have observed that in these seemingly simple scenarios a user often experiences long latency and prolonged device wake time. Some examples of these

---

[1] Recognizing that modern wearables include a wide spectrum of devices, we target the commercially available, interactive devices that support third party apps. They have small displays and tiny batteries, and are intended for daily use.
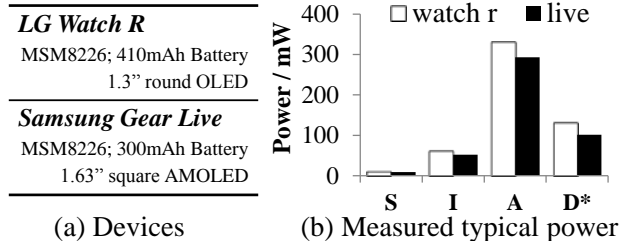
| LG Watch R |
| MSM8226; 410mAh Battery |
| 1.3" round OLED |
| Samsung Gear Live |
| MSM8226; 300mAh Battery |
| 1.63" square AMOLED |

(a) Devices

(b) Measured typical power

Figure 1: *The devices under test. Power in (b): S-device suspended and screen off; I-device idle and screen off; A-device recognizing "Ok Google"; D*-display power only, a white screen with default backlight*

problems include our findings that launching a minimalist app takes more than 1 second, receiving a simple weather update keeps the wearable intermittently working for 5 seconds, and turning the wrist reduces the device standby time by almost 2 minutes.

To unearth the root causes, we equip ourselves with a suite of standard, customized, and home-made tools. With this toolkit, we drill down into the software stack and trace a wide spectrum of activities.

We have the following interesting and sometimes surprising findings: *i)* the OS functions that were intended for managing complex user apps and display overlay incur the major overheads; *ii)* a large amount of short CPU idle periods significantly delay a device's entry to sleep; *iii)* the OS fails to coordinate various software activities for minimizing user-perceived latency; *iv)* UI animation dominates CPU usage during simple user interactions. Our findings suggest that a wearable OS should be engineered with the unique usage pattern in mind. This may be done by trimming down legacy software layers and directly supporting the core use scenarios.

We have made the following contributions: we identify a set of core use scenarios for benchmarking interactive wearables, describe a drill-down profiling workflow, and present a series of insights into the root causes of inefficiency.

2 Background

Wearables have fundamentally different usage patterns than handhelds. According to the UI design guidelines published by Apple [1], users' interactions with wearable devices focus on light tasks, which often last less than 10 seconds. This means that users are likely to use their wearable often, expecting fast responses every time.

Among a couple of wearable OSes, Android Wear is the one with the most public information. Inheriting most of its architecture from of its predecessor - Android for handheld, Android Wear features a redesigned UI for watches. According to Google [10], Android Wear is

intended to support a small set of interaction scenarios, including Cards (for notifications), Actions (for smartphone control), Contextual, and Voice command. At the time of writing (Apr 2015), the published Android Wear source is insufficient for generating a functional build, raising multiple interesting challenges that will be discussed in Section 4.

We test two popular Android Wear devices, the LG Watch R and the Samsung Gear Live, as summarized in Figure 1. We find two hardware trends worth noting: although they both feature quad Cortex-A7 cores at a maximum clock rate of 1.2GHz, the stock OS only uses one core fixed at 700 MHz. Unlike handheld devices where the display power often dwarfs the SoC power, the display power of wearables is comparable to that of SoC, making energy-efficient processing even more important.

3 Core Use Scenarios

Our driving motivation is that the use of wearables is dominated by a small number of core use scenarios, on which system profiling and design should focus. We identify a representative set of scenarios that fall into three categories.

Notification: The device has some new information, e.g. weather update, that may deserve user's attention. To display a notification, a wearable device often presents brief texts or static images in a non-disruptive fashion, e.g. as Cards [10]. In daily use, notifications can be frequent and diverse.

Sensing: The device acquires contexts, e.g. user physical activities, by sampling and processing I/O data periodically. As a wearable system is heavily driven by various contexts [4], these scenarios are frequent and diverse.

Direct Interaction. The user briefly manipulates UI such as scrolling or navigating among Cards [4] or query information through voice command. Each direct interaction often lasts for a couple of seconds.

These scenarios under test are summarized in Table 1. These are by no means exhaustive, but we hope we have captured typical and representative ones.

4 Profiling Methodology

Focusing on these core use scenarios, we seek to answer the two aforementioned questions through systematic profiling.

To obtain both an aerial view and interesting close-ups, we need to cover a wide range of system components and do so at different temporal and spatial granularities. However, the high overhead coming from fine granulated profiling is likely to skew our measurements,

Table 1: The core use scenarios under test. Measurements are collected from LG Watch R externally.

	Scenario	Description	Duration (D) or Latency (U) /ms	Energy (E) /mJ Power (P) /mW
bkgnd	update	A minimalist watch face is updated with a new minute value.	D: 435	E: 63
	notif	Receive a weather Card from the phone (over Bluetooth).	D: 5100	E: 592
	motion	User's wrist motion wakes up the device.	D: 7170	E: 1515
sensing	accel	A minimalist program sampling the accelerometer. Screen is off.		P: 83
	heart	A minimalist program sampling the heart rate sensor. Screen is off.		P: 105
	baro	A minimalist program sampling the barometer. Screen is off.		P: 113
Interaction	lch.set	Screen is on; touch to launch "System Settings" (preloaded in mem).	U1:967 U2:300	E: 485
	lch.calc	Screen is on; touch to launch a calculator app (light).	U1:950 U2:342	E: 672
	lch.game	Screen is on; touch to launch DeadlySpikes (heavy).	U1:283 U2:2708	E: 1634
	voice	Screen is on; speak "Ok Google" to activate voice command.	U1:242 U2:525	E: 769
	touch	Touch an asleep watch to light up the watch face.	U1:225 U2:933	E: 1627
	navi	Activate the watch; pop up the "Weather" card; navigate and dismiss.	U1:92 U2:92	E: 4824

(From power monitor).…….………………………………**P**: average power; **E**: energy; **D**: the duration of device being awake
(From slow motion video)..**U1**: latency from user input end to UI animation start; **U2**: latency from UI animation start to its end
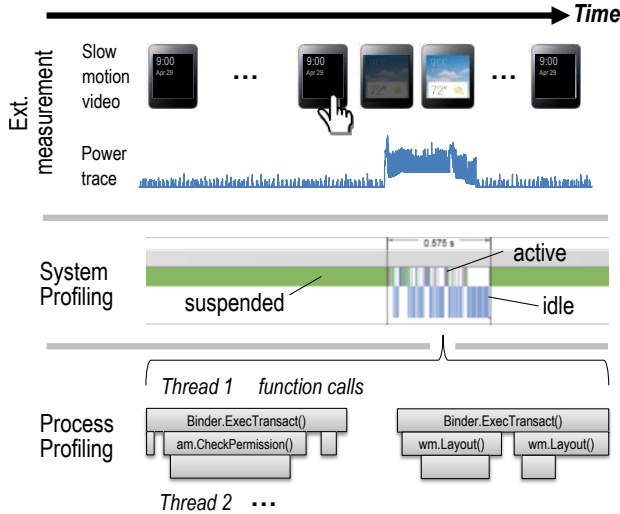

Figure 2: *An overview of our drill-down profiling methodology (with an example scenario)*

as will be shown in our experiments discussed in Section 4.1.

To address this challenge, our profiling follows a drill-down approach done at three levels. Each level has an increasingly narrower scope and finer granularity, where the higher level steers the focus of the lower level. They are summarized below and shown in Figure 2.

P1 **External measurement.** Without tapping into the device, we measure power and latency as the *ground truth*.

P2 **System-level profiling.** Zooming into specific time windows, we trace events that happen inside the OS (both kernel and daemons) and across processes.

P3 **Process-level profiling.** Further zooming into specific processes, we capture the function call history.

The multi-level profiling raises an extra challenge: in order to correlate the outcome across levels (e.g. mapping a long delay observed in P2 to specific function calls traced in P3), multiple levels of profiling must be turned on in the same run, in which the overhead from a lower level may skew the measurements taken at a higher level. To have non-skewed measurements, we repeat each profiled scenario multiple times and apply one extra level of profiling *incrementally*: first run with P1 only, the next with P1 and P2, and the final run with all three.

4.1 Profiling Workflow

Next we sketch our profiling methodologies at each level.

External measurement: We focus on getting the *ground truth* on two metrics: user-perceived latency and device power. For latency, we shoot slow motion videos of the wearable device under test and measure the interaction latencies by manually analyzing video frames. To do so, we use an iPhone 5s shooting at 120fps, which provides sufficient temporal resolution for identifying long latencies in the order of hundreds of milliseconds.

To measure power, we replace a device's battery with one Monsoon power monitor, a standard way for measuring mobile device power consumption.

System-level profiling: Focusing on interesting time windows (e.g. long latency, recurring activities) revealed in the external measurement, we use system-level profiling to learn the overall system workload and pinpoint resource-demanding system components. We use ftrace, a Linux kernel facility that provides API for both user and kernel code to generate globally ordered trace records. We have experimentally confirmed that ftrace introduced little power and latency overheads, which are even smaller than the standard deviations of the external measurements.

We tap into a set of pre-existing ftrace tracepoints in the OS. In the kernel, these include events about context switches, CPU idle states, binder IPC transactions, and block I/O. In the Android framework, these include user touch inputs, activity manager, window manager, and display. To further gain visibility into system suspend and resume transitions, we backported the suspend tracepoints from Intel's SuspendResume project [13] to our kernels.

As a result, one captured system-level trace is a list of ftrace events with their global timestamps and the identifiers of their enclosing contexts, such as threads.

Process-level profiling: System-level profiling helps to identify resource-demanding processes, e.g., those using long CPU time. However, this information is still too coarse-grained. A daemon process, for example, often run various OS services on behalf of different user apps. To further track the root causes of inefficiency we need to discover what functions were responsible for the excessive resource use. Usually, such function information can be extracted from program's debugging symbols, which is a method used by popular profiling tools such as Linux perf. Unfortunately, these symbols are absent on the production Wear devices being profiled.

To overcome this limit, we leverage the fact that most important Android processes – from apps to OS daemons – run atop Android runtime (ART), which has a built-in function tracer. Once turned on, the tracer will keep a history of function calls happened in a given process: the function name, the entry and exit timestamps, and the enclosing objects and threads. The function and object names provide fine-grained information for us to infer the workload semantics.

In this function trace, each timestamp has two versions generated by two clocks: the global clock, a rough equivalence to the wall clock; a thread-local clock, which only ticks when the thread is running. The dual clocks are proven useful: we use the global clock for correlating function calls with system-level activities and use the thread clock to determine a function's CPU usage.

The process-level profiling incurs a high overhead, e.g., it extends the launch latency in *lan.set* by 1.7x. This justifies our method of incrementally applying process-level profiling discussed earlier.

Post-analysis: We collect the absolute numbers of latency by only turning on the system-level tracing in running scenarios. In order to further break down the latency, we run the scenarios again and collect both the system-level and process-level traces at the same time, whose timestamps are taken from the same global clock.

To correlate the system-level and process-level traces, we build an analyzer that automatically reads in the system-level trace, identifies "hotspot" time windows based on hand-coded rules (e.g., app launching, device wakeup), and uses the identified time windows to filter function calls in the process-level trace.

4.2 Tool Customizations & Caveats

Due to the wide scope and great details we seek to cover, no publicly available profiler works for us out of box. To achieve our goal, we set up the profiling environment by deep customizing public tools, patching the kernel to overcome limits, and building new utilities.

Enabling untethered tracing: The official Android profilers, e.g., systrace for system-level tracing and traceview for process-level tracing, all require a wearable device to be "tethered" (i.e. attached to the development PC through USB) streaming data throughout the entire profiling session. Unfortunately, tethering prevents a device from entering suspend, a power state vital to almost all core use scenarios.

To enable untethered tracing, we have modified the official profilers and complemented them with our helper scripts for both PC and device. As a result, the modified profilers can start (both system-level and process-level) profiling on a tethered wearable device. Once started, tracing will continue even after the device is disconnected from the PC and undergoes various power state transitions. When the tracing is finished, device can be tethered again and the trace buffered in device's memory can be transferred to the PC.

Unifying global clocks: As stated previously, an essential step in our profiling methodology is to correlate the system-level events with the process-level events. This requires all related events to carry timestamps from one unified global clock. Despite that ART uses monotonic clock for the process-level trace, ftrace, the system-level tracer, provides no option of using a monotonic clock as in Linux 3.10. Since we cannot rebuild ART due to the lack of full Wear source, we have patched the kernels to add the monotonic clock support to ftrace. Related utilities are also modified accordingly.

Tracing without root privilege: As expected, capturing system-level events with ftrace requires root privilege. Due to the security feature introduced in Android 4.4, any root shell obtained through the well-known "device rooting" must depend on a user-level daemon call daemonsu [5], which, according to our observation, runs periodically and brings CPU out of idle spuriously. Rewriting Android's default security property does not work on Wear: it results in an unbootable image.

To eliminate the need for daemonsu, we patched the kernel to relax the permissions on the ftrace interface, and killed daemonsu before any profiling.
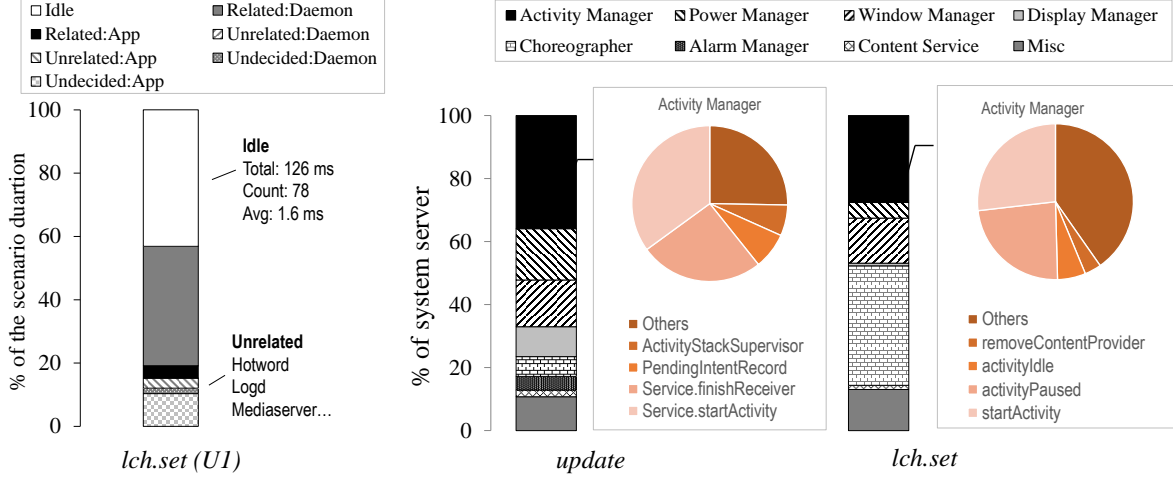
(a) A breakdown of global CPU usage   (b) A breakdown of CPU usage of `system server`, a key OS daemon

Figure 3: CPU usage during test scenarios. Device: LG Watch R. See Table 1 for scenario description.

5 Top Findings

Our top-down profiling essentially puts the core scenarios under a microscope, opening the gate to a number of new findings. Next, we present a couple of discovered symptoms and then the major causes of them.

Symptoms: inefficient pacing and sluggish racing

The external measurement described in Section 4 has answered the first question in Section 1, by showing two serious symptoms.

The *background* scenarios ("pacing") are often highly inefficient: to complete a simple task, the system is woken up from suspension and works at an intermittent schedule for a prolonged period. In *update*, all the useful work is changing the displayed minute on a minimalist watch face UI. However, this simple task keeps the device awake for around 400 ms, of which 88.7% is CPU busy time. In this per-minute task, 63 mJ is consumed; in one day, this reduces the standby battery life by 1.5 hours. The situation is even more severe in *notif*: to receive a weather update from smartphone, the device is kept awake for around 5 seconds (of which 22.5% is CPU busy time) and consumes around 600 mJ.

Direct interactions ("racing"), on the other hand, often see long latencies. As shown in Table 1, touching to wake a device takes 1.1 seconds (*touch*); launching the lightweight Settings activity that already resides in memory takes 1.3 seconds (*lch.set*); launching a full-blown app is even more sluggish (*lch.game*, 3 seconds). Given that most user interactions only last for a couple of seconds (§2), these latencies are unacceptably long. Adding more CPU resources does not help much: with four cores running at the highest frequency, the latency in *lch.set* is only reduced to 1.0 second, which is still unacceptable.

OS overhead dominates execution

A majority of the above latency is contributed by the Android Wear OS, most notably its user-space daemons. In stage U1 of *lch.set*, the execution of system server, one key OS daemon, constitute 38.9% of user wait time as shown in Figure 3(a). The situation is similar in *update*, where 55.9% of device wake time is consumed by this OS daemon.

Inside system server, as shown in Figure 3(b), the top CPU consumers include the manager functions for app activities, windows and power. These sophisticated managers are a legacy of handheld, where the overheads were warranted by the diversity and complexity of apps and windows. However, they become heavy burdens for the few, lightweight tasks on wearable.

Sleep procrastination

When the system is briefly awake for a simple job (which almost always involves multiple processes), ideally, the system should finish everything related to the job in a non-stop fashion and go back to sleep as soon as possible. However, our observation is just the opposite – the device wake duration is interspersed with a large amount of short CPU idle periods, which extends the wake time for no good. In *lch.set (U1)*, as Figure 3(a) shows, the scenario duration consists of ∼40% of idle periods, each of which only lasts for 1.6 ms on average. This is also true for simpler scenarios, such as *update* in which 25 idle periods constitute 11.3% of the device total wake time.

We want to stress that the observed idling is interleaved with busy executions and is thus different from the "lingering" tail time that has been reported previously. It is unlikely due to I/O neither, e.g. none of the 25 idle periods in *update* are coincident with network or block op-

erations. Our initial suspects are animation or IPC communication, which we will verify in future exploration.

Uncoordinated activities

By design, after a user input is received, the OS should coordinate various activities to favor servicing the user input. Unfortunately, within such critical time windows a number of unrelated activities are often obstructing the output production. By examining process scheduling history and tracing binder IPC messages, we are able to categorize CPU usage based on the relevance to the pending user requests. Figure 3 (a) shows the case of stage U1 of *lch.set*, where at least 5% of total CPU busy time is due to activities that are known to be unrelated to the launching task. In addition, 12% CPU time is consumed by "undecided" activities that no evidence supporting their relevance to the launching task. This suggests that the OS lacks the key notion of user-perceived latency and thus is unable to optimize towards the notion.

Overwhelming UI animation

UI animation provides appealing eye candy and hides latency. However, our results show that animation often incurs high overhead in user's short interactions with wearables. In *notif.navi*, when the user is swiping to switch display Cards, 15.6% of CPU time is dedicated to the app renderer thread, which, through a rough estimation, accounts for at least 10% of total system power consumption. In reality, the animation power should further include the GPU power which is often significant. Thus, if UI animation will remain as an important UI element, it seems promising to redesign the OS animation framework for much higher efficiency, as will be discussed in Section 6.

6 OS Design Implications

The revealed causes imply a large room for efficiency and latency improvement. For instance, if in *update* only the related app code is executed, the wake duration can at least be reduced by 60%, leading to a 61.4% energy reduction for the scenario. Overall, we believe a 2x increase in power efficiency and 2x reduction in latency for core use scenarios is feasible through careful OS designs, some of which are discussed below.

Cleaning cruft: Android Wear inherits most of the OS architecture from handhelds, which we already demonstrated as clumsy in Section 5. It seems reasonable to purge such "cruft" from the OS by aggressively trimming down the Android core.

Interaction-centric: Driven by the unique UI pattern, we advocate to engineer a wearable OS with the short and impromptu uses in mind. For instance, the OS should track user-perceived latencies and minimize them

on-the-fly. This is much harder than CPU scheduling: one user activity often spans multiple processes and is serviced by a number of randomly selected OS daemon threads. This requires the OS to have a first-class notion for latency and reorganize existing resource management mechanisms around such a notion.

Specialized for core scenarios: A small set of core use scenarios may deserve specialized support. For instance, rendering smooth UI animation requires both CPU and GPU to sprint in rhythm and is often intensive. Fortunately, the workload is fairly predictable. Thus, instead of periodically producing new frames, the OS may speculatively render frames or even replay pre-recorded ones. This allows similar visual effects at a much lower cost, thus qualifying weaker processors for intensive UI animation.

7 Concluding Remarks

Future Profiling Puzzles Through this preliminary study, we have demonstrated modern wearables' inadequacies in efficiency and performance, and have identified a series root causes to account for some inadequacies. Compared to the answered questions, however, we may have raised more to be addressed in future study, e.g. why do numerous CPU idle periods exist in a "sprinting" period, why can't extra CPU resources help to aggressively cut down latency, and how does network and GPU activities affect efficiency and performance.

OS Design Challenges While our findings justify aggressive (or even radical) changes in wearable OS, we argue for maximizing compatibility with handheld OS in doing so. Due to the huge efforts invested in handheld OSes and the app programming framework, a wearable OS should be able to incorporate code from handheld OS when appropriate. Down the road, both OSes are likely to coevolve; any features added to one should be ported to the other with ease. A clean-slate approach to OS design or programming paradigms is unlikely welcomed.

Related Work Much prior work has recognized the significance of interactive systems and has studied key aspects including energy efficiency [3, 17], latency [6, 21], thread-level parallelism [9, 2], architectural characteristics [15], and file I/O behaviors [12, 16]. Although none directly addresses interactive wearables, their methodologies have sparked our study. In particular, our scenario-driven profiling echoes the argument for application-centric workload analysis [12].

Usage pattern has been a driving force in designing interactive systems. As smartphones become pervasive, multiple studies have already revealed the uniqueness

and diversity in its usage [8, 19, 7], in particular the frequent, short user interactions throughout daily use [20]. These studies motivate our efforts on interactive performance of wearables, where interactions are even more brief and user's attention is more precious.

Recent academic studies on wearables are mostly app-specific, focusing on customizing one device for a particular use, e.g. daily sensing [14, 11] and health [18]. While we recognize the necessity of app-specific wearables, it is urgent to understand OS design for wearables that act as computing platforms.

Acknowledgement

References

[1] Apple. Apple watch human interface guidelines. `https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/WatchHumanInterfaceGuidelines/`, 2015.

[2] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proc. Int. Symp. Computer Architecture (ISCA)*, 2010.

[3] A. Carroll and G. Heiser. The systems hacker's guide to the galaxy energy usage in a modern smartphone. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, 2013.

[4] E. Connolly, A. Faaborg, H. Raffle, and B. Ryskamp. Designing for wearables. Google I/O, 2014.

[5] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch press, 2014.

[6] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 1996.

[7] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. ACM Internet Measurement Conference (IMC)*, 2010.

[8] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2010.

[9] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2000.

[10] Google. Ui patterns for android wear. `https://developer.android.com/design/wear/patterns.html`, 2014.

[11] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2014.

[12] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2011.

[13] Intel. Intel suspendresume project. `https://01.org/suspendresume`, 2015.

[14] N. Lane, P. Georgiev, C. Mascolo, and Y. Gao. Zoe: A cloud-less dialog-enabled continuous sensing wearable exploiting heterogeneous computation. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2015.

[15] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows nt. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[16] K. Lee and Y. Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, 2012.

[17] R. LiKamWa, Z. Wang, A. Carroll, F. X. Lin, and L. Zhong. Draining our glass: An energy and heat characterization of google glass. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014.

[18] A. Parate, M.-C. Chiu, C. Chadowitz, D. Ganesan, and E. Kalogerakis. Risq: Recognizing smoking gestures with inertial sensors on a wristband. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[19] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.*, 38(3):15–20, Jan. 2011.

[20] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2012.

[21] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2008.