Understanding the Characteristics of Android Wear OS

Renju Liu
Purdue ECE
liu396@purdue.edu

Felix Xiaozhu Lin
Purdue ECE
xzl@purdue.edu

Abstract

Interactive wearable devices bring dramatic changes to apps and hardware, leaving operating system (OS) design in the mist. To this end, we thoroughly examine the execution efficiency of Android Wear, a popular wearable OS. By running a suite of fifteen benchmarks, we profile four system aspects: CPU usage, idle episodes, thread-level parallelism, and microarchitectural behaviors. We present the discovered inefficiencies and their root causes, together with a series of widespread, yet unknown OS design flaws. Towards designing future wearable OSes, our study has yielded a generic lesson, key insights, and specific action items.

1. INTRODUCTION

Interactive wearable, as exemplified by smart watches, is a newcomer to the spectrum of mobile computers. Being in close proximity to users, it further extends the boundary of mobile-cloud service, integrating computing even tighter with our daily lives.[1] In the year 2014, nearly three million of Android Wear and Apple watch devices have been shipped [7].

Wearable, as compared to existing mobile devices such as smartphones and tablets, exhibits unprecedented characteristics. On top of the software/hardware stack, the user frequently interacts with the wearable throughout daily life; each interaction is dedicated to a simple task and only lasts for a couple of seconds [2, 45, 10]. At the lowest level, a wearable device's small form factor diminishes the screen and battery; CPU runs slower, possesses fewer cores, and embraces simpler microarctechitures.

Operating systems (OS) for wearable, by contrast, evolve rather slowly. They often feature renovated, watch-friendly user interfaces (UI) while reusing most of the core components from hand-

[1]Recognizing the diversity of modern wearable devices, in this paper we use "wearable" to refer to commercially available, interactive devices that support third party apps and are intended for mass market.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MobiSys'16, June 25 - 30, 2016, Singapore, Singapore

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4269-8/16/06. . . $15.00

DOI: http://dx.doi.org/10.1145/2906388.2906398

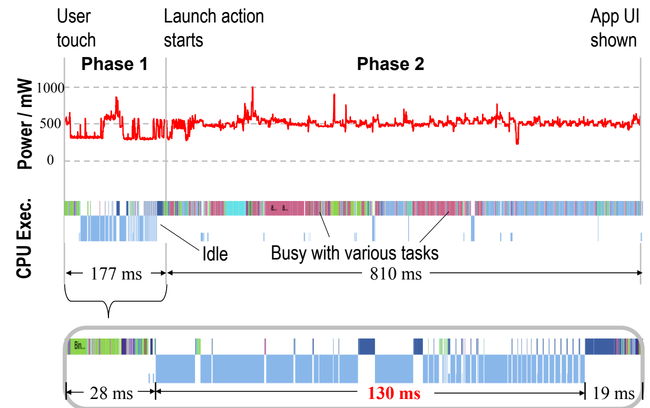


Figure 1: The timeline of launching the "Settings" app on Android Wear, showing both anomalous idle episodes (*not* due to I/O) and intensive CPU execution

held [10, 47] for compatibility and engineering ease. There was neither a clear understanding of how such a design choice impacts wearable efficiency nor a set of guidelines for wearable OS design.

Our pilot study [31] of Android Wear, one of the most popular wearable OSes, presented a framework for analyzing the OS behaviors and discovered preliminary evidences of design inadequacy: many simple scenarios were surprisingly CPU-intensive; numerous idle episodes slowed down system response and wasted energy. This pilot study, however, offered no thorough analysis. Beyond Android Wear, the Apple watch is also reported to show unsatisfactory battery life [50] and slow response [6], although little is known about its internals.

The high power and poor responsiveness are mostly due to *execution inefficiency*. Figure 1 shows a real case observed on an LG smart watch: while the device is awake, the user touches the screen to launch "Settings", a simple app that has already resided in memory. As shown in the Figure, the wearable device takes 987 ms before showing the UI of "Settings" to the user, which is unacceptably long, given that one interaction only lasts for seconds. In phase 1, i.e. from user touch until the launch action starts, the CPU exhibits anomalous idle spanning 130 ms, which we confirmed was not due to well-known reasons such as I/O; in phase 2, the OS performs the launch action and the CPU remains busy for about 810 ms. Note that the CPU, an ARM Cortex-A7 clocked at 780 MHz, is far from wimpy.

To demystify the execution inefficiency, this paper presents a systematic characterization of Android Wear. We focus on its ma-

jor components – the OS shell and daemons – based on two rationales: these OS components dominate the overall CPU execution, as shown in our pilot study [31] and confirmed by the new evidence in this paper; these OS components are likely to remain as the core of future Android Wear systems.

Through running a suite of fifteen benchmarks on two cutting-edge wearable devices, we examine four key system aspects: CPU usage, idle episodes, thread-level parallelism (TLP), and microarchitectural behaviors. For each aspect, our examination covers the global statistics, their fine-grained breakdown, and the explanation of the observation. We have the following major findings:

Many costly OS services are likely software cruft. In the heavy OS execution, a significant fraction of CPU cycles are spent in the legacy code managing app life cycles, complex window layout, or animation. Given the simplicity of wearable apps, much of the code complexity is likely unwarranted.

Basic data structures are hot spots. Tens of hot functions, in particular those manipulating basic data structures, account for most CPU usage of the major OS services.

OS inadequacies and performance overprovision cause anomalous idle. OS flaws in managing voice UI and the power states of I/O peripherals lead to excessive idle time, slowing down system response and device's entry to deep sleep. In continuous interaction, the CPU and GPU performance provision exceeds the compute demand of simple animation, leaving plentiful (>50%) idle time.

System exhibits substantial TLP, showing the need for multicore. The concurrent execution of apps and OS daemons leads to TLP that is up to 2.2. This exceeds typical smartphone workloads and is even on a par with many desktop workloads. This disproves the common belief that wearables only need a single core and warrants the use of multicore.

Complex OS code mismatches the microarchitecture. The OS execution sees high rates of misses in the L1 icache, iTLB, and branch prediction. This mismatch roots in the large OS code base and the poor code locality; it is further exacerbated by the scaled-down microarchitecture of wearable CPUs.

In addition to these high-level findings, our investigation has yielded *i)* a large set of quantitative results such as CPU usage breakdown for individual OS services and the lists of hot functions, and *ii)* the discovery of multiple OS flaws that affect numerous commercial devices, have serious impact, but were unknown before. By presenting them, we clearly point out the targets of future software and hardware optimization.

Lesson & Insight. Our characterization shows a generic lesson regarding the origin of the discovered OS inefficiency: many existing designs that traded execution efficiency for engineering ease or flexibility now cause major bottlenecks on wearables. Towards evolving the wearable OS, we contribute a key insight: today, the inefficiency within individual OS components is the first-order concern. Accordingly, towards making the OS efficient, it is more rewarding to fix individual OS components in question rather than embracing a new OS structure.

We have made the following contributions:

- We present a comprehensive suite of benchmarks covering typical use of interactive wearable devices and a workflow for analyzing the benchmarks.
- For the first time, we thoroughly quantify a commodity wearable OS, present the outcome in detail, and report a series of OS inadequacies that had a wide impact but were unknown before.
- Through an in-depth analysis of the results, we reveal the root causes of the observed inefficiency, discuss their implications, and contribute a set of new guidelines for future wearable software system design.

All the code, data, and benchmark videos are at:

`http://xsel.rocks/p/wear`

2. BACKGROUND & MOTIVATION

Next, we highlight the wearable's unique usage that drives our benchmark design, discuss the hardware trends, and describe the Android Wear OS, the focus of our investigation.

2.1 Usage Patterns

Wearable introduces unprecedented usage patterns. According to the app design guidelines by major vendors [2, 17, 45], users interact with wearable devices frequently throughout daily use; each interaction is short, often less than 10 seconds, and is dedicated to a simple task, e.g., glancing over an email notification. Due to the limited content that can be displayed on one screen, users spend a short time on one screen before switching to the next. Accordingly, a wearable system should cater to such frequent, brief interactions with minimal latency and high efficiency.

2.2 Device Hardware

Compared to handheld, wearable is "leaner" on most hardware specifications, for which we identify three noteworthy trends.

First, the battery capacity is tiny. Ranging from 200 to 400mAh, it is almost one order of magnitude smaller than a typical smartphone battery which is around 2000 mAh. This calls for high energy efficiency.

Second, processing power dwarfs display power. On smartphones, display power dominates the system power. Compared to a smartphone display, a wearable display has 20 – 40× fewer pixels, fundamentally changing the proportion of display power. Our pilot study [31] shows that while a smart watch consumes up to 4 Watts during interaction, its display consumes no more than 150 mW. As a result, optimizing software activities becomes the first priority in quest of energy efficiency.

Third, the CPU is much simpler. To pursue high efficiency without losing software compatibility, wearable devices often embrace a scaled-down CPU that still remains architecturally identical to handheld's CPU. For instance, almost all commodity smart watches, from the Android devices [29] to the Apple Watch [1], use ARM Cortex-A7, a partial dual-issue, in-order core implementing the pervasive ARMv7 ISA. The core is often clocked at 500 – 800 MHz.

2.3 Android Wear OS

Android Wear is one of the most popular OSes for interactive wearables. Although not completely open-source (unlike Android for handheld), it is the wearable OS with the most public information. Targeting the common usage of smart watches, Android Wear supports third-party applications and features a resigned system UI, including Card for notifications, Context streams, and voice input. Assuming a companion smartphone is nearby, Android Wear wire-

lessly offloads to the smartphone heavy compute tasks, e.g., voice recognition.

The apps on Android Wear, despite their renovated UI, follow Android's conventional programming paradigm: they are written in Java, compiled ahead-of-time, and executed atop the managed Android Runtime.

Major OS components. Same as on handheld devices, most of Android Wear's *personality* is implemented as userspace processes. The most important ones are three:

- *System Server* is the key daemon hosting the core OS services, such as those managing app life cycles, display layout, and security policy. It is implemented with around 110K SLoC in over 100 files. It is managed, i.e., running atop the Android Runtime.
- *Surface Flinger*, the daemon controlling UI animation, periodically composites bitmaps rendered by multiple apps into the final framebuffer. It is implemented with around 11K SLoC in 81 files.
- *Clockwork* is the OS shell that implements the system UI such as app launcher and voice input. Its source code is unpublished at the time of writing (Dec. 2015). It runs atop the Android Runtime and is likely written in Java.

Underneath the userspace is a mostly vanilla Linux kernel enhanced by a handful of Android-specific facilities, e.g., WakeLocks [20] and Binder IPC [18]. Enforced by the kernel, all apps and OS processes are sealed in separate address spaces and may communicate through the Binder IPC. Within one given process, threads often communicate through shared message queues.

3. BENCHMARK SCENARIOS

The usage of wearables is dominated by a small set of scenarios [31, 25]. Accordingly, we conduct scenario-centric profiling, a proven approach to characterizing interactive systems [23, 27]. By augmenting the core scenarios identified in our pilot study [31], we design a suite of benchmarks with three goals: the benchmarks should cover common wearable usage as described in app design guidelines [2, 10]; they should exercise a variety of system aspects such as computing, IO, and power management; they should be simple and require minimal user inputs to ease reproducing.

As summarized in Table 1, our benchmark suite consists of fifteen benchmarks falling into the following four categories.

- *Wakeup.* Stimulated by internal or external events, a wearable device transits out of suspended mode and presents brief information. No user input follows the device wakeup. Since wakeup happens frequently throughout daily usage, energy efficiency is the most important metric.
- *Single input.* A waking wearable device responds to a single input from the user, e.g., touch or voice command, a common pattern of brief interactions. Because the user is waiting, the device needs sprinting to achieve low UI latency.
- *Continuous interaction.* Users are interacting with the device continuously, e.g. navigating among Cards [10]. The resultant UI animation requires the device to produce a steady stream of graphic frames, which often requires a synergy between CPU and GPU.
- *Sensing.* A set of minimalist programs sample and process sensor data periodically to collect context information, which is known to drive wearable app execution [10]. These programs keep their UI as a black screen with no updates.

Latency and power consumption. As the measurement in Table 1 shows, these conceptually simple scenarios often exhibit long user-perceivable latency and substantial power consumption. These two metrics, as exemplified by the case in Figure 1, are tightly coupled and are ultimately determined by the system's execution efficiency. This motivates our characterization as will be presented below.

4. METHODOLOGY

We next sketch our overall experiment setup and describe the four key aspects under profiling.

Experiment setup. We run all the benchmarks on two state-of-the-art Android Wear devices, the LG Watch R [29] and the Samsung Gear Live [46], both using Qualcomm's APQ8026 system-on-chip (SoC) and running the stock Android Wear 5.0 "Lollipop". Although the SoC is equipped with $4\times$ Cortex-A7 cores, three are forced offline by both device vendors – a common practice among Android Wear watches. We will examine the CPU details in Section 5.3 and 5.4.

In the rest of this paper, we present the quantitative measurements from the LG Watch R; yet, all the discoveries, e.g., OS inefficiency, have been confirmed on both devices unless otherwise stated.

Power measurement. While it is common to measure a smartphone's power by interposing its battery interface, doing so for wearable is challenging: the latter's batteries have tiny contacts and are incompatible with commodity power monitors [38]. To this end, as shown in Figure 2, we carve out a compatible interface circuit from a smartphone battery by the same manufacturer and use the interface as an adapter between the smart watch and an external power monitor. This technique was unknown to the community before, to the best of our knowledge.

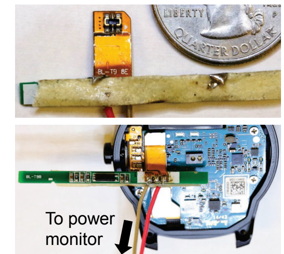


Figure 2: Top: the battery interface carved out from Nexus 5; bottom: the interface (flipped) connected to the LG watch R.

Toolset. To examine system behaviors at different levels and granularities, we have employed a set of tools: systrace [19] for capturing global system events such as scheduling, I/O activities, and IPC; the Android Runtime's built-in function tracer for recording function call history in individual processes; the Linux perf for sampling CPU performance counters. We have further customized these tools for collecting additional information and overcoming the limitation of the current Android Wear [31]. All the customized tools are open for download as mentioned in Section 1.

Tackling profiling overhead. The major profiling overhead comes from event tracing: each tracepoint saves in memory a timestamp and a small amount of event information that is at most tens of bytes. While the overhead is negligible in tracing the relatively sparse system events, it can be overwhelming in tracing function invocations.

We tackle the overhead in two ways. First, in quantifying global system behaviors, we only rely on system events; in order to under-

Table 1: The benchmark suite. The power and energy are measured as described in Section 4; all time values are measured by systrace [19]. Some of the scenarios and their measurement were previously reported in our pilot study [31] and are included here for completeness. The slow-motion videos of the benchmarks are available as mentioned in Section 1.

	Scenario	Description	Duration (D) or Latency (P) /ms	Energy (E) /mJ Power (P) /mW	# of Tasks
Wakeup	update	A minimalist watch face is updated with a new minute value. Device gets suspended.	D: 364	E: 61	81
	notif	Receive a weather Card from the phone (over Bluetooth). Device gets suspended.	D: 4645	E: 539	135
	wrist	User's wrist motion wakes up the device.	D: 407	E: 155	113
	touch	Touch a sleeping watch to light up the watch face.	D: 184	E: 69	102
Single Input	lch.set	Screen is on; touch to launch "System Settings" (preloaded in memory).	P1:177 P2:810	E: 440	92
	lch.calc	Screen is on; touch to launch a calculator app (light).	P1:202 P2:520	E: 376	99
	lch.game	Screen is on; touch to launch "DeadlySpikes" (heavy).	P1:182 P2:1236	E: 775	107
	palming	Screen is on; cover the screen with palm so that the device is suspended.	D: 2343	E: 556	130
	voice	Screen is on; Bluetooth is off; speak "Ok Google" to activate the voice command UI.	D: 1671	E: 1675	65
Interact.	game	The interval between two adjacent touches in a busy game, "DeadlySpikes".	D: 409	E: 241	62
	notes	Speak the word "hello" for 3 times; user waits until the notes recognized and saved.	D: 3897	E: 2449	89
	navi	Screen is on; pop up a set of two "Weather" cards; navigate and dismiss.	D: 4119	E: 2400	67
Sensing	accel	A minimalist program sampling the accelerometer at the default rate (5Hz). Screen is blank.		P: 83	33
	heart	A minimalist program sampling the heart rate sensor at the default rate (5Hz). Screen is blank.		P: 105	21
	baro	A minimalist program sampling the barometer at the default rate (5Hz). Screen is blank.		P: 113	28

P1: from user touch to launch start; **P2**: from launch start to UI displayed

stand their causes, we collect function trace from extra runs. Second, in quantifying function-level activities, we deduct an overhead of 4 μs from each traced function invocation. The overhead is due to bookkeeping each function's entry and exit, and is measured and verified as constant regardless of the traced functions; specifically, we measured the overhead as the delay of tracing a no-op function.

CPU Usage

We collect and analyze CPU usage at two granularities.

Task-level breakdown. To identify the tasks[2] that are the major CPU consumers, we build an analyzer to scan all traced scheduling events and attribute the CPU busy episodes to individual tasks accordingly.

Function-level breakdown. To further locate the performance hot spots in *System Server*, the managed OS daemon where most services run, we tap into the function trace and attribute the CPU usage to distinct managed functions. More specifically, we employ the following two metrics, commonly used in Android app profiling, to characterize one invocation of a managed function:

- *Exclusive CPU cycles* are spent in the function's own code, i.e., excluding any of its subroutines. Intuitively, the total exclusive cycles of a given function imply the potential benefit from optimizing the function *per se*;
- *Inclusive CPU cycles* are spent in the function's code as well as in all subroutines being called. Intuitively, the total inclusive cycles of a given function imply the potential benefit from eliminating invocations to this function.

[2]Following Linux's lingo, we use *tasks* to refer to scheduling entities, regardless of whether they are in the same or separate address spaces.

Note that both metrics include the time spent in both user and kernel spaces; they do not cover the time when a task is off CPU due to being scheduled out.

Idle Time Analysis

In a pilot study of Android Wear [31] we have noticed excessive idle episodes. Although interactive systems are known for being intermittently idle, e.g. due to user think, the amount and duration of the observed idle episodes are unusual.

To reveal the root causes of all observed idle episodes, we first attempt to match them with system events known to cause idle, e.g. I/O and power management. Accounting for the remaining idle episodes, however, is more challenging: they often root in OS service's stalling in serving app's requests. To further obscure the true cause, as the service is stalled, the kernel will schedule in other unrelated tasks, making the CPU intermittently busy as exemplified by phase 1 of Figure 1.

To address this challenge, we build `IdleChecker`, an analyzer that helps mapping anomalous idle episodes to the responsible code regions, based on a simple rationale: the function calls and IPC transactions spanning an anomalous idle episode are suspicious. In a nutshell, `IdleChecker` runs the following steps for each idle episode.

First, it identifies suspicious app tasks that are blocked throughout the entire idle episode but run after the episode. Second, for each suspicious task, it identifies two suspicious CPU time quanta: the one right before the idle episode and the one right after it. Third, `IdleChecker` examines the suspicious quanta, looking for IPC transactions spanning across the idle episode. Finally, the analyzer identifies the function invocations that either coincide with the IPC or span across the idle episode. It maps the names of the identified functions, which are preserved in the trace, back to the source code.

`IdleChecker` effectively narrows down the investigating scope

from tens of thousands of function invocations per benchmark down to a couple of functions, making our idle analysis tractable.

Thread-level Parallelism (TLP)

Despite the popular belief that one CPU core is enough for wearable workloads, we seek to experimentally test it by measuring thread-level parallelism (TLP), a metric widely used for gauging an interactive system's need for core count. Intuitively, TLP is the average number of busy CPU cores during the non-idle time. It is formally defined as follows [15]:

$$TLP = \sum_{i=1}^{n} i * c_i/(1 - c_0)$$

In the formula, c_0 is the total time when no threads are running; c_i stands for the time when exactly i threads are running simultaneously; n is the number of cores available.

To measure TLP, we override the device vendor's default configuration and force all four cores online, each running at their default clock rate. We repeat each scenario three times, identify the busy episodes of every core from the traced scheduling events, and calculate TLP based on the formula above.

Microarchitectural behaviors

Little work has been done on the microarchitectural behaviors of wearable workloads, making the microarchitecture design a mystery. To this end, we characterize the microarchitectural behaviors of the major OS components, due to their dominating CPU usage and importance in the current and future Android Wear systems. By using the Linux perf, we sample the performance counters of the Cortex-A7 CPU on our test devices. The full-fledged performance counters allow us to observe branch prediction, cache, and TLB in all benchmarks.

5. RESULTS

Next, we present the characterization results on the four aspects. For each aspect, we first highlight the key findings, explain them with evidence or case studies, and finally discuss their implications on future software or hardware design. Note that each experiment reported has been repeated at least 3 times.

5.1 CPU usage

Despite their conceptual simplicity, most benchmarks show substantial CPU utilization, as shown in Figure 3(a). In fact, the CPU often keeps spinning for tens to hundreds of ms without a break, as exemplified by the case shown in Figure 1. This observation raises one top profiling question: *where do these CPU cycles go?*

5.1.1 Global CPU Usage

• *Intensive OS execution often dominates the global CPU usage.*

As shown in Figure 3(a), the percentages of the OS-consumed CPU cycles in the total busy cycles range from 48% (*game*) to 93% (*wrist*). A further breakdown among the OS daemons, as shown in Figure 3(b), indicates that two daemons are in particular heavy: *System Server*, which hosts most OS services; *Surface Flinger*, which composites bitmaps for painting UI. It is worth noting that even for the *sensing* benchmarks where the app UI is blank, the OS seems to run *Surface Flinger* occasionally in order to update the entire system UI.

Overall, the observation confirms our intuition: since wearable apps are simple, the cost of OS execution becomes overwhelming. This observation also justifies our OS-focused investigation as stated in Section 1.

5.1.2 OS Services

• *Many costly OS services are likely software cruft.*

Due to the significance of *System Server*, we further examine how its enclosed OS services contribute to the CPU usage. We do this by studying each OS service's inclusive CPU cycles as defined in Section 4. As shown in Figure 3(c), among a variety of OS services, the most CPU-intensive ones are: activity manager, which manages the life cycles of apps; window manager, which manages the complex UI layout of multiple apps; power manager, which enables app to control device power state through WakeLock [20]; choreographer, which controls the timing of UI animation. These sophisticated OS mechanisms, despite being essential to smartphones, e.g. in managing app multitasking or handling screen rotation, become *cruft* in serving simple wearable apps that have minimalist UI.

5.1.3 Hot Functions in System Server

• *The distribution of hot functions is highly skewed.*

• *Manipulating basic data structures consumes substantial CPU cycles.*

• *Legacy OS functions may become serious performance bottlenecks.*

To gain further insights into *System Server*, we break down its CPU usage at the function level. We study the *exclusive* cycles of functions as defined in Section 4. As shown in Figure 4, the distribution of CPU time over all functions is heavily skewed. Although the total number of distinct functions per benchmark ranges from several hundreds to a few thousands, the top 5 "hot" functions constitute more than 20% of *System Server*'s CPU cycles in most benchmarks; the top 50 hot functions constitute more than 50%. This skewed distribution implies highly concentrated performance hot spots. The skewness is rare in commodity OS workloads: for instance, prior measurement shows that the Linux kernel often does not see a small set of functions dominating CPU usage [43].

In understanding the skewness, we have found a small set of functions – those manipulating basic data structures – are hot in almost all benchmarks. As shown in the Total column of Table 2, these functions collectively contribute a significant fraction to *System Server*'s CPU usage, up to 45%. In benchmarks such as *notes*, these functions are even more CPU-hungry than those implemented by the code of *System Server*.

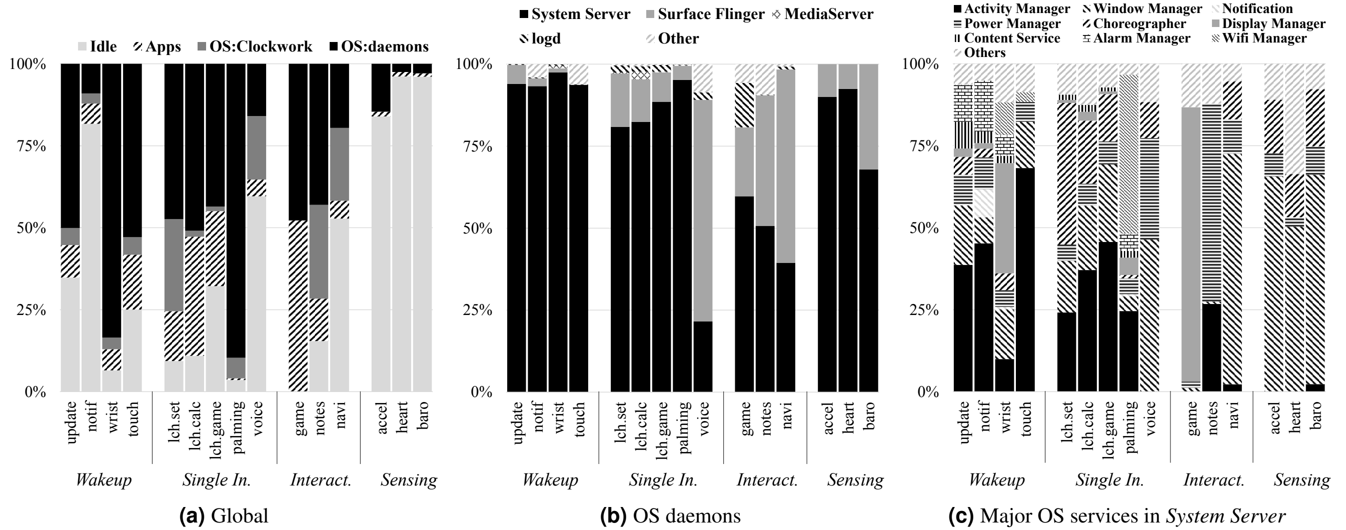


Figure 3: The CPU usage breakdown

A further breakdown shows that these functions concentrate on tens of basic data structures implemented in a couple of software packages, as summarized in Table 2:

- `java.core`: the utility data structures from Java's core API, such as arrays, collections, maps, and dictionaries;
- `java.string`, Java's string implementation;
- `android.util`, the utility data structures, such as SparseArray and ArraySet, from the Android framework libraries;
- `threadlocal`, thread-local variables and objects;
- `android.msgq`, message queues for inter-thread communication.

Although one single invocation of them is as short as a couple of μs, the large number of invocations per benchmark, sometimes multiple thousands, makes their CPU usage substantial as a whole. This can be observed in the full lists of top hot functions in Table 3.

5.1.4 Discovered OS Execution Bottlenecks

In addition to the operations on basic data structures, we found that multiple hot functions, among the top ones listed in Table 3, correspond to serious bottlenecks existing on various OS aspects.

Ad-hoc backlight effect. A mobile OS often varies the screen brightness as it sees fit. Yet, we have noticed `setLight..()`[3], the OS function setting the screen backlight, is expensive, taking more than 100 ms or 78M CPU cycles per affected benchmark. A deeper investigation holds a vendor library (`light.lenok.so`) responsible: in setting the backlight to a new level, the library emulates a transition effect by modulation, for which it repeatedly calls the kernel driver and thus increases the execution cost by $50\times$. This function is invoked by *System Server* every 35 ms to realize another level of light modulation. Together, they create an expensive anomaly we dubbed "modulating of modulation". This bottleneck, rooting in a device-specific library, affects the LG device but not the Samsung one.

UI layout. Due to the limited content that can be shown on a small display, a user often navigates among "Cards", or app windows, frequently. This new UI pattern, however, is poorly supported by the expensive OS mechanism for window switch. Every time a new window is brought to the foreground, *System Server* re-layouts the UI: it updates each app window's display properties, including size, location, and visibility, and runs transition animation. At the heart of this workflow is `perf..Layout()`, a giant function catering to the complex UI layout of multiple apps. On our test devices, each invocation takes 1 ms (780K CPU cycles); in one benchmark that lasts for a few seconds, this function alone may take up to 10 ms (7.8M cycles).

Maintaining low-memory killer. To reconcile app memory demand with the physical memory limit, the OS maintains a low-

[3]For all shortened function names mentioned in this section, see Table 3 for their full names.

Table 2: The percentages of *System Server*'s CPU cycles spent on manipulating basic data structures

	Benchmark	java.core	java.string	android.util	threadlocal	android.msgq	Misc	Total
Wakeup	update	2.6%	5.5%	4.5%	6.7%	6.1%	1.6%	**27.0%**
	notif	4.8%	8.5%	2.9%	8.5%	2.7%	3.3%	**30.7%**
	wrist	4.9%	5.2%	2.1%	3.6%	4.2%	2.0%	**22.0%**
	touch	2.8%	6.1%	8.7%	4.1%	**10.9%**	3.7%	**36.3%**
Single Input	lch.set	2.3%	3.8%	5.5%	6.0%	4.1%	2.5%	**24.2%**
	lch.calc	2.7%	4.2%	4.5%	7.0%	3.0%	3.0%	**24.4%**
	lch.game	2.3%	3.9%	5.1%	7.6%	2.9%	2.5%	**24.3%**
	palming	4.5%	2.1%	3.0%	3.2%	2.0%	2.9%	**17.7%**
	voice	1.4%	3.0%	3.7%	**20.3%**	1.2%	2.2%	**31.8%**
Interact.	game	0.0%	0.1%	1.8%	**36.3%**	0.9%	2.6%	**41.7%**
	notes	4.2%	4.1%	2.7%	**18.7%**	**12.2%**	3.2%	**45.1%**
	navi	1.0%	3.8%	5.3%	7.3%	3.1%	3.4%	**23.9%**
Sensing	accel	1.7%	4.0%	5.1%	5.9%	3.0%	1.5%	**21.2%**
	heart	1.6%	4.6%	5.5%	5.7%	4.3%	1.7%	**23.4%**
	baro	4.9%	6.1%	4.8%	5.1%	2.5%	5.9%	**29.3%**

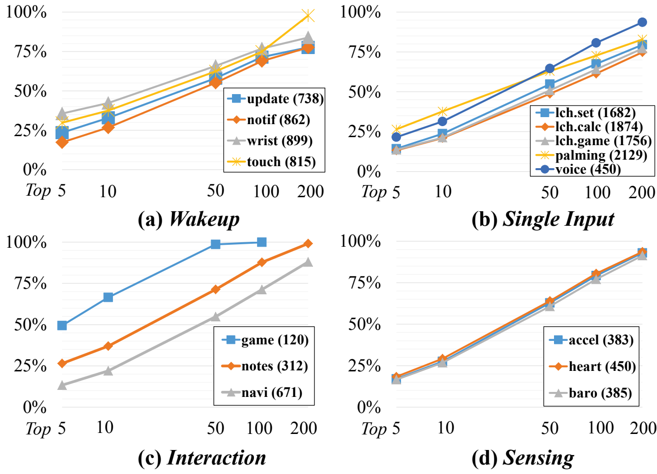


Figure 4: The percentages of CPU cycles spent by *System Server* on its top N hot functions (x-axis, logarithmic). Each subfigure is for one benchmark category. The legends contain the numbers of distinct functions invoked in each benchmark.

memory killer that works based on per-process threshold values. At run time, the OS may kill any process whose threshold is above the amount of available physical memory. By design, the per-process thresholds have to be adjusted in the event of significant changes to memory use, e.g., app startup, window switch, process termination, etc. To do the adjustment, *System Server* executes `computeOom..()` for each process, which turns out to be a heavy burden. For instance, in *update* or *wrist*, this function has been invoked for over 150 times; this function alone consumes around 15 ms or 11M CPU cycles.

Object reflection. A powerful mechanism, reflection enables program code to inspect its objects at run time [49]. However, its lazy behavior may incur significant overhead in wearable's brief use. In suspending a device, *System Server* stops the wireless networking service (among other OS services) by operating the service's state machine. For bookkeeping, *System Server* uses reflection to retrieve the name of a state object (`getSimpleName()`). The reflection leads to a flood of 33,000 function invocations, most spent on loading and parsing annotations from the object's class file (`getAnnotationReader()`). As a result, each reflection takes around 40 ms (3.1M cycles), deferring the device's entry to suspended mode in an expensive way. This accounts for the high CPU usage of WiFi manager in *palming* as shown in Figure 3(c).

5.1.5 Design Implications

Our findings clearly suggest that *i)* OS should be the primary target of system optimization and *ii)* optimizing the small set of hot functions is likely rewarding. In particular, the functions manipulating basic data structures should receive priority efforts, towards which we foresee a few promising approaches: hand-tuned implementation; exploiting SIMD hardware, e.g. ARM NEON [3], for vector operations; hardware-accelerating the most common data structures, an approach shown to benefit intensive mobile apps [55].

The discovered OS bottlenecks might already be known as non-cheap, but were still adopted by the current OS in favor of engineering ease, flexibility, visual appeal, etc. However, our results have shown that they now become unacceptably expensive for wearable devices. This observation urges a rethink of such OS design decisions. In order to do so, we contribute the following new guidelines: *i)* the software cruft for the complex runtime environment and UI layout should be aggressively trimmed down; *ii)* the OS should be frugal at spending CPU cycles on "add-on" user experience; *iii)* lazy or on-demand operations, despite their wide use for efficiency, may hurt system responsiveness when they come in the way of short user interactions.

5.2 Idle Episodes

- *The idle episodes are plentiful and of a variety of lengths.*
- *Idle anomalies are mainly contributed by improper OS implementations and performance overprovision during interaction.*

As shown in our pilot study [31] and discussed in Section 4, idle anomalies hurt both user experience and energy efficiency. Next, we present a quantification of them and describe our discovery of their root causes.

We select nine out of fifteen benchmarks whose idle time is larger than 5% of the benchmark duration. The statistics of idle time, as shown in Table 4, exhibit four unusual characteristics. First, over the duration of one benchmark, the accumulative idle time can take a significant fraction up to 87%. Second, the number of idle episodes is large, often multiple hundreds per benchmark. Third, many of the episodes are as short as a couple of ms, not enough for the device to enter low-power mode. Fourth, numerous idle episodes even emerge in CPU-bound benchmarks, e.g. launching an in-memory app (*lch.set*).

As summarized in Table 4, our investigation reveals a variety of causes, which, together, account for more than 91% of all the observed idle time. In addition to a couple of well-known causes such as user think, storage I/O or app policies, we have discovered two groups of causes originated in the OS design, as discussed below.

5.2.1 Improper OS Designs

Surprisingly, a large fraction of idle episodes are incurred by OS design flaws. We next describe the two most notable cases that have high impact on user experience and energy efficiency.

Interference from voice UI. *Clockwork*, the OS shell, hosts an always-on voice UI and thus is consuming a stream of audio input. Unfortunately, managing the voice UI incurs a long delay. When a user launches a new app through *Clockwork*, the foreground *Clockwork* releases its system resources and moves to the background. In doing so, *Clockwork* sends an IPC request to *MediaServer*, an OS daemon, asking to terminate the audio input stream. By design, this protocol is synchronous for security. On one hand, *Clockwork* stalls launching until the request is completed. On the other hand, *MediaServer* handles the request conservatively: its core function `setOutputDevice()` waits for the audio input pipeline to completely drain so that no leftover audio samples will enter the newly launched app. The draining holds the foreground app launch for around 120 ms, constituting up to 19% of the user-perceived latency in app launch benchmarks. Affecting both of our test devices, this flaw is responsible for the idle anomaly shown in phase 1 of Figure 1. Through experiments, we further confirmed that this flaw is introduced by Android Wear, i.e. not existing on smartphone: we

Table 3: Top 5 hot functions of System Server based on an average over three benchmark runs. Functions operating basic data structures are shaded. See Appendix for brief descriptions of these functions.

update	#Calls (mean/SD)		Total Time (mean/SD)	
computeOomAdjLocked	176	6.35	15.11	0.15
removeObserverLocked	152	0	9.74	0.52
MessageQueue.next	40	0.58	9.52	0.32
Values.cleanUp	98	2.08	6.68	0.22
updateOomAdjLocked	19	0	6.55	0.22

notif	#Calls (mean/SD)		Total Time (mean/SD)	
ProcessCpuTracker.collectStats	1	1	25.06	25.39
Values.cleanUp	220	48.39	15.06	3.45
computeOomAdjLocked	189	42.15	14.61	3.31
AlarmManager$Batch.remove	285	70.47	12.59	3.19
PendingIntent.equals	488	109.7	9.74	2.45

wrist	#Calls (mean/SD)		Total Time (mean/SD)	
LightImpl.setLightLocked	23	4.16	164.08	30.1
MessageQueue.next	130	7.57	15.53	1.12
parseProcWakelocks	1	0	14.4	0.08
computeOomAdjLocked	152	69	13.97	6.31
Values.cleanUp	121	11.36	8.36	0.62

touch	#Calls (mean/SD)		Total Time (mean/SD)	
LightImpl.setLightLocked	20	12	124.02	63.26
MessageQueue.next	124	51.62	15.67	8.35
Values.cleanUp	140	8.49	9.51	0.73
String.equals	525	78.49	8.74	1.10
setHalInteractiveModeLocked	6	2.12	7.34	0.96

lch.set	#Calls (mean/SD)		Total Time (mean/SD)	
SparseArray.get	2098	122.57	17.33	0.9
MessageQueue.next	70	3.46	13.21	0.66
Values.cleanUp	174	3.79	11.15	0.13
Parcel.writeLongArray	1	0	9.34	0.63
animateLocked	42	3.21	9.03	0.97

lch.calc	#Calls (mean/SD)		Total Time (mean/SD)	
Values.cleanUp	1	0	53.01	8.94
Parcel.writeLongArray	165	1.15	11.35	0.1
SparseArray.get	1	0	10.84	0.98
MessageQueue.next	1079	20.43	9.12	0.2
Parcel.writeLong	45	2.31	7.71	0.92

lch.game	#Calls (mean/SD)		Total Time (mean/SD)	
Values.cleanUp	220	22.11	14.96	1.27
SparseArray.get	1453	62.78	12.42	0.66
Parcel.writeLongArray	1	0	10.94	0.53
performLayoutAnd PlaceSurfacesLockedInner	9	0.58	9.41	0.48
MessageQueue.next	42	1	7.63	0.56

palming	#Calls (mean/SD)		Total Time (mean/SD)	
DirectByteBuffer.get	6040	1060.08	92.78	12.34
DirectByteBuffer. checkIsAccessible	6586	1155.05	85.94	11.25
computeOomAdjLocked	709	1.53	64.74	1.9
DirectByteBuffer. checkNotFreed	6933	1215.1	51.09	5.85
MemoryBlock.isAccessible	6586	1155.05	49.62	5.61

voice	#Calls (mean/SD)		Total Time (mean/SD)	
Values.cleanUp	41	5.03	2.81	0.35
Binder.execTransact	19	3.06	1.14	0.15
ThreadLocal.get	41	3.06	1.11	0.12
performLayoutAnd PlaceSurfacesLockedInner	1	0	0.85	0.01
ThreadLocal$Values.put	39	3.06	0.83	0.08

game	#Calls (mean/SD)		Total Time (mean/SD)	
Values.cleanUp	163	11.72	12.31	0.93
DisplayInfo.writeToParcel	54	3.21	10.83	0.64
Parcel.writeInt	1181	70.72	5.82	0.31
ThreadLocal.get	163	11.53	4.39	0.29
Values.put	163	11.72	3.84	0.28

notes	#Calls (mean/SD)		Total Time (mean/SD)	
ThreadLocal$Values. cleanUp	393	33.41	28.08	2.68
MessageQueue.next	67	6.03	27.76	2.41
MessageQueue. enqueueMessage	133	12.01	14.63	1.35
ThreadLocal.get	526	45.49	14.18	1.50
String.equals	292	38.08	10.26	0.99

navi	#Calls (mean/SD)		Total Time (mean/SD)	
performLayoutAnd PlaceSurfacesLockedInner	6	0	6.22	0.39
ThreadLocal$Values.clean Up	79	12.22	5.88	0.87
InputEventReceiver. finishInputEvent	60	5	5.02	0.44
SparseArray.get	533	58.29	4.9	0.59
consumeBatchedInputEvent s	51	4.51	4.57	0.34

accel	#Calls (mean/SD)		Total Time (mean/SD)	
performLayoutAnd PlaceSurfacesLockedInner	5	0	4.73	0.08
SparseArray.get	443	5.77	3.86	0.17
setInputWindows	10	0	2.69	0.12
MessageQueue.next	13	3.21	2.5	0.68
Values.cleanUp	31	4.04	2.22	0.28

heart	#Calls (mean/SD)		Total Time (mean/SD)	
performLayoutAnd PlaceSurfacesLockedInner	1	0	0.93	0.02
SparseArray.get	88	0	0.73	0.01
MessageQueue.next	3	0	0.6	0.04
setInputWindows	2	0	0.54	0.04
addInputWindowHandleLw	16	0	0.41	0

baro	#Calls (mean/SD)		Total Time (mean/SD)	
performLayoutAnd PlaceSurfacesLockedInner	5	0	4.58	0.23
SparseArray.get	440	0	3.78	0.19
setInputWindows	10	0	2.45	0.02
addInputWindowHandleLw	80	0	2.11	0.12
updateInputWindowsLw	10	0	1.96	0.1

have not observed similar idle anomalies on the Nexus 5 running Android Lollipop.

Legacy support for device suspending. The current kernel mechanism for device suspending is decades-old; initially designed for desktop and later adapted to handheld devices, the mechanism is now cumbersome for wearable's frequent, brief wakeups. Our measurement shows that one suspending action spends a continuous period of around 200 ms in kernel, a non-negligible overhead comparable to one interaction itself that often lasts for a couple of seconds (§2.1). Over the course of suspending, around 130 ms is in small idle episodes, each lasting for a couple of ms.

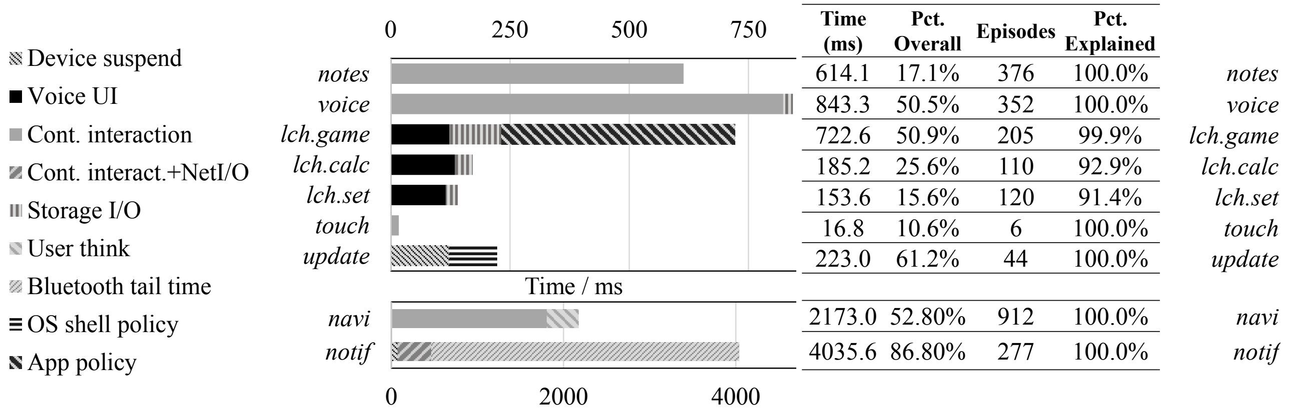
Our investigation shows power management for I/O peripherals as the deeper cause: in suspending a wearable device, the kernel checks all peripherals to ensure they are ready to enter low-power mode. Since a subset of peripherals take tens or even a couple of hundred ms to be ready, the CPU has to repeatedly poll them and sits idle in between. These peripherals include display controller, battery temperature monitor, and MMC host controller, based on our measurement. Note that this inefficiency roots in the Linux kernel and is not tied to specific wearable devices.

5.2.2 *Continuous Interaction*

Recall that continuous interaction and the resultant UI animation, e.g. scrolling a web page, used to be among the most intensive smartphone workloads [48]. In animating simple wearable UI, however, performance overprovision has produced numerous idle episodes.

As a user touches the device screen, the CPU processes a sequence of discrete input events, collaborates with the GPU to periodically render graphic frames, and sits idle between adjacent frames. In our benchmarks, almost all frames in continuous animation are rendered within 8 ms each, which is less than 50% of the 16 ms per-frame rendering budget (the displays of our test devices, as in most today's mobile devices, are refreshed at 60 Hz).

Table 4: Idle time in benchmarks. (Left) a breakdown based on their causes; (Right) the total idle time; its percentage over the benchmark duration; the number of episodes; the percentage of idle time that we provide explanations for. Note that the two bar graphs have different x-axis scales (time).



	Time (ms)	Pct. Overall	Episodes	Pct. Explained	
notes	614.1	17.1%	376	100.0%	*notes*
voice	843.3	50.5%	352	100.0%	*voice*
lch.game	722.6	50.9%	205	99.9%	*lch.game*
lch.calc	185.2	25.6%	110	92.9%	*lch.calc*
lch.set	153.6	15.6%	120	91.4%	*lch.set*
touch	16.8	10.6%	6	100.0%	*touch*
update	223.0	61.2%	44	100.0%	*update*
navi	2173.0	52.80%	912	100.0%	*navi*
notif	4035.6	86.80%	277	100.0%	*notif*

As a result, many short idle episodes come hand in hand with continuous interaction. For instance, in *navi* that lasts for 4119 ms, performance overprovision contributes 1805 ms idle time in about 630 episodes.

5.2.3 Design Implications

To cope with the excessive idle time, our findings suggest either eliminating their origins or utilizing them more efficiently.

Hunting OS inefficiencies. Fixing the *discovered* inefficiencies requires a rethink of the respective OS aspects, e.g. security policy and power management. What is even more challenging is to uncover the *unknown* inefficiencies: despite our success in identifying a few key cases, more subtle, non-deterministic ones are likely to exist. While the spirit of `IdleChecker` discussed in Section 4 is likely to apply, analyzing massive traces captured *in the wild* will be vital.

To this end, it is crucial to have a lightweight tracer that captures important software activities – from system events to function calls – related to anomalous idle episodes. Designing the tracer, despite a reminiscence of smartphone app tracking [44], raises two unique challenges: first, tracing should only be activated upon the manifest of anomalous idle, otherwise the amount of information will be overwhelming; second, tracing should span across multiple processes, most notably the OS daemons. Ultimately, we envision that the wearable OS, through integration with tools, can offer accountability of idle episodes, clearly pinpointing which software entity causes an idle episode and for what reason.

Filling idle time with useful work. As static power keeps increasing on future devices [13], it is compelling to fill the power-hungry idle episodes, e.g., those happened in continuous interaction, with useful work. We expect that two approaches are promising. First, compared to handheld, the wearable UI has fewer elements and much less variation in rendering time. These characteristics, together with the observed performance overprovision, suggest to lower the performance of CPU and GPU, e.g., by reducing their clock rates. This will shrink idle episodes and improve system energy efficiency.

Second, the idle episodes during interaction are ideal opportunities for predictive execution. Our rationales are twofold. First, since wearable usage is driven by user's daily routines, many tasks can be foreseen, e.g., retrieving calendar from the paired phone. Such prediction is already demonstrated as effective on smartphones [53, 33]. Second, due to the periodic rendering activity, the idle episodes show regular, predictable timing patterns. This simplifies scheduling predictive tasks and minimizing their contention with the foreground UI animation.

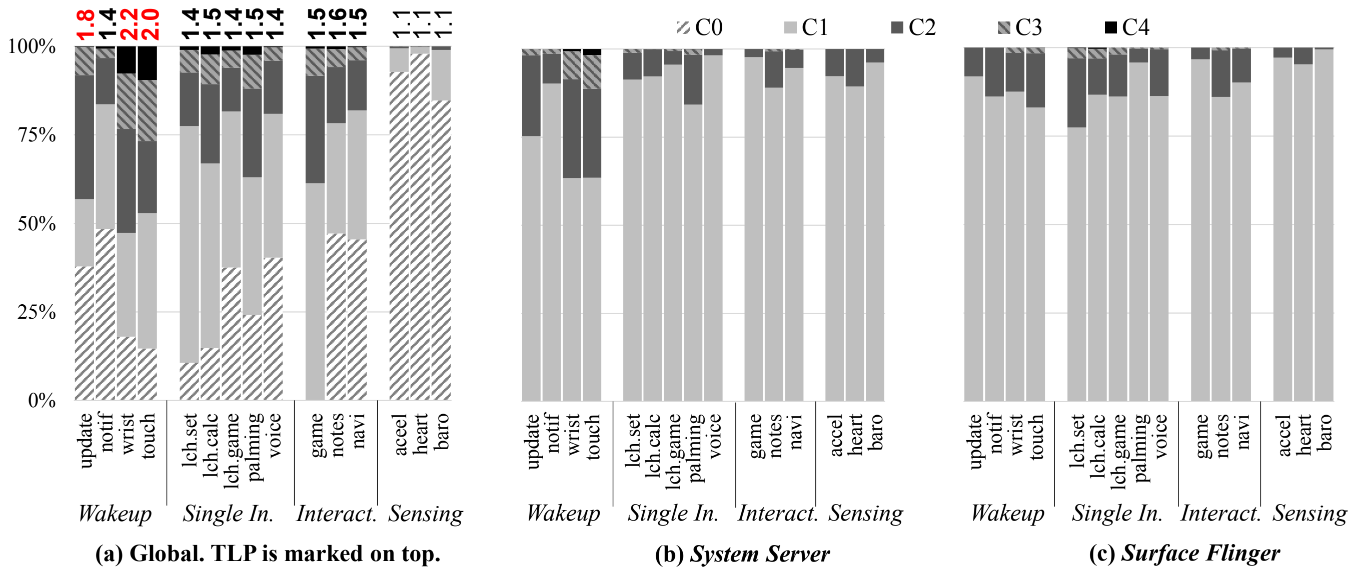
5.3 Thread-level parallelism

- *Short interactions exhibit substantial TLP, which is on a par with desktop workloads.*
- *While apps are mostly single-threaded, OS daemons contribute to TLP significantly.*
- *A wearable device needs at least two cores.*

As described in Section 4, we measure TLP by forcing all four CPU cores on at their default clock rate of 787 MHz. This is essentially a "what-if" study: *what core utilization can a wearable system achieve, if it has plentiful cores?*

Our experiments dispel the "one core is enough" myth mentioned in Section 4: despite that a wearable device is physically equipped with multiple cores, all but one are forced offline by the vendor. As shown in Figure 5(a), most benchmarks exhibit TLP higher than 1.5; for three *Wakeup* benchmarks the TLP rises to around 2. This TLP is unexpected to us, as it exceeds that of typical smartphone workloads (1.44 on average [16]), and is on a par with or even higher than many desktop workloads including Microsoft Office (1.2), 3D gaming (1.6), and web browsing (2) [4] . The concurrency values c2 – c4 shown in Figure 5(a) indicate that more than one core are in use for a significant amount of time.

Why is the TLP high? A closer examination reveals two causes of the high TLP. First, wearable's short span of interaction creates bursty TLP. Recall that compared to smartphones, user interaction periods on wearables are much shorter (§1). Although similar TLP peaks have been observed on smartphones [16], the peaks on wearable constitute a much larger fraction of the interaction span, leading to a higher TLP on average.



(a) Global. TLP is marked on top. **(b) *System Server*** **(c) *Surface Flinger***

Figure 5: The system-wide TLP and the concurrency values. C_i indicates the time fraction when i threads are simultaneously running over (a) the whole benchmark duration or (b & c) the corresponding process execution duration.

Table 5: A comparison of popular mobile CPU cores

	Cortex-A7 *Current wearable*	***Cortex-A9*** *Smartphone*	***Cortex-A53*** *Next-gen wearable*
ISA	ARMv7A	ARMv7A	ARMv8A
Inst. Pipeline	8-stage in-order	8-stage out-of-order	8-stage in-order
Br. Predictor history buffer	256-entry	4096-entry	3072-entry
TLB entries	L1: 10I/10D L2: 256	L1: 32I/32D L2:128	L1: 10I/10D L2: 512
L1$ (I/D)	32KB/32KB	32KB/32KB	8-64KB/8-64KB
L2$	1024KB	512-1024KB	128-2048KB

Second, the OS daemons, which often dominate CPU usage (§5.1), contribute to the TLP substantially. As standalone processes, these daemons receive app requests over IPC and serve them in parallel with the app execution. In addition, an individual OS daemon is multi-threaded and moderately parallel, e.g., to run multiple service instances simultaneously. This can be seen from the concurrency values of two major OS daemons shown in Figure 5(b) and (c).

The two causes also account for the particularly high TLP in most *Wakeup* benchmarks: these benchmarks are OS-intensive as shown in Figure 3(a); in response to a wake up event, multiple OS services run in order to initialize the hardware or prepare themselves for the subsequent app execution, creating a surge of TLP.

The wearable apps, on the other hand, are mostly single-threaded. Even the most parallel app, "Deadly Spikes" in benchmark *game*, runs one thread for 93% of the time, i.e., its c1 value is as high as 93%.

Design Implications

OS structure. Our TLP measurement suggests the existing structure of Android Wear, i.e. realizing the OS personality as multi-process, multi-threaded daemons, is a viable design. By promoting TLP, the OS structure is capable of exploiting mobile multicores which have already demonstrated the advantage of energy efficiency [9]. Furthermore, the cost of app/daemon IPC is still a much lower-order concern as compared to the expensive hot functions shown in Section 5.1.

CPU core count. A wearable device needs to use at least two cores. The measured TLP already shows the need for a second core; eliminating the idle anomalies identified in Section 5.2 may further boost TLP for the affected benchmarks, e.g., in *Single Input*. To accommodate the lower, but still non-negligible, demand for more than two cores (c3 and c4 in Figure 5(a)), a wearable device may embrace even weaker cores to form 2+1 or 2+2 big.LITTLE clusters.

One may wonder the impact of multicore on device cost or energy efficiency. First, the use of extra cores will incur no extra cost: as mentioned in Section 4, the commodity Android wear devices already possess quad cores – three of them are just forced off. Second, previous study has already shown that catering to parallel workloads with extra cores will benefit, rather than hurting, energy efficiency [9], as the device can finish tasks sooner and fall into sleep. We expect the extra cores, when they become idle, can be aggressive power-gated in order to further reduce their static power.

5.4 Microarchitectural behaviors

- *A significant mismatch exists between the OS and CPU microarchitecture, particularly in L1 icache, iTLB, and branch predictor.*
- *The mismatch is largely due to the OS code complexity, and will not be eliminated by a unilateral enhancement of wearable CPU.*

We have listed three typical mobile CPUs in Table 5. Among them, Cortex-A9, a smartphone CPU, already showed microarchitectural inefficiency in running popular smartphone apps [21]. Cortex-

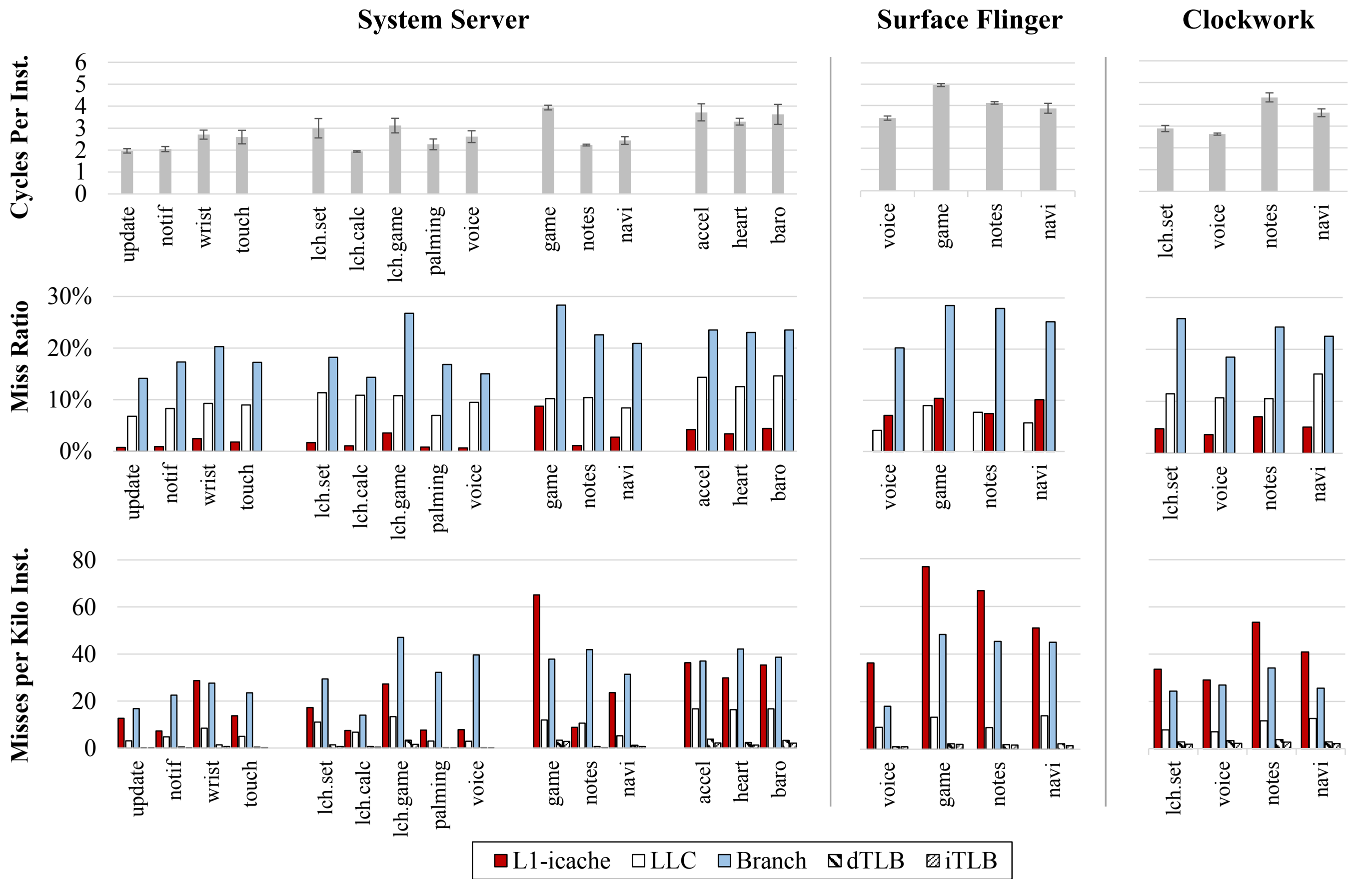


Figure 6: The measured microarchitectural behaviors of three major OS processes, covering their both user and kernel execution. Note that the LLC miss ratio is normalized to the total LLC references.

A7, a lighter version of smartphone CPU, is used as the *de facto* core for modern wearables. Cortex-A53 is a promising candidate for the next-generation wearables.

Atop our A7-based devices, we characterize the microarchitectural behaviors of the three major OS components, which collectively dominate the global CPU usage (§5.1). Note that for each OS component, we only show the benchmarks in which the component uses more than 10% of the system CPU cycles.

Perhaps contrary to common perception, the seemingly simple wearable workloads severely suffer from microarchitecture inefficiency. As shown in Figure 6, the cycles-per-instruction (CPI) ranges from 2 to 5, much higher than CPU-bound smartphone workloads such as web rendering (1.3 – 2.5) [21, 5]. The inefficiency also far exceeds that of desktop workloads: the CPI of SPEC INT on mobile CPUs rarely goes above 2 [5].

Through an analysis of the microarchitecture events, we have made three observations.

First, L1 icache and iTLB misses are high. As shown in Figure 6, *Surface Flinger* and *Clockwork* (and *System Server* in some benchmarks) exhibit a L1 icache misses-per-kilo-instructions (MPKI) ranging from 30 to 76. As reference points, the L1 icache MPKI in smartphone web rendering is around 28 [21, 26, 5] and in heavy smartphone gaming it is around 75 [21]; note that these misses were observed on A9, whose L1 cache is no larger than A7 as shown in Table 5. The fundamental reason lies in the complex OS codebase and its poor code locality, a nature likely inherited from the handheld OS [21]. In addition, the small iTLB of A7 may have worsened the situation.

Second, branch misprediction is substantial. Across all the benchmarks, the misprediction ratio is up to 28%, significantly higher than various smartphone workloads (all below 14% [26]) and SPEC INT (rarely above 17% [21]). The likely reasons are *i)* the complex OS execution paths and *ii)* the simple branch predictor in A7.

Last, the L1 dTLB and L2 cache are moderately effective. The misses are lower than data-intensive smartphone workloads, e.g., map or photo viewer [26], which have much larger working sets than wearable workloads.

Design Implications

The serious mismatch motivates two possible remedies. First, the CPU may evolve towards satisfying the demand of the OS. This is exemplified by an enhanced branch predictor of A53 shown in Table 5. However, one cannot satisfy all the microarchitectural demands, especially a larger L1 icache, due to the wearable's tight power and area constraints. Second, and more importantly, the observed inefficiencies – in icache, iTLB, and branch predictor – all root in the high complexity of the OS code. Our findings in Section 5.1 already suggest that the OS contains plentiful cruft. Should the OS be trimmed down to match the simplicity of its apps, the need for scaling up microarchitecture will significantly diminish.

6. RELATED WORK

Characterizing interactive workloads. Recognizing the significance of interactive systems, a large body of prior work has studied the key aspects including energy efficiency [8], latency [12, 54], thread-level parallelism [15, 4], and file I/O behaviors [23, 28]. Although none directly addressed wearables, their methodologies have been retrofit in our study.

Extensive study has been done towards understanding smartphone workloads. Power characterization has been a central topic [8], which highlights the importance of energy efficiency. Smartphone benchmarks, such as BBench [21], MobileBench [40], and Moby [26], often show non-traditional microarchitectural inefficiencies, including high TLB miss rate and icache miss rate. Our work shows that wearable workloads inherit many these behaviors due to the heavy OS execution. Gao *et al.* find that smartphone workloads show limited TLP, concluding that they need no more than two cores [16]. Our study shows that TLP in a wearable system can be even higher due to bursty OS workloads. ProfileDroid [52] contributes an approach for charactering smartphone apps at multiple layers; compared to it, our work addresses whole system efficiency with a focus on the OS components.

Little prior work has characterized wearable workloads. Among them, our prior work [30] studies the power and heat behaviors of Google glass. Our pilot study [31] on Android Wear reports power measurement, describes our custom toolset, and presents preliminary evidence of inefficiency. Our work [36] studies computational resource waste of the circular display commonly seen on wearables. Min *et al.* studies the battery usage of smart watches [37]; WearDrive [25] creates synthetic benchmarks to shed light on wearable storage. Compare to them, this work provides a holistic understanding of wearable systems.

Diagnosing mobile software anomalies. Various tools have been built to analyze smartphone software malfunctions, including quick battery drain [42, 34], sluggish GUI [32], and app crashes [24]. They enable systematic analysis of smartphone apps and inspire our work. Compared to them, we target a brand new platform – wearable; our diagnosis focuses on the anomalies originated in the OS, which have much wider impact than app anomalies.

While our `IdleChecker` (§4) may evoke user transaction trackers for smartphone such as AppInsight [44], it differs on three aspects: an offline diagnosis tool, `IdleChecker` does not instrument apps; targeting a system-level analysis, it takes all tasks into account instead of focusing on one single app; it identifies suspicious source code instead of just reporting delay numbers.

Quantifying microarchitectural behaviors. Much work has quantified microarchitectural behaviors of various workloads, including data-intensive workloads [51], scale-out servers [14], desktop apps [27], and smartphone apps [21, 26], We retrofit their methodologies and use their data as reference points for comparison.

Novel wearable apps. There is a large body of recent work on wearable, focusing on novel algorithms or app-specific systems. For instance, RisQ [41] and TypingRing [39] target gesture recognition; iShadow [35] tracks gaze in real time; Ha *et al.* build wearable for cognitive assistance [22]; Cornelius *et al.* focus on user identification [11]. While we recognize their importance, we focus on the OS internals of mass-market wearables.

7. CONCLUSIONS

We present an in-depth analysis of Android Wear, one of the most popular wearable OSes. Targeting OS design, we examine four key aspects – CPU usage, idle episodes, TLP, and microarchitectural behaviors – in fifteen benchmarks. We have reported detailed quantification, distilled a number of new findings, and discovered serious OS inefficiencies that were widespread but unknown before. Together, our results clearly point out the system bottlenecks for immediate optimization and have strong implications on future wearable system software and hardware design.

Acknowledgement

This work was supported in part by NSF Award #1464357 and by a Google Faculty Award. The authors thank the anonymous reviewers and the paper shepherd, Vishnu Navda, for their useful feedbacks.

8. REFERENCES

[1] ANANDTECH. The apple watch review. http://www.anandtech.com/show/9381/the-apple-watch-review, 2015.
[2] APPLE. Apple watch human interface guidelines. https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/WatchHumanInterfaceGuidelines/, 2015.
[3] ARM. ARM architecture reference manual, armv7-a and armv7-r edition, 2014.
[4] BLAKE, G., DRESLINSKI, R. G., MUDGE, T., AND FLAUTNER, K. Evolution of thread-level parallelism in desktop applications. In *Proc. Int. Symp. Computer Architecture (ISCA)* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 302–313.
[5] BLEM, E., MENON, J., AND SANKARALINGAM, K. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on* (Feb 2013), pp. 1–12.
[6] BUSINESS INSIDER. The first round of apple watch apps weren't that great, but that's all about to change. http://www.businessinsider.com/why-the-first-apple-watch-apps-are-slow-2015-6, 2015.
[7] BUSINESSINSIDER. Apple just kneecapped google's smartwatch efforts, 2015.
[8] CARROLL, A., AND HEISER, G. The systems hacker's guide to the galaxy energy usage in a modern smartphone. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2013), APSys '13, ACM, pp. 5:1–5:7.
[9] CARROLL, A., AND HEISER, G. Mobile multicores: Use them or waste them. *SIGOPS Oper. Syst. Rev. 48*, 1 (May 2014), 44–48.
[10] CONNOLLY, E., FAABORG, A., RAFFLE, H., AND RYSKAMP, B. Designing for wearables. Google I/O, 2014.
[11] CORNELIUS, C., PETERSON, R., SKINNER, J., HALTER, R., AND KOTZ, D. A wearable system that knows who wears it. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 55–67.
[12] ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proc. USENIX Conf. Operating Systems Design and*

Implementation (OSDI) (New York, NY, USA, 1996), OSDI '96, ACM, pp. 185–199.

[13] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. In *Proc. Int. Symp. Computer Architecture (ISCA)* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[14] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 37–48.

[15] FLAUTNER, K., UHLIG, R., REINHARDT, S., AND MUDGE, T. Thread-level parallelism and interactive performance of desktop applications. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 129–138.

[16] GAO, C., GUTIERREZ, A., RAJAN, M., DRESLINSKI, R., MUDGE, T., AND WU, C. A study of mobile device utilization. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2015).

[17] GOOGLE. Ui patterns for android wear. https://developer.android.com/design/wear/patterns.html, 2014.

[18] GOOGLE INC. Binder. http://developer.android.com/reference/android/os/Binder.html.

[19] GOOGLE INC. Systrace. http://developer.android.com/tools/help/systrace.html.

[20] GOOGLE INC. Wakelock. http://developer.android.com/reference/android/os/PowerManager.WakeLock.html.

[21] GUTIERREZ, A., DRESLINSKI, R., WENISCH, T., MUDGE, T., SAIDI, A., EMMONS, C., AND PAVER, N. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on* (Nov 2011), pp. 81–90.

[22] HA, K., CHEN, Z., HU, W., RICHTER, W., PILLAI, P., AND SATYANARAYANAN, M. Towards wearable cognitive assistance. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 68–81.

[23] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proc. ACM Symp. Operating Systems Principles (SOSP)* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 71–83.

[24] HU, G., YUAN, X., TANG, Y., AND YANG, J. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 18:1–18:15.

[25] HUANG, J., BADAM, A., CHANDRA, R., AND NIGHTINGALE, E. B. Weardrive: Fast and energy-efficient storage for wearables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 613–625.

[26] HUANG, Y., ZHA, Z., CHEN, M., AND ZHANG, L. Moby: A mobile benchmark suite for architectural simulators. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on* (March 2014), pp. 45–54.

[27] LEE, D. C., CROWLEY, P. J., BAER, J.-L., ANDERSON, T. E., AND BERSHAD, B. N. Execution characteristics of desktop applications on windows nt. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 1998), ISCA '98, IEEE Computer Society, pp. 27–38.

[28] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the Tenth ACM International Conference on Embedded Software* (New York, NY, USA, 2012), EMSOFT '12, ACM, pp. 23–32.

[29] LG USA. Design comes full circle. http://www.lg.com/us/smart-watches/lg-W110-g-watch-r, 2014.

[30] LIKAMWA, R., WANG, Z., CARROLL, A., LIN, F. X., AND ZHONG, L. Draining our glass: An energy and heat characterization of google glass. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2014), APSys '14, ACM, pp. 10:1–10:7.

[31] LIU, R., JIANG, L., JIANG, N., AND LIN, F. X. Anatomizing system activities on interactive wearable devices. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2015), APSys '15, ACM, pp. 18:1–18:7.

[32] LIU, Y., XU, C., AND CHEUNG, S.-C. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1013–1024.

[33] LYMBEROPOULOS, D., RIVA, O., STRAUSS, K., MITTAL, A., AND NTOULAS, A. Pocketweb: Instant web browsing for mobile devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 1–12.

[34] MA, X., HUANG, P., JIN, X., WANG, P., PARK, S., SHEN, D., ZHOU, Y., SAUL, L. K., AND VOELKER, G. M. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 57–70.

[35] MAYBERRY, A., HU, P., MARLIN, B., SALTHOUSE, C., AND GANESAN, D. ishadow: Design of a wearable, real-time mobile gaze tracker. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 82–94.

[36] MIAO, H., AND LIN, F. X. Tell your graphics stack that the display is circular. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2016), HotMobile '16, ACM, pp. 57–62.

[37] MIN, C., KANG, S., YOO, C., CHA, J., CHOI, S., OH, Y., AND SONG, J. Exploring current practices for battery use and management of smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers* (New York, NY, USA, 2015), ISWC '15, ACM, pp. 11–18.

[38] MONSOON. Monsoon power monitor. http://msoon.com/LabEquipment/PowerMonitor.

[39] NIRJON, S., GUMMESON, J., GELB, D., AND KIM, K.-H. Typingring: A wearable ring platform for text input. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2015), MobiSys '15, ACM, pp. 227–239.

[40] PANDIYAN, D., LEE, S.-Y., AND WU, C.-J. Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on* (Sept 2013), pp. 133–142.

[41] PARATE, A., CHIU, M.-C., CHADOWITZ, C., GANESAN, D., AND KALOGERAKIS, E. Risq: Recognizing smoking gestures with inertial sensors on a wristband. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 149–161.

[42] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. What is keeping my phone awake? Characterizing and

detecting no-sleep energy bugs in smartphone apps. In *Proc. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012), pp. 267–280.

[43] PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 335–348.

[44] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 107–120.

[45] SAMSUNG. Samsun gear application programming guide. http://img-developer.samsung.com/contents/cmm/Samsung_Gear_Application_Programming_Guide_1.0.pdf, 2014.

[46] SAMSUNG. Samsung gear live. http://www.samsung.com/global/microsite/gear/gearlive_design.html, 2014.

[47] SAXENA, SUNIL. Tizen architecture overview. Linux Foundation Collaboration Summit, 2012.

[48] SIEVERS, D., AND DUCA, N. Chrome on mobile @ 60fps. GPU Technology Conference, 2014.

[49] SMITH, B. C. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[50] TECHRADAR. Apple watch battery life: How many hours does it last? http://www.techradar.com/us/news/wearables/apple-watch-battery-life-how-many-hours-does-it-last–1291435, 2015.

[51] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ZHENG, C., LU, G., ZHAN, K., LI, X., AND QIU, B. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (Feb 2014), pp. 488–499.

[52] WEI, X., GOMEZ, L., NEAMTIU, I., AND FALOUTSOS, M. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2012), Mobicom '12, ACM, pp. 137–148.

[53] YAN, T., CHU, D., GANESAN, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 113–126.

[54] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First class support for interactivity in commodity operating systems. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 73–86.

[55] ZHU, Y., AND REDDI, V. J. Webcore: Architectural support for mobileweb browsing. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 541–552.

Appendix: Brief descriptions of top hot functions listed in Table 3

computeOomAdjLocked Computing the Oom value for a given process.

consumeBatchedInputEvents Consuming a batch of input events.

DirectByteBuffer.get Reading one byte from a memory-mapped buffer.

InputEventReceiver.finishInputEvent Finishing the handling of an input event.

MessageQueue.next Retrieving an inter-thread message.

Parcel.writeLongArray Serializing an array of Long integers to an IPC parcel.

parseProcWakeLocks Parsing wakelock statistics in file `/proc/wakelocks`.

performLayoutAndPlaceSurfacesLockedInner Updating system UI and readjusting the look of each app window.

ProcessCpuTracker.collectStats Collecting CPU usage statistics.

removeObserverLocked Removing a given content observer.

setHalInteractiveModeLocked Informing the power HAL layer about the change of interactive mode.

setLightLocked Setting the screen backlight level.

SparseArray.get Get an element from a SparseArray.

updateOomAdjLocked The wrapper of computeOomAdjLocked(). It updates the OOM values for all the processes.

Values.cleanUp Cleaning up the garbage-collected thread-local variables.