Drowsy Power Management

Matthew Lentz
University of Maryland
mlentz@cs.umd.edu

James Litton
University of Maryland
litton@cs.umd.edu

Bobby Bhattacharjee
University of Maryland
bobby@cs.umd.edu

Abstract

Portable computing devices have fast multi-core processors, large memories, and many on-board sensors and radio interfaces, but are often limited by their energy consumption. Traditional power management subsystems have been extended for smartphones and other portable devices, with the intention of maximizing the time that the devices are in a low-power "sleep" state. The approaches taken by these subsystems prove inefficient for many short-lived tasks common to portable devices, e.g., querying a sensor or polling a cloud service.

We introduce Drowsy, a new power management state that replaces "awake." In the Drowsy state, not all system components are woken up, only the minimal set required for a pending task(s). Drowsy constructs and maintains the minimal task set by dynamically and continuously inferring dependencies between system components at run-time. We have implemented Drowsy within Android, and our results show a significant improvement (1.5-5x) in energy efficiency for common short-lived tasks.

1. Introduction

The functionality of portable wireless devices, including smartphones, tablets, sensors, and embedded hardware, is often limited by energy. Software executing on these devices, including the OS and system software, is designed to conserve energy. For instance, Android operates in the low-energy "sleep" state by default; applications must actively notify the system that they require the device to stay "awake" (or *on*) by holding locks that prohibit the system from sleeping. Enhancements in power management techniques have enabled portable devices with small batteries to provide service for hours during use, and for days when idle.

A large proportion of energy savings comes from device-specific hardware power management, e.g., displays have a lower-power mode, CPUs have low-power speed governors, and on-board peripherals can transition to low-power modes. Yet, surprisingly, such intermediate power states have largely not made their way into software; rather, power management for processes is binary. Neither processes nor the kernel can run when the CPU is powered down, and when the system is active, *all* tasks are resumed and may be run. As a result, even with hardware support for power management techniques, devices remain highly inefficient when processing short, interrupt-driven wakeups.

The crux of the problem lies in the observation that current power management mechanisms were primarily designed in an era of solely human-driven wakeups, which tend to last on the order of minutes to hours. However, mobile devices demand a fundamental reconsideration of this assumption: periodic tasks, such as those that poll on-board sensors, rely on interrupt-driven wakeups and last on the order of tens to hundreds of milliseconds. Existing mechanisms to reduce power rely on human-timescale assumptions by using heuristics for the hardware-supported power management to reduce consumption. These techniques do not scale down for supporting such short-lived, machine-driven computations that touch few devices.

We introduce a new kernel power-management state, *Drowsy*, as a replacement for the *on* state. When transitioning from "sleep" to *drowsy*, the OS resumes only the *necessary* tasks and devices to handle any work performed prior to returning to sleep. For instance, if the system wakes up in response to a hardware real-time-clock (RTC) interrupt event, then with Drowsy, only the devices related to the RTC, necessary system services, and the user-level process that should receive the alarm signal are woken up. All other devices in the system remain in a low-power mode, unaware that the system has transitioned to *drowsy*, and all unaffected software, including device drivers, system services, and user processes remain "frozen." The system transitions back to "sleep" after the event triggered by the hardware interrupt is handled.

A primary design goal for Drowsy is to not perturb user-space, in that no user-space applications should require modification, source-code analysis, or recompilation for the system to operate in the *drowsy* state. Therefore Drowsy has to infer dependencies between devices and tasks (i.e., kernel-

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815414

visible threads) at runtime. We describe different methods for transparently determining these dependencies at runtime.

We present an implementation of Drowsy within Android. Android has no well-defined interfaces that explicitly declare dependencies involving tasks and devices; we describe how to instrument the Android kernel to efficiently infer dependencies dynamically at runtime. We show that for many common tasks, Drowsy can reduce Android energy consumption during the wakeup cycle by a factor of up to 5x with a commensurate reduction in the elapsed time for handling such tasks. We use a custom circuit, detailed in Appendix A, to precisely measure energy consumed by small devices. The circuit is able to synchronize kernel events with the energy traces, and is capable of measuring extremely low-power draws, such as when the device is suspended. Our measurements show that the improvements due to Drowsy translate to significant gains in overall battery life.

The rest of the paper is structured as follows: Section 2 provides background on Android power management. Section 3 describes Drowsy's design, followed by details of our Android implementation in Section 4. Section 5 presents our evaluation in which we compare Drowsy with stock Android. We discuss related work in Section 6, and conclude in Section 7.

2. Power Management Background

The popularity of portable devices has led to various power management states (beyond *on* and *off*) being incorporated into OS kernels. The additional states are *"sleep"* states, which try to conserve energy by placing portions of the system in low-power modes. Two common *"sleep"* states are *suspend* and *hibernate*, which are defined in the ACPI standard [4]. *Suspend* halts executing processes, places devices in a lower powered state, and leaves RAM powered with the CPU disabled. *Hibernate* is similar to *suspend*, except that the OS writes the RAM image to non-volatile storage when hibernating, allowing the system to be fully powered down.

The mechanisms underlying Drowsy build on how modern kernels transition between power management (PM) states. We review these transitions between different power states using Linux as an example. Our description follows the Linux kernel 3.4 source tree. For more information on the *suspend* state in Linux, see Brown et al. [8].

2.1 State Transition in Linux

In Linux, *task* denotes a kernel-visible thread that is either part of a user-space process or is a kernel-space thread. Tasks can be "frozen," which renders them unschedulable until they are "thawed." All user-space tasks and a subset of kernel tasks are freezable, indicated by a "freezable" flag in the task state.

Each peripheral device (or simply device) is comprised of a device driver and a hardware device. The driver is kernel-resident software that exposes interfaces for accessing the hardware device, and supports power management (PM) methods for transitioning between power states. The driver registers handlers with the OS for callbacks when specific interrupt requests (IRQs) are sent by hardware devices. In addition, there are many *virtual* devices which consist of only a device driver and are not backed by any actual hardware device. These virtual devices provide various utilities (e.g., /dev/random for cryptographically secure random number generation) and serve to abstract the actual underlying hardware devices (e.g., /dev/rtc).

on *to* suspend When transitioning to the *suspend* state, Linux first synchronizes all filesystem buffers with their backing stores. Afterwards, the kernel freezes all user-space and freezable kernel-space tasks, as detailed in Section 4.2.

Once all tasks are frozen, the system begins to suspend all active devices. The kernel PM subsystem maintains a list of devices in the order that they are registered. This list is a topological ordering of the device tree maintained by the OS, which represents the parent-child relationships between devices. Suspending devices occurs in multiple phases, traversing the list from tail to head (children are handled before parents). The kernel:

1. instructs each device to prepare for a transition, which prevents registration of new child devices.
2. requests each device to stop I/O, save state, and enter a low-power mode.
3. disables device IRQ handlers and allows devices to finalize the transition without the possibility of IRQ handler invocation.

At the end of this process, all devices are suspended in a low-power state; some devices may completely disable themselves, while others remain active enough to generate wakeup interrupts [23]. Finally, the OS disables non-boot CPU cores and invokes architecture-specific callbacks to finalize the state transition, setting RAM in self-refresh mode and instructs the boot CPU to wait for a wakeup interrupt.

suspend *to* on Modern OSes, especially those running on portable devices, allow the system to be woken up by a variety of devices that generate wakeup interrupts (e.g., Bluetooth controller). After an interrupt is generated by a hardware device, the system wakes up through a resume transition back to the *on* state.

The resume transition mirrors the *suspend* process, starting with the enabling of non-boot CPU cores. Devices power up starting with the head of the device list, (parents are handled before children), reversing the order of operations during suspend. Next, the resume routine thaws processes, which enables them to run (as detailed in Section 4.2).

We integrated Drowsy into the Android kernel, which itself has sophisticated power management built on top the Linux kernel. In the rest of this section, we describe Android power management in detail.

2.2 Android Power Management

Android inherits the Linux PM subsystem and supports three global system power states: *on*, *off*, and *suspend*. In order to conserve energy, Android makes *suspend* the default state, i.e., the system suspends unless a task or device driver has explicitly requested that it stay awake. From the *suspend* state, various wakeup events transition the system to the *on* state, such as a user pressing the power button, an incoming call, or the real-time clock (RTC) alarm. After a wakeup event is handled completely (e.g., user finished interacting with the device), the system transitions back to the *suspend* state until the next wakeup event occurs. We term this sequence of events (i.e. *suspend* to *on* → event handling → *on* to *suspend*) as a *wakeup cycle*.

Wakelocks Android needs tasks to specify when it is safe to transition back to *suspend*. It cannot simply transition to *suspend* when the CPU is idle, because even with the processor completely idle, it is not necessarily the case that the wakeup event has been handled. An active task may be blocked waiting on I/O, such as a weather application waiting for a response from the remote server. *Wakelocks* are named resources that can be requested by tasks or device drivers. While any wakelock is held the system does not suspend. When all wakelocks are released, the system immediately begins transitioning to *suspend*.

Scheduling Wakeups To allow applications to wake up at a specified time, Android introduces wakeup alarms via the `/dev/alarm` virtual device and the `AlarmManager` system service. Applications register an `Intent`[1] to be broadcast by the `AlarmManager` at a specified time. The `/dev/alarm` device driver manages these alarms by using OS-provided high-resolution timers; when the system suspends, this driver uses the RTC device to trigger a wakeup at the next alarm time.

Early Suspend Users expect the device to go to sleep when they press the power button, but if a wakelock is held this cannot occur. Instead, Android allows device drivers to register *early suspend* (and *late resume*) PM callbacks, which notify the subscribed drivers when the user-perceptible sleep state changes. For example, if the device is on and the user presses the power button to stop interacting with the device, the system invokes all early suspend callbacks (e.g., display and backlight power down). After early suspend, and once all wakelocks are released, the system transitions to the *suspend* state.

Illustrative Example Figure 1 shows the various steps within Android for a weather application that periodically wakes up and polls a remote weather service. The Android `system_server` provides utility services for user-space applications, including acquiring/releasing wakelocks (via `PowerManager`) and setting alarms (via the `AlarmManager`). The weather application sets a future wakeup alarm via the `system_server`, which in turn communicates with drivers and hardware devices as shown. The system suspends once all wakelocks are released, which involves freezing all tasks (canceling the blocked `ioctl`) and suspending all devices (which includes setting the hardware RTC alarm). Later, the RTC device triggers a wakeup interrupt that eventually reaches the `AlarmManager` via the restarted `ioctl` call. The `AlarmManager` notifies the weather application by sending it an `Intent`; upon receiving the `Intent`, the application acquires a wakelock, and then performs its necessary actions before releasing its wakelock.

When the system resumes as a result of the interrupt from the RTC device, Android wakes up all tasks and devices. However, the tasks and devices involved in handling the wakeup event are extremely limited, essentially just those represented in the figure (as well as the networking device).

[1] `Intents` are messages that consist of an action and (optionally) associated data, which either specify a destination or are routed based on content. They are the primary IPC mechanism in Android.

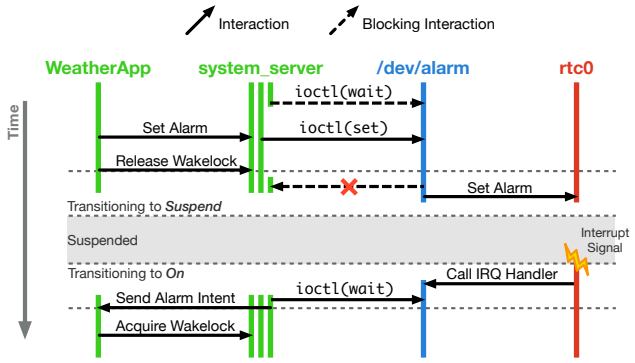

Figure 1: Message sequence of system interactions for an application that periodically checks the current weather forecast in Android. Vertical lines represent tasks and devices, where breaks indicate when each is frozen or suspended. Arrows represent the interactions between tasks and devices.

3. Drowsy Power Management

Drowsy aims to reduce energy consumption by minimizing the number of tasks and devices that are woken up in each wakeup cycle (i.e., *suspend* to *on* → event handling → *on* to *suspend*). We define *wake set* to be the set of all tasks and devices that are awake. A *minimal* wake set exists for each wakeup cycle, and is comprised of *only* the tasks and devices required to handle the actions performed within the cycle. In traditional OSs such as Linux and Android, once the system transitions to the *on* state, the wake set includes all OS tasks and devices. This choice of wake set is functionally correct, but is typically much larger than the minimal wake set.

Drowsy constructs a minimal wake set by identifying necessary tasks and devices for inclusion to the existing wake set, which initially consists of the device whose IRQ

triggered the wake event. This process takes place while transitioning to (and operating in) the *drowsy* state. To ensure that this set is both minimal, Drowsy must (at runtime): 1) determine the dependencies between system components, 2) preserve dependency state during the *drowsy* to *suspend* transition.

3.1 Model

To capture dependencies in the system, we propose a model that decomposes the system into tasks, resources, and devices. A task represents a single thread of execution and has an associated state, which denotes whether the task is running, blocked, or frozen. Resources are system components without a thread of execution; for example, in Linux, these include files and sockets. Devices are special resources that have an additional state which notes whether the underlying hardware device is in a suspended or resumed state.

Resources expose *interfaces* for task interaction (e.g., `open`, `close`, `read`, `write`, `ioctl`), which are system calls in user-space (or direct function calls in kernel mode). Since the OS mediates access to resources, a task must acquire a handle to the resource prior to any interaction. Typically, the handle is acquired using a variant of `open`. In the course of a task's interaction with a resource, e.g., a `read` or a `write`, the task may block until a specific "wait condition" on the resource is satisfied. Wait conditions are satisfied as a result of interactions from one or more other tasks on the shared resource. For example, a task may block `read`ing a descriptor, and can subsequently continue when another task `write`s to the resource.

Devices have an associated software driver, which implements the interface to interact with the device. A task can execute the driver code to interact with the device. Hardware devices must be resumed before their driver code is executed (otherwise the system could fault). In general, driver code may interact with other resources, which may require multiple hardware devices to be resumed.

In the process of suspending the system, the OS places tasks in a frozen state and devices in a suspended state. Drowsy determines which tasks and devices must be included in the wake set (and thus thawed or resumed) to maintain correct behavior of the system *as if all tasks and devices are included in the wake set*. To do so, Drowsy monitors interactions that allow it to infer dependencies involving tasks and devices.

3.2 Dependencies

There are two types of dependencies that Drowsy must track: when a task depends on device (denoted by T→D), and when a task depends on a wait condition (denoted by T→W). For a T→D dependency, a task depends on a device being in a resumed state while calling any of the functions in the software driver. For a T→W dependency, a task depends on a wait condition being satisfied by another task before being able to continue executing.

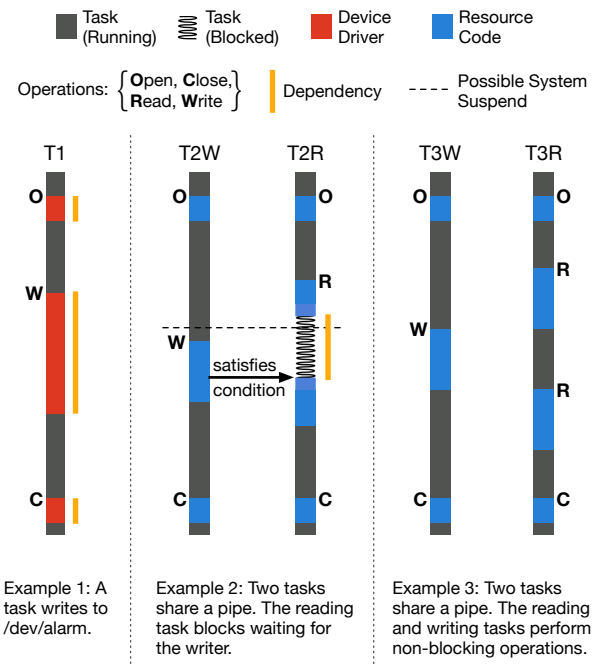

Figure 2: Examples of the types of interactions and the resulting dependencies. Each vertical segment represents a thread of execution, running code from associated with a task, device, or resource. Lines beside the task's thread of execution denote dependencies.

We describe a method of tracking dependencies and constructing the minimal wake set based on monitoring all resource accesses in the system. For a T→D dependency, a dependency exists *for the duration that the task is accessing the device*, e.g., the time during which an `open`, `read`, etc. call is being executed. For a T→W dependency, a dependency exists while the task is blocked on a wait condition, e.g., the interval during which a process waits on a descriptor after issuing a blocking `read`.

Drowsy detects a T→D dependency when a task tries to access a device. If the device is not resumed, Drowsy resumes it and adds it into the wake set; otherwise, if the device is resumed, it is already in the wake set Drowsy detects a T→W dependency when a task blocks. T→W dependencies require no immediate action; instead, when another task later satisfies the wait condition, Drowsy thaws the dependent task and adds it to the wake set.

Note that when a task is thawed, it may access (or be in the process of accessing) one or more devices. Drowsy detects these T→D dependencies and resumes the devices as necessary.

We illustrate dependency tracking in Figure 2, which contains examples based on Android. The figure shows three separate instances of interactions between tasks and a resource. The open and close operations acquire and release the resource. Example 1 shows a task opening the

`/dev/alarm` device file, writing to it, and later closing it. Here, Drowsy detects a T→D dependency, and resumes the device prior to the `open` system call invoking driver code. Further T→D dependencies are detected for subsequent system calls, but since the device is already resumed, the wake-set is not modified. (Runtime power management *may* suspend a device after it has been resumed if it is unused; in this case, the device driver would re-resume the device prior to subsequent system calls.) Finally, recall that the dependencies exist *only* for the duration of the system calls.

Example 2 shows two tasks that open and share the same named pipe, either write to or read from it, and later close it. Drowsy detects a T→W dependency when the T2R task attempts to read from the empty pipe; the task must block waiting for data to be made available by the writing task. Later, the T2W task writes data to the pipe, which satisfies the wait condition of the T2R task (and allows it to continue executing). The system may suspend and then resume at the point noted by the horizontal line: T2R was blocked and T2W was running. Since T2W was frozen from a running state, Drowsy thaws T2W during the resume. Afterwards, when T2W writes data to the pipe, Drowsy thaws T2R (and adds it to the wake set) since its wait condition is now satisfied.

Example 3 is similar to Example 2, except that the operations are now non-blocking. Task T3R performs two reads: the first fails due to an empty pipe, while the second succeeds because T3W had since written to the pipe. Because the pipe here does not have a wait condition (the reads are non-blocking), there is no T→W dependency even when the pipe is empty.

This method of tracking dependencies by individual resource accesses results in a minimal wake set by construction. Devices are added to the wake set (and resumed) if and only if a task depends on a device, and tasks are added (and thawed) if and only if another task satisfies their wait condition.

Instead of tracking individual accesses, a simpler method would be to track only resource acquisition and release (i.e., instrumenting only open and close, but not read or write). Dependencies would be declared over the entire interval between acquisition and release. This method would also be functionally correct but would lead to unnecessarily large wake sets. Tasks may acquire many resources, but use very few during any given wakeup cycle. This is particularly pronounced on Android due to commonly shared system services such as Binder [5], which is the core IPC mechanism in Android and is accessed via the `/dev/binder` device. All Java applications on Android utilize Binder, thus if one is added into the wake set then all other Java applications would need to be woken up.

3.3 State Transitions

Assume the system is in the *drowsy* state, and may suspend. In suspending the system, both Android and Drowsy freeze all tasks and suspend all devices in the wake set; however, since Drowsy operates using the minimal wake set, it performs less work. In Drowsy, devices transition to suspend under the same conditions as in Android.

Later, a device sends a wakeup IRQ to the CPU; this IRQ initiates the transition from the *suspend* to *drowsy* state. During the transition to (and while remaining in) the *drowsy* state, Drowsy constructs the minimal wake set for the current wakeup cycle. This process of adding tasks and devices to the wake set is similar to how the OS supports paging, thawing tasks and resuming devices (memory pages) on an as-needed basis.

Wakeup IRQ Upon resume, the wake set consists of a single kernel task that executes state transitions. This task re-enables IRQs. The processor receives the IRQ, which activates the IRQ handler task and resumes any devices associated with the IRQ. The IRQ handler may cause other tasks and devices to be resumed and thawed as necessary.

Previously Running Tasks To preserve correctness, Drowsy must always thaw all previously running tasks during the *suspend* to *drowsy* transition. Absent static analysis of each task's source code, Drowsy cannot predict which resource a task will interact with (which may or may not block), as shown in the example next.

```
wakelock.acquire();
while true do
    wakelock.release();
    <system may suspend here>
    client = socket.accept();
    wakelock.acquire();
    if client ≥ 0 then
        handle(client);
```

Algorithm 1: Runnable task resumed by Drowsy.

Consider the pseudocode presented in Algorithm 1. This task waits for an incoming client connection, acquires the wakelock, communicates with the client, then finally releases the wakelock. The system may suspend immediately after the task releases its wakelock, while the task is still running prior to blocking in the `accept` call. In Android, an incoming connection will wake up the entire system, allowing the task to continue executing (calling `accept`, which immediately returns). However, if Drowsy were to not thaw previously running tasks, the task would be left frozen because a dependency does not *yet* exist (as `accept` has not been called). In general, very few tasks are running at the point the system transitions to the *suspend* state; most tasks are blocked waiting on some event.

4. Drowsy on Android

In this section, we describe how we implemented Drowsy on Android. We provide an overview of the interactions be-

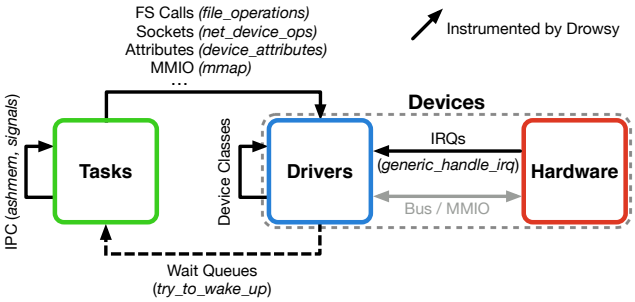

Figure 3: Overview of interactions system components with interfaces present in Android.

tween system components in Android, and identify the interfaces that Drowsy must instrument. Afterwards, we describe the new PM transitions between *suspend* and *drowsy*. Finally, we reflect on the lessons learned from our implementation of the *drowsy* state in the Android kernel.

4.1 Android Instrumentation

Drowsy tracks dependencies involving tasks and devices by instrumenting the interaction interfaces that expose such dependencies. Figure 3 shows the major interfaces, kernel structures, and calls that Drowsy instruments. Interactions between drivers to their associated hardware device do not need to be instrumented, since the hardware device is necessarily awake while driver code is executing in a task's thread of execution. We describe the interfaces identified for all Drowsy-instrumented interactions next.

Task to Driver Interactions There are a number of interfaces through which a task can interact with a device driver. Most of these interfaces depend on the type of device: block, character, or network. All of these interactions expose T→D dependencies.

For drivers that export block or character access to the device, tasks can invoke system calls on the files present in the `/dev/` directory. Additionally, block devices are accessed through their mounted partitions. Drowsy intercepts these types of calls by instrumenting the associated `file_operations` structure in the kernel. While parameters for such operations do not directly point to the underlying device, we utilize information contained in the file's inode: the type of device (character or block), and the device major/minor numbers.

Network devices are intercepted by instrumenting the `net_device_ops` kernel structure. In four cases, we found functions that could be invoked in an IRQ (or software IRQ) context. In these cases, if the associated device is not already awake, Drowsy invokes the PM routines to resume the device; however, the system may fault since the PM routines assume they are run in process context (and thus can block). In each of these cases, we identified a precursor function that executes outside of IRQ context, which we instrumented to resume the device prior to entering the software IRQ context. For example, packet transmissions culminate in a call to the `.ndo_start_xmit` function, which runs in IRQ context; Drowsy instruments the precursor function `dev_queue_xmit`, which enqueues packets prior to transmission.

Drivers and device classes can export attributes via the `sysfs` interface, e.g. `/sys/class/leds/red/brightness`. Tasks can view and modify device attributes through system calls on these files. Drowsy instruments the associated `device_attribute` or `bin_attribute` structures in the kernel to intercept these calls.

Tasks may also access devices directly through their I/O memory regions after calling `mmap` on a given device file. Typically this is done in the case of graphics devices. To handle these types of accesses, Drowsy instruments both the `mmap` syscall and the page fault handling code. Drowsy identifies the regions to track by checking when `mmap` invocations meet two conditions: the call corresponds to a device file, and the returned region is flagged for memory mapped I/O. Drowsy modifies the page protection bits for such regions to capture accesses that occur prior to the resume of the corresponding device.

Task/IRQ to Resource Interactions In Android, each kernel resource may have an associated `wait_queue` data structure in order to keep track of all tasks waiting (blocked) on some wait condition to be satisfied. These `wait_queues` expose T→W dependencies.

If a task interacts with a resource and a wait condition is not met, the task is placed on the associated `wait_queue` and unscheduled. When the wait condition is satisfied by another task, there is a call to `try_to_wake_up` in order to schedule the waiting tasks. Drowsy monitors this interaction by instrumenting the `try_to_wake_up` function, which sets tasks as runnable in the kernel (allowing them to be scheduled).

Hardware Device to Driver Interactions A hardware interrupt signal sent to the processor invokes the associated IRQ handler routine, which was registered by the driver through `request_irq`. This interaction leads to a T→D dependency.

We require drivers to additionally include the device as a registration parameter, such that Drowsy can maintain a mapping between IRQs and devices. Drowsy intercepts the lowest-level kernel interrupt trampoline routine, and ensures the device is awake before delivering the IRQ. The interrupt is postponed if the device was not already resumed, making use of existing mechanisms for handling pending interrupts.

Driver to Driver Interactions Device drivers expose parent-child relationships to the OS, which aggregate to form the device tree. There are three ways drivers interact with one another: 1) device to ancestral device, 2) through device class interfaces, and 3) direct invocation. All of these lead to T→D dependencies.

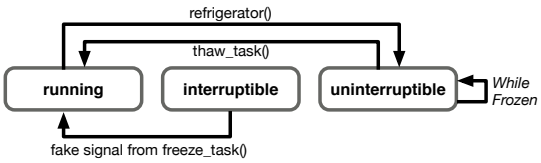

Figure 4: Task state transitions during Android suspend and resume transitions.

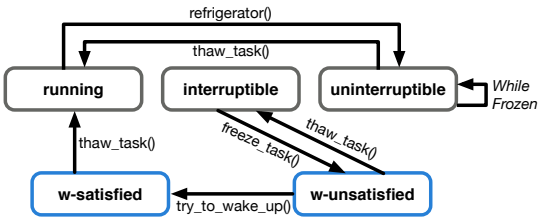

Figure 5: Task state transitions during Drowsy suspend and resume transitions; `w-unsatisfied` and `w-satisfied` are new Drowsy-only states.

Communication with ancestral devices is the most common. For example, when drivers need to communicate with their hardware device through their bus, which is an ancestral device, they do so through their bus's driver. Since device wakeups proceed from parent to child, all ancestral devices are already awake.

A device class exposes an interface associated with a general type of device (e.g., `input` class for devices that handle user inputs). Specific devices can present themselves to the OS and other drivers as a more general device by wrapping themselves in the appropriate device class. Drowsy wraps device class operations to ensure that the underlying device is awake before these calls proceed.

Direct invocation is when a driver invokes a function directly in another driver, which is possible because all driver modules are linked into the kernel image, providing no isolation. Handling these ~10 instances [2] required manually modifying drivers where appropriate.

4.2 *Drowsy* to *Suspend*

With Drowsy, the Linux *suspend* transition (as detailed in Section 2.1), is largely unaffected except for the handling of tasks. Since Drowsy leaves unused devices in a suspended state, only those devices that are currently resumed need to be handled; for these devices, suspend proceeds as in Android.

Figure 4 shows task state transitions during suspend and resume in Android. Tasks that can be scheduled are in the `running` state. Tasks that are blocked are in either the `interruptible` or `uninterruptible` state. Tasks in the `interruptible` state transition to the `running` state if its wait condition was satisfied (e.g., descriptor is `readable`) or if the task receives a signal (e.g., `SIGINT` from console). Tasks in the `uninterruptible` state are in the midst of an atomic operation in the kernel, and will not transition to the `running` state until the operation completes.

Tasks can be in any one of these three states at the start of suspend. To freeze a task, the kernel sends it a "fake" signal regardless of task state. `running` tasks context switch into kernel mode, and invoke the `refrigerator` function which sets their state as `uninterruptible`. `uninterruptible` tasks in the `refrigerator` transition state only as a result of the `thaw_task` function. Upon receiving the signal, `interruptible` tasks transition to `running` and follow the same path as previously `running` tasks. `uninterruptible` ignore the fake signal until their atomic operation completes, after which they are in the `running` (and follow the same path). Upon thaw, *all* tasks are placed in the `running` state and system calls are restarted for `interruptible` tasks.

Tasks previously in the `interruptible` state, normally the vast majority of resumed tasks, restart their system call and block again since their wait condition is likely not satisfied. Drowsy avoids this extra computation and needless device resumes by introducing two new states: `w-unsatisfied` and `w-satisfied`. The basic idea is to leave `interruptible` tasks on their `wait_queues`, and only thaw them when their wait condition is satisfied. Figure 5 shows task state transitions during suspend and resume in Drowsy. The `w-unsatisfied` state is for `interruptible` tasks whose wait condition was not satisfied prior to suspend. Upon resume, these tasks restored to the `interruptible` state. (It is not strictly necessary to transition `w-unsatisfied` tasks back to `interruptible`, but we do so due to ease of implementation.)

Interestingly, there are cases in which a wait condition is satisfied after the system starts suspending yet prior to the completion of the suspend. Here, the system should suspend since there is no active wake lock, but these tasks should transition to `running` upon resume since their wait condition is satisfied. Drowsy places these tasks in the `w-satisfied` state.

4.3 *Suspend* to *Drowsy*

Once a wakeup IRQ is generated by a hardware device, the system begins resuming as before, starting with the enabling of non-boot CPUs.

Unlike Linux PM, Drowsy avoids explicitly resuming devices during this transition, instead relying on inferred dependencies to wake devices on demand[3] . Drowsy then iterates over the set of all tasks, handling each depending on its current state. The tasks in the refrigerator, as well

[2] We found these (not necessarily exhaustive) instances through manual debugging of symptomatic device drivers.

[3] Resuming the RTC class device injects the timekeeping subsystem with the amount of time passed while the system was suspended. Therefore, the RTC (and its dependencies) must be woken up on each resume.

as any tasks in the `w-satisfied` state, are added to the wake set (and thus woken up). The tasks that remain in the `w-unsatisfied` state across the *suspend* transition are reverted back to their previous `interruptible` state. At this point the PM transition is complete. Throughout this transition, and while the system remains in the *drowsy* state, Drowsy monitors the interaction interfaces as described earlier in order to resume devices and thaw tasks as needed.

There are two ways that devices can be woken up: task- and IRQ-originated interactions. In task-originated wakeups, the interaction with the device must occur through one of the interfaces described earlier in this section. IRQ-originated wakeups cannot be interrupted, which prevents the invocation of device resume routines inline (as they *may* block). When the device associated with the IRQ is not in the wake set, Drowsy uses a separate worker thread that runs in process context to resume the device. In the process of resuming the device, the IRQ handler is re-enabled and the kernel resends the pending interrupt.

4.4 Lessons Learned

We integrated Drowsy into the Android 4.2.2 kernel for the Nexus 4 smartphone. The result of our work is a relatively small set of changes (4608 LOC), that efficiently supports Drowsy. Our version of the Drowsy-enabled Android kernel is stable, and supports all devices on a modern phone, including the cellular, WiFi, and Bluetooth networking stacks, and all sensors. As Drowsy requires no changes to user-space applications, the complete Android runtime is supported; Drowsy devices can and do run all Android applications.

Retrofitting a new run state into a fully featured, general purpose OS for a mobile platform was a major effort. Along with the usual interfaces (file system, sockets, shared memory), Drowsy had to address many undocumented interactions such as direct driver-to-driver calls. Drowsy changes a basic assumption within the system: that every task and device is available when the system is *on*. Changing this assumption revealed several bugs in drivers (e.g., conflating level- and edge-triggered IRQs) and esoteric ways in which drivers are related to each other.

Drowsy relies on the ability to wake up devices and resend an IRQ later if the IRQ is disabled when triggered (meaning the devices are still suspended). Unlike edge-based IRQs, level-based IRQs are not resent as the level is maintained by the hardware interrupt controller. In over fifty cases, device drivers incorrectly specified an IRQ to be handled as a level-based IRQ when it is in fact edge-based (and vice-versa). This caused the interrupt to be correctly masked in Drowsy, whereas these were (incorrectly) delivered in Android, causing the bug to not manifest itself. In some cases, including the I2C bus, the driver did not specify its relationship to its parent device, forcing Drowsy to fail to resume all necessary devices. Again, these latent bugs do not manifest themselves in stock Android.

I/O Event	Description
Alarm RTC (ALM)	Wakeup and set up next alarm
BT Connection (BT2)	Handle an incoming Bluetooth connection
Pull Weather (PUL)	Fetch weather update from remote server
Push Notify (PSH)	Push notification messages to the phone
Sensor (SEN)	Sample the accelerometer sensor

Table 1: I/O events used in the evaluation.

5. Evaluation

In this section, we begin by measuring the costs (in time and energy) associated with PM transitions in stock Android, and analyze the overhead of transitions relative to completing common periodic tasks. Next, we analyze the dependencies for common periodic tasks, and demonstrate that tracking dependencies only by resource acquisitions results in overly-broad wake sets. Afterwards, we investigate the overhead of Drowsy instrumentation in the kernel, and the new costs for PM transitions in/out of the *drowsy* state. Next, we analyze the time speedup and energy efficiency gains of Drowsy over stock Android for common periodic tasks. Finally, we evaluate the battery life improvements of Drowsy.

5.1 Platform

We use the Nexus 4 smartphone (released 2012) for most of the evaluation, The Nexus 4 contains a 1.5GHz quad-core CPU with 2GB of RAM, with seven different sensors and four wireless radios [2]. Drowsy is implemented on the Android 4.2.2 kernel for the Nexus 4, which is a fork of version 3.4 of the mainline Linux kernel. We also evaluate stock Android PM on two prior generations of devices: the Nexus S (2010, Android 4.1.2, last supported) and the Galaxy Nexus (2011, Android 4.2.2).

5.2 Methodology

We developed a simple Java application which continuously acquires a wakelock, performs specific I/O, and releases the wakelock (allowing the system to suspend). The I/O events we evaluated are listed in Table 1. For the SEN event, we register and unregister a `SensorEventListener` during each wakeup to avoid unnecessary energy consumption while the system is suspended. For each event, we only enable the required radios; this means that only Bluetooth is enabled for BT2 and only WiFi is enabled for PUL and PSH. Each experiment consists of 40 such iterations for each I/O event. As part of performing evaluations on a fully-working system, other applications and system services may perform periodic tasks that are separate from our benchmarks. To account for this, we discarded wakeup cycles that did not correspond to the periodic tasks run by our microbenchmarks.

To gather timing measurements, we instrument appropriate points in the kernel's PM transition handling code to log the current timestamp. For measuring energy consumption, we collect a trace of the phone's power consumption which

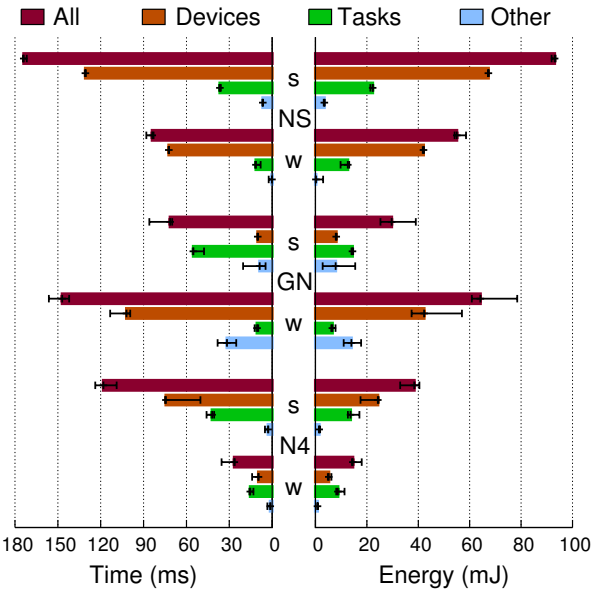

Figure 6: Breakdown of time and energy for the major components of the suspend (s) and wakeup (w) for the ALM I/O event. These measurements are performed on three devices: the Nexus S (NS), Galaxy Nexus (GN), and Nexus 4 (N4). The bars and whiskers represent the 25^{th}, 50^{th} and 75^{th} percentiles.

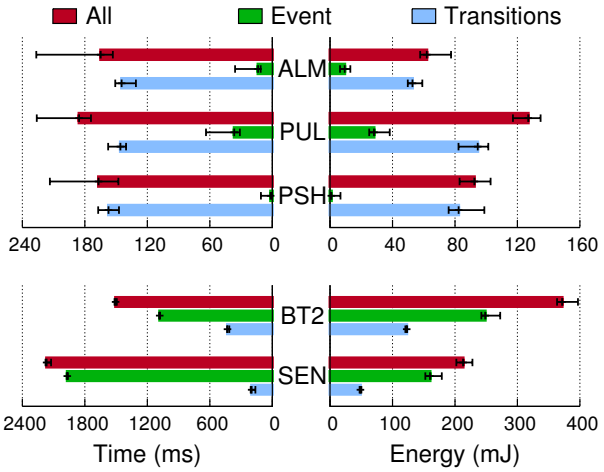

Figure 7: Breakdown of time and energy spent in PM transitions and event handling for common I/O events on the Nexus 4. The bars and whiskers represent the 25^{th}, 50^{th} and 75^{th} percentiles.

we synchronize with the timestamp log. We include more detailed information on our measurement setup in Appendix A.

5.3 Stock Android Overhead

Figure 6 shows the time and energy consumed by the PM transitions for the ALM I/O event across three generations of Nexus devices: the Nexus S (NS), Galaxy Nexus (GN), and Nexus 4 (N4). The result breaks down the time and energy measurements for handling of *tasks* and *devices* with the remainder aggregated into *other* (e.g., syncing buffered file-system data, disabling non-boot CPU cores).

Most of the time and energy involved in PM transitions is spent thawing/freezing tasks and resuming/suspending devices, accounting for over 97% of the time for the Nexus 4 and Nexus S smartphones, and 81% for the Galaxy Nexus. The Galaxy Nexus contains a dual-core processor, but it is unable to efficiently disable and enable cores. This overhead accounts for the high "Other" category for the Galaxy Nexus. While the Nexus 4 contains a quad-core processor, the new software includes appropriate hardware and driver support to enable and disable cores on-demand without as much overhead.

We use the latest device in our experiments (Nexus 4) for the rest of the results. In Figure 7, we plot the time and energy associated with the entire wakeup cycle ("All"), as well as individually for the event and the PM transitions.

Several of the short-lived events witness high variance in time (and to a lesser extent in energy), as demonstrated by the width of the whiskers representing the range of 25^{th} to 75^{th} percentile values. We attribute this high variance to `mpdecision`, a closed-source user-space application from Qualcomm that enables/disables CPU cores and modifies their operating frequency. Our logs show that `mpdecision` varies the number of active cores and the CPU frequencies during these tests, even though the work performed in each wakeup is the same. Finally, the PUL event is affected by network latency from the smartphone to the remote server, which ranged from 21ms to 97ms.

For all events except for BT2 and SEN, the median Android state transition costs are much larger than the median event costs, with push notifications being the largest at a 37x energy consumption ratio. This overhead severely limits the efficiency of periodic tasks, even when they access a remote server.

Even for BT2 and SEN, PM transitions are a significant factor, representing 32% and 23% of the total energy costs respectively. BT2 requires two wakeup cycles to handle the incoming connection; the additional wakeup cycle is a result of the Bluetooth controller informing the OS that the connection ended (even though `close` was called in the initial wakeup cycle). Note that Bluetooth Low Energy (BLE) [6] is not available in Android 4.2.2 via the Java APIs, nor does the device driver for the Nexus 4 export it as a Bluetooth HCI network device (accessible via the `socket` interface). We estimate that the more-efficient communication mechanisms present in BLE (tens-hundreds of millisecond scan times) would cause it to incur even higher transition overheads with respect to the event handling.

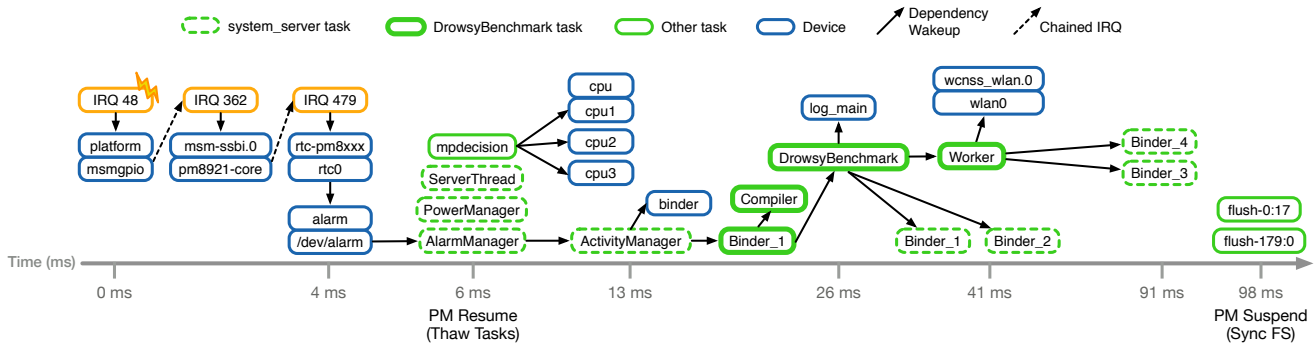

Figure 8: The dependency graph for the PUL I/O event. The directed edges represent interactions that resulted in the wakeup of a destination components.

5.4 Dependency Analysis

Figure 8 shows the dependency graph for the PUL I/O event, which is based off of the weather application example described in Section 2.2. In the dependency graph, vertices represent tasks and devices, while directed edges mean that the destination was added to the wake set as a result of an interaction by the source. We only include tasks and devices that appear in 90% or more of the wakeup cycles for a given periodic task, due to interference from other events that occur while the system is running.

This figure shows Drowsy at runtime, as it dynamically resumes tasks and devices on an as-needed basis. The Android `system_server` process provides many system services, such as `PowerManager` and `AlarmManager`. The `DrowsyBenchmark` process contains our benchmarking application for evaluating various I/O events, with `Worker` as a thread created to perform the periodic tasks. Both processes contain `Binder_*` tasks, which are used by Binder for interprocess communication. The `Compiler` task is invoked by the Dalvik VM for Just-in-Time compilation.

The I/O event wakeup is initiated by a series of IRQ events, which represent an alarm triggered by the `rtc-pm8xxx` device. Afterwards, the `rtc0` driver notifies the `alarm` driver of the IRQ, which signals a wakeup on the "alarm pending" `wait_queue` (occupied by `AlarmManager`). At this point, the PM resume routine thaws previously `running` tasks (e.g., mpdecision) or tasks in the `w-satisfied` state. Afterwards, the `AlarmManager` process broadcasts an `Intent`, which culminates in the execution of `DrowsyBenchmark` and its `Worker` task. The `Worker` then opens a HTTP connection to the remote weather server, which utilizes the WiFi radio on the device. Finally, once the `Worker` (and all other tasks/drivers) release their wakelocks, the PM suspend routine takes place. During suspend, the syncing of file system buffers involves the two separate `flush-*` threads, each handling a particular block device.

This example involved only 16 devices and 15 tasks. In contrast, the stock Android PM would have woken up 846 devices and ~800 tasks. Note that in these experiments,

	Tasks		Devices	
I/O Event	**Count**	**%**	**Count**	**%**
ALM	14	1.8	14	1.7
BT2	12	1.4	16	1.9
PUL	15	1.8	16	1.9
PSH	5	0.6	9	1.1
SEN	21	2.6	27	3.2
Power Button	135	16	77	9.1
Incoming Call	142	18	97	12

Table 2: The sizes of the minimal wake sets, broken down into tasks and devices, for various I/O events. The percentage listed is in comparison to the total number of tasks or devices registered at that time.

Drowsy and Android both run entirely on the stock factory image, with no user-installed applications. In practice, Android may have to freeze and thaw hundreds more tasks depending on the apps installed and executed by the user.

Table 2 shows the total number of tasks and devices contained in the minimal wake set constructed by Drowsy for all of the I/O events. The result also includes the percentage of tasks and devices Drowsy wakes up relative to the total number of tasks and devices registered with the OS (which is what Android PM would wake up).

For additional context, we consider two events: a user pressing the power button to wake the device, and an incoming phone call. Even though these two events involve a significantly higher fraction of components (between ~3-30x) compared to the periodic short-lived events, it is instructive to note that a majority of the components are left suspended.

Tracking Dependencies by Resource Acquisitions We also measured dependency tracking by `open` and `close`, instead of individual accesses. We examined the set of open read-write files on the system to determine what percentage of tasks would be dependent on one another. The thirteen most commonly shared resources are listed in Table 3.

The `/dev/__properties__` file is similar to the Windows registry and is used by applications on Android to save configuration information. This dependency alone would sug-

File	%	File	%
/dev/__properties__	83	/dev/cpuctl/a/tasks	69
/dev/null	83	/sys/k/d/tr/trace_marker	69
/dev/log/events	78	/dev/cpuctl/a/b/tasks	68
/dev/log/main	78	/dev/urandom	66
/dev/log/radio	78	/system/bin/app_process	66
/dev/log/system	78	/dev/alarm	40
/dev/binder	70		

Table 3: The top 13 most commonly opened read-write files and the percentage of tasks that have them open. Paths are abbreviated whenever the expansion was unambiguous to save space.

gest that a naive implementation would over-approximate the minimal wake set so completely that it hardly differs from Android at all. Even if an implementation of Drowsy selectively tracked dependencies on `/dev/__properties__` and `/dev/binder` by individual accesses, a requirement to meaningfully reduce the wake set, other devices and files present on the system represent highly connected nodes in the dependency graph. Constructing (hundreds or thousands of) special cases would likely be more difficult than our implementation based on tracking individual resource accesses.

5.5 Drowsy Instrumentation Overhead

We first evaluate the overhead imposed by Drowsy's instrumentation of Android. Checking if a device is awake requires only one "if" statement, with no locking required for the common cases (outside of IRQ-based wakeups). When tasks interact with a character or block device, Drowsy must only additionally determine if it is a storage-backed file or if it is a device file (i.e., `/dev/` files) which allows Drowsy to check if the appropriate device is awake.

We measured the time it takes to complete `read` and `write` system calls, as well as the time it takes to broadcast an `Intent`. For file-backed operations, we used `/dev/null` to read or write a page of data. For Binder, we measured the broadcast of an intent with no registered subscribers. These experiments are conservative, as they maximize the relative overhead of Drowsy because very little work is done.

Since the time taken for a single system call is on the order of our measurement granularity, $\sim 30\mu$s using the most accurate clock source (`getrawmonotonic`), we measure the time taken to complete 1,000 calls and compute the average. We run 10,000,000 total operations, which corresponds to collecting 10,000 such averages. Table 4 lists the median times taken by each operation, in addition to ranges representing the 95% confidence interval for the measurements. These results show that Drowsy's overhead from instrumenting system calls is negligible.

Drowsy Transitions Figure 9 shows the overhead of the major components of PM transitions under Drowsy. This result is analogous to the breakdown of Android PM transitions in Figure 6.

Operation	Android (ns)	Drowsy (ns)	Change (%)
Read	$53,288^{+122}_{-30}$	$53,288^{+31}_{-30}$	+0.00
Write	$56,279^{+0}_{-30}$	$56,340^{+0}_{-0}$	+0.11
Intent	$329,451^{+382}_{-411}$	$330,291^{+366}_{-458}$	+0.25

Table 4: Median time taken for operations involving syscalls with Drowsy-wrapped interfaces. The ranges corresponding to 95% confidence intervals for the median values.

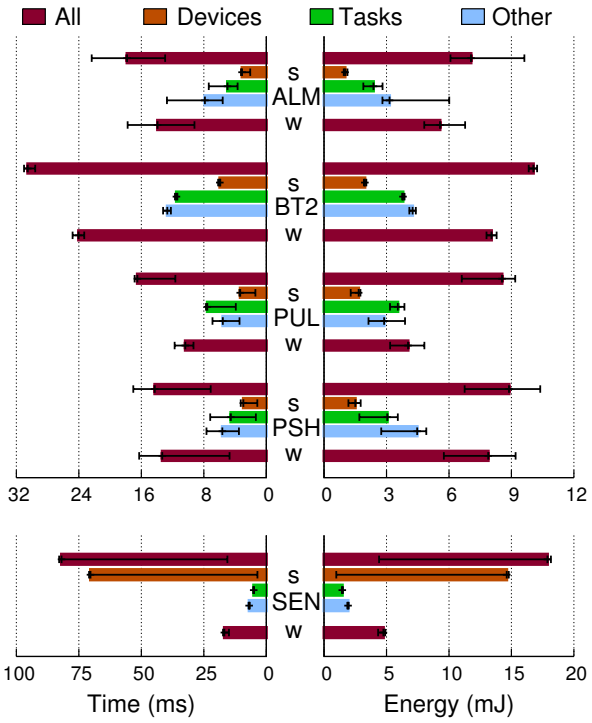

Figure 9: Energy costs associated with the major portions of the suspend (s) and wakeup (w) transitions handled by Drowsy PM, for a variety of I/O event wakeups. The bars represent the median values, while the whiskers correspond to the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile values.

Suspend transitions are factored as before, considering time and energy spent handling devices and tasks and the remaining portions aggregated in the "Other" category. However, for wakeup transitions, we are only able to report on the aggregate measurements during the PM transition into the *drowsy* state. Drowsy constructs the wake set throughout the transition to (and while remaining in) the *drowsy* state, with devices and tasks woken up in small batches (e.g., a device and its ancestors). As these batched wakeups can take time on the order of our power sampling interval (500μs for 2kHz), we cannot accurately break down the costs further, nor examine the time and energy costs beyond the initial transition. We account for these hidden costs later in this sec-

tion, when we analyze the time and energy associated with the entire wakeup (i.e., PM transitions and event handling).

Figure 9 shows that Drowsy dramatically reduced the time and energy associated with the PM transitions in comparison to Android. For instance, The ALM I/O event requires a median 118ms time and 38mJ energy for the suspend transition (see Figure 6). By instead suspending from the *drowsy* state, Drowsy achieves a 6.6x speedup in time and 5.3x increase in energy efficiency.

5.6 Drowsy vs. Android – Wakeup Cycles

Figure 10 shows the factors of improvement for Drowsy over stock Android in terms of time and energy consumed during a wakeup cycle (which includes the event handling). In addition to the stock Android and Drowsy configurations, we also evaluate three other configurations: Android+, Android-NoMPD, and Drowsy-NoMPD. Android+ incorporates non-Drowsy changes that we have implemented, preventing file system buffer synchronization from taking excessively long (due to checking for completion every 250ms) and using an exponential backoff between checks to determine if all processes are frozen instead of a fixed 10ms interval. The Android-NoMPD and Drowsy-NoMPD configurations disable the `mpdecision` service, set one CPU core active, and use the `powersave` frequency scaling governor.

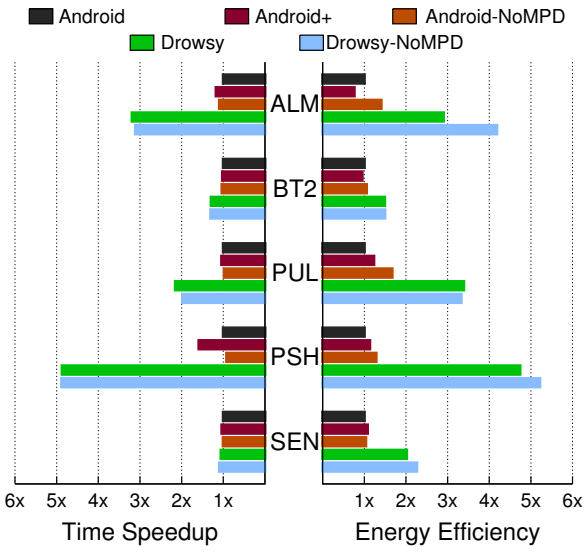

Figure 10: Comparison between Drowsy and Android for the total time and energy involved in the handling of various I/O events. The bars represent the improvements in the medians, and are normalized to the stock Android configuration.

Results show that Drowsy is at least 1.5x as energy efficient as Android. For the ALM, PSH, and PUL periodic tasks, Drowsy is 3-5x as energy efficient as Android. Even though BT2 is an outlier due to the large amount of time and energy spent in handling the Bluetooth connection, the 1.57x gain is substantial; we expect Drowsy will provide even greater energy savings for BLE. Additionally, Drowsy is over 2x as energy efficient as Android for SEN, even though the total time of the wakeup cycle does not change substantially. In this case, Drowsy actually enables more energy efficiency within the event handling itself by allowing `cpuidle` to opportunistically place the CPU in a deep idle state. We note that these are the worst-case results for Drowsy, since the devices were running no additional user-space applications beyond what the factory image includes.

Configuration	Avg. Power (mW)
None	14.34
Bluetooth	22.65
WiFi	24.52

Table 5: Average power consumed while in the *suspend* state for different radio configurations.

5.7 Drowsy vs. Android – Battery Life

In order to quantify Drowsy's improvement in battery life over Android, we first examine the daily battery life consumption when the smartphone executes only a single periodic task (from Table 1). We use an equation-based approach to observe the efficiency gains across a wide spread of timing intervals for the periodic tasks. Equation 1 represents the fraction of battery life consumed in a day, with constants and parameters as follows:

1. $P_{suspend} \equiv$ Average power consumed while in the *suspend* state.
2. T_{wc},$E_{wc} \equiv$ Average time and energy consumed during the entire wakeup cycle for handling the periodic task.
3. $E_{battery} \equiv$ Amount of energy stored in the battery. For the Nexus 4, this is equal to 28,800,000 mJ.
4. $i \equiv$ Interval in seconds between executions of the periodic task.

$$BL_{daily}(i) = \frac{86400\left(\left(1-\frac{T_{wc}}{i}\right)P_{suspend}+\frac{E_{wc}}{i}\right)}{E_{battery}} \quad (1)$$

Earlier measurements from the microbenchmarks provide us with E_{wc} and T_{wc} for each periodic task. We measured $P_{suspend}$ for a various configurations and provide the values in Table 5. The power consumed while suspended grows based on the set of enabled radio interfaces, as they perform their own periodic tasks on their hardware controllers. For example, Bluetooth in discoverable mode briefly listens for incoming requests every 1.28 seconds.

In order to validate our approach, we compare the percentage of daily battery life consumption calculated by Equation 1 to actual measurements. For each I/O event, we capture a power trace for one hour and extrapolate the battery life consumption to an entire day. For the purposes of

	I/O Event	Eq. 1 (%)	Actual (%)	Diff (±%)
Android	ALM	5.83	6.26	-6.80
	BT2	31.64	31.83	-0.63
	PSH	9.78	9.87	-0.87
	PUL	9.85	10.36	-4.87
	SEN	9.52	9.60	-0.83
Drowsy	ALM	4.94	5.11	-3.33
	BT2	29.50	29.67	-0.58
	PUL	8.67	8.47	+2.45
	PSH	8.08	8.31	-2.70
	SEN	6.27	6.76	-7.25

Table 6: Validation of calculated daily battery life consumption from Equation 1 in comparison to the actual measured (and extrapolated) consumption.

our validation, we use a 10 second event interval across the board and use the Android-NoMPD and Drowsy-NoMPD configurations to avoid additional variance. As shown in Table 6, most of the differences between computed and measured values are roughly 5%. Part of the difference is due to wakeups generated by the OS or other applications, which we do not filter in these long-running experiments. Since BT2 involves two wakeup cycles, we include the energy consumed by the Bluetooth controller between these wakeup cycles in E_{wc} (and correspondingly the time in T_{wc}).

Figure 11 shows the improvement of Drowsy over Android in terms of battery life for individual periodic tasks. As in the validation, we compare the performance of the Drowsy-NoMPD and the Android-NoMPD configurations. All lines approach the asymptote at 0% improvement as the interval between events grows, as Drowsy does not affect power consumption while the system remains in a suspended state.

Drowsy improves battery life the most when considering the SEN event, approaching near 100% improvement at a 3 second interval between events. This follows from Figure 10, which shows that Drowsy is over 2x as efficient as Android, and the fact that the SEN wakeup cycle takes 2.2 seconds to complete (close to the 3 second interval). Drowsy also improves battery life significantly for the ALM, PUL, and PSH events.

These results isolate different types of wakeups. Typically, the applications on the system will use more then one; the benefits of Drowsy will aggregate across all applications.

6. Related Work

Prior work has addressed power management efficiency across both hardware and software domains.

Hardware Current mobile systems-on-chip (SoCs) [3, 15] support aggressive processor power management in hardware, allowing smart drivers to take advantage of idle periods while in the working state [14]. Additionally, these idle states are used more often due to timer coalescing, which attempts to group running tasks together to maximize idle time. Some SoCs contain co-processor elements, which are

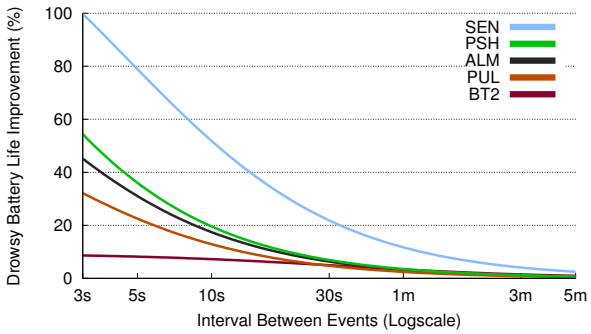

Figure 11: Drowsy battery life improvement in comparison to Android, for individual periodic tasks with intervals between 3 seconds and 5 minutes.

low-power processors used to offload tasks from the main processor [7, 17, 19]. In smartphones, they are used for continuous-sensing tasks, constantly collecting and processing sensor data. Drowsy complements these advances by improving suspend and wakeup transitions for when the entire system can sleep.

Software Independent of the system-wide PM state transitions, individual drivers can perform runtime PM while in the working state to transition to a suspended state [22]. Existing runtime PM requires all tasks and devices to be resumed at the start of each wakeup, and relies on heuristics (from the device driver, or central PM agent [24]) to enter runtime suspended states which may not trigger during short-lived wakeups. Xhu et al. [24] introduced techniques to provide automatic runtime PM for devices at the OS-level, without the need for device driver writers to explicitly interact with the runtime PM subsystem. Drowsy complements such techniques for making devices more energy efficiency, as it focuses on improving short-lived, machine-driven wakeups.

Recognizing that system-wide PM state transitions are costly, Motorola created the QuickWakeUp [9] driver to improve performance for periodic driver tasks by not fully waking up the system. Since drivers (and devices) are generally self-contained, QuickWakeUp does not require any dependency tracking. Drowsy improves upon QuickWakeUp by supporting wakeup events that involve user-space applications. Since Drowsy avoids resuming unnecessary tasks and devices in a transparent manner, essentially *all* drivers in the system get QuickWakeUp-like benefits with Drowsy.

Wright et al. [20, 21] propose to selectively resume devices based on detailed information provided by the task associated with an incoming Wake-On-LAN packet. Drowsy provides similar benefits, while also addressing task wakeup (which we showed has non-trivial overhead for smartphones in Section 5). Additionally, we demonstrate that it is feasible to provide the *drowsy* state while remaining completely transparent to user-space applications that run on disparate

hardware platforms, and requiring relatively little modification of device drivers.

Meisner et al. proposed PowerNap [12], a PM approach to reduce energy consumption in data center servers by remaining in a low-power "nap" state and transitioning to a high-performance active state when packets arrive. They postulated that if 1-10ms transition times were available, their approach would provide significant power savings based on real-world traces. They work with Dreamweaver [13], proposing a power-aware scheduling approach to coalesce idle and busy periods over multiple CPU cores. Dreamweaver relies on an available co-processor to monitor incoming network packets and to wake the system up (as appropriate) to handle queued packets. These approaches are similar to Android PM, whereby the system remains in a low-power state between wakeups generated by the hardware devices. These results, which assume fast suspend and resume, motivate broader uses for Drowsy. As part of our future work, we will port Drowsy to stock Linux kernels and evaluate gains in other environments.

Several systems [10, 11, 16, 18] interpose between applications and the sensing platform to identify which devices should be used to answer queries regarding the user's context. ACE [16], for instance, allows applications to query context attributes, using relationships between sensor data and context changes to infer which sensors should be used. Drowsy complements these approaches as it will only wake up the requested devices, instead of the entire system as in existing Android.

7. Conclusion

In this paper, we have introduced a new kernel power-management state, *drowsy*, as a replacement for the *on* power state. Drowsy tracks dependencies between system components at runtime, and dynamically resumes only those tasks and devices that are required. We describe different methods for tracking dependencies at runtime, and show that a commercial kernel (Android) can be retrofitted to support Drowsy. Our Android implementation is efficient, and improves energy consumption 1.5-5x for common short-lived tasks. This is a remarkable result, because the Drowsy kernel is fully functional, and yet significantly improves the power consumption for a highly optimized OS that is deployed on tens of millions of devices.

Our paper and evaluation mostly focuses on mobile devices, but prior work [12, 13] has noted potential benefits for faster PM transitions in data centers. As part of future work, we plan on porting Drowsy to stock Linux kernels and evaluating the energy efficiency gains in other environments.

The source code for our implementation is publicly available at:

`http://www.cs.umd.edu/projects/drowsy`

Acknowledgments

We thank Dave Levin, Peter Druschel, Pete Keleher, Neil Spring, Brandi Adams, the anonymous reviewers, and our shepherd Landon Cox for their helpful comments on the paper. We also thank Aaron Schulman, who participated in the initial discussions for this work. This work was partially supported by the United States National Science Foundation (award numbers: IIS-0964541 and CNS-1314857.)

A. Power Measurement Setup

Figure 12 shows the circuits and equipment involved in collecting a power trace for the phone. We use three digital multimeters (DMMs), which are triggered using a common function generator to obtain coherent samples. One DMM measures the voltage across the terminals of the phone (V_{Phone}). Another DMM measures the voltage (V_{Shunt}) across a precision shunt resistor ($R_{\text{Shunt}} = 0.18\Omega$); the measurements are used to compute the current I_{Phone} which is equal to $\frac{V_{\text{Shunt}}}{R_{\text{Shunt}}}$. The coherent measurements of V_{Phone} and I_{Phone} form the power trace.

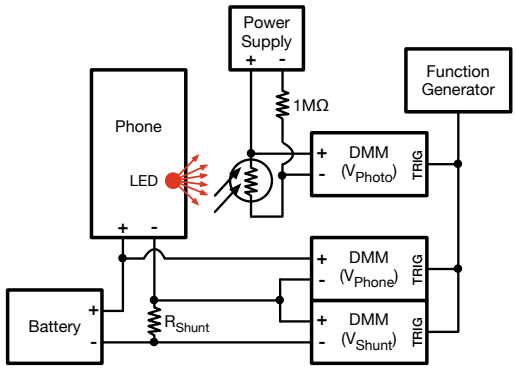

Figure 12: The circuits and equipment used to collect the phone's power trace and allow for synchronization with the software timestamp log.

To measure the energy consumption for individual portions of each wakeup cycle, we must synchronize the power trace with our log of timestamped events. This requires an externally measureable signal such that we can associate the timestamp when such a signal is generated to its appearance in the power trace. We use the notification LED with an affixed photoresistor to provide this signal, by toggling the LED on for 3ms. The toggling of the LED lowers the resistance of the photoresistor, which represents a drop in the voltage (V_{Photo}) measured by a third DMM. This must be performed during each wakeup cycle, since the OS relies on the RTC (with a timing granularity of 1 second) to account for the time passed while in a suspended state. Since we did not have a third DMM available, we instead used an oscilloscope to probe V_{Photo} as well as V_{Shunt}. We triggered waveform captures based on the drops of V_{Photo}, and aligned these captures with the power trace based on the V_{Shunt} waveform.

All of the measurements were sent back to a host computer over the LAN. We extended available scripts [1] from researchers working in the Embedded Systems Research Lab at University of Michigan, which allow for remote control of the measurement equipment.

References

[1] Measurement and Control Library written in Python to control HP, Agilent, and LeCroy devices. https://github.com/lab11/mclib.

[2] Nexus 4 Tech Specs. https://support.google.com/nexus/answer/2840740.

[3] Snapdragon S4 Processor Whitepaper. http://www.qualcomm.com/media/documents/files/snapdragon-s4-processors-system-on-chip-solutions-for-a-new-mobile-age.pdf.

[4] Advanced Configuration and Power Interface Specification. http://acpi.info/DOWNLOADS/ACPI_5_ErrataA.pdf, 2013.

[5] Binder Documentation. http://developer.android.com/reference/android/os/Binder.html, 2015.

[6] Bluetooth Low Energy. https://developer.bluetooth.org/TechnologyOverview/Pages/BLE.aspx, 2015.

[7] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[8] A. L. Brown and R. J. Wysocki. Suspend-to-RAM in Linux. In *Ottawa Linux Symposium (OLS)*, 2008.

[9] F. Jocelyn. New Feature Proposal "quickwakeup". https://patchwork.kernel.org/patch/58064/, 2009.

[10] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2008.

[11] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[12] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[13] D. Meisner and T. F. Wenisch. Dreamweaver: Architectural Support for Deep Sleep. *ACM SIGPLAN Notices*, 47(4):313–324, 2012.

[14] A. W. Min, R. Wang, J. Tsai, M. A. Ergin, and T.-Y. C. Tai. Improving Energy Efficiency for Mobile Platforms by Exploiting Low-power Sleep States. In *ACM International Conference on Computing Frontiers*, 2012.

[15] R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor, S. Thomas, B. Veera, Y. Singh, and S. Ramachandra. Experiences with Power Management Enabling on the Intel Medfield Phone. In *Ottawa Linux Symposium (OLS)*, 2012.

[16] S. Nath. ACE: Exploiting Correlation for Energy-efficient and Continuous Context Sensing. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[17] B. Priyantha, D. Lymberopoulos, and J. Liu. Little Rock: Enabling Energy Efficient Continuous Sensing on Mobile Phones. Technical report, Microsoft Research, 2010.

[18] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2009.

[19] R. Want. Always-on Considerations for Mobile Computing. In *Design Automation Conference (DAC) Workshop*, 2010.

[20] E. Wright, N. Bila, E. de Lara, and A. Goel. Effective Use of Sleep States with Context-aware Selective Resume. Pre-print draft. http://www.eecg.toronto.edu/~ashvin/publications/caesar.pdf, 2013.

[21] E. Wright, E. de Lara, and A. Goel. Vision: The Case for Context-aware Selective Resume. In *Mobile Cloud Computing and Services (MCS)*, 2011.

[22] R. J. Wysocki. Technical Background of the Android Suspend Blockers Controversy. http://lwn.net/images/pdf/suspend_blockers.pdf, 2010.

[23] R. J. Wysocki and A. Stern. Device Power Management. https://www.kernel.org/doc/Documentation/power/devices.txt, 2014.

[24] C. Xu, F. X. Lin, Y. Wang, and L. Zhong. Automated OS-level Device Runtime Power Management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.