# Reducing Power Consumption and Latency in Mobile Devices Using an Event Stream Model

STEPHEN MARZ and BRAD VANDER ZANDEN

November 18, 2015

## Abstract

Most consumer-based mobile devices use asynchronous events to awaken apps. Currently, event handling is implemented in either an application or an application framework such as Java's VM or Microsoft's .NET, and it uses a "polling loop" that periodically queries an event queue to determine if an event has occurred. These loops must awaken the process, check for an event, and then put the process back to sleep many times per second. This constant arousal prevents the CPU from being put into a deep sleep state, which increases power consumption. Additionally, the process cannot check for events while it sleeps, and this delay in handling events increases latency, which is the time that elapses between when an event occurs and when the application responds to the event. We call this model of event handling a "pull" model because it needs to query hardware devices or software queues in order to "pull" events from them. Recent advances in input devices support direct, informative interrupts to the kernel when an event occurs. This allows us to develop a much more efficient event handling model called the "Event Stream Model" (ESM). This model is a push model that allows a process to sleep as long as no event occurs, but then immediately awakens a process when an event occurs. This model eliminates the polling loop, thus eliminating latency-inducing sleep between polls and reducing unnecessary power consumption. To work properly, the ESM model must be implemented in the kernel, rather than in the application. In this paper, we describe how we implemented the ESM model in Android OS. Our results show that with the event stream model, power consumption is reduced by up to 23.8% in certain circumstances, and latency is reduced by an average of 13.6 milliseconds.

# Chapter 1

# Introduction

Mobile devices use asynchronous events, such as a finger gesture, a finger tap, or a cellphone "ring" event, to trigger a response from an appropriate application. Existing operating systems for mobile devices place the onus on the application to determine when an event has occurred. Applications currently use a *pull model* to detect events. This model requires an application to periodically poll an event queue located in kernel space to determine if an event exists. If an event does exist, the application must retrieve and decode the event before responding to the event. This periodic polling of the event queue is known as a *polling loop* and is a fairly large source of power consumption [1]. Indeed, numerous complaints have been made about mobile devices because some running applications can completely drain the battery in a relatively short amount of time [2]. This phenomenon has given rise to "app-killers", which are applications designed to close idle, power consuming applications.

The reason that polling loops drain the battery needlessly is that they must rouse the CPU several times a second to check for events, even when no events exist. Mobile devices spend a large portion of their time idling in someone's pocket or while the user reads the screen [3], and it would be helpful if the event handling system could be altered so that it does not constantly rouse the CPU when no events are present. It is important to note that if the polling loop does not check the event queue several times per second, then an application will incur an unacceptable amount of latency, which is the time that elapses between the occurrence of an event and the event being processed (see figure 1.1).

A number of recent hardware improvements are making it possible to consider alternative event handling paradigms in which the onus for initiating event-handling is transferred from the application to the kernel. These innovations include: 1) removing legacy, non-interrupting hardware devices and replacing them with more efficient, interrupt driven devices [4], 2) adding more efficient, power-saving instructions to the CPU [5], and 3) adding a low-power, shadow core to the CPU [6].

By removing legacy hardware devices, the kernel knows immediately when an event occurred and which device originated the event. For example, the uni-

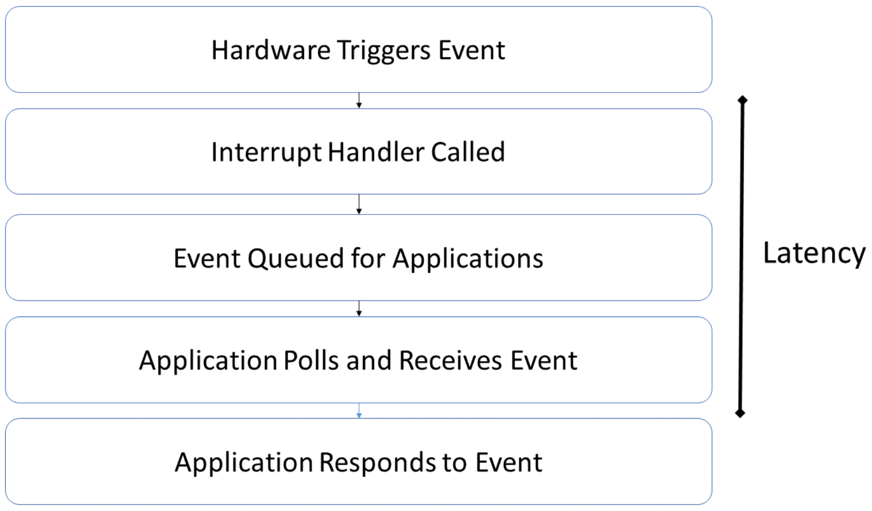
Hardware Triggers Event
Interrupt Handler Called
Event Queued for Applications
Application Polls and Receives Event
Application Responds to Event
Latency

Figure 1.1: Latency is the duration between an event notification and the start of the response to the event.

versal serial bus (USB) is replacing older serial peripherals and is now a common mainstay in the mobile world. These older peripherals required the operating system to periodically poll them since those devices have no method to push notifications to the kernel. Instead, some of these devices would send a generic interrupt to the CPU, and the CPU would then have to poll all hardware to determine which device caused the interrupt. In extreme cases, the hardware would not even interrupt the CPU and thus, would place the polling responsibility onto the operating system. This significant drawback necessitated the event pull model. In contrast, the newer devices allow the CPU to sleep until an event occurs and then to awaken once an event occurs and push the event to the appropriate application.

The introduction of power-saving CPU instructions means that if the CPU can sleep until an event occurs, then we can keep the kernel, and hence the device hardware, in a much more power efficient, deep sleep state during the long periods in which a mobile device is idle.

Finally, the introduction of lower-power, shadow cores means that event handling can be managed on these cores. The higher-power main cores can then be dedicated to applications and, therefore, can sleep while these applications are not receiving events. In contrast, if event-handling activity originates in these applications, as it does with the existing pull model, then the main cores must be constantly roused to request events, thus preventing them from entering deep sleep states for long periods of time.

In this paper, we present the design and implementation of a push-based event handling model that could be used in future mobile OSes. We call this model the Event Stream Model (ESM) and implemented and tested it in the Android kernel. It sleeps until it receives an interrupt-driven event from a device, at which point it identifies which application should receive the event and forwards the event to that application. However, this relatively simple description of the model masks many implementation issues, such as scheduling and separation of responsibilities between the kernel and the application, that will be described in the rest of the paper.

We developed our event stream model for Linux and adapted it to Android API level 21. We chose Android since it is derived from Linux, and it is the most widely used open source operating system for mobile devices [7]. This decision gave us full access to the Android and Linux source code that we then modified to implement the ESM model.

Our results show that with the event stream model, software contributed power consumption is reduced by up to 23.8% and latency is reduced by an average of 13.6 milliseconds. These power and latency numbers are produced by comparing the ESM with a polling loop which delays for 16ms, which is the event polling delay for the Android GUI system. These power saving numbers are much more pronounced with an idle application or when events are received at a slow rate.

The rest of this paper describes the implementation and testing of the ESM model and is organized as follows. Chapter 2 describes related work and chapter 3 outlines a number of deficiencies of the existing pull model. Chapter 4 presents an overview of the ESM model and chapter 5 presents its detailed implementation. Chapter 6 presents the results of our experiments that compare the power consumption and latency of the ESM and event pull models. Finally, chapter 7 presents our conclusions and ideas for future work.

# Chapter 2

# Related Work

In this section, we examine two primary areas of related work: 1) existing event handling models, and 2) hardware oriented power saving techniques that are used primarily in mobile devices.

## 2.1 Event Handling Models

### 2.1.1 Kernel-Level Event Models

In this subsection, we examine various partial push models for events that have been implemented in the kernel. However, they all have the shortcoming that an application must still pull events from the kernel using a polling loop.

Kernel objects (known as "kobjects") are used in Linux as a device package [8]. Kobjects have a notification model that devices use when they are added to or removed from the kernel. This notification occurs via a "uevent". These events can be sent to a bus from which a userspace program pulls the event. This is a type of push model that is implemented in the kernel, but it pushes to a bus rather than to an application where the event is handled. Hence, the bus is more of an aggregator than an event handler. Moreover, these events are defined by the kobjects package and are not the native events generated by the hardware devices. Therefore, the kobject notification model is not flexible enough for our purposes.

Rossi [9] outlined the issues with Linux's select() command, which is one of the two principle ways of handling events in Linux. Specifically, it takes time to detect activity on event or file descriptors, which is proportional to the size of the array of descriptors. He notes that "This increases the application latency and leads to a decrease in the overall system performance". We also note that the select() command does not eliminate continuous iterations of a polling loop. Instead, this command moves the polling loop from the application into the kernel.

Megapipe [10] and epoll [11] are software solutions that were designed to improve the efficiency of determining if data (events) exist on a socket or file descriptor, and epoll is now a principle way of handling events in Linux. Megapipe and epoll bundle many different file descriptors together, such as network sockets and event queues, so that an application only has to poll one queue rather than all of the queues individually. Although this improves the efficiency of the pull model, it does not eliminate the polling loop or the power inefficiencies inherent with the polling loop.

FreeBSD implemented KQueues, which use a push model to receive events which are then pushed into a queue [12]. Like Megapipe, KQueues can be used to queue events into a single queue so that the application can pull from one event queue rather than pull from many event queues. However, the application layer is still required to use a polling loop in order to monitor the queues and to retrieve events.

POSIX signals, such as the terminate, kill, and segmentation fault signals, implement a form of the push model [13]. If an application wants to catch a signal, it must register a function, known as a signal handler, that the kernel calls when it sends a signal to the application. However, the signal system does not pass the type of in-depth information required for handling events. Because of this limitation, we cannot reuse the signal system to push events to applications. An additional problem with using the signal system to perform event handling is that the signal system interrupts the application when a signal is received. For any event model, the application must only be pushed an event when it has indicated that it is ready to receive events. Otherwise, events could be received out of order or when the application is not ready to handle them (e.g., when it is in the middle of handling another event).

### 2.1.2 Application-Level Event Models

The application-level event models are typically located inside of virtual machines or library packages. They act as middle men between the kernel and the application. They use a pull model to obtain events from the kernel but use a push model to transmit the events to an application. In other words, as soon as they obtain an event, they immediately dispatch it to the application they are supporting. The application models we survey in this section are widely used in Linux and mobile devices and offer an insight into how they retrieve events from the operating system and propagate them to the application.

Android uses a distributed GUI library that is layered on top of their Dalvik or ART virtual machine [14]. This virtual machine is the middleman between the application and the underlying operating system. Events from the kernel are polled by the virtual machine and then stored into its own internal queues. The virtual machine then runs an event dispatch thread (EDT) that pushes the event to the actual application.

A generic event model for ubiquitous computing called the *ontology event model* was presented to diagram events and break them into atomic pieces [15]. It does not provide specific event handling algorithms for hardware and software, but instead it talks about how to package events so that software can capture every aspect of an event in an efficient way. Its emphasis on modeling events generically means that their event model is not sufficient for an end-to-end (operating system to application) event model, such as the one we are proposing, but it does provide some helpful insights on ways to package events.

High level programming languages that intrinsically support asynchronous event handling by including language support for common GUI design patterns, including the observer and callback patterns, are becoming increasingly common [16, 17]. C# goes fairly far in implementing these recommendations; however, while such native language support eases the application programmer's job, it does nothing to eliminate the polling loop.

Many researchers have been working on a real-time event handling system for Java in order to make event handling nearly instantaneous [18, 19]. Currently, Java's event handling model is similar to a push model; however, the point in time where Java executes an event handler cannot be precisely determined and is contingent upon Java's event dispatch thread [20]. Furthermore, as with Android's Dalvik and ART, the Java VM is simply a middleman that must poll for events from the kernel before finally pushing them to its client applications.

Event processing of distributed systems is also a large focus with high performance computing [21]. Cugola's distributed systems event processing model shows some interesting ways to improve event packaging by having the hardware input devices provide notifications of events and including useful information with these events. For example, a temperature monitoring device would include a timestamp and an indication of the temperature at that timestamp. However, since our goals and their goals differ dramatically, we are unable to use any of their actual proposed improvements, but we do gain useful insight on the feasibility of our ESM model.

### 2.1.3 Summary

Although many partial push models for event handling have been previously proposed, none provide a complete end-to-end push model that starts with an event originating at an input device and ends with the application consuming the event. These previous push models all place the locus of control for event handling in the application, whereas our push event stream model places the focus of control for event handling in the kernel. This transfer of control to the kernel has implications for the event scheduling algorithms that we discuss later in this paper.

## 2.2 Power Saving Software Techniques

Most mobile operating systems use certain techniques to reduce power consumption when a mobile device is idle. For example, the LCD screen is a major contributor to power consumption [22] and one common optimization is to dim and then turn off the screen when the device is left idle. This dimming and power down is performed by the OS and is called dynamic power management [23]. While this is a clever method to conserve power, it does not address wasteful power consumption while the device is running or while waiting for the idle timers to trigger the power-down state. In fact, it causes mobile device operating systems to implement aggressive sleeping policies, which is a contributor to increased event handling latency [24].

Mobile CPUs and hardware have multiple power states that determine which pieces of the entire hardware package are enabled and how much energy they are consuming [25]. Using these power states, an operating system can shut off hardware devices not being used and, hence, conserve power.

Many adaptive power algorithms that mix idle timing and power state management to further reduce power consumption have been developed for mobile devices [26]. However, these algorithms are reactive in that they do not conserve power by changing how power is being used, but rather when power is being used. Adaptive power algorithms that use reinforcement machine learning to manage the power states of the hardware are also being seen in some mobile devices [27]. However, the inputs to the multilayer artificial neural network suffer the same drawback as Shih and Wang's algorithm in that they react to when power is being used instead of how power is being used.

A dynamic, on-demand power system that allows for high-performance and power conservation when using multiple cores was developed in response to the enormous power draw required for high-performance computing. The importance of a new model was apparent since the current on-demand system incurs latency when cores are brought up and down based on CPU demand [28].

A case study was performed that investigated several different systems-level approaches to reducing power [29]. The main finding of this study was that software solutions, such as changing the network buffering layer, have real impacts on the power consumption of mobile devices.

Finally, Android linked the event polling rate to the vertical refresh rate, meaning the rate at which the graphics processing unit (GPU) redraws its framebuffer. The point was that there is no reason to handle an event while it is undetectable by the user. If a hardware vertical refresh signal is present, Android uses it to trigger an event poll. However, if the hardware vertical refresh signal is not present, Android defaults to a 16 millisecond sleep per event poll. This roughly corresponds with a 60 Hz ($16.6\overline{6}$ms) refresh rate that is common for mobile device screens. However, this event linkage only works for those events that update the display since there are many other events that must be processed but do not necessarily provide feedback to the user.

# Chapter 3

# The Pull Model Inefficiencies

As explained earlier in the paper, the pull model retrieves events from the operating system by periodically polling an event queue (see figure 3.1). Unfortunately, polling loops force an uncomfortable compromise between latency and CPU usage. If an application selects a relatively short polling delay, the application may uselessly consume the CPU when no events are present. Conversely, if the applications selects a relatively long polling delay, there may be a noticeable increase in latency. Most applications choose shorter delays that minimize latency. Several improvements have already been made to the polling loop to combat some of the inefficiencies we outline [30]. However, the inherent power consumption required to poll for events has not been eliminated.

The polling loops in the ART/Dalvik virtual machine, the Java "Hotspot" virtual machine, and the X11 display server in Linux all use operating system techniques to prevent an application from dominating the CPU. One of these techniques is to allow the operating system to block the application from running. A blocked application will not be scheduled to run until the condition causing the block is satisfied.

Unfortunately, this blocking is illusory. The Android and Linux operating systems provide two blocking methods, the *select* system call and the *epoll* system call, both of which perform their own polling loops. For example, the select system call pauses by yielding to the scheduler and moving onto the next process. However, the loop is reentered when the scheduler reschedules the select system call. Thus, using these two methods simply transfers the polling loop to the kernel. The select function allows the programmer to specify a timeout which will cause the select function to stop polling and return control back to the application. However, the select system call must still poll for events prior to the timeout expiring.

The polling loop runs at the speed at which the scheduler returns to the polling loop. While it is difficult to calculate an accurate time, the polling loop

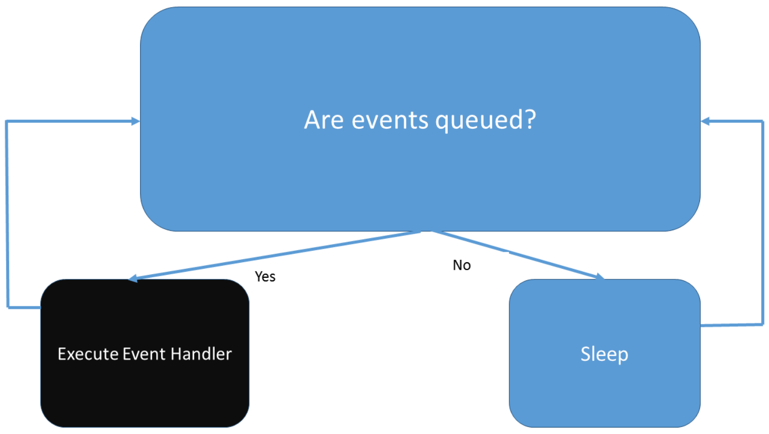

Figure 3.1: The Polling Loop: When no events exist, the polling loop delays, or sleeps in order to prevent excessive CPU usage. If an event exists, an event handler is called to respond to the event and then the event queue is checked again. If events arrive at a faster rate than the speed at which the event handler executes, the polling loop does not sleep but instead calls the next event handler.

delay is typically around 10 to 100 milliseconds when all factors are included. The large discrepancy between a requested delay and the actual delay is because the scheduler attempts to prioritize processes by the amount of processor time they have already used [31]. Thus, even if an application requests a 16ms polling delay, the actual delay could be much longer.

# Chapter 4

# Overview of the Event Stream Model

As noted earlier, the event stream model takes advantage of interrupt-driven events to change the direction of event propagation. Rather than having the application or its surrogates ask the operating system for the existence of events, the event stream model pushes them directly to the application. We call it the event stream model since it reflects the stream model used in graphics processor units (GPUs). The idea is that when data is in motion, it is more efficient to keep the data in motion until it reaches its end destination. In our event model, the data are the events.

In this section, we present an overview of the event stream model and its general improvements over the pull model. Then in the next section we present its detailed implementation.

## 4.1 General Overview

In the ESM model, input devices generate vectored interrupts that forward an event to the kernel. An interrupt routine asks the device to provide details about the event (the interrupt does not include these details), packages the information it receives into an event record and passes this record to an event interpreter. The interpreter further packages the event record into a form recognizable by an application and identifies which application, if any, should receive the event. After identifying an application waiting for the event, the event interpreter calls the event dispatcher which is responsible for running the application's event handler. The event dispatcher is only called if the application is ready to receive events. If the application is busy, the event is queued in the application's private event queue. The next two sections describe two new technologies, interrupt vectoring and shadow cores, that make the ESM model feasible.

## 4.2 Interrupt Vectoring

The ESM relies on the input hardware devices to immediately notify the kernel when an event occurs. Until recently, the ARM-based CPU has had to poll hardware devices to detect events by setting up an interrupt table that includes entries for handling a hardware interrupt and a fast hardware interrupt. Operating system programmers write large, catch-all interrupt routines for these two entries. When a CPU interrupt occurs, a generic interrupt notification tells the operating system that a hardware event occurred. The operating system must then poll each device to determine which one actually caused the interrupt.

Interrupt vectoring is a recent addition to the mobile device world even though such hardware is widely used on desktop computers. It has two virtues that are important to the ESM model. First, the interrupt controller tells the CPU which hardware device caused the interrupt by relaying a “vector number”. The interrupt handler can then directly query the device that caused the interrupt in order to find out details about the event. Second, the interrupt controller can be configured to target certain CPUs or CPU cores, such as the shadow core (discussed next), so that the other cores are left undisturbed.

## 4.3 Event Handling Shadow Core

A shadow core, companion core, or the +1 core (depending on what the manufacturer calls it) is a separate core inside of a mobile CPU that uses a fraction of the power that the main CPU cores use at the expense of the core’s processing speed [32]. Our ESM model uses the shadow core to handle events while the main CPU cores are sleeping. For example, if the device is idle and an event occurs, the shadow core will awaken and handle the interrupt (event). This has two benefits: first, the shadow core uses less power, and second, the shadow core awakens faster and goes to sleep faster than a main CPU core, which means that the initial event is handled more rapidly, even though the shadow core is slower [24]. Favoring the shadow core in these circumstances has a positive impact on reducing power consumption in mobile devices [33].

The shadow core’s decreased performance may have an impact on latency should the event stream model continually favor the shadow core even when the device is fully awake (see section 6.2). Therefore, the shadow core is only used by the event stream model when the application’s CPU is sleeping. In order to target specific cores, we use the configurable interrupt controller, such as the ARM Generic Interrupt Controller or NVIDIA’s in-house interrupt controller, to determine which CPU cores will handle events. By contrast, in the traditional pull model, the applications are running on the main cores and they, rather than the OS, initiate the polling, so the main CPUs cannot be put to sleep as quickly. This is another factor that gives rise to the ESM model’s savings in power consumption.

## 4.4 General Improvements

The main improvement our ESM provides is the elimination of all polling loops, including the polling loops present in the operating system, while the application is blocked. Since an application does not have to continuously poll to determine if an event exists, the power consumed by polling loops is eliminated.

The second improvement is that the ESM permits more deterministic dynamic power management (DPM). Rather than using sleep timers for the polling loop, applications that use the event stream model are binary: if an event occurs, the application is running, otherwise it is not. Thus, if a certain device has not been used for some period of time, the OS can confidently power it down, knowing that no application is actively using it. In the pull model, the OS cannot confidently power down a device because it must keep pinging it without incurring a large latency penalty. So, instead, the operating system resorts to very aggressive sleeping policies which could increase latency. We do note that there are some circumstances where the automatic power down is not preferred, such as while the user is reading the screen. We, therefore, are unable to eliminate all "wake locks" or software implementations that prevent the kernel from putting the hardware devices to sleep. However, even in these situations, our ESM frees the kernel from having to guess if an application is running useful instructions or only polling for events.

The third improvement is a streamlined propagation route. In the pull model, an event may end up in at least three queues–the input device queue, the OS event queue, and the event queue for the application to which the event is ultimately directed. In real world situations, there are several more queues that must be polled in the middleware frameworks, such as Java's VM or the X11 display server. Typically, the OS must use a polling loop to move an event from the input device queue to the OS's event queue and the application must use a polling loop to move the event from the OS's event queue to the application queue (see figure 4.1). In the ESM model, the event is immediately forwarded to an event handler in the kernel, which in turn immediately identifies the application to which the event should be directed (see figure 4.2). If the application is idle, it immediately handles the event, or else the event is queued onto the application's private queue. By eliminating the "hops" between queues, the ESM model can reduce the latency in handling the event.

The fourth improvement is the removal of event queue contention, since in the pull model, the kernel maintains a single event queue for each event and different applications vie for access to this queue. This contention requires locking to enforce control and exclusivity. For example, an application may lock the mouse event queue, thus preventing any other application from polling it. Currently, this is resolved by giving exclusive access to a middleware application, such as the display server, which in turn distributes the events to each GUI application. However, this prevents applications that are separate from the display server from polling the event queues locked by the display server. The ESM model eliminates these worries by removing the event-specific queues and by removing the need for elaborate locking schemes.

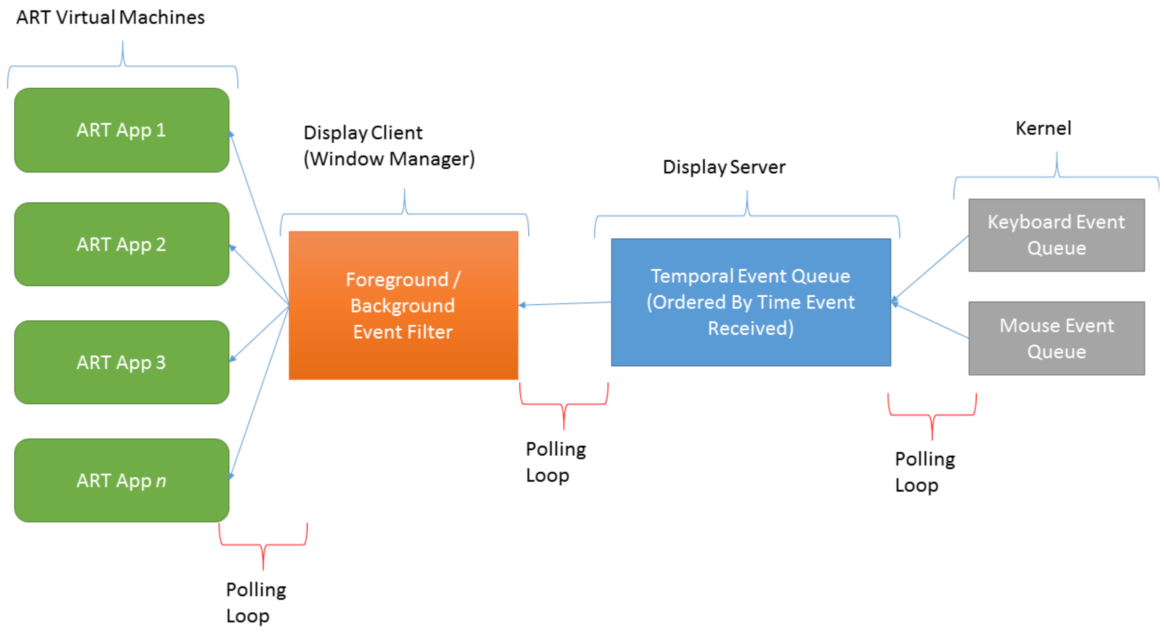

Figure 4.1: Events in the pull model are manipulated by many layers. The kernel queues initially store the events. The display server polls these queues and moves the events to its queue. The display server is required to gather events that are GUI specific, such as mouse clicks or keyboard events. Then, the event is passed to the window manager where it is examined to see if that application can receive an event (e.g. if the app is covered by another app, it cannot receive events). The event is queued if the application is a foreground application and discarded if the application is a background application. Unlike desktop machines, mobile device applications cover the screen completely, so only foreground applications can receive events. Finally the application's virtual machine polls the window manager for the events specific to that application.

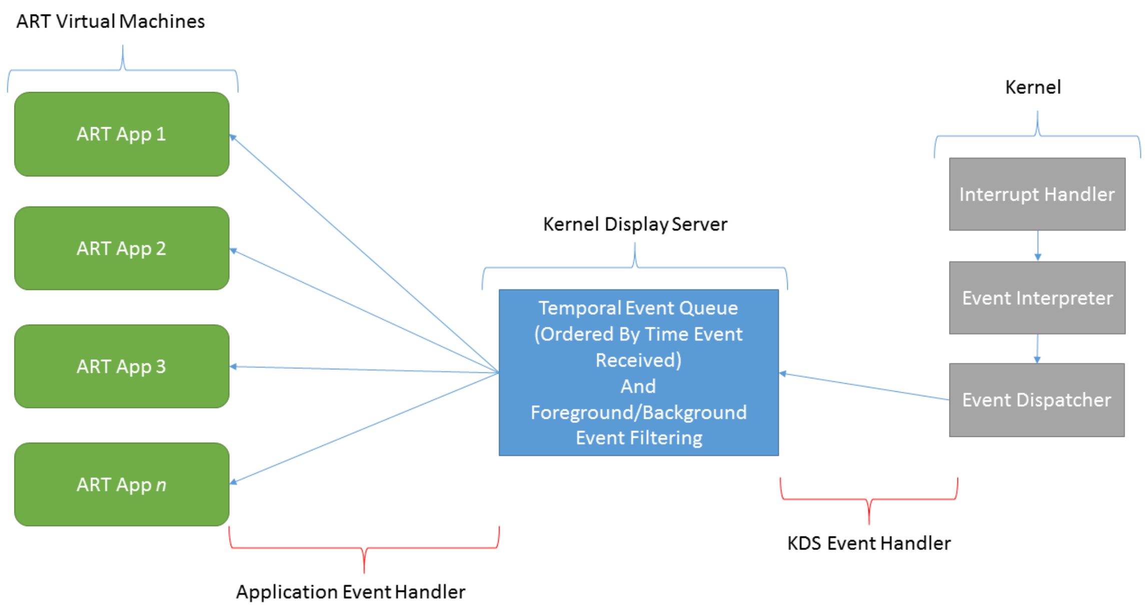

Figure 4.2: The push model is a direct event propagation model and, hence, contains no polling loops. The virtual machines register the events they are interested in, and the kernel directly pushes the event to the virtual machine through the kernel display server (KDS). The KDS takes care of filtering events for those applications that are exposed (i.e. in the foreground) and can receive events. The kernel display server is implemented in the kernel and is discussed in section 7.

The fifth improvement is the ability to use a vectored interrupt controller in order to target the low power core to handle events (see sections 4.2 and 4.3). When the mobile device is in a deep sleep, all of the CPU cores are put into low power states or even powered off completely. With the ESM, when a hardware event occurs, the interrupt controller awakens only the CPU's shadow core, which runs the interrupt routine based on the interrupt vector that the controller gave to the CPU. The main CPU cores are kept in a deep sleep while the lower power, shadow core packages the event. If no applications are listening for the event, the shadow core discards it. In this scenario, the main CPU cores are never awoken, and therefore, the power that would be required to do so is conserved.

# Chapter 5

# Implementation

This section describes the algorithms we used to implement the event stream model, and outlines several challenges we faced and our solutions to them. The implementation for the ESM is broken into two parts: a kernel part and an application part. The kernel part pushes events from the input devices to the applications' event handlers or event queues. The application part allows an application to register event handlers with the kernel and to notify the kernel that it is ready to receive events.

## 5.1 Modifying the Kernel

The ESM kernel implementation consists of three routines: 1) device interrupt handlers and drivers that initially handle the event and collect information about the event, 2) an event interpreter that packages the event information into a standardized data structure that can be handled by an application and finds those application(s) interested in the event, and 3) an event dispatcher that calls the application's event handler. Figure 5.1 demonstrates how these routines cooperate to handle a keyboard event.

Our ESM implementation is modeled after the POSIX signaling system. We could not directly use the POSIX system because POSIX signals interrupt the running process, which we do not want to do if the application is processing a pre-existing event, and they also do not pass enough information to the application. Our ESM modifies the POSIX model by: 1) notifying the application only if it is ready to receive events, and 2) packaging an event into an event structure that gives the application's event handler enough information to handle the event.

We needed to modify the kernel's data in several ways to handle the ESM model. First, we introduced a new EV_WAIT process state that tells the kernel that the application is ready to receive events. The EV_WAIT state is necessary to prevent the kernel from pushing new events to the application before the application has had a chance to finish handling the previous event. The

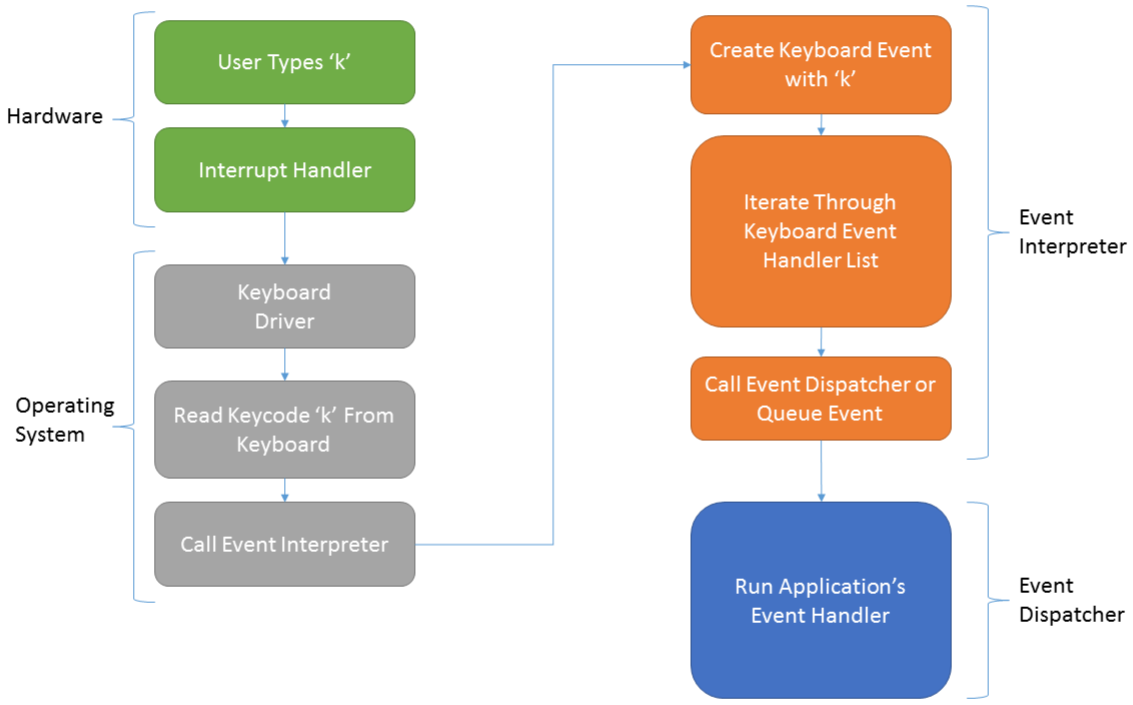
Hardware
User Types ‘k’
Interrupt Handler
Operating System
Keyboard Driver
Read Keycode ‘k’ From Keyboard
Call Event Interpreter
Create Keyboard Event with ‘k’
Iterate Through Keyboard Event Handler List
Call Event Dispatcher or Queue Event
Event Interpreter
Run Application’s Event Handler
Event Dispatcher

Figure 5.1: The path a keyboard event follows to an application using the ESM model.

addition of this state allows the application to handle events in order and to finish the processing of one event before starting the processing of the next event. The application is provided with a kernel call that sets the application's state to EV_WAIT when it is ready to receive another event (see the esm_wait call in Section 5.2). The kernel changes this state to a running state when it pushes an event to the application. Currently, there is no timeout that could be set to get the application out of the EV_WAIT state; however, an application in the EV_WAIT state is by default interruptible, meaning that an interrupting source, such as a signal, can force the process back into a running state. This prevents the application from deadlocking and allows the application to be forcibly terminated should it be "stuck" waiting for events.

The second addition to the kernel's data is an application event-handler list that associates an application's event handler with an event. For each event, the list contains a list of application event handlers that are interested in that event. This list is consulted whenever an event occurs to find which event handler(s) to call to handle that event.

Finally, each application has its own first-in, first-out (FIFO) event queue that is stored in the application's task structure. This queue ensures that events are handled in order. It is only used if an event occurs when the application is not in the EV_WAIT state. Otherwise, the event is pushed directly to the application's event handler.

### 5.1.1 Modifying the Interrupt Routines

The entry point to our ESM model is the first point where software is aware of an event, which is inside an interrupt routine (see algorithm 1). These routines are registered with the CPU so that when the hardware generates an interrupt, the CPU automatically and immediately starts to execute the interrupt routine.

One problem we faced in these routines was that we could only determine where the event came from, but not the details of the event (i.e. which key was pressed on the keyboard). Therefore, we added routines to the driver system (e.g., for the mouse and keyboard) to handle these situations. Only after events were passed through the driver system did we know what the details of the events were.

### 5.1.2 Event Interpreter

After the event is translated by either the interrupt routine or the driver, the event interpreter uses the kernel's application event handler list to determine those applications that have registered event handlers for that event. If there are no applications willing to handle the given event, then the event is silently discarded, and no further action is taken. However, if there are applications interested in the event, the event interpreter makes separate calls to the event dispatcher for each interested application. The interpreter does as little work as possible because the event interpreter is called in the interrupt state where further interrupts are disabled. It is good operating system design practice

**Algorithm 1:** The interrupt routine calls the event interpreter unless a driver is attached. If a driver is attached, the event interpreter is called after the driver adds the event's details. A raw translation from an interrupt vector (for those without a driver handler) is a table lookup and provides minimal event details.

```
Data: interrupt – the interrupt vector number
interrupt_routine(interrupt)

driver_handler = get_driver_handler(interrupt);

if driver_handler != NULL then
    event = driver_handler();
else
    event = default_event_lookup_table[interrupt];
end

esm_interpret(event);
```

to limit the amount of time the CPU is in this state and hence, the event interpreter hands off the majority of the work to be done to the event dispatcher. Pseudocode for the event interpreter is presented in algorithm 2.

**Algorithm 2:** The event interpreter is responsible for finding those applications that are waiting to receive an event. It then calls the event dispatcher to push events to the applications. The context shown here is performed while the CPU is in the interrupt state. For this reason, the dispatcher is called in a separate kernel thread.

```
Data: event – the event that was just received
Data: application_list – The kernel's application event handler list
esm_interpret(event)

foreach application in application_list[event] do
    esm_dispatch(event, application);
end
```

### 5.1.3 Event Dispatcher

The event dispatcher is responsible for running the event handler routine that an application registered with the kernel (see algorithm 3). If the application is not ready to receive events, the dispatcher queues the event onto the application's private event queue, and no further action is taken. The event dispatcher is necessary to avoid blocking conditions or deadlock conditions and to limit the amount of time the CPU is in the interrupt state where further interrupts are disabled. For example, if the event dispatcher starts to run an event handler and that handler sleeps or hangs, then all other events are blocked until the sleep or hang is resolved. Therefore, a separate event dispatcher runs for each application that has an event handler routine registered with the kernel. The dispatcher notifies the user space program, which in turn runs the event handler. This ensures that the application's memory locations and process context is fully restored before the event handler is executed. Once the event handler returns, the dispatcher automatically pushes the next queued event to the application or places the application in the EV_WAIT state if no such event exists. It does this via a call to the esm_wait function, which is described in section 5.2.

**Algorithm 3:** The event dispatcher will immediately push events to the application if it is waiting. Otherwise, the event dispatcher queues the event onto the application's private event queue. The context shown here is performed out of the interrupt state and is scheduled like any other kernel thread.

```
Data: event – the event which comes from the event interpreter
Data: application – the application to which to push the event should be
      pushed
esm_dispatch(event, application)
if application is in the foreground then
    if application.state = EV_WAIT then
        application.state = RUNNING;
        handler = application_list[event][application];
        handler(event);
        esm_wait(application); // see Algorithm 5 for esm_wait
        implementation
    else
        application.enqueue(event);
    end
end
```

## 5.2 Modifying the Application

The application must use two system calls to coordinate with the operating system, one which registers event handling routines with the kernel and one which tells the kernel that it is ready to receive events. The application may interact directly with the kernel or indirectly through a virtual machine or framework, such as Android's Dalvik or ART, or Microsoft's .NET framework (i.e., the application may use the virtual machine's existing registration methods and the virtual machine would be modified to make the registration calls to the kernel).

The application will first execute any code that is needed to display the initial graphics and establish the initial program state. During this time, it will make one or more kernel calls to register events in which it is interested and the callback procedures that should be called when these events occur. When the application finishes the initialization of its graphics and determines it is ready to receive events, it will notify the kernel. The kernel will put the application into the event waiting state (EV_WAIT) meaning that events can be pushed directly to the application's event handler.

The application interfaces with the kernel through two new system calls: esm_register and esm_wait. The esm_register function is responsible for adding the application's event handlers to the kernel's event list. This call takes two parameters, the event to register, and the event handler to register. If the application passes NULL as the event handler, the esm_register call removes the application from the event's list. Algorithm 4 presents the detailed implementation of the esm_register function.

The second system call, esm_wait, is used to tell the kernel that the application is ready to receive events. This routine prevents race conditions or potential out-of-order event handling. For example, if an application is running an event handler for the keyboard and another key event is pushed before the processing for the previous key is completed, then the event handling for the next key stroke should be delayed until the processing for the first key stroke is completed. The esm_wait function sets the application's process state to EV_WAIT to indicate that the application is ready to receive events. Any other process state signifies that the application is not ready to receive events and will cause further events to be queued on the application's private event queue. Algorithm 5 shows pseudocode for the esm_wait function.

**Algorithm 4:** esm_register links an event to an event handler. If the event handler is NULL, then the event is deregistered from the application.

```
Data: event is the event to register and event_handler is a pointer to the
      function that handles the event
Data: application_list – The kernel's application event handler list
esm_register(application, event, event_handler)

if event_handler = NULL then
    delete application_list[event][application];
else
    application_list[event][application] = event_handler;
end
```

**Algorithm 5:** If events are queued, the next event is immediately pushed to the application. Otherwise, the application is put into the EV_WAIT state until an event occurs.

```
esm_wait(application)

if application.event_queue is EMPTY then
    application.state = EV_WAIT;
else
    event = application.event_queue.pop();
    esm_dispatch(event, application);
end
```

# Chapter 6

# Testing and Results

This section presents the results we have obtained by testing our event stream model with the Android operating system, API level 21. We used the NVIDIA Tegra TK1 [6] to perform our tests because it is used by developers to create apps for mobile devices that use NVIDIA chips and because it contains all of the cutting edge features that make the ESM feasible. Our tests compared the ESM and event pull models in two areas of interest: 1) power consumption and 2) latency.

We created two applications in order to examine power consumption and latency. The first application interrupts the CPU with 2 mouse click events per second and a variable number of mouse movement events per second (customizable from 0 to 100 movements per second). Typically, the user would use finger taps and finger movements; however, these events are translated to the more traditional mouse click and mouse movement events in the Android OS. This gesture tracking application simulates a user reading (or scrolling through) text messages, Facebook posts, or electronic books. The second application is a "keyboard" like application. It simulates keyboard input (including on-screen keyboards) by varying the number of key inputs per minute. We posited that the ESM model would show less latency than the pull model and somewhat less power consumption, since at lower levels of user activity, there could still be brief idle periods.

## 6.1 Methodology

In this section we describe our methodology for measuring the power consumption and latency of applications.

### 6.1.1 Power Consumption

We tested the power consumption by the NVIDIA TK1 device with two independent methods to give tight, verifiable power consumption data. The first method is our ESM modified version of *AppScope*, which is a software based power consumption monitoring program [34] and [35]. AppScope uses performance counters and other software mechanisms to determine power consumption, thus using CPU time, performance counters, etc. as proxies for power consumption. The second method measured the battery output before and after our software tests and the difference became the power consumption [36]. We performed our tests with both the traditional pull model and with our event stream model. Both of our tests give us results in milliwatt (mW) units, thus allowing us to compare power consumption in a common, useful format.

### 6.1.2 Latency

To measure latency, we originally sought to use the operating system's process accounting system, but determined that our results were influenced by the scheduler. The scheduler's influence is intrinsic for both process scheduling and I/O scheduling inside of the kernel [37]. The default scheduler is called the Completely Fair Scheduler (CFS) which attempts to give an equal share of CPU time to each process [38]. Therefore, our testing process could be starved of CPU time if the scheduler determines the testing process has consumed more than its fair share of CPU. Applications using the pull model are particularly likely to be starved of CPU time because their polling loop consumes CPU time, and as a result they may be scheduled less frequently than the polling loop desires. For example, with a polling loop of 16ms, we would not expect the latency to exceed 16ms, but it can because of scheduler-induced latency. Scheduler-induced latency for the ESM model is mitigated because its applications are not penalized since they do not have a polling loop. In order to accurately measure total latency, we needed to capture both scheduler-induced latency and the latency that would be recorded by the process accounting system. Hence, we used a wall clock timer that records when an event is received by an input device and when the event is finally handled by the application.

More specifically, our testing platform used two high resolution timers (HRTs) that are built into the NVIDIA K1, ARM-based CPU. We set these timers to measure time within a one (1) millisecond resolution. For our purposes, this provided us with a significant precision to obtain meaningful results. The first timer was set to a fixed 1 kHZ (1,000 HZ) rate and was used to provide a wall clock timer. This timer operated by automatically increasing its internal counting register by 1 for every $\frac{1}{1,000}$ seconds, which gave us the one (1) millisecond precision. The second timer was an event timer and was used to interrupt the CPU and simulate an actual event. When the event occurred, the wall clock timer's counter register was recorded. Then, when the event handler began executing, the wall clock timer's counter register was also recorded. The difference between the two recordings gave us our latency reading. It should be noted that while we do artificially send events by using a timer, the application's response to the event is fully genuine.

## 6.2 Results

This section presents the results of our experiments using the testing applications we described above.

### 6.2.1 Power Consumption Results

Figure 6.1 shows the empirically measured CPU time that is required to service polling loops of various lengths for an idle application running on a single core for 120 seconds. It also shows the empiricially measured CPU time that is required by an idle application running the ESM model. As we posited, the ESM model consumes no CPU, and hence no power, whereas the pull model consumes increasing amounts of CPU, and hence power, as the length of the polling loop decreases. At the normal length of an Android polling loop, 16ms, an idle application consumes 12.14 seconds of CPU time, or roughly 10% of the CPU.

Our second set of tests involved simulations of user mouse and keyboard activity in applications. The AppScope simulation of battery consumption (figure 6.2.1) and the empirical measurement of battery consumption (figure 6.2.1) are remarkably similar. They show that power consumption with the pull model changes moderately between slowly received events and quickly received events. We attribute this to the fact that when the events are occurring infrequently, the polling loop itself consumes CPU time and hence, much of the power. As events occur more frequently, the polling loop no longer consumes the most power, but rather the event handling itself consumes the most power. The results also show that when events are relatively infrequent, the ESM shows its greatest benefit, with a 23% improvement in power consumption over the pull model. The power consumption numbers for the pull and ESM models start to converge as events occur more frequently, which makes sense since the application's event handlers will be almost continuously active for both models.

Finally, the results show that as the polling loop delay increases, the power consumption numbers for the pull model significantly decrease. However, the tradeoff is that the latency also increases (see section 6.2.2). One reason for the decreased power consumption is because of the power spike associated with "spinning up" the software layers, the CPU, and the kernel from a sleeping state. In the pull model, these power spikes are mitigated with a longer polling loop delay since the power spike averages over lower power consumption numbers. On the other hand, quickly occurring events reduce power consumption associated with event polling, since both the polling loop and ESM can quickly handle the events without having to delay or put the device to sleep. This confirms our main hypothesis about the event stream model, which is that power consumption can be reduced by leaving events in motion (i.e. streaming). We can also conclude that the pull model's multiple event queues and polling loop stages have a large influence on both latency and power consumption.

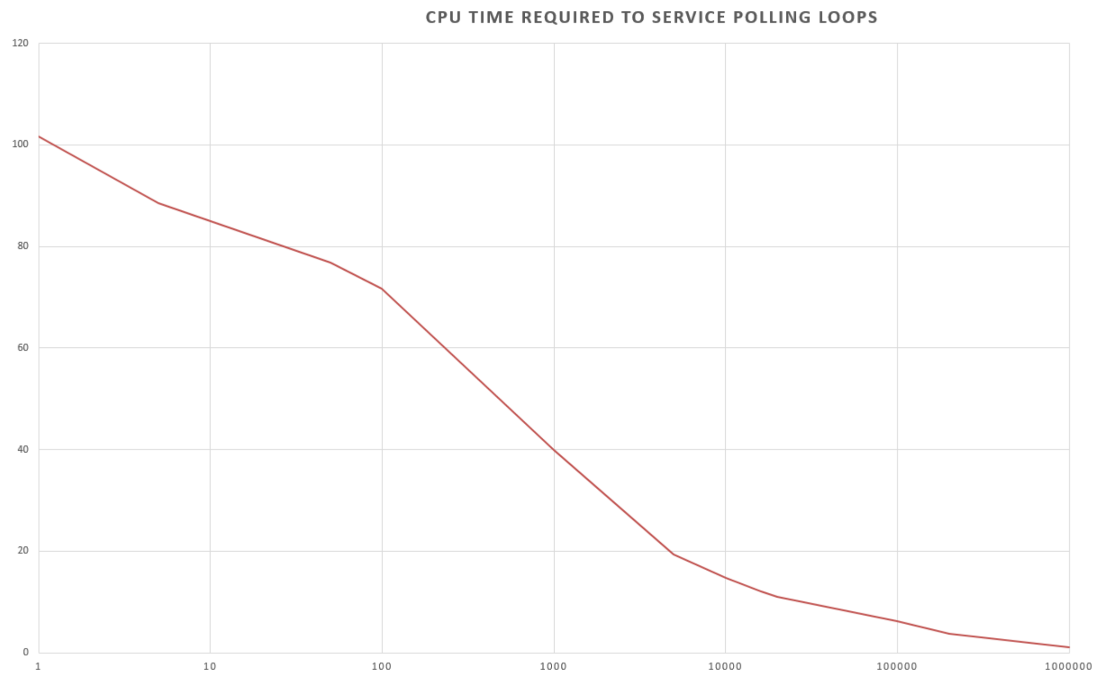

Figure 6.1: A log-scale representation of the amount of CPU time (Y axis in seconds) that is required to service the polling loop on a single core for 120 seconds with a varying amount of delay (X axis annotated in microseconds [$\mu$s]). The application using a push model, such as the ESM, is never context switched to, so it does not require the CPU to check the event queue when no events exist. In contrast, the pull model requires the OS to execute a polling loop that consumes varying amounts of CPU time depending on its sleep delay. We added a 16,000 $\mu s$ (16 milliseconds) sleep delay to our results in order to show the event polling loop in an Android-powered phone.

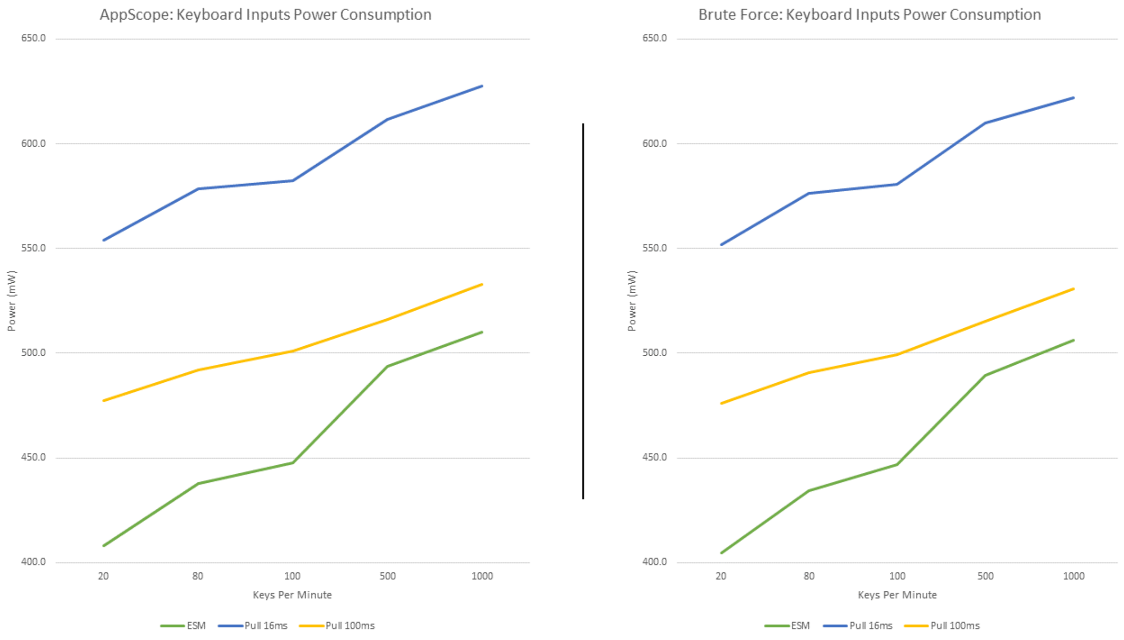

Figure 6.2: The keyboard inputs are by keys per minute. Roughly translated, 1000 keys per minute are 17 keys per second. This graph shows that the ESM model continually outperforms all pull models.

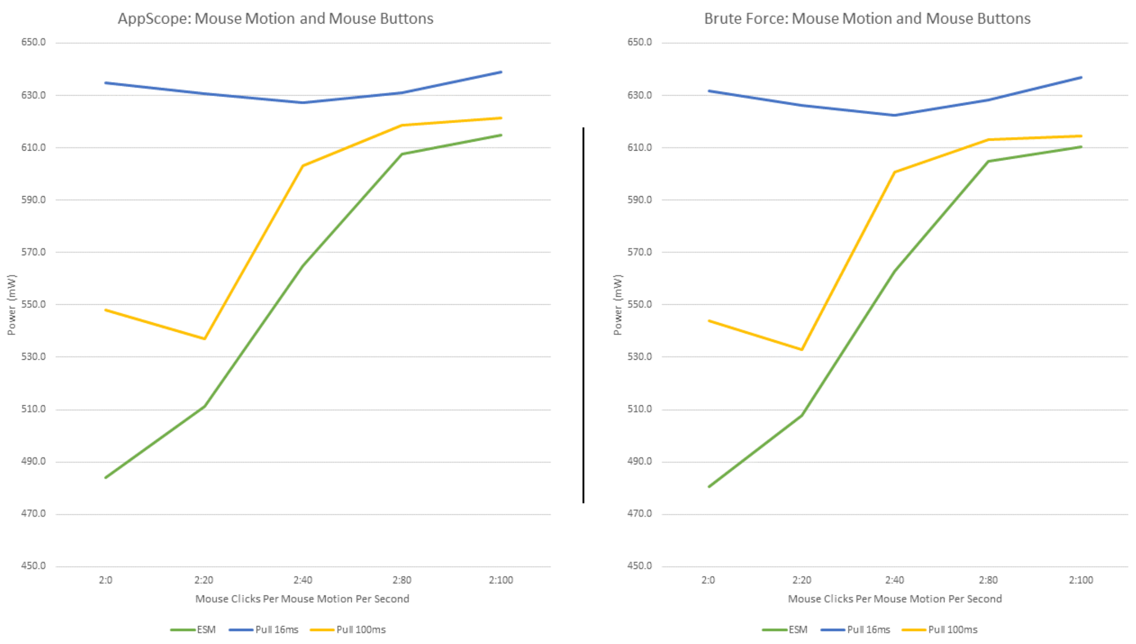

Figure 6.3: The X-axis shows the number of mouse inputs per second (2 mouse clicks plus some number of mouse moves). Initially more events cause a slight dip in power consumption for the polling loops since the polling loops are not constantly executing. Then, as the events occur more rapidly, the CPU is constantly occupied, thus causing the numbers to converge for both the ESM model and the two variations of the pull model.

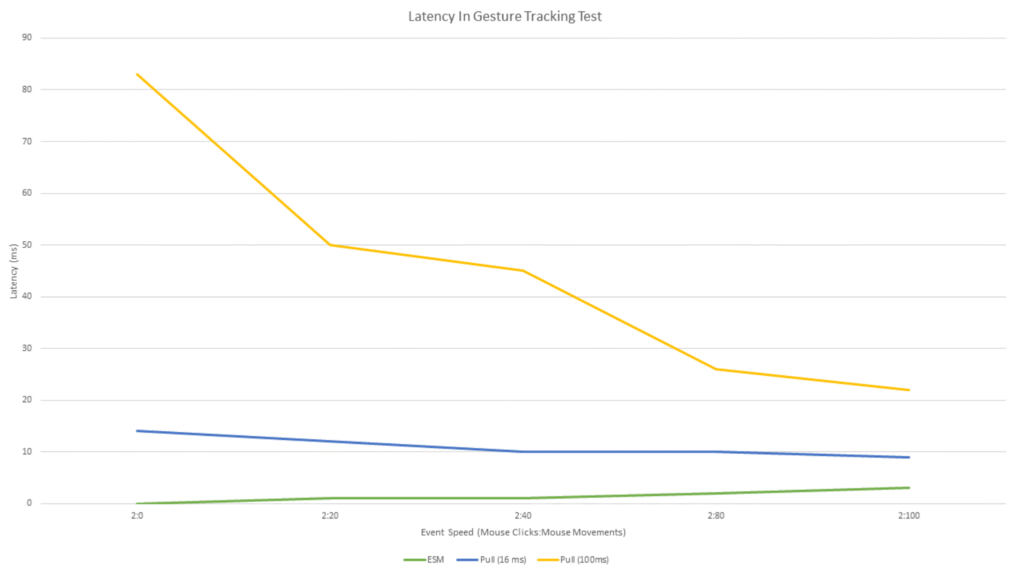

Figure 6.4: The gesture tracking latency test was performed 10 times, and the results were averaged. This graph shows that the pull model's loop delay dominates the latency numbers when events are slowly being received. The numbers converge as events occur more rapidly since the likelihood of a polling loop delay diminishes.

### 6.2.2 Latency Results

For the gesture tracking application, the ESM model reduces latency by an average 9.6ms versus the most commonly used 16ms polling loop (see figure 6.4). The greatest reduction, 14ms, occurs when only gesture clicks are received with no motion events, but even when 100 motion events are received per second, the ESM model still achieves a 6ms reduction in latency. The figure also shows that as the length of the polling loop increases, the latency associated with the pull model increases markedly. Thus, a longer polling loop can decrease power consumption, but at an increase in latency that most users will find unacceptable.

For the keyboard input application, the ESM model reduces latency by an average of 13.6ms versus the most commonly used 16ms polling loop. Users cannot type as quickly as they can swipe a finger across a screen and hence the keyboard inputs are triggered at a far slower rate than the gesture tracking application. Thus the likelihood that an event would occur while the polling loop is sleeping is far greater than when events are quickly being received. This increases the latency since an application cannot poll for events while the polling loop is sleeping.

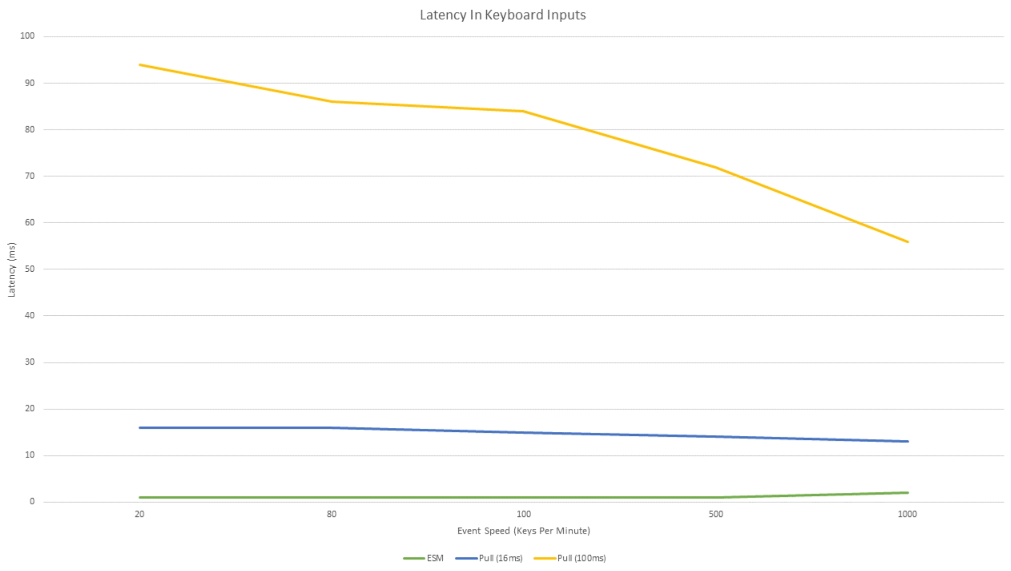

Figure 6.5: The keyboard latency test was performed 10 times, and the results were averaged. This graph shows that the pull model's loop delay dominates the latency numbers when there are few events per second. The push model shows the biggest jump in latency as the event frequency increases and the event handlers cannot completely finish an event before the next one arrives. In contrast the pull model shows a decrease in latency because events may be queued and hence no polling loop delay occurs as these events get processed.

# Chapter 7

# Conclusion and Future Work

Mobile operating systems employ a myriad of techniques to ensure that the battery isn't drained by an idle device. For example, an operating system will record the last user input to determine when to shut down hardware that is not being used. However, before the hardware can be shut down, the polling loops used by the existing event pull model will consume CPU time. Additionally, if the user sporadically uses the device, the timer is reset before the device can enter a deep sleep, and the polling loops continue to consume battery power. Our event stream model (ESM) eliminates the polling loops used by the event pull model and hence, reduces the device's power consumption while the device is idle, but before it enters a deep sleep state. Additionally, if the device is employed intermittently, the OS may be able to place the device in a lighter sleep state that will consume less power. This is not possible with the existing pull model because the polling loops keep rousing the CPU.

The components required for the ESM model, such as the power saving CPU instructions, the vectored interrupts that push events to the kernel, and the shadow cores, are becoming increasingly available on mobile devices, and hence, the ESM model presents an opportunity for the designers of mobile OSes to improve power consumption by moving event handling into the kernel.

The ESM model may require several power-saving techniques currently implemented in software to be tweaked or redone. With our event stream model, most of the current wake locks that prevent the device from sleeping are not needed, and many "no-sleep" bugs that Vekris et. al [39] have identified are eliminated. Some proactive applications (i.e. those that continually display updated information, such as a stock ticker, etc.) will still require these wake locks in order to stop the device from going to sleep. However, even with these applications, our ESM could be used to save power without resorting to a more aggressive sleep policy [40].

To further improve the efficiency of the ESM, we are currently developing a kernel display server (KDS) to be used in tandem with the ESM that we hope will allow for better dynamic power management in terms of the LCD and other hardware devices. Display servers are traditionally implemented in an application framework or GUI library rather than in the kernel, which forces the display server to coordinate event passing to and from the kernel. This combination should really start reducing the power consumption numbers since it will tackle the much larger power-hungry devices, such as the LCD, wi-fi, and GSM.

Our ESM/KDS combination would take advantage of the fact that GUIs on mobile devices behave differently than GUIs on desktop devices. Specifically on desktop devices, a foreground window (the top-most window) may not completely cover a background window associated with another application. Therefore, the display system may need to deal with repaints from both the foreground and background applications. With mobile devices, this is not a concern since foreground applications completely cover the screen, and therefore, we can treat applications that are in the background completely differently than those in the foreground. With the KDS system, the drawing thread will be suspended when the application is in the background, thus removing the need to calculate complete or partial redraws. This complements the ESM in that events can also be suspended depending on the viewable state of the application. For example, if an application is in the background, it cannot receive finger tapping or mouse events. In the existing pull model, where event handling and display management is handled in the application or an application framework, the OS does not have sufficient control or knowledge to be able to suspend background applications, and therefore, it is up to the application framework's implementation to implement sufficient power saving techniques. Hence, these applications still poll for events through the display server (or the application's framework), even though they cannot handle them. This constant polling further increases power consumption as we detail in section 3. If we move the display server and the event model into the kernel and categorically schedule GUI traffic separately, we could suspend these background applications and better manage other sources of power consumption, such as the display, wi-fi, and GSM. For example, the OS might be able to shut down wi-fi if the foreground application does not require it without having to resort to aggressive sleep timers and policies. Thus, mobile devices could see a significant power consumption reduction and a more fluid sleep policy by using to our ESM model.

# Bibliography

[1] I. Alagöz, C. Löffler, V. Schneider, and R. German, "Simulating the energy management on smartphones using hybrid modeling techniques," in *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance - 17th International GI/ITG Conference, MMB & DFT 2014, Bamberg, Germany, March 17-19, 2014. Proceedings*, pp. 207–224, 2014.

[2] B. Stewart, "Is your Android app getting enough sleep?," *O'Reilly*, August 2011.

[3] G. Öquist and K. Lundin, "Eye movement study of reading text on a mobile phone using paging, scrolling, leading, and rsvp," in *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia*, MUM '07, (New York, NY, USA), pp. 176–183, ACM, 2007.

[4] ARM, *ARM Generic Interrupt Controller*, 2.0 ed., 2013.

[5] M. Bhadauria and S. A. McKee, "Optimizing thread throughput for multithreaded workloads on memory constrained cmps," in *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, (New York, NY, USA), pp. 119–128, ACM, 2008.

[6] NVIDIA. `http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html`, 2015.

[7] B. Holtsnider and B. D. Jaffe, *IT Manager's Handbook, Third Edition: Getting Your New Job Done.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2012.

[8] G. Kroah-Hartman, *Everything you never wanted to know about kobjects, ksets, and ktypes*, 2007.

[9] F. Rossi, "An event mechanism for linux," *Linux J.*, vol. 2003, pp. 7–, July 2003.

[10] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, (Hollywood, CA), pp. 135–148, USENIX, 2012.

[11] R. Strebelow and C. Prehofer, “Analysis of event processing design patterns and their performance dependency on i/o notification mechanisms,” in *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, MSEPT’12, (Berlin, Heidelberg), pp. 54–65, Springer-Verlag, 2012.

[12] J. Lemon, *KQUEUE*. FreeBSD System Calls Manual, 2013.

[13] M. Kerrisk, *POSIX Signals Manual*. Linux Documentation Project, 3.82 ed., March 2015.

[14] A. Singh, “Google introduces ART(android runtime) in KitKat,” 2014.

[15] H. Park, Y. Lee, B. Noh, and H. Lee, “Design of an event system adopting ontology-based event model for ubiquitous environment,” in *Proceedings of the 5th International Conference on Soft Computing As Transdisciplinary Science and Technology*, CSTST ’08, (New York, NY, USA), pp. 620–626, ACM, 2008.

[16] C. R. Gimenes das Neves, E. M. Guerra, and C. T. Fernandes, “Language support for asynchronous event handling in the invocation call stack,” in *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, (New York, NY, USA), pp. 177–180, ACM, 2011.

[17] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: Safe asynchronous event-driven programming,” *SIGPLAN Not.*, vol. 48, pp. 321–332, June 2013.

[18] M. Kim and A. Wellings, “Asynchronous event handling in the real-time specification for java,” in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES ’07, (New York, NY, USA), pp. 3–12, ACM, 2007.

[19] M. Kim and A. Wellings, “Efficient asynchronous event handling in the real-time specification for java,” *ACM Trans. Embed. Comput. Syst.*, vol. 10, pp. 5:1–5:34, Aug. 2010.

[20] J. Chen, “Formal modelling of java gui event handling,” in *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM ’02, (London, UK, UK), pp. 359–370, Springer-Verlag, 2002.

[21] G. Cugola, A. Margara, M. Pezzè, and M. Pradella, “Efficient analysis of event processing applications,” in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS ’15, (New York, NY, USA), pp. 10–21, ACM, 2015.

[22] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *USENIX Annual Technical Conference*, (Boston, MA, USA), pp. 271–284, jun 2010.

[23] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "Marte profile extension for modeling dynamic power management of embedded systems," *J. Syst. Archit.*, vol. 58, pp. 209–219, Apr. 2012.

[24] W. L. Bircher and L. John, "Predictive power management for multi-core processors," in *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, (Berlin, Heidelberg), pp. 243–255, Springer-Verlag, 2012.

[25] S. Irani, S. Shukla, and R. Gupta, "Online strategies for dynamic power management in systems with multiple power-saving states," *ACM Trans. Embed. Comput. Syst.*, vol. 2, pp. 325–346, Aug. 2003.

[26] H.-C. Shih and K. Wang, "An adaptive hybrid dynamic power management algorithm for mobile devices," *Comput. Netw.*, vol. 56, pp. 548–565, Feb. 2012.

[27] U. A. Khan and B. Rinner, "Online learning of timeout policies for dynamic power management," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 96:1–96:25, Mar. 2014.

[28] A. Shayesteh, G. Reinman, N. Jouppi, T. Sherwood, and S. Sair, "Improving the performance and power efficiency of shared helpers in cmps," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, (New York, NY, USA), pp. 345–356, ACM, 2006.

[29] T. L. Martin, D. P. Siewiorek, A. Smailagic, M. Bosworth, M. Ettus, and J. Warren, "A case study of a system-level approach to power-aware computing," *ACM Trans. Embed. Comput. Syst.*, vol. 2, pp. 255–276, Aug. 2003.

[30] Q. Zhan, W. Zhao, Y. shao, J. Zhuang, and Y. Chen, "A novel multi-task software architecture applied in the intelligent insulin injector - an improved polling loop," in *Proceedings of the 2011 First International Workshop on Complexity and Data Mining*, IWCDM '11, (Washington, DC, USA), pp. 5–9, IEEE Computer Society, 2011.

[31] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 184, pp. 82–83, August 2009.

[32] A. Gomez, C. Pinto, A. Bartolini, D. Rossi, L. Benini, H. Fatemi, and J. P. de Gyvez, "Reducing energy consumption in microcontroller-based platforms with low design margin co-processors," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, (San Jose, CA, USA), pp. 269–272, EDA Consortium, 2015.

[33] R. Rodrigues, I. Koren, and S. Kundu, "Performance and power benefits of sharing execution units between a high performance core and a low power core," in *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, VLSID '14, (Washington, DC, USA), pp. 204–209, IEEE Computer Society, 2014.

[34] C. Yoon, D. Kim, W. Jung, C. Kang, and H. 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)*, (Boston, MA), pp. 387–400, USENIX, 2012.

[35] X. Chen, Y. Chen, M. Dong, and C. Zhang, "Demystifying energy usage in smartphones," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, (New York, NY, USA), pp. 70:1–70:5, ACM, 2014.

[36] C. Wang, F. Yan, Y. Guo, and X. Chen, "Power estimation for mobile applications with profile-driven battery traces," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, (Piscataway, NJ, USA), pp. 120–125, IEEE Press, 2013.

[37] K. Salah, A. Manea, S. Zeadally, and J. M. Alcaraz Calero, "On linux starvation of cpu-bound processes in the presence of network i/o," *Comput. Electr. Eng.*, vol. 37, pp. 1090–1105, Nov. 2011.

[38] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the linux scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 34–43, July 2008.

[39] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs," in *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2012.

[40] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff, "Hypnos: Understanding and treating sleep conflicts in smartphones," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 253–266, ACM, 2013.