Power Sandbox: Power Awareness Redefined

Liwei Guo*
Purdue ECE

Tiantu Xu*
Purdue ECE

Mengwei Xu
Peking University

Xuanzhe Liu
Peking University

Felix Xiaozhu Lin
Purdue ECE

ABSTRACT

Many apps benefit from knowing their power consumption and adapting their behaviors on the fly. To offer apps power knowledge at run time, an OS often meters system power and divides it among apps. Since the impacts of concurrent apps on system power are *entangled*, this approach not only makes it difficult to reason about power but also results in power side channels, a serious vulnerability.

To this end, we introduce a new OS principal called power sandbox, which enables one app to observe the fine-grained power consumption of itself running in its vertical slice of the hardware/software stack. The observed power is insulated from the impacts of other apps. Our contribution is a set of lightweight kernel extensions that simultaneously i) enforce the power sandbox boundaries and ii) confine entailed performance loss to the sandboxed apps. Our experiences on two embedded platforms show that power sandboxes simplify reasoning about power, maintain fairness among apps, and minimize power side channels, thus facilitating construction of power-aware apps.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; • **Software and its engineering** → **Operating systems**; **Power management**;

KEYWORDS

Operating systems; Embedded systems; Energy efficiency; Power awareness

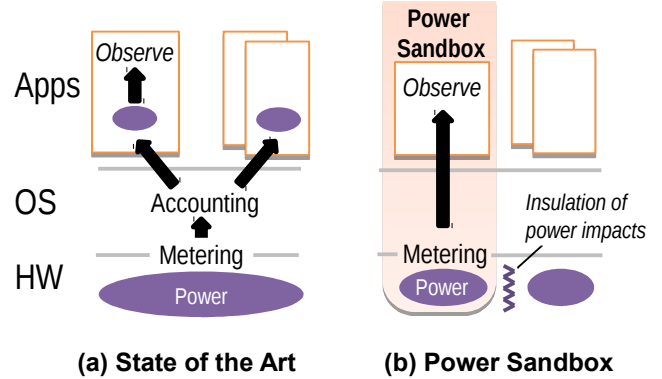*Both authors contributed equally to the paper

Figure 1: An overview of this work

1 INTRODUCTION

The quest for app power awareness[1] has lasted over a decade [28]: an app, as one or a group of user processes, demands to observe its power consumption online, in order to adapt its behaviors accordingly to lower power or higher efficiency. Traditionally, an operating system (OS) supports app power awareness through a two-step approach at run time as shown in Figure 1(a). First, the OS meters system power by either consulting a power model [25, 26, 59, 70, 71, 81, 96, 97] or performing *in situ* direct measurement [11, 28, 29, 79, 84]. Second, it divides the metered power into per-app shares, based on certain heuristics chosen at the OS development time.

Despite recent advances in fine-grained power metering [11, 79], the above approach suffers from two key inadequacies.

(1) **Reasoning difficulty**: it fails to provide power observations that are easy for apps to reason about and act upon.

(2) **Security vulnerability**: it creates power side channels [10], allowing attackers to learn a victim app's security-sensitive behaviors.

The latter inadequacy is already shown by prior work [37, 58, 95] and will be further demonstrated in this paper (§2.5): from its observed GPU power, an attacker app can infer what website a co-running victim browser is visiting. Such inference's success rate is 6× higher than random guess. Fine-grained power metering further *exacerbates* this vulnerability.

In summary, the two inadequacies are becoming the major obstacles towards app power awareness.

Our key observation is that the metered system power contains *entangled* impacts from concurrent apps, and the impacts cannot

---

[1]Power awareness and energy awareness are often used interchangeably in prior work. To highlight power knowledge at fine temporal granularity, we use power awareness in this paper unless stated otherwise.

be separated cleanly. Such power entanglement is rooted in work-conserving OSs that aggressively multiplex apps on hardware resources. Unfortunately, the existing approach to app power awareness copes with power entanglement *reactively* at best without attempting to eliminate it.

To this end, we advocate a fresh perspective on OS support for power awareness, as illustrated in Figure 1(b). First, the OS supports any app to observe the power of the app running in its *vertical environment* (i.e., its vertical slice of the software/hardware stack) and hence prompts the app to suit the vertical environment. Furthermore, the OS insulates the app's power observation from the impacts of other apps.

Following this perspective, we propose a new abstraction called power sandbox, or psbox for short. A psbox allows the enclosed app to observe the collective power of the app itself and its vertical environment at fine temporal granularities. In this observation, the only possible contributions of concurrent apps are periods of idle power. The OS enforces psbox as the only way for apps to observe power: one app may enter or leave its psbox freely, but is only allowed to observe power when it is in the psbox. Free of power entanglement, the resultant power observation is not only amenable to reasoning but also minimizing power side channels. We stress that power sandbox *insulates* app power impacts but does not *isolate* their executions: all apps, inside a psbox or not, share the same system image as usual.

To support psbox, we have addressed two primary challenges:
Enforcing psbox boundaries   We make the OS kernel respect psbox boundaries in resource multiplexing. The kernel does so with two extensions: i) it grants a psbox exclusive use of resource partitions at fine spatial or temporal granularities, called *resource balloons*; it meters and reports the power of resource balloons; ii) it virtualizes hardware power states for every single psbox.
Confining performance loss to sandboxed apps   As other mechanisms for online resource monitoring [1, 46], psbox comes with runtime costs, which is mainly due to lost sharing opportunities. In response, a core mechanism of psbox is to confine the costs to the sandboxed apps and therefore ensures performance fairness among all the apps, regardless of their usage of psbox. This mechanism is both powerful and critical: assuming two apps co-running on a multicore equally share the CPU time and one app enters its psbox, the unsandboxed app continues to enjoy its original share normally – half of the total CPU time, despite the reduction in combined CPU utilization.

The OS kernel confines performance loss with two techniques. It tracks the lost sharing opportunities and fully charges the loss to the sandboxed app, disadvantaging this app in future resource competitions. It encapsulates resource balloons as normal scheduling entities and therefore reuses most of the existing kernel infrastructure for scheduling.

We intend psbox to be a "pay-as-you-go" service for apps: apps use psbox to periodically sample power or to selectively monitor power of key execution phases. Based on their power observation, apps make power-aware decisions, which remain valid even after they leave psbox. In most of their lifetime, they run outside of psbox without overhead.

Atop a recent Linux kernel and two embedded platforms, we implement psbox for a variety of major hardware components, including CPU, GPU, DSP, and WiFi interface. psbox keeps an app's power observations highly consistent even when the app co-runs with other different apps. Across these runs, the app's energy, as observed by the app itself, differs by less than 5%; by contrast, energy shares reported by a prior approach differ by up to 60%. In a benchmark of three co-running computer vision apps, the use of psbox by one app leads to 10% total throughput loss. The confinement of performance loss is robust: in a test with extremely high resource contention, despite the throughput of the sandboxed app dropping by 4×, the other co-running app only experiences 1% throughput loss.

Based on psbox, we present an end-to-end use case. We build a virtual reality app (in 2K SLoC) that periodically observes its power and dynamically trades its fidelity level for lower power, demonstrating how psbox facilitates the construction of power-aware apps.

This paper has made the following contributions:

- We present an analysis of existing approaches to app power awareness, demonstrate the inadequacies, and identify the cause as power entanglement. In response, we present a novel OS principal called power sandbox (psbox) that supports an app to observe the power of itself and its vertical environment.
- We enforce psbox and confine its performance cost to the sandboxed app. We do so through a suite of techniques: resource ballooning, power state virtualization, and tracking/charging the lost sharing opportunities.
- On top of a recent Linux kernel, we implement psbox for CPU, GPU, DSP, and WiFi interface. Our evaluation shows that psbox reliably insulates power impacts, incurs minor cost, maintains fairness, and facilitates the construction of power-aware apps.

The full source code of psbox is available at:
<center>http://xsel.rocks/p/psbox</center>

2   A CASE FOR A NEW OS PRINCIPAL

We next analyze the design space of supporting app power awareness. First, we distill the essential power knowledge needed by apps (§2.1). Next, we examine the classic two-step approach, showing that while metering is becoming accurate and efficient (§2.2), accounting encounters a fundamental difficulty which we dub power entanglement (§2.3 – §2.5). To address the difficulty, we advocate eliminating power entanglement and empowering apps to observe exactly what they need to know. This motivates a new OS principal (§2.6).

2.1   Power awareness: what matters to apps?

We first examine what power knowledge has been required by existing app adaptation strategies. Figure 2 illustrates the key concepts of app power-aware adaptation.

App cares about its own power impact   By design, most adaptation strategies focus on optimizing one app's behaviors. By exploiting the app's domain knowledge, these strategies reduce the app's power impact, which will be translated to a similar reduction in the system power or energy. Often, apps demand to know their

power impacts at fine temporal granularities in order to map the power to short-lived software activities [63, 70, 79, 81].

This app-centered approach is extensively taken by prior work: an app optimizes its own code execution efficiency [17, 18, 63, 79, 101], reduces the power impact of its own I/O activities [23, 65, 78], or does both simultaneously [7, 14].

App adapts to suit its vertical environment   As illustrated in Figure 2, a vertical environment incorporates hardware conditions, system software configurations, user preferences, etc.

Figure 2: Concepts in app power-aware adaptation

For higher power efficiency, prior systems adapt to various factors of a vertical environment. Code generators adapt to CPU microarchitectures [63]. Mobile/-cloud offloading [17, 18] and mobile data compression [7] adapt to the comparative efficiency of CPU and wireless link. Content fidelity [16, 28, 60] and algorithm accuracy [38] adapt to user preferences. Network transfer scheduler adapts to network conditions [23, 65, 78] or app preference [6]. Web page or game rendering adapts to user perception [39, 101]. As such, power-aware adaptation decisions inherently depend on the app's vertical environment.

By contrast, app-centered adaptation rarely considers "horizontal" factors such as peer app activities, which would require apps to have not only deep knowledge of each other but also mutual trust. As a result, power-saving opportunities from horizontal cooperation (e.g., app co-execution [102], cooperative I/O [91], request piggybacking [53]) are more limited, and are often exploited at the OS level. These are complementary to the app-centered adaptation under discussion.

Comparative power drives actions   To make an adaptation decision, an app often chooses one action out of multiple candidates by comparing their power impacts. In existing power-aware systems, these alternative actions include program partitioning plans [17, 18, 59], code generation strategies [63], middleware configurations [60], graphics rendering strategies [39], network transfer plans [23, 65, 78], hardware component combinations [14], and compression algorithms [7].

Summary: essential power knowledge   We summarize the power knowledge that is essential to app adaptation as follows:

(1) An app demands to observe power consumption of itself and its vertical environment at fine temporal granularity. It is often indifferent to the power impacts of peer apps.

(2) An app must be able to compare the above power observations quantitatively.

Unfortunately, this essential power knowledge mismatches what apps are learning from the current approach to power awareness. Next, we examine this approach, in particular its two key steps.
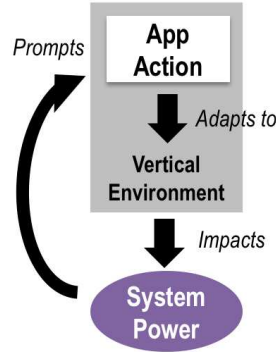
2.2 Fine-grained power metering is getting easier

System-level power metering[2] used to be the major challenge towards power awareness. While most prior work metered power using models [25, 26, 59, 70, 71, 81, 96, 97], such modeling for modern hardware is increasingly difficult, due to processor heterogeneity, variation in fabrication [83], and changing operating conditions [56].

Fortunately, direct measurement, the alternative metering method, starts to show high promise. Besides the known benefits of high rate (>10KHz) and accuracy (in mW) [56], recent work demonstrates that direct measurement can be efficient and therefore *in situ*, by offloading periodic power sampling and pre-processing to low-power microcontrollers [11, 79]. Fine-grained, inexpensive power metering enables characterization of short-lived software activities, and is likely to become a common feature of future hardware platforms. We will discuss this in detail in §8.1.

## 2.3 Accounting is hard due to power entanglement

Even though system power can be metered at a high resolution, attributing it to separate apps encounters a fundamental difficulty:

> **Power entanglement:** In a work-conserving OS that aggressively multiplexes apps on hardware, concurrent apps impact the hardware power simultaneously, and the impacts become inseparable.

We identify three major causes for power entanglement:

- *Spatial concurrency in hardware*   Multiple apps concurrently use disjoint hardware resources for which power can only be metered as a whole. Note that such power metering scopes are often hardware design choices. We show this with a simple experiment in Figure 3(a). On a dual-core CPU with one power rail, we measure the whole CPU power, and compare i) only running one process on core 0 to ii) additionally running a second instance of the same process on core 1. As shown in the figure, one cannot simply extrapolate the former run's power, e.g., by doubling it, to get that of the latter run. This is because in the latter run, the power impacts of two active CPU cores are entangled, as has also been confirmed by prior work [102].

- *Blurry request boundary*   Many hardware components, notably accelerators and I/O, accept requests from CPU and execute the requests asynchronously. Since CPU lacks visibility into the execution durations of in-flight requests, it cannot differentiate their power impacts. In Figure 3(b), we show the durations of three consequent GPU commands and the GPU power. The commands' durations are to the best of CPU's knowledge. Each duration starts when the command leaves the OS and enters the GPU, and ends when the OS is notified command completion by a GPU interrupt. Although we expect that the power of command 2 is similar to that of command 3 (they are of the same type), command 2 significantly overlaps with command 1 in time and their

---

[2]In this paper, we use "metering" to refer to both physically measuring energy and inferring energy through software models.
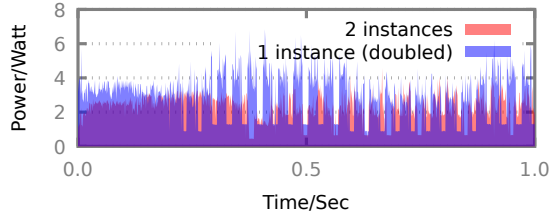
(a) Total CPU power of two co-running process instances, one on each core, compared to 2× power of one instance running alone. Hardware: 2×core Cortex-A15
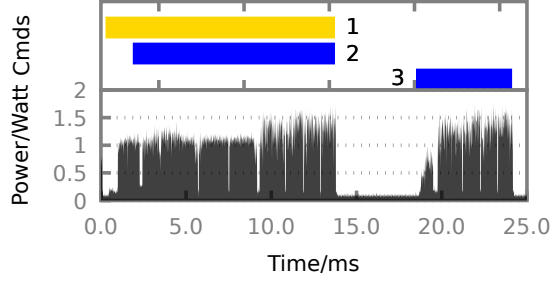
(b) A sequence of three GPU commands (top) and the total GPU power (bottom). Commands of the same type have the same color. Hardware: PowerVR SGX544MP
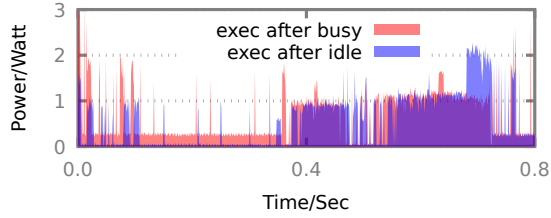
(c) Comparison of CPU power of the same app when it runs after a CPU idle period and when it runs after a busy period.

Figure 3: Examples of power entanglement

power impacts are hence entangled. The OS is incapable of separating power of these two commands.

- *Lingering power state*   Software workloads may prompt changes in the hardware power state, which will affect the power of subsequent workloads. In Figure 3(c), we compare the CPU power when one app runs in two different scenarios: running after the CPU has been idle for a while; running right after the completion of another busy workload. The latter scenario incurs noticeably different power, as the CPU clock rate raises prior to the app execution. Similar effects exist in transmission power of wireless interfaces.

Power entanglement exists no matter *how* power is metered, either through modeling or direct measurement. In particular, modeling suffers from all the causes above, as most existing modeling techniques infer system-level power from *aggregated* hardware activities, e.g. total LLC misses read from performance counters [81]. High-rate direct measurement does not help either, since the above

causes prevent obtained power samples from being attributed to apps, as we will demonstrate in evaluation (§6).

Existing approaches are inadequate   Existing accounting mechanisms cope with power entanglement *reactively* at best. They divide system power among apps using a variety of heuristics: even splitting [94], attributing each app's marginal contribution [25], attributing based on app hardware utilization [100], or attributing to the app that uses the hardware most recently [70].

These heuristics are useful for *system-level* energy accounting, in that they encapsulate the beliefs or policies of the OS designers. However, they are unable to address the aforementioned major obstacles in *per-app* power awareness, since no accounting heuristics can eliminate power entanglement that has *already* occurred.

2.4 Power entanglement creates reasoning difficulty

Existing accounting mechanisms provide per-app power shares that are difficult for apps to reason about or reproduce. For instance, merely based on its power share, one app can hardly tell why one network transmission consumes more energy than others of the same length (which could be because the OS charged the WiFi tail energy to this particular transmission [70, 96]); or why multiple invocations of the same function show much different power behaviors (which could be because varying workloads ran concurrently on other CPU cores).

One may suggest that besides dividing the system power, OS should open up its accounting internals to apps, e.g., publishing the hardware usage of concurrent apps and the accounting heuristics used by the OS. This will create more problems. i) Besides reasoning about power, app developers now need to reason about power accounting heuristics. As the heuristics become non-trivial (e.g. based on cooperative game theory [25]), app development soon becomes a daunting task. ii) Revealing apps' hardware usage to each other may create security vulnerabilities.

2.5 Power entanglement creates security vulnerability

Dividing system power among apps may reveal their power behaviors to each other. When the apps are mutually distrusted, this creates a known vulnerability called power side channels [10]: by observing the power of a victim app, an attacker app may learn the victim's security-sensitive behaviors, such as encryption and authentication procedures [51, 52, 95], GPS usage [58], or GUI state [95].

We next demonstrate that power entanglement can be exploited through power side channels, showing GPU power leaks a browser's deep information – which website it is visiting. We co-run two apps: a browser (victim) is scripted to open the Alexa top10 websites; an attacker app, while executing light GPU workloads as camouflage, attempts to infer what website the browser is opening. We train the attacker *once* with the GPU power traces collected when the browser runs alone, labeled by website URLs. In subsequent runs, the attacker infers the websites based the similarity between its known and observed GPU power activities. The similarity is measured with DTW, a well-known algorithm for time-series analysis [2].

Our results show that the attacker's success rate of inference is 60%, 6× higher than random guess. This is because different web pages tend to generate different GPU workloads, and hence unique power signatures.

2.6 Design choices

We advocate an OS principal for any power-aware app to observe the collective power of the app itself and its vertical environment. Specifically, the OS should achieve three objectives:

(1) *Insulate app power observation*   The OS shields an app's power observation from the impacts of other apps, and hence eliminates power entanglement for this observation. The OS does so by adjusting resource multiplexing.

(2) *Preserve vertical environment*   The OS keeps an app vertical environment unchanged, whether the app is using the OS service for observing power or not. This enables apps to make valid adaptation decisions based on their insulated power observations.

(3) *Track and charge cost*   The OS charges any overhead or lost multiplexing opportunity in insulating power observations to the requesting app. This ensures fairness among all apps despite their different usages of the service.

Following these choices, we introduce a new OS principal called *power sandbox*, or psbox, as will be presented below.

3 SYSTEM OVERVIEW

psbox is an OS principal enclosing one power-aware app, i.e., one or a group of user processes. It is the only way for any app to observe power. More specifically, a psbox exposes an interface of *virtual power meter* to the enclosed app, from which the app may read real-time power consumption incurred by the app and its vertical environment. In this observed power, the only possible contributions of concurrent apps are periods of idle power.

```
1  // Create a power sandbox
2  box=psbox_create(HW_CPU /* optional */);

3  psbox_enter(box);
4  // Continuous collection of power
       samples
5  psbox_sample(box, &buf, NUM_SAMPLES);
6  // One-time query of energy
7  energy = psbox_read(box);
8  psbox_leave(box);
```

Listing 1: The psbox User API

Intended usage of psbox   Since a psbox's overhead is charged to the sandboxed app, we expect apps to use psbox as a "pay as you go" service. They use psbox to *periodically* sample power, or *selectively* monitor power during interesting execution phases, and leave psbox for full-speed execution. An app makes power-aware decisions according to its psbox's virtual power meter. After the app leaves the psbox, its decisions remain valid, since the OS preserves the app's vertical environment (§2.1). The app only pays the price of psbox during a small fraction of its execution time.

We would like to stress this "pay as you go" power sandboxing is complementary to, and may coexist with, the OS mechanisms that optimize multiplexing of *power-unaware* apps for combined efficiency [40, 66, 102].

The app interface   Apps access psbox through the API summarized in Listing 1. An app creates a psbox and binds it to a set of hardware components of which power is reported (line 2). The granularity of hardware sets is determined by the possible power metering scopes as supported by hardware. For example, the hardware can be a subset of CPU cores sharing one measurable power rail [5]. During execution, the app is at liberty to enter or leave the psbox (line 3 and 8).

When it is in psbox, the app may query the psbox's virtual power meter. Similar to accessing CPU performance events [62, 90], the app may collect power samples in a user-provided buffer (line 5) or poll to get the accumulated energy (line 7). Unlike existing CPU events (including the power events [21]), all psbox power readings are timestamped. These timestamps come from a standard clock that apps can access through the clock_gettime() syscall. This allows apps to readily map power readings to software activities at fine granularities. Depending on metering methods, the timestamp resolution can be as high as 10 μs, as will described Section 5.

Kernel enforces psbox boundaries   The kernel eliminates power entanglement for a psbox. To do so, the kernel allocates spatial and temporal partitions of hardware resources at fine granularities, and grants *exclusive* use of them to the psbox. We term these partitions resource balloons, which are exemplified by a set of CPU cores and a time slice of the WiFi interface. Having established the boundaries for resource balloons, the kernel meters the corresponding hardware power, through either direct measurement or modeling (§2). The kernel then reveals the metered hardware power to the psbox's virtual power meter.

Kernel confines performance loss   A psbox incurs performance overhead. Most notably, the exclusive use of resource balloons likely leads to hardware under-utilization. The kernel tackles the overhead in two ways. On one hand, the kernel reduces the overhead by keeping resource balloons small, as will be shown in Section 6. More importantly, the kernel confines the overhead to the sandboxed app and minimizes the impact on apps outside the psbox. To do so, the kernel tracks the *lost sharing opportunity* due to resource ballooning, bills it to the sandboxed app, and properly disadvantages the sandboxed app in future competitions for accessing the hardware.

4 KERNEL SUPPORT

To support psbox, we face a twist of two challenges: i) eliminating power entanglement (§2.3) by changing how the kernel multiplexes concurrent apps on hardware; ii) integrating the changes into mature kernel mechanisms to avoid disruptive modifications. To address the first challenge, we present a model for extending kernel drivers; to address the second challenge, we describe how to apply the model to the kernel subsystems that manage major hardware components. For brevity, the remainder of this paper refers to these kernel subsystems as *drivers* in general.

4.1 The driver model

We propose two lightweight extensions to existing drivers.

Resource ballooning Resource multiplexing must respect psbox boundaries. More specifically, the kernel must confine spatial concurrency and asynchronous requests, two major causes of power entanglement (§2.3). To this end, we retrofit the concept of memory ballooning for virtual machines [89]. The kernel allocates fine-grained resource partitions, called *resource balloons*, and makes them exclusive to a psbox. The kernel schedules resources balloons together with other normal apps, enforces balloon boundaries, and meters the power of resource balloons for the psbox.

We next describe two types of balloons. In the discussion, we use psbox⟨App, *hw*⟩ to denote a psbox bound to hardware *hw* and enclosing an app App. We use $\overline{App}$ to refer to all other apps outside the psbox.

- *Spatial balloon* is for confining spatial concurrency on OS-schedulable, preemptable resources, most notably CPU cores. It prevents App and $\overline{App}$ from using *hw* simultaneously. To do so, when granting the access of *hw* to App, the OS schedules in a spatial balloon that occupies all the resources in *hw*, which effectively exclude $\overline{App}$ from *hw*.
- *Temporal balloon* is for confining request asynchrony on accelerators and I/O devices. It prevents App and $\overline{App}$ from having in-flight requests submitted to *hw* simultaneously. To do so, when granting App the access to *hw*, the OS schedules in a temporal balloon, a time slice during which the OS only dispatches the requests from App to *hw*. At the start and end of the temporal balloon, the OS drains in-flight requests by holding back new requests until *hw* completes the existing ones.

A key advantage of resource balloons is they appear as normal scheduling entities to the existing kernel infrastructure. Hence, they keep most of the latter oblivious and therefore unmodified. i) The kernel's existing accounting mechanism does not differentiate the portion of *hw* used by App from the portion intentionally kept idle by the balloons, e.g. unused CPU cores or stalled GPU cycles. The kernel simply bills all the resource occupied by the balloons to App. ii) The kernel's existing schedulers, e.g. for CPU or for network packets, still enjoy full freedom of choice: they are at liberty to decide whether and when to schedule a balloon on *hw*, and may freely multiplex $\overline{App}$ on *hw* without constraints.

Figure 7 in evaluation shows resource balloons in action.

Power state virtualization Enclosed in a psbox⟨App, *hw*⟩, App should neither observe any lingering power state (§2.3) on *hw* nor leave any residual state after using *hw*. To this end, the OS keeps a virtual copy of the power state of *hw* for each psbox (and a separate copy for $\overline{App}$). Upon scheduling in a resource balloon on *hw*, the OS puts *hw* in the power state in which the psbox left *hw* previously; when scheduling out the resource balloon, the OS extracts the hardware power state and saves it for the psbox.

To make this idea practical, we put hardware power states into two categories, depending on the costs of the related state transitions, and treat them differently:

- *Off/suspended states*, in which devices lose power or remain in deep sleep. Examples include CPU deep sleep that retains no cache content, or GPS cold start without any locked satellite. Exiting these power states often requires expensive

hardware operations, e.g. device initialization. Once a device exists such an off/suspended state, it often remains in an operating state for a long period, as described below.

- *Operating/idle states* are rough equivalents of P and C states in ACPI [36], which control performance settings of a working device or power saving of an idle device. A device can switch among these states at low cost and with low delay (often sub-milliseconds). Examples include CPU frequencies and WiFi transmission power levels.

The kernel virtualizes *operating/idle* states and reports the corresponding hardware power to psboxes. By contrast, it neither virtualizes *off/suspended* states nor reveals the power pertaining to these states. The rationales are as follows. First, reconstructing off/suspended states for each psbox can be prohibitively expensive, e.g. it requires to cold restart a GPS device for each new psbox. Furthermore, it is unsafe to reveal unvirtualized off/suspend hardware states to apps, which would allow a malicious app to infer the device usage, e.g. whether other apps have just used GPS for localization, through power side channels (§2.5). Hence, for the durations when *hw* is off/suspended, the kernel simply feeds psbox with samples of *hw*'s idle power. To App, *hw* appears idle.

4.2 Applying the driver model

According to our model, a driver takes on two new responsibilities for psbox:

(1) Enforcing resource balloon boundaries, including virtualizing power states;
(2) Tracking lost opportunities of resource sharing and counting them against App.

Beyond these two, balloon scheduling is handled by existing kernel mechanisms transparently.

Multicore CPU

We build spatial balloons into the CPU scheduler. A typical multicore CPU scheduler runs multiple instances, one for each core and managing a runqueue of local tasks (processes or threads). To choose the next running task, an instance picks the one with the best scheduling credit. Scheduling credits are often computed from tasks' recent CPU usage. For scalability, scheduler instances rarely communicate.

To enforce spatial balloons for psbox⟨App, *hw*⟩, the CPU scheduler coschedules tasks of App on all the cores of *hw*. If the runnable tasks in App are fewer than the cores, the scheduler runs dummy tasks on the remaining cores to force them idle.

To do this, an existing multicore scheduler faces twofold challenges. First, it needs to decide when to start and end a coscheduling period across a set of cores. However, in current designs each scheduler instance schedules its local tasks independently. Second, according to CPU cycles spent in coscheduling, the scheduler needs to discount scheduling credits, and hence ensure fairness between App and $\overline{App}$ across all the cores. However, in current designs an instance focuses on maintaining fairness among its local tasks.

While the idea of coscheduling is long known [69], the above challenges were still considered unaddressed on multicore, especially the fairness concern [20]. To address the challenges, we introduce a new notion of *scheduling loan* with three key ideas: i) we allow a scheduler instance to pick a task T for execution even if T does not have the best scheduling credit among all the local runnable tasks; ii) in order to be picked, T must get a loan to triumph other runnable tasks and pay back the loan with its future credits; iii) all tasks in $\boxed{App}$ share their scheduling loans.

Our augmented multicore scheduler works as follows.

- *Scheduling entities*: Similar to a Linux cgroup, a psbox has a set of scheduling entities $\{E\}$, one entity on each core. An entity E_i encompasses all tasks in $\boxed{App}$ on core i and keeps a collective scheduling credit. The kernel schedules E_i together with other normal tasks.

(1) *Schedule in*: Same as in current designs, the scheduler instance on core i picks E_i when E_i has the best scheduling credit. The instance further picks a task within E_i to run.

(2) *Task shootdown*: The scheduler instance thus requests all other cores to schedule in their corresponding entities in $\{E\}$. It does so by sending inter-processor interrupts to all other cores. Upon request, the scheduler on a remote core j picks E_j: it calculates Δ_j, the initial loan of E_j, as the difference between E_j's current scheduling credit and that of the most favorable task on core j (which would otherwise run). After shootdown, all tasks in $\overline{App}$ are off CPU and a coscheduling period for $\boxed{App}$ starts.

(3) *Running & loan update*: During coscheduling, scheduler instances bill local CPU cycles to the corresponding entities in $\{E\}$. When any scheduler instance, e.g., the one on core i, is invoked for rescheduling, it takes the chance to calculate the extra loan needed by E_i to warrant E_i's continue use of core i, and add this new loan to Δ_i.

(4) *Schedule out*: The coscheduling of $\boxed{App}$ continues until none of $\{E\}$ has the best credit on their corresponding cores, i.e., they all need extra loans to continue. At that time, the scheduler simultaneously schedules out all E_i from all the cores, by performing another shootdown.

(5) *Loan redistribution & repayment*: When scheduling E_i out, a current scheduler design will adjust E_i's credits based on the time E_i just runs. We further make $\boxed{App}$ pay back the loans that have accumulated during the preceding coscheduling period. To provide long-term fairness over all the cores, all entities in $\{E\}$ evenly split their total loans. The scheduler redistributes the loans within $\{E\}$, which will disadvantage $\boxed{App}$ in future scheduling.

Accelerators

Accelerators, such as GPU and DSP, execute commands offloaded from the CPU. The lowest CPU/accelerator interface is often a shared command queue. To exploit hardware parallelism, the command queue is asynchronous: CPU may dispatch multiple commands to the queue, and will be notified by the accelerator on the completion of these commands.

In multiplexing apps on an accelerator, the corresponding driver schedules their commands. The driver picks one app's pending commands for dispatch, based on the scheduling credits of all apps, e.g., their recent accelerator usages, and the driver's scheduling policy. To support psbox, we bake temporal balloons (§ 4.1) in the driver. We augment how the driver switches among commands of different apps and bills the accelerator usage; meanwhile, we keep any scheduling policy intact. In a nutshell, i) the augmented driver treats $\boxed{App}$ as a single app in scheduling; ii) the driver avoids dispatching commands of $\boxed{App}$ as long as any commands from $\overline{App}$ are outstanding; iii) the driver bills any resultant lost opportunity in utilizing the accelerator to $\boxed{App}$; iv) the driver further virtualizes the accelerator's operating frequency, its most important power state, for each psbox.

We next describe how the driver schedules in and out a temporal balloon.

(1) *Drain others*: When the driver's scheduling policy decides to dispatch commands for $\boxed{App}$, the driver buffers all subsequent requests (from both $\boxed{App}$ and $\overline{App}$) until the accelerator hardware notifies the completion of all existing commands. During this phase, the driver bills the unutilized portion of the accelerator (e.g., idle DSP cores) to $\boxed{App}$ as if the portion was actually used by $\boxed{App}$.

(2) *Flush psbox*: After draining outstanding commands, the driver sends out any buffered command for $\boxed{App}$, which may have accumulated during phase 1.

(3) *Serve psbox*: The driver directly dispatches all the subsequent requests from $\boxed{App}$ to the accelerator while buffering the ones from $\overline{App}$.

(4) *Drain psbox*: When the driver's scheduling policy decides that $\overline{App}$ deserves the access of accelerator, it drains any outstanding commands from $\boxed{App}$ in a way similar to phase 1. Over the course of phase 2–4, the driver bills the usage of entire accelerator to $\boxed{App}$.

(5) *Flush others*: The driver sends out any buffered commands from $\overline{App}$, which may have accumulated in phase 4, in their queueing order. Thereafter, it buffers all subsequent commands from $\boxed{App}$ while dispatching ones from $\overline{App}$ directly.

The above design integrates well with existing schedulers, yet are not tied to any specific definition of fairness or scheduling policy. A challenge to demonstrating this, however, is that many production accelerator drivers use simple scheduling policies, e.g., round-robin dispatch, which do not guarantee fairness. In our implementation described in Section 5, we have built fair queueing schedulers as baseline designs for GPU and DSP on our test platform, and augment the schedulers for supporting psbox.

Wireless interfaces

Wireless network interfaces (NICs) such as WiFi interface, are asynchronous by nature. Often, apps trap into the kernel to deposit their packets into their corresponding kernel buffers; the driver incorporates a packet scheduler to dispatch these packets into a unified transmission queue, from which the driver will send packets to the NIC in order. The packet scheduler determines scheduling credits for apps based their total sent bytes; it ensures fairness through its queueing discipline, e.g., the Linux fq_codel.

We tap into the packet scheduler to realize temporal balloons for NICs. We realize packet draining phases similar to accelerators as described above, while holding back packets in per-socket buffers instead of a global queue. To better assess lost sharing opportunities, the packet scheduler inspects packets that are buffered due to temporal balloons. It identifies any buffered packets that *could have* been dispatched without the balloons. Based on the total bytes in these packets, the driver discounts the scheduling credit of $\boxed{App}$ as a penalty for the lost opportunities.

A particular challenge is making packet reception respect psbox boundaries (§4.1). To achieve this, the NIC should *defer* receiving the packets that are not destined to the current temporal balloon, a function unsupported by commodity wireless NICs. Because of this, our current implementation is limited in insulating power impacts of receiving different packets. Yet, we have observed that such reception deferral can be achieved by exploiting virtual MAC addresses, an emerging feature of recent WiFi NICs [13, 15, 92]: the driver creates one virtual MAC for each psbox and switches among virtual MACs as it switches among temporal balloons.

Wireless NICs often have non-trivial power state that must be virtualized. Fortunately, modern WiFi NICs [86] often expose the control of power states to the OS. Hence, we augment the WiFi NIC driver to virtualize power states including transmission modes and power saving timer, and drive an independent state machine for each psbox. We recognize that cellular (4G) NICs have uncontrollable power states [41] which we will discuss in Section 7.
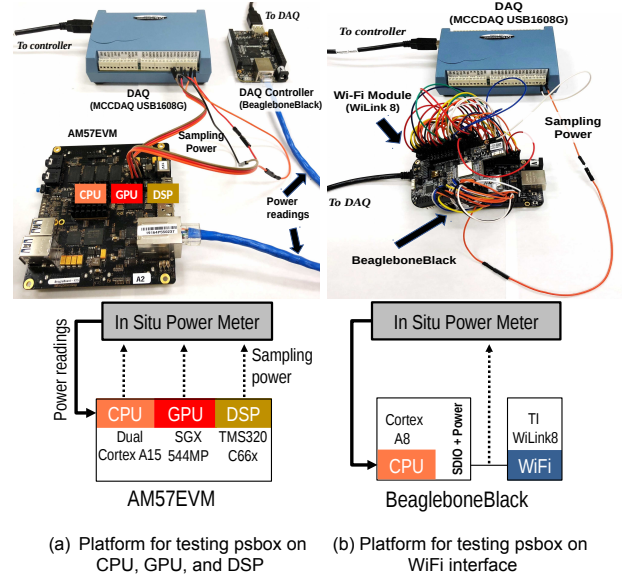
5 IMPLEMENTATION

We have built psbox into the Linux kernel 4.4 with about 2250 SLoC. We have assembled two hardware prototypes capable of measuring each of the major hardware components *in situ* and separately, as shown in Figure 4. The power sampling is as fast as 100KHz. Besides acquiring power samples, the power meter and the CPU synchronize their respective clocks to align power samples with software activities. It is worth noting that the purpose of our hardware prototypes is for evaluating psbox; they are not intended to be free-roaming devices as other systems [11, 79].

CPU We build psbox into the Linux completely fair scheduler (CFS) [61]. Although a CFS instance is able to schedule a process group (cgroup) as one scheduling entity, it does not coordinate multiple scheduler instances. We encapsulate each power sandbox in a Linux cgroup, and coordinate the tasks within through IPI.

GPU We implement psbox for PowerVR SGX544, a mobile GPU on the platform in Figure 4(a). Due to diversity of modern GPUs, we further evaluate psbox atop Qualcomm Adreno420 on Nexus 6. The two GPUs belong to different families, and have very different hardware/software stacks.

For both GPUs, we tap into their GPU command queues to implement fair schedulers in the spirit of the Linux CFS [61]: our scheduler tracks per-app virtual GPU runtime and favors GPU commands from the app that has the minimal virtual GPU runtime. Atop the schedulers, the drivers enforce temporal balloon boundaries differently, based on their existing structures: since SGX544 directly dispatches commands from syscall contexts to GPU, the driver buffers app *locking requests*; by contrast, since Adreno330

(a) Platform for testing psbox on CPU, GPU, and DSP

(b) Platform for testing psbox on WiFi interface

Figure 4: Our prototype hardware platforms used in evaluation. *In situ, per component* **power metering (through four distinct power rails) is built atop a Cortex-A8 controlling MCCDAQ USB1608G [57] sampling at 100KHz. Time synchronization is over GPIO (not shown). In (b), the Beaglebone Black acts as both the target system and the DAQ controller.**

buffers GPU commands in per-process queues before dispatching them, the driver buffers *commands* from apps.

DSP We implement psbox for TI c66x, a popular multicore DSP that supports OpenCL. During execution, CPU dispatches DSP commands, e.g., task execution or cache flush, via a kernel-managed command queue. Similar to GPU, we enforce resource partitions atop a fair scheduler along the command queue. The driver further inspects DSP commands for tracking their dispatch and completion time.

WiFi We build psbox for the TI WiLink8 NIC with a wl1837 chip as shown in Figure 4(b). The chip accepts packets and commands from CPU over SDIO, and runs its own firmware to implement MAC layer and below.

We build temporal resource partitions into the Linux's fair packet scheduler and virtualize the NIC power state in the driver. Despite the NIC's support of multiple MACs, when we switch MAC at run time the NIC resets and loses its association with base station. Therefore, the lack of true MAC virtualization defeats our effort in insulating energy impacts of receiving different packets, as described in Section 4.2.

6 EVALUATION

We evaluate the drivers augmented for psbox reported in Section 5 using benchmark apps summarized in Table 5. The evaluation answers the following questions:

§6.1 Does psbox eliminate power entanglement?
§6.2 How does psbox impact app performance?

	Benchmark	Description
CPU	bodytrack	A vision program tracking human body move (P)
	calib3d	Camera calibration and 3D reconstruction (O)
	dedup	Compressing data stream with deduplication (P)
GPU	browser	A webkit browser opening a Google homepage (T)
	magic	Rendering a "magic lantern" scene at 60fps (V)
	cube	Rendering a rotating cube scene at 60fps (Q)
	triangle	A synthetic app drawing 100k triangles /sec offscreen
DSP	sgemm	Single-precision matrix-multiplication (T)
	dgemm	Double-precision matrix-multiplication (T)
	monte	Monte Carlo simulation. (T)
WiFi	browser	A Links browser opening a Yahoo homepage
	scp	Transmitting a 50MB data file over ssh
	wget	Transmitting a 50MB data file over http

Figure 5: Benchmark apps used in evaluation. P-PARSEC 3; O-OpenCV 3.1; T-TI am57 SDK; V-PowerVR SDK; Q-Qt SDK

§6.3 Does psbox confine throughput loss to sandboxed apps?
§6.4 Does psbox facilitate building power-aware apps?

6.1 Elimination of power entanglement

Methodology To test each driver, we run a set of scenarios as shown in Figure 6. Designating a benchmark app *App* to be power-aware, we first run *App* alone and then co-run it with other apps. For co-running scenarios, we compare psbox to an existing kernel-level accounting mechanism [96] without psbox. This prior mechanism derives *App*'s power by dividing each system power sample among co-running apps based on their hardware usages in each power sampling interval. Note that we implement this prior mechanism favorably by tracking hardware usage at the lowest software level and at very fine granularities ($10\mu s$, $10\times$ smaller than prior work [11, 79]).

Our experiments demonstrate that psbox achieves its primary goal of eliminating power entanglement. As shown in the figure, no matter whether *App* is executed alone or co-executed with different apps, psbox keeps *App*'s power observations highly consistent, e.g., preserving significant power spikes and dips. By contrast, the power shares produced by the prior mechanism exhibit significant variations. The power differences are reflected in that of the accumulated energy: while the energy values reported by psbox are less than 5% within each other in most scenario sets, that of the prior approach can be as high as 60%. This also supports our argument in Section 2.3: existing accounting approaches are fundamentally inadequate, despite of the high metering rate. Note that psbox does not seek absolute *reproducibility* of power observations, which is difficult, if not impossible, on commodity computers. This is because OS resource scheduling and app behaviors are not guaranteed to be the same across different runs.

We further show the details of resource multiplexing, without and with psbox. As shown in Figure 7, psbox creates spatial and temporal balloons on CPU and DSP, respectively, and hence makes resource multiplexing respect the psbox boundaries. Outside of these balloons, the kernel multiplexes other apps freely as usual.

6.2 Performance impact

Latency increase All apps in the system may experience extra latency in some of their hardware access, if the hardware access happens to trigger resource balloon switch. Our implementation keeps the extra latency relatively low. Throughout our benchmark scenarios, the CPU scheduling latency is increased by tens of μs for task shootdown; the command dispatch latencies for GPU and DSP are increased by 1.8 ms and 100 ms on average, respectively.

The increased latency for WiFi packet transmission can be long, sometimes hundreds of ms. We found this is likely due to internal notification batching by the firmware of the WiLink NIC on the platform in Figure 4(b). In addition, the platform's wimpy CPU also contributes to interrupt handling latency. The combined software and hardware behaviors prolong draining phases.

Throughput loss As mentioned in Section 3, the exclusion of resource balloons may lead to lost sharing opportunities, which will reduce the total throughput on hardware. In our experiments, the total throughput loss can be noticeable, ranging from 0.9% (WiFi) to 9.8% (CPU). In face of the hardware throughput loss, we next discuss how well psbox maintains fairness among apps.

6.3 Confinement of throughput loss

Our system maintains throughput fairness among apps which may have different usages of psbox. To ease the comparison of app throughput loss, we co-run multiple instances of the same app. We show the throughputs of all the apps in Figure 8. When one app enters its psbox, it is the only one experiencing throughput loss; in comparison, the throughputs of other co-executing apps remain largely unaffected despite the total throughput decrease. Note that this is achieved without changing existing scheduling policies. This validates our key design of fully charging lost sharing opportunities to the sandboxed app (§4.2). We further test the robustness of our fairness guarantee under extremely high resource contention: we test the GPU driver, by co-running *browser* (in psbox) with *triangle*, a synthetic, intensive benchmark. Our results show that while the GPU throughput of *browser* drops by $4\times$ due to excessive draining time, that of triangle only decreases by 1%.

6.4 An end-to-end use case

We demonstrate the efficacy of psbox on a virtual reality (VR) scenario derived from a SDK demo (2K SLoC) [85]. The VR scenario lets a human user move her hand in order to control animated water waves. Two CPU tasks are running continuously. The *gesture* task processes video frames, identifies hand contours, and recognizes hand gestures. The *rendering* task translates the recognized gestures to wind directions, generates Phillips spectrum and 2D IFFT, and keeps refreshing a height map for animating the waves.

We, as app programmers, set to make *rendering* power-aware, so that it can trade the rendering fidelity (e.g. framerate, resolution) for lower power at run time. Without psbox, reasoning about the power of *rendering* is difficult due to power entanglement, as shown in Figure 9. To worsen the problem, the *gesture* task's workloads (and hence its power impacts) largely vary based on inputs, i.e., the number of contours in a frame. With psbox, the *rendering* task observes its power without the varying impacts of *gesture*. By adjusting the rendering fidelity based on its power observation in psbox, *rendering* achieves a wide range ($8.9\times$) of power, from 90mW to 800mW.
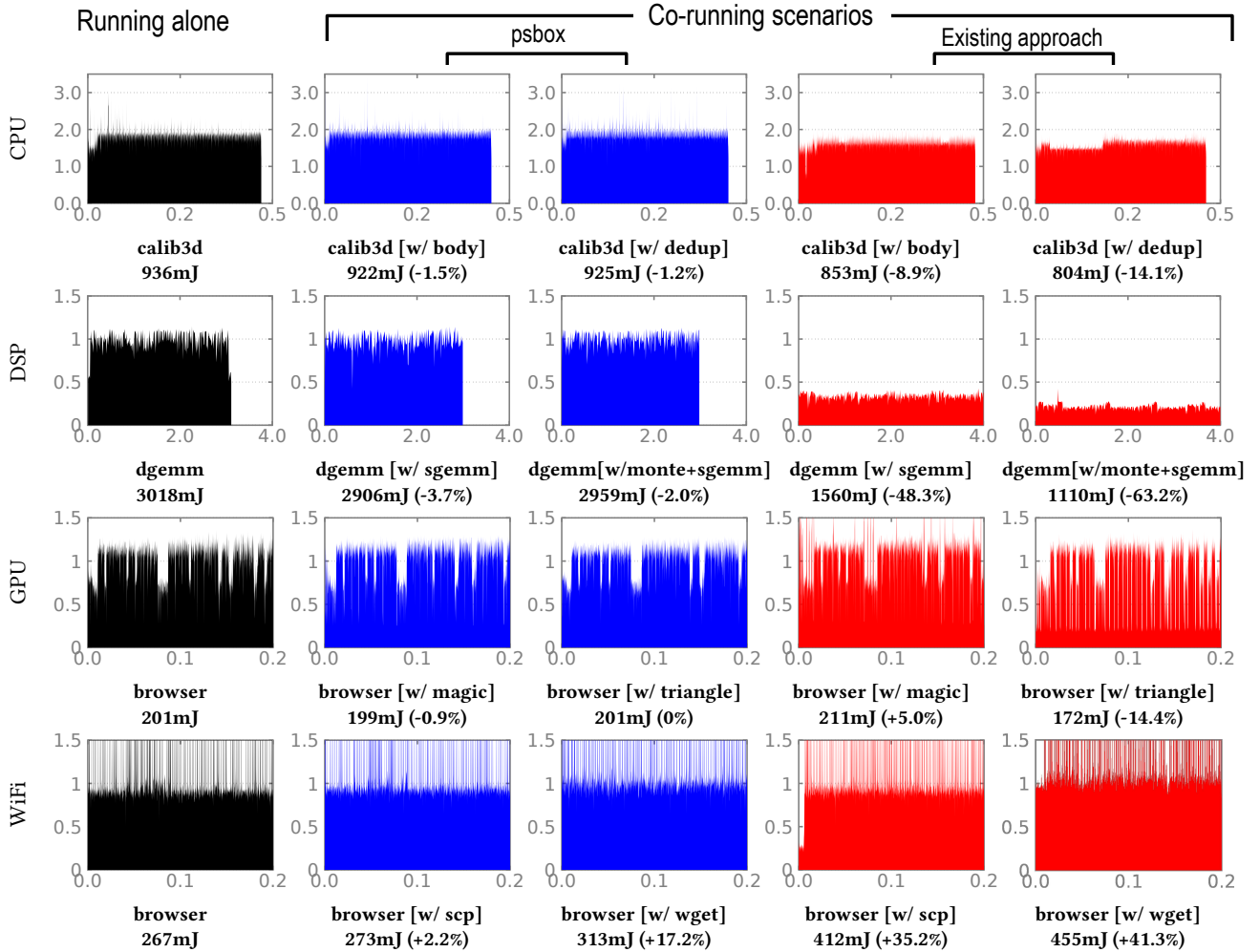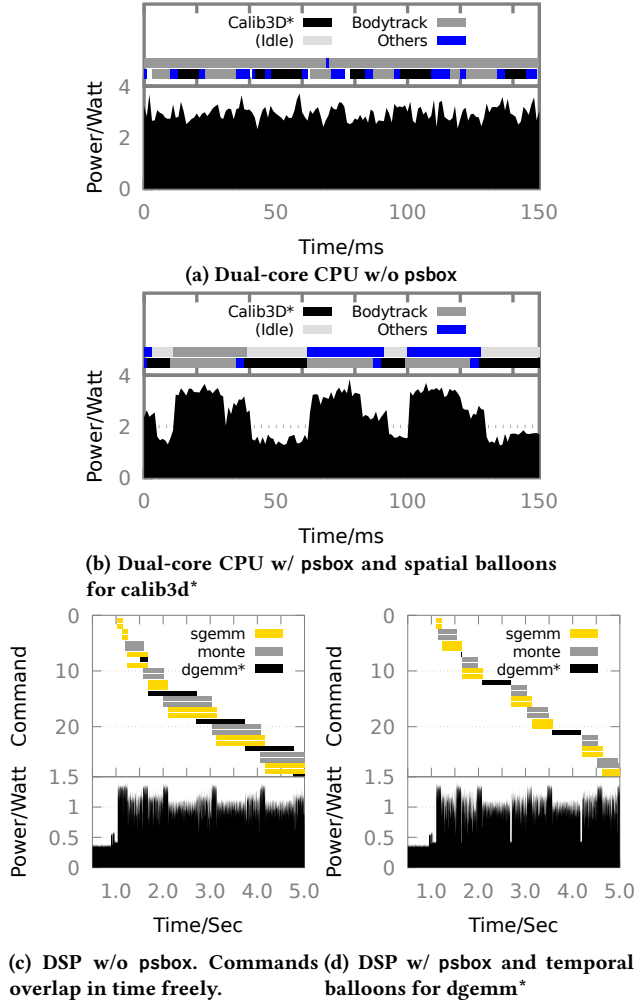
Figure 6: Power of the benchmark scenarios. In all plots, x-axis: Time/Sec; y-axis: Power/Watt. In each row: even as an app co-runs with different apps (column 2–5), psbox provides it with consistent power observations (column 2 & 3), which are close to the power of the app running alone (column 1). This contrasts to the power attributed to the app by an existing accounting approach (column 4 & 5). The numbers under each plot show the app's total energy and the difference compared to the energy when the app runs alone. Some plots cannot display full length of power activities due to space limit.

Without psbox isolation, the *rendering* task will mistakenly take entangled power impacts into account. This incorrect power knowledge will mislead the app's power adaptation strategy, lowering energy efficiency or user experience. This VR scenario demonstrates the benefit from insulating power impacts.

7 LIMITATIONS & DISCUSSIONS

Support psbox on extra hardware (1) **Display** may consume more than 50% of energy of a smartphone or tablet [12]. Fortunately, modern displays, notably OLED, are known free of power entanglement: each pixel contributes to the total power independently with little lingering power state [24]. Hence, OS may simply divide the display power among apps based the pixels produced by each app [70]. (2) **GPS** power is unaffected by concurrent uses once the device is operating. Therefore, the kernel can safely reveal GPS

hardware power, except when the GPS is in off/suspended state, to individual psboxes. This avoids expensive power state virtualization as described in Section 4.1. (3) **Cellular interface** While temporal balloons for cellular interface (4G) can be constructed in a way similar to WiFi NICs (§4.2), a unique challenge is for the kernel to virtualize a cellular interface's power state [96]. In practice, the state transitions of a cellular interface are not controllable by the OS, but by the cellular standard that must be agreed with cellular towers. To this end, psbox will be made feasible on cellular interfaces through future hardware support. (4) **DRAM** consume 5% – 25% of system energy [8, 12]. Given that DRAM power is often metered at the level of DIMM [19] or controller [21], it is possible to realize psbox on DRAM through temporal balloons, However, it is challenging to track app DRAM usage and ensure fairness, for which the OS may need to consult hardware performance counters.

(a) Dual-core CPU w/o psbox

(b) Dual-core CPU w/ psbox and spatial balloons for calib3d*

(c) DSP w/o psbox. Commands overlap in time freely.

(d) DSP w/ psbox and temporal balloons for dgemm*

Figure 7: Resource multiplexing and the resultant system power, before and after one app* enters psbox. (a)(b): CPU schedule and power. When Calib3D runs, the system power consumption is lower because Calib3D's psbox forces the other CPU core to stay idle. (c)(d): DSP commands and power.

Userspace OS daemon   Our current implementation focuses on kernel drivers. In other systems especially Android, multiplexing of app requests also happens in user-level daemons. It is possible to build psbox into these daemons by making their request multiplexing respect psbox boundaries.

Power-aware entities other than apps   Some scenarios define alternative entities for power awareness, e.g. a user request served by multiple processes or even machines [30, 81]. psbox may enclose these entities in addition to an app. To do support this, each involved process or machine, as points of multiplexing, must be augmented to respect psbox boundaries.

Alternative OS mechanisms for supporting psbox   Besides our Linux-based instantiation of psbox (§4.2), there are existing OS mechanisms that are absent in the mainline Linux yet suiting the need of enforcing psbox. First, scheduler activations [3] help move
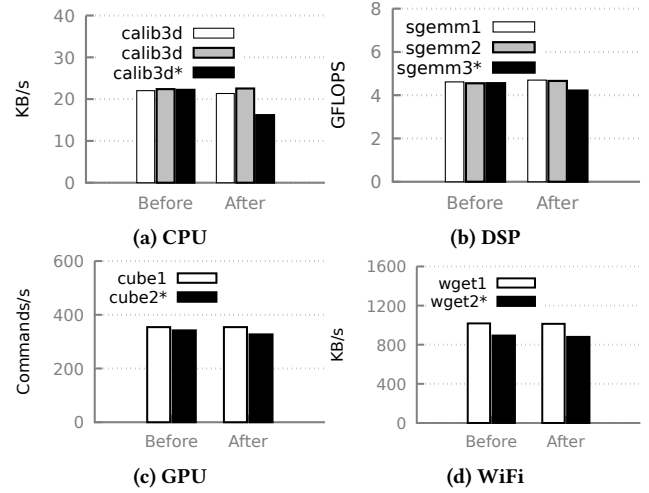
Figure 8: Throughputs of co-running app instances, before and after one instance (marked with *) enters psbox.
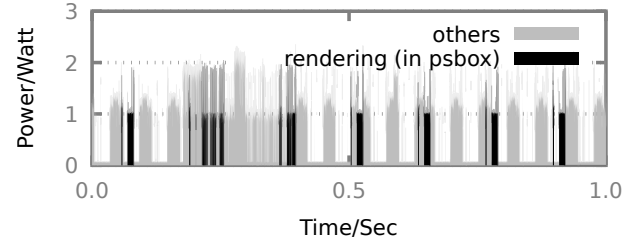
Figure 9: CPU power of a VR scenario. The rendering task enters psbox to observe its power and adapts accordingly

much of the CPU scheduling logic for psbox to user space. With such a mechanism, an app in its psbox spawns dummy threads to occupy unused cores for enforcing the balloon boundary; as the app's actual threads suspend/resume, the kernel notifies the app through upcalls, which adjust the number of dummy threads accordingly. Second, gang scheduling [20], commonly seen in real-time kernels, directly supports executing all threads in a psbox (a gang) simultaneously and enforces mutual exclusion among gangs. Third, systems like Dune [9] creates *per-app* virtualized views of the baremetal CPU hardware. This idea can be further extended to create per-app views of baremetal I/O devices, e.g. WiFi NIC. The virtualization cost can be further reduced by only enforcing power insulation (as required by psbox) while eschewing strong state isolation.

8   ROAD TO EXISTING ECOSYSTEMS

To bring psbox and the power awareness into today's mobile and embedded ecosystems, the major challenges are twofold: i) processing high-rate power data with low hardware cost and ii) reusing mature APIs. We next discuss how these can be achieved by leveraging the *existing* software/hardware support for sensor data processing.

8.1 Hardware support

We next discuss how situ power metering (§ 2.2) can be realized atop existing hardware platforms with little addition.

Integrating with existing sensor hubs To harness rich sensors, most modern mobile/embedded devices incorporate sensor hubs, whose overall market is projected to exceed 2 billion units [42]. Sensors hubs are dedicated, extremely efficient processors for pre-processing sensor data, typically incarnated as Arm Cortex-M MCUs. As the volume of sensor hubs grows, their cost keeps decreasing: it is several US dollars per unit at the time of writing. They are penetrating most of the mobile/embedded SoC market.

By their design, sensor hubs directly suit pre-processing of power samples. A Cortex-M0 sensor hub clocked at 32 MHz consumes as low as 13 mW, and is capable of real-time processing of power data sampled at 1 KHz [50]. Such a sampling frequency already exceeds what is demanded by existing power-aware systems [28, 29, 79], and is able to differentiate microscopic power activities, e.g. scheduler context switch as shown in Figure 7.

Asymmetric cores We recognize that there exist mobile/embedded devices that do not have sensor hubs (yet). To increase efficiency of pre-processing power samples, they can leverage the lower-power cores in modern Arm architecture, e.g. big.LITTLE and DynamIQ [64]. The trend of increasing architectural asymmetry promises better processing efficiency.

Utilizing low-cost power sensors Modern mobile devices are already sensor-rich. For instance, the recent iPhone X has eight sensors of different types [43], ranging from the accelerometer to proximity sensor. Often, it is the types of sensors that differentiate mobile devices. While existing sensors are for *extrospection*, we believe it is equally valuable and feasible for the devices to additionally incorporate power sensors for *introspection*.

Power sensors can be very cost effective. The simplest power sensor can be a shunt resistor accompanied by an analog-to-digital converter (ADC); the latter can be further integrated into an on-chip I/O controller [5]. The combined cost is less than $1 [87]. Standalone current sensing ICs provide additional design convenience. At minor cost (around $1 per unit) [44], such an IC can be attached to a device's I2C bus with little extra hardware complexity. A typical current sensing IC [45] is capable of sampling three power rails at 500KHz simultaneously and returns digitalized power samples. They are already pervasive on experimental devices including Tegra X1 [67], X2 [68], Odroid XU3 [35].

8.2 Software support

To foster its adoption, psbox can further leverage the existing software infrastructure. This includes mature API frameworks and processing algorithms of sensor samples.

High-level sensor APIs Mobile OSes such as Android and iOS support tens of sensor types. They already offer mature APIs for apps to retrieve sensor data and subscribe to sensor events [4, 33]. The psbox native interface, as presented in Section 3, can be further wrapped under such APIs, adding a new "power" sensor type. For instance, through calling Android's `SensorManager.registerListener`, an app is able to retrieve power samples or register callbacks for "high power" events. This is exactly how today's apps monitor existing sensors such as accelerometers.

To cater to app-defined power events, existing sensor APIs can be further augmented with simple temporal predicates [73]. Through embedded scripting languages such as Lua or Javascript, the apps can specify events such as "frequent power spikes" or "power keeps increasing". The predicates are continuously evaluated over power samples by the OS or the sensor hub.

Sensor hub runtime As discussed before, processing of power samples can be offloaded to sensor hub hardware for efficient execution. Fortunately, there exist rich runtime software on sensor hubs that facilitates such offloading. First, existing commodity sensor hub runtimes, e.g. SenseMe [74], are already mature; they provide an arsenal of signal processing algorithms, e.g. denoising, that can pre-process power samples with high efficiency. Second, recent research has proposed a variety of techniques for simplifying new code development for sensor hubs. For instance, our work Reflex [55] creates a software distributed shared memory between CPU and sensor hubs; MobileHub [80] automatically learns sensor events and produces event detection code for sensor hubs; Sidewinder [54] supports composition of parameterized, pre-defined algorithms for sensor hubs. These rich techniques are applicable to development of power data processing algorithms for sensor hubs.

9 RELATED WORK

Power metering Much work infers power from software-visible events, such as syscall activities [70, 71], kernel activities [96], hardware states [26, 59, 81, 82, 94, 97, 100]. They often construct linear models either during development [59, 70, 71, 81, 96, 97, 100] or at run time [26, 82, 94]. Although convenient, energy modeling is limited by complex hardware [56] and high variation in semiconductor process [83]. Intel RAPL [21] is a CPU feature: the firmware monitors hardware activities and infers power based on pre-defined models. Yet, lacking timestamps, RAPL power samples can hardly be mapped to software activities at fine time granularity [22, 34]. Direct measurement allows accurate power metering through external multimeters [28, 29], fine-grained hardware instrumentation [84], smart switching regulators [27], smart battery interfaces [11, 79], and specialized metering circuits [32, 88]. Regardless of metering approaches, power entanglement is inevitable as explained in Section 2, which necessitates power sandboxes.

Power accounting heuristics As mentioned in Section 2, prior work attributes power using various heuristics. Eprof [70, 71] attributes lingering tail power to the last triggering entity. HaPPy [99] splits hyperthreading CPU power based on per-thread aggregated cycles. Ghanei *et al.* [31] track asynchronous hardware use and evenly divides power among concurrent apps. Dong *et al.* [25] attribute energy based game theory. Power Containers [81] meters per core power, while evenly splitting the power of shared resources among active cores. Joulemeter [47] models per-VM power in the server by inferring system power from hardware activities reported by OS or performance counters. However, without eliminating power entanglement, they suffer from the inadequacies described in Section 2. It is also difficult to apply the performance counter-based approaches to many accelerators and I/Os that lack performance counters.

OS-level power management Power management has been a key OS responsibility. Odyssey [28, 29] enables the OS to guide

apps for energy-aware adaptation. ECOSystem [97] and Cinder [77] present OS-level abstractions for energy. Koala [82] builds energy models in the kernel and sets performance/efficiency dynamically. Rao *et al.* [75] build a controller to balance performance loss and energy saving, based on application-specific data profiled offline. OS also manages power for accelerators [72] and I/O devices [93, 98].

However, none prior work presented virtualized power view to individual apps.

Power side-channel attacks Prior work exploits power side channels to steal private information on smartphones [95], recover cryptographic keys [51, 52], reveal mobile user geolocations [58], and leak information across virtual machines [37]. However, few systems prevent power side channels through active resource management as we do.

OS resource scheduling Several proposals on scheduling are in particular related to psbox. GPU scheduling has been advocated for long. TimeGraph [48] prioritizes and isolates performance of competing apps. PTask [76], Gdev [49], and Menychtas *et al.* support fair sharing of GPU. ShuffleDog [40] prioritizes UI tasks through resource scheduling. SmartIO [66] reduces app delay by prioritizing disk reads over writes. Energy discounted computing [102] co-schedules tasks to improve total system efficiency. Complementary to psbox, these scheduling proposals target performance or efficiency for power-unaware apps.

10 CONCLUSIONS

An app's power observation should be insulated from the impacts of concurrent apps. We introduce power sandbox, a new OS principal capturing the power of the enclosed app and its vertical environment. To support power sandbox, our key techniques are two: to allocate exclusive resource partitions at fine granularities and bill the lost sharing opportunities; to virtualize hardware power states. Our experience shows that power sandbox simplifies reasoning, eliminates security vulnerability, and still ensures fairness among apps.

ACKNOWLEDGMENTS

REFERENCES
[1] Perf. https://perf.wiki.kernel.org/index.php/Tutorial.
[2] *Dynamic Time Warping*, pages 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
[3] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.
[4] Apple. Core Motion. https://developer.apple.com/documentation/coremotion.
[5] ARM. 64 bit juno arm development platform. http://www.arm.com/files/pdf/Juno_ARM_Development_Platform_datasheet.pdf, 2014.
[6] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.
[7] Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, August 2006.
[8] Luiz AndrÃ Barroso, Jimmy Clidaras, and Urs HÃűlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2013.
[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proc. USENIX OSDI*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.
[10] Eric Brier, Christophe Clavier, and Francis Olivier. *Correlation Power Analysis with a Leakage Model*, pages 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
[11] Niels Brouwers, Marco Zuniga, and Koen Langendoen. Neat: A novel energy analysis toolkit for free-roaming smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 16–30, New York, NY, USA, 2014. ACM.
[12] Aaron Carroll and Gernot Heiser. The systems hacker's guide to the galaxy: energy usage in a modern smartphone. In *Proc. of the 4th Asia-Pacific Workshop on Systems (APSYS)*, page 5. ACM, 2013.
[13] Microsoft Hardware Dev Center. Virtual wifi in kernel mode. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/virtual-wifi-in-kernel-mode/, 2017.
[14] Geoffrey Challen and Mark Hempstead. The case for power-agile computing. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 15–15, Berkeley, CA, USA, 2011. USENIX Association.
[15] R. Chandra, P. Bahl, and P. Bahl. Multinet: connecting to multiple ieee 802.11 networks using a single wireless card. In *IEEE INFOCOM 2004*, volume 2, pages 882–893 vol.2, March 2004.
[16] Hui Chen, Bing Luo, and Weisong Shi. Anole: A case for energy-aware mobile application design. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 232–238, Washington, DC, USA, 2012. IEEE Computer Society.
[17] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
[18] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proc. USENIX/ACM MobiSys*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.
[19] Zehan Cui, Yan Zhu, Y. Bao, and M. Chen. A fine-grained component-level power measurement method. In *2011 International Green Computing Conference and Workshops*, pages 1–6, July 2011.
[20] Nikunj A. Dadhania. Gang scheduling in cfs. https://lwn.net/Articles/472797/, 2011.
[21] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.
[22] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 455–470, New York, NY, USA, 2016. ACM.
[23] Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 29–40, New York, NY, USA, 2013. ACM.
[24] M. Dong and L. Zhong. Chameleon: A color-adaptive web browser for mobile oled displays. *IEEE Transactions on Mobile Computing*, 11(5):724–738, May 2012.
[25] Mian Dong, Tian Lan, and Lin Zhong. Rethink energy accounting with cooperative game theory. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 531–542, New York, NY, USA, 2014. ACM.
[26] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.
[27] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 283–294, April 2008.
[28] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM.

[29] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[30] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA, 2008. USENIX Association.

[31] Farshad Ghanei, Pranav Tipnis, Kyle Marcus, Karthik Dantu, Steve Ko, and Lukasz Ziarek. Os-based resource accounting for asynchronous resource use in mobile systems. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 296–301, New York, NY, USA, 2016. ACM.

[32] Bartosz Golaszewski. sigrok: Adventures in integrating a power-measurement device. http://events.linuxfoundation.org/sites/events/files/slides/ELC_pres_bgolaszewski.pdf.

[33] Google. Sensors Overview. https://developer.android.com/guide/topics/sensors/sensors_overview.html.

[34] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, April 2013.

[35] Hardkernel. Odroid xu3: Board detail. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127&tab_idx=2, 2014.

[36] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Acpi - advanced configuration and power interface. http://www.acpi.info/.

[37] H. Hlavacs, T. Treutner, J. P. Gelas, L. Lefevre, and A. C. Orgerie. Energy consumption side-channel attack at virtual machines in a cloud. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 605–612, Dec 2011.

[38] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, New York, NY, USA, 2015. ACM.

[39] Mohammad Hosseini, Alexandra Fedorova, Joseph Peters, and Shervin Shirmohammadi. Energy-aware adaptations in mobile 3d graphics. In *Proceedings of the 20th ACM International Conference on Multimedia*, MM '12, pages 1017–1020, New York, NY, USA, 2012. ACM.

[40] G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu. Shuffledog: Characterizing and adapting user-perceived latency of android apps. *IEEE Transactions on Mobile Computing*, PP(99):1–1, 2017.

[41] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[42] IHS Inc. Led by iphone 6s, sensor hubs market is growing fast, ihs says, ihs markit press release, 2017.

[43] Apple Inc. iPhone X, Tech Specs. https://www.apple.com/iphone-x/specs/, 2017.

[44] Texus Instruments. Ina231, ina3221 triple-channel, high-side measurement, shunt and bus voltage monitor with i2c and smbus-compatible interface. http://www.ti.com/lit/ds/symlink/ina3221.pdf, 2016.

[45] Texus Instruments. Ina3221, 28-v, bi-directional, zero-drift, low-/high-side, i2c out current/power monitor w/ alert in wcsp. http://www.ti.com/product/INA231, 2018.

[46] John Levon. OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/about/.

[47] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[48] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[49] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[50] Kionix. Kx23h-1035: Arm-based sensor hub with accelerometer. http://www.kionix.com/product/KX23H-1035, 2014.

[51] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[52] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

[53] Nicholas D. Lane, Yohan Chon, Lin Zhou, Yongzhe Zhang, Fan Li, Dongwon Kim, Guanzhong Ding, Feng Zhao, and Hojung Cha. Piggyback crowdsensing (pcs): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 7:1–7:14, New York, NY, USA, 2013. ACM.

[54] Daniyal Liaqat, Silviu Jingoi, Eyal de Lara, Ashvin Goel, Wilson To, Kevin Lee, Italo De Moraes Garcia, and Manuel Saldana. Sidewinder: An energy efficient and developer friendly heterogeneous architecture for continuous mobile sensing. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 205–215, New York, NY, USA, 2016. ACM.

[55] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proc. ACM ASPLOS*, pages 13–24, New York, NY, USA, 2012. ACM.

[56] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[57] Measurement Computing. USB-1608G Series User's Guide, 2012.

[58] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 785–800, Washington, D.C., 2015. USENIX Association.

[59] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[60] Shivajit Mohapatra, Nalini Venkatasubramanian, Nikil Dutt, Cristiano Pereira, and Rajesh Gupta. Energy-aware adaptations for end-to-end video streaming to mobile handheld devices. In E. Macii, editor, *Ultra Low-Power Electronics and Design*. Springer Science & Business Media, 2007.

[61] Ingo Molnar. [patch] modular scheduler core and completely fair scheduler. http://lwn.net/Articles/230501/, 2007.

[62] Philip J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. http://icl.cs.utk.edu/~mucci/papiex.

[63] L. Mukhanov, D. S. Nikolopoulos, and B. R. d. Supinski. Alea: Fine-grain energy profiling with basic block sampling. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 87–98, Oct 2015.

[64] Nandan Nayampally. ARM DynamIQ: Expanding the possibilities for artificial intelligence. 2017.

[65] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 323–336, Berkeley, CA, USA, 2008. USENIX Association.

[66] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 287–300, New York, NY, USA, 2015. ACM.

[67] Nvidia. Jetson tx1 voltage and current monitor configuration application note. https://developer.nvidia.com/embedded/tegra-2-reference, 2017.

[68] Nvidia. Tegra x2: Technical reference manual. https://developer.nvidia.com/embedded/tegra-2-reference, 2017.

[69] John K. Ousterhout. Scheduling techniques for concurrebt systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*, pages 22–30, 1982.

[70] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[71] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[72] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated CPU-GPU power management for 3D mobile games. In *Proc. of the 51st Annual Design Automation Conference (DAC)*, pages 40:1–40:6, 2014.

[73] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[74] QuickLogic. SenseMeâĐć - Sensor Algorithm Library for Mobile Devices. https://www.quicklogic.com/technologies/sensor-hub/senseme/.

[75] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong. Application-specific performance-aware energy optimization on android mobile devices. In *2017 IEEE*

International Symposium on High Performance Computer Architecture (HPCA), pages 169–180, Feb 2017.

[76] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[77] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[78] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 85–96, New York, NY, USA, 2010. ACM.

[79] Aaron Schulman, Tanuj Thapliyal, Sachin Katti, Neil Spring, Dave Levin, and Prabal Dutta. Stanford CS battor: Plug-and-debug energy debugging for applications on smartphones and laptops. Technical report, 2016.

[80] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 227–238, New York, NY, USA, 2015. ACM.

[81] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 65–76, New York, NY, USA, 2013. ACM.

[82] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proc. of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 289–302, 2009.

[83] Guru Prasad Srinivasa, Rizwana Begum, Scott Haseley, Mark Hempstead, and Geoffrey Challen. Separated by birth: Hidden differences between seemingly-identical smartphone cpus. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, HotMobile '17, pages 103–108, New York, NY, USA, 2017. ACM.

[84] T. Stathopoulos, D. McIntire, and W. J. Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 383–394, April 2008.

[85] Texas Instruments. Processor SDK Demos Video Analytics. http://processors.wiki.ti.com/index.php/Processor_SDK_Demos_Video_Analytics.

[86] Texas Instruments. WL18x7MOD WiLink 8 Dual-Band Industrial Module âĂŞ Wi-Fi, Bluetooth, and Bluetooth Low Energy, 2015.

[87] Texus Instruments. Ads7040: Ultra-low-power ultra-small-size sar adc. http://www.ti.com/product/ADS7040, 2017.

[88] Patrick Titiano. Leveraging open-source power measurement standard solution. http://events.linuxfoundation.org/sites/events/files/slides/Leveraging_Open-Source_Power_Measurement_Standard_Solution_0.pdf.

[89] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[90] Vince Weaver. The unofficial Linux Perf Events web-page. http://web.eece.maine.edu/~vweaver/projects/perf_events. Last accessed: Dec. 12, 2013.

[91] Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative i/o: A novel i/o semantics for energy-aware applications. *SIGOPS Oper. Syst. Rev.*, 36(SI):117–129, December 2002.

[92] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. Virtual wifi: Bring virtualization from wired to wireless. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 181–192, New York, NY, USA, 2011. ACM.

[93] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated os-level device power management for socs. In *Proc. ACM ASPLOS*, New York, NY, USA, 2015. ACM.

[94] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 43–55, Lombard, IL, 2013. USENIX.

[95] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A study on power side channels on mobile devices. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, pages 30–38, New York, NY, USA, 2015. ACM.

[96] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 387–400, Boston, MA, 2012. USENIX.

[97] Heng Zeng, Carla Schlatter Ellis, and Alvin R Lebeck. Experiences in managing energy with ecosystem. *IEEE Magazine Pervasive Computing*, 4(1):62–68, 2005.

[98] Shuang Zhai, Liwei Guo, Xiangyu Li, and Felix Xiaozhu Lin. Decelerating suspend and resume in operating systems. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, HotMobile '17, pages 31–36, New York, NY, USA, 2017. ACM.

[99] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 211–217, Philadelphia, PA, 2014. USENIX Association.

[100] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[101] B. Zhao, W. Hu, Q. Zheng, and G. Cao. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):761–774, March 2015.

[102] Meng Zhu and Kai Shen. Energy discounted computing on multicore smartphones. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 129–141, Denver, CO, 2016. USENIX Association.