8-2016

Reducing Power Consumption and Latency in Mobile Devices using a Push Event Stream Model, Kernel Display Server, and GUI Scheduler

Stephen Gregory Marz
University of Tennessee, Knoxville, stephen.marz@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Stephen Gregory Marz entitled "Reducing Power Consumption and Latency in Mobile Devices using a Push Event Stream Model, Kernel Display Server, and GUI Scheduler." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Brad T. Vander Zanden, Major Professor

We have read this dissertation and recommend its acceptance:

Wei Gao, Jian Huang, Peiling Wang

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Reducing Power Consumption and Latency in Mobile Devices using a Push Event Stream Model, Kernel Display Server, and GUI Scheduler

A Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Stephen Gregory Marz
August 2016

Acknowledgements

I would like to thank my doctoral committee members, Dr. Wei Gao, Dr. Jiang Huang, and Dr. Peiling Wang for their valuable support and guidance for my research and subsequent degree. Dr. Qing "Charles" Cao initially served as a committee member and provided helpful advice at the outset of my dissertation. I would like to add a special thanks to Dr. Brad Vander Zanden for serving as my advisor, editor, and coach throughout this entire doctoral experience. Finally, I would like to thank my family and my friends who have supported me.

Abstract

The power consumed by mobile devices can be dramatically reduced by improving how mobile operating systems handle events and display management. Currently, mobile operating systems use a pull model that employs a polling loop to constantly ask the operating system if an event exists. This constant querying prevents the CPU from entering a deep sleep, which unnecessarily consumes power.

We have improved this process by switching to a push model which we refer to as the event stream model (ESM). This model leverages modern device interrupt controllers which automatically notify an application when events occur, thus removing the need to constantly rouse the CPU in order to poll for events. Since the CPU rests while no events are occurring, power consumption is reduced. Furthermore, an application is immediately notified when an event occurs, as opposed to waiting for a polling loop to recognize when an event has occurred. This immediate notification reduces latency, which is the elapsed time between the occurrence of an event and the beginning of its processing by an application.

We further improved the benefits of the ESM by moving the display server, a central piece of the graphical user interface (GUI), into the kernel. Existing display servers duplicate some of the kernel code. They contain important information about an application that can assist the kernel with scheduling, such as whether the application is visible and able to receive events. However, they do not share such information with the kernel. Our new kernel-level display server (KDS) interacts directly with the process scheduler to determine when applications are allowed to use the CPU. For example, when an application is idle and not visible on the screen, the KDS prevents that application from using the CPU, thus conserving power. These combined improvements have reduced power consumption by up to 31.2% and latency by up to 17.1 milliseconds in our experimental applications. This improvement in power consumption roughly increases battery life by one to four hours when the device is being actively used or fifty to three-hundred hours when the device is idle.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile devices are an important part of the everyday lives of millions of users and are increasingly supplanting desktop computers as the primary method for accessing online material (Frias-Martinez & Virseda, 2012; Maurer, Hausen, Luca, & Hussmann, 2010). However, unlike desktop machines, mobile devices have a limited power supply that if not properly managed, can leave a user with an unusable device late in the day.

Many mobile operating systems are derived from an existing desktop operating system and thus inherit many of the disadvantages of desktop operating systems (Searls, 2010). For example, many desktop operating systems are designed to run on a wide array of different hardware configurations. As such, desktop operating systems are designed for a generic machine and are therefore unable, or at least unwilling, to exploit an advantage one machine might have over another (Sungjoo Yoo & Jerraya, 2003). Moreover, Raman and Chakraborty (2008) find that a solution which fits a desktop machine may not fit a mobile device no matter how cleverly it is designed.

One such solution that works poorly on mobile devices because of its power consuming features is the event handling system present in most desktop and mobile operating systems. Nearly all operating systems use some set of system calls to communicate events between the kernel and the application (H.-c. Lee, Kim, & Yi, 2011). With very few exceptions, this communication occurs in one direction: the application initiates the request for events and the kernel responds. The application does so by polling the operating system, and in turn, the operating system executes its own polling loop which periodically scans the kernel's devices or internal queues in order to determine if any events have occurred (see Figure 1). This polling loop periodically awakens the CPU to perform this device scan, hence preventing the CPU from entering its most power-efficient, deep-sleep state when the mobile device is otherwise idle. The CPU's inability to enter a deep-sleep period during idle periods unnecessarily consumes power and drains the device's battery (Alagöz, Löffler, Schneider, & German, 2014).

To illustrate the power draining characteristics of a polling loop, consider a mobile device sitting idly in a user's pocket, which is where many mobile devices spend a large portion of their time (Öquist & Lundin, 2007). During this time, the mobile device must still poll for events, thus draining the battery's power. While there are many techniques used in mobile devices to mitigate the negative effects of the polling loop, none completely eliminate unnecessary polling, including in the operating system itself (See Appendix A). In fact, this phenomenon has given rise to "app-killers", which are applications designed to close idle, power consuming applications that needlessly poll for events on an idle device (Stewart, 2011).

A second drawback of polling loops is that they force system or application programmers to carefully select the polling frequency in order to avoid incurring either excess latency or bogging down the CPU as it executes the polling loop. For example, if the event queue is checked frequently, the CPU will spend most of its time checking the event queue and hence be slow to perform other tasks or enter a lower power state. On the other hand, if the event queue is checked infrequently, events will start accumulating in the event queue, and users could notice a delay in the application's responsiveness. This delay is often called *latency*.

We define latency as the delay between the time an event arrives at a device and the time that an application starts processing the event. An alternative definition for latency is the delay between the time an event is received by a device and the time the application presents a response to the user. However, the kernel cannot control the length of time that an application requires to compute its response to an event. It can only control the length of time that elapses before an application starts processing an event. Since the latter measure is what we are interested in minimizing in this dissertation, we have adopted the latter definition for latency.

Technological Advances Enabling the ESM Model

The reason polling loops still exist is that for many mobile devices, the hardware peripherals do not have a method for notifying the operating system that an event has occurred. For example, older hardware devices will change one of their internal registers to signal that an event has occurred, but the operating system must then scan the hardware in order to determine which hardware device set its register. Since the OS must "pull" events from the hardware devices using a polling loop, this type of event handling model is often called a "pull" model.

Recent advances in mobile hardware devices have allowed us to re-think how the OS can handle events more efficiently. Specifically, new hardware devices can directly tell the CPU that an event has occurred, thus "pushing" events to the OS. This evolution has occurred in two steps. The first step was to have the hardware device generate an interrupt by triggering a pin on the CPU. The interrupt caused the CPU to jump to a certain block of code called the interrupt handler. However, the operating system still had to determine which hardware device generated the event by scanning all of the hardware devices attached to the mobile device. Hence, polling loops could not be eliminated since events still had to be pulled from the device.

The second step and newest improvement for mobile devices is the “vectored interrupt controller” or VIC. The vectored interrupt controller allows the hardware to both interrupt the operating system and to provide an identification number so that the operating system knows what hardware device caused the event. Hence, the OS no longer needs to pull events from devices by scanning the hardware to determine which device caused the event. Instead, the input devices can “push” their events to the OS. Desktop computers already perform this type of interrupt vectoring, but input devices on mobile platforms are just now starting to catch up with desktop devices, such as with the Generic Interrupt Controller (GIC) provided by ARM for mobile devices.

Another recent technological improvement in mobile devices is the incorporation of more power-conserving CPU instructions (Bhadauria & McKee, 2008). The ARM CPU, which dominates the mobile device consumer marketplace, continues to improve the ways an operating system can coordinate with the CPU and other hardware devices in order to reduce unnecessary power consumption. Polling loops do not allow a mobile OS to make efficient use of these power saving instructions since they constantly rouse the CPU and prevent it from being placed in a deep sleep state. However, our ESM uses the new interrupt driven hardware devices to eliminate the polling loop, which in turn allows us to place the CPU in a deep sleep when the applications on a mobile device are idle.

The final technological improvement for mobile devices that we exploit in this dissertation is the introduction of new, lower power consuming CPU cores known as “shadow cores”, +1 cores, or big.LITTLE (Kim, Kim, Geraci, & Hong, 2014). Shadow core, +1, and big.LITTLE describe the same concept of a lower power consuming CPU core, but the implementation is vendor dependent. The lower power consuming cores trade performance for increased power efficiency. The ESM exploits this development by using the smaller cores to process events and determine if any applications need to be notified, thus leaving the big, power hungry cores undisturbed and in a deep sleep state if no application cares about an event (the small cores would discard the events in these cases).

Contributions

This dissertation exploits the enabling technological advances described in the previous section by developing the following three new and interrelated techniques for reducing the power consumption and latency of mobile devices:

1. A push model, called the Event Stream Model (ESM), for pushing events by hardware input devices to an application without using a polling loop.
2. A kernel level implementation of a display server that coordinates event handling between the ESM and the application, and that provides information about GUI applications to the scheduler.

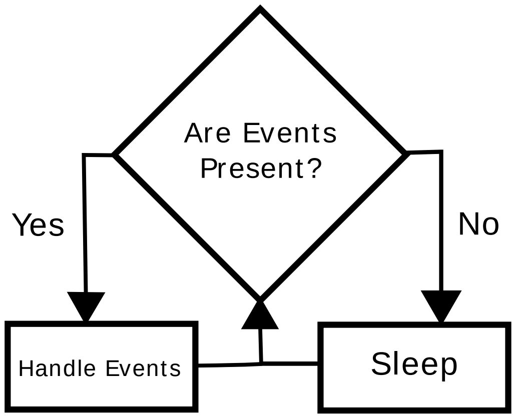


Figure 1: A polling loop is a continuous loop which polls for events. If events are present, the loop retrieves the event and handles them. If there are many events, the loop never sleeps and continuously handles the events until the queue is exhausted. On the other hand, if the queue is empty, the polling loop delays by sleeping in order to rest the CPU before the queue is rechecked for events.

3. An improved scheduler that takes advantage of information about GUI applications, such as whether they are visible and whether they are in the foreground or background, to make more intelligent scheduling decisions that save power and reduce latency

An important element of this dissertation’s contributions is that the power consumption and latency savings are automatically realized by the kernel without explicit intervention by the application programmer. Previous efforts at using software to reduce power consumption have relied on the application programmer to make explicit interventions. Since power consumption is rarely a selling point for applications, application programmers have tended not to take advantage of language features that might reduce power consumption. Thus, the techniques described in this dissertation provide a way for software to obtain power consumption savings while freeing application developers to focus on features that are more important for marketing their apps.

The Event Stream Model

This dissertation makes two improvements to the existing event handling models in mobile operating systems. First, we take advantage of the recent advances in hardware devices to implement a push model. Second, we implement the push model in the kernel, as opposed to either the application or the GUI library used by the application. Specifically, we have developed and implemented a push-style event handling model in the kernel of the Android API Level 22 OS (known as “Lollipop” or version 5.1). We chose Android due to its dominance in the marketplace for both consumers and research (Holtsnider & Jaffe, 2012). Android’s kernel is derived from Linux and uses the same scheduling and event handling algorithms as the desktop version of Linux. In particular, the existing Android kernel uses a pull model with a polling loop.

In Chapter 3, we describe our kernel implementation of a push model, and then in Chapter 6, we present results that compare its performance with that of Android’s existing pull model on a 32-bit, NVIDIA TK1 reference board (NVIDIA, 2015). The TK1 is a cutting-edge, mobile development board that contains the hardware, including a vectored interrupt controller, required to make the push model work (See Appendix B). The TK1 design reference is used in the NVIDIA Shield tablet computer and is also used by developers creating future consumer devices, so we are testing with a “real world” reference board.

The Kernel Display Server

The event stream model by itself significantly reduces both latency and power consumption (Marz & Vander Zanden, 2015). However, we discovered that the display servers used by GUI applications offer another area for improvement. Currently, a middleware software package, called the *display server,* is responsible for coordinating an application’s activity with the kernel, including coordinating event handling. For example, the display server keeps track of the stacking order of applications (i.e., which applications are on top and which are on the bottom) and is also responsible for providing a method for individual applications to handle events. One source of inefficiency arises from the fact that the display server is implemented in user space and is therefore separated from the kernel. This bifurcation forces the display server to duplicate some of the kernel’s functionality. For example, the display server contains its own event queues and state machine in order to track how an application should be displayed and interact with other applications. Additionally, the display server implements a graphics driver that is responsible for communicating with the kernel’s hardware graphics driver. This forces a vendor to write a driver for the display server using the display server’s API and another driver for the kernel using the kernel’s API. This complicates the programmer’s job and may discourage certain vendors from optimizing their hardware for a particular display server and by association a particular operating system.

In response to the inconvenient and inefficient nature of current application-level display servers for mobile devices, we have designed and developed a new display server in the kernel that we call the *Kernel Display Server* (KDS). It contains a unified graphics/video driver, hence removing the need to write two different drivers, and co-locates its event queues and event system with the kernel through the ESM, which is also implemented in the kernel. Therefore, the effective distance an event must travel between the hardware and the application is shortened, the number of queues is reduced, and as a result, power consumption and latency are reduced.

The Categorical GUI Scheduler

We have capitalized on the placement of the KDS in the kernel to improve the scheduling of GUI applications. We call our improved scheduler the *guiS* or *GUI scheduler*. GUI applications execute differently than traditional, console-based applications and might have several interconnected pieces of code that handle items, such as events, drawing, and processing. While these pieces are unified under one application, each section of code has a different focus (e.g., some attend to the user, some attend to the kernel, and some attend to the window manager). The KDS uses this knowledge to categorize the tasks that are performed by the different code segments of a GUI application. This enables guiS to skip certain code segments if executing these code segments would have no effect. For example, the GUI scheduler avoids executing the drawing code for an application that is not visible. This smarter scheduling removes the need for many aggressive sleep policies, which can cause applications to incur additional latency. In fact, many applications use *wake locks* to force the mobile device to stay awake in order to mitigate the latency penalty from the aggressive sleeping policies; however, this comes at the cost of increased power consumption since the mobile device cannot go to sleep while "locked" awake. Wake locks are necessary for certain applications that cannot tolerate aggressive sleeping policies, such as when the user is using a movie playing application. The application programmer enables many of these wake locks, which means that careless programmers could mistakenly force the CPU to remain in the highest power consuming state, even when there are no tasks for it execute.

Summary

Figure 2 shows how the event stream model interacts with the kernel display server. We will present experimental results in Chapter 6 that show that the new event stream model, kernel display server, and GUI scheduler work synergistically to reduce power consumption by up to 31.2% and to reduce latency by up to 17.1 milliseconds in our experimental mobile device apps.

Use Cases

The ESM, KDS, and guiS show improvements over the pull model in power consumption and latency whenever events arrive in an irregular fashion. The reason is that the CPU can rest between events and can then immediately respond to an event when it occurs. In contrast, the pull model must constantly poll for events, thus rousing the CPU and increasing power consumption, while also forcing the CPU to wait to process an event until the end of the polling loop’s sleep interval, thus increasing latency. Our push model shows a marked improvement in power consumption when events arrive in short bursts, followed by long periods of idle time, such as occur in social media applications where users may engage in a burst of activity followed by long periods of browsing. For example, a person might either quickly enter some number of keyboard or voice input events to compose a text message or email response, followed by periods of inactivity spent browsing text or email messages or figuring out how to compose a response to such a message. During the long idle times our push model allows the CPU to rest whereas a pull model will constantly rouse the CPU with its polling loop. While this dissertation focuses on the event handling in graphical user interfaces, it is quite likely that our push model would work well in other “bursty” situations that currently use polling loops, such as cell phone usage and wi-fi usage. Finally our push model improves power consumption when no events are occurring but an app is performing computationally expensive activities, such as decoding video. In this case our push model allows the CPU to stay focused on the computationally expensive task, rather than periodically switching contexts to check for non-existent events. This allows the CPU to either finish the task more quickly, or to perform the task at a more leisurely pace that is less taxing on the CPU.

The one use case where it makes little difference whether the pull or push model is used is when an app receives events at pre-specified intervals, such as when it is performing sensor readings for items like velocity or temperature. For example, if an application wants to display miles per hour, the application would likely sample the velocity sensor as a multiple of an hour (e.g., one sample per second or one sample per thirty seconds). In this case the sleeping period for the polling loop can be set so that it exactly corresponds to the period between sensor readings and hence it neither unnecessarily rouses the CPU or experiences much latency (there might be slight latency due to timing drift caused by scheduling but it would typically be negligible). We note that even in these situations, our push model might show a slight improvement in both power consumption and latency, since the event would be delivered directly to the application rather than passing through the multiple event queues required by the pull model.

Organization of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 discusses related work in the areas of event models and GUI applications. Chapter 3 describes the event stream model, Chapter 4 describes the kernel display server, and Chapter 5 describes the GUI scheduler, which is a subsystem of the kernel display server. Chapter 6 describes our testing platform and testing programs, and presents our experimental results. Finally, Chapter 7 presents our conclusions and recommendations for future work.

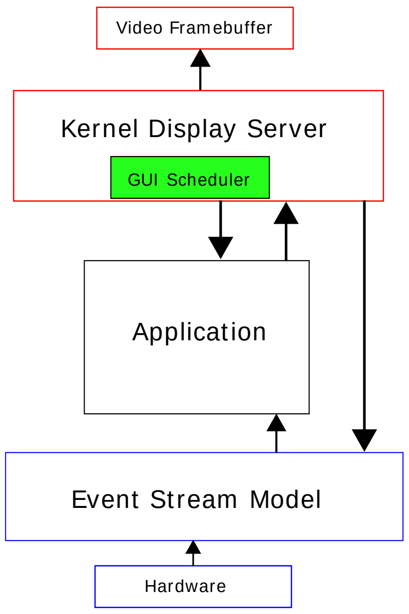


Figure 2: The ESM accepts events from hardware devices and pushes events into the GUI application. The KDS tells the ESM to enable/disable certain events depending on the state of the GUI application, such as its foreground/background status, which is controlled by the GUI scheduler. Finally, the KDS automatically coordinates with the ESM, but events are pushed directly to the application.

Chapter 2

Related Work

Our review of related work focuses on five distinct areas that are related to our work: (a) kernel and application event models that are used or have been proposed for use with mobile devices, (b) graphical display servers, (c) mobile operating systems, (d) schedulers whose main goals are to reduce latency or to reduce power consumption, and (e) power aware programming.

Kernel Level Event Models

Event models found in mobile operating systems utilize a pull model due to its simplicity and its wide-range of supporting architectures. Therefore, it is no surprise that most of the research into event models for mobile devices are improvements over the traditional pull models.

The majority of research in event modelling has concentrated on how to mitigate the inefficiencies of the polling loop by aggregating events from multiple devices into a smaller set of event queues that can then be polled by an application. The idea is that rather than forcing multiple applications to poll every hardware device queue, the OS can poll the hardware devices on behalf of multiple applications and store the results in a smaller set of event queues. Each application can then poll this smaller set of queues in search of events it should process. The benefit is that polling even one event queue is CPU intensive, and hence consumes power.

Rossi (2003) first highlighted the problem of having each application poll every hardware device using Linux's *select* and *poll* system calls. The select call blocks the application from running until it detects an event on one of the hardware devices. It might seem that blocking the application would permit the CPU to sleep, but unfortunately this is not true because the select call executes its own polling loop. Each iteration of the loop takes time to detect activity on event or file descriptors, which is proportional to the size of the array of descriptors. Rossi notes that “this increases the application latency and leads to a decrease in the overall system performance". Despite their inefficiencies, the *select* and *poll* system calls remain one of the two principle ways of handling events in Linux.

Megapipe (Han, Marshall, Chun, & Ratnasamy, 2012), epoll (Strebelow & Prehofer, 2012), and KQueues (Lemon & Manual, 2013) are event aggregating software solutions that were designed to improve the efficiency of determining if data (events) exist on a socket or file descriptor (event queue). Megapipe is a theoretical model that was not implemented in an actual system that we could find, and epoll is the actual realization of the Megapipe model. Epoll is now the second principle way of handling events on Linux systems and is used by the Android OS. KQueues represents a somewhat similar solution but was implemented in FreeBSD Unix. All three models 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 querying improves the efficiency of the pull model, it does not eliminate the polling loop or the power inefficiencies inherent with the polling loop. In fact, these three systems solve a problem that is orthogonal to the push/polling loop problem. They solve the problem of how to efficiently retrieve events from a large number of event queues (e.g., network sockets, input events, and file descriptors), but they do not solve the problem of how to stop an application from ``spinning" in a polling loop while waiting for an event to occur. Our research is aimed at stopping the application from spinning in a polling loop and thus consuming power unnecessarily.

The Linux kernel objects system (known as ``kobjects") is used in Linux as a device package (Kroah-Hartman, 2007). Kobjects has a notification model that devices use when they are added to or removed from the kernel. This notification occurs via a ``uevent", which can be sent to a bus where an application finally pulls the event. Hence, the bus acts as an aggregator of events. Unfortunately, these events are defined by the kobjects system and are not native events generated by the hardware devices. Therefore, the kobject notification model is not flexible enough for our purposes.

EVDEV or “Event Device” is the primary event aggregating system in Linux and Linux-derived operating systems, such as Android (Pavlik, 2001). EVDEV is written in the kernel and is responsible for queueing events for a particular input device (e.g., mouse or touchpad). EVDEV creates a special file for each input device that is then used by the application (or delegate, such as epoll) to poll and to retrieve events.

Inotify, or “inode notify”, is a limited kernel level event model used to aggregate file change events and to notify applications when a file changes (Love & Zhen, 2015). It works similarly to EVDEV in that it populates an event queue with a notification which then can be polled by an application. The kernel’s actions are limited to populating an event queue with the event notifications, which means that mobile devices are still forced to use battery power in order to poll the event queue.

Parthasarathy (2006) provided a mechanism for software to gracefully reduce the energy consumed when it is not being fully utilized and involves reducing power consumption at the hardware level. It also provides key insights into how to reduce power consumption at the software level, including how to efficiently handle events from hardware and how to efficiently handle polling an entire bus of hardware devices for events. Since our research strives to remove polling hardware altogether, we used this paper as a guide for what type of hypothetical power numbers could potentially be realized by redesigning the event system in the operating system kernel.

Finally POSIX signals, such as the terminate, kill, and segmentation fault signals, implement a form of the push model (Kerrisk & Project, 2015). 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 use 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).

Application Level Event Models

Most GUI applications are written on top of middleware that hides from the application programmer the system calls required to fetch events from the kernel and to display the application graphics on the screen. The Java Virtual Machine (VM) is one such piece of middleware. It provides the application programmer with an event push model (Chen, 2002). Using the Java application framework, a programmer creates an event-handling function that is used to respond to an event and then registers that function with the event system. When the Java VM receives an event from the kernel, it finds the event handling functions that are interested in the event, and calls them, thus pushing the event to the event handlers. We have adopted these two features, callback functions and function registration, in the event stream model presented in this dissertation. However, our event system is broader in scope since it extends from the initial hardware event all the way to the application. In contrast, Java provides a push model only after an event has arrived at the application level. The Java virtual machine still uses a polling loop to pull events from the kernel and display server. It is only after Java polls and retrieves an event that the event is pushed to the application programmer's event handler, and thus this type of push model is only used to make the programmer's job easier (Cugola, Margara, Pezzè, & Pradella, 2015).

GIMP Drawing Kit (GDK) and GLIB are GUI middleware libraries developed by the open source group GIMP that provide high-level wrappers around the underlying display server and event model API, for window managers such as X11 or Wayland (Nilsson, 2015). The goal of GDK and GLIB are to allow a programmer to develop portable applications, meaning application that may be used on several different operating systems or architectures with little to no modifications. However, these libraries still use a pull model, which means that while they hide the event polling loop from the programmer, the polling loop still exists and increases the power consumption and latency of GDK/GLIB applications.

Mobile Operating Systems

There are three commonly used mobile device operating systems in use today: (a) Google Android, (b) Apple iOS, and (c) Windows Mobile. This section examines these operating systems and the system design principles they incorporate. To our knowledge, none of these operating systems implement a kernel display server, event stream push model, or categorical GUI scheduler.

The Android operating system is developed and distributed by Google, Inc. and is the most popular mobile operating system. Android is developed from the mainline Linux kernel but is modified for the Android mobile platform. However, the graphics package for application developers is a Java virtual machine called Dalvik or, more recently, ART or *Android Runtime* (Singh, 2014).

The graphics interface that Android employs is the typical foreground/background system where the user interacts with one application at a time but also where many applications may run in the background. Each application runs on a separate instance of the Dalvik or ART virtual machine and connects to the graphics interface via two special processes: (a) surface flinger, which is used to draw GUI components to the screen, and

(b) input flinger, which is used to aggregate and to pump GUI events to the applications. The input flinger system uses Linux's "epoll" system to poll for and to retrieve events from the Linux kernel, which means that an event polling loop is used for each application, regardless of whether it is in the background or the foreground.

Microsoft Windows Mobile is a mobile version of Windows that is designed for the mobile ARM processor. It is the principle operating system for many ARM-based tablet computers. Windows Mobile provides a public API method that suspends a running thread while it waits for events, but its implementation is a proprietary trade secret.

Microsoft has recently released a new operating system called Windows 10 Mobile which is designed for portability with its desktop cousin, Windows 10 ("Windows 10 Experience" 2016). While much is still unknown about the implementation of this new operating system, application development is done through Microsoft's .NET framework and applications may be developed using several different programming languages, such as Visual Basic or C#. However, the .NET framework still uses the underlying Windows 10 API (written in C and C++) as its GUI event handling system. Further examination makes it evident that some sort of "message pump" (aka polling loop) is used by applications to poll and retrieve events from the Windows kernel.

Apple iOS is a popular mobile operating system developed by Apple, Inc. and is a competitor to Android and to Windows Phone. Apple iOS' operating system design and practices are also proprietary, so the implementation of most of the operating system is a trade secret. However, Apple provides a guide for users to save energy on their smart phones, which focuses on how to allow your phone to get the maximum amount of sleep ("Maximizing Battery Life and Lifespan," 2016), The guide provides several tips for saving battery power, such as turning down the brightness of the screen, reducing the time that elapses between when the last input event is received from the user and when the device powers itself down, closing applications that are not being used, and turning off extraneous peripherals. These suggestions reduce the usability of the mobile phone in exchange for improved battery life. Fortunately, by eliminating the costly event polling loops, our proposed ESM may improve the battery life of a mobile device without the user having to restrict how the device is used.

Graphical Display Servers

This section examines some of the most common display servers used in today's GUIs. To our knowledge, there are no kernel-level display servers in any modern mobile operating system, so this section includes only display servers implemented in user space (i.e., as part of an application).

The most common display server for Unix environments is the X.org display server, which is the modern fork of the X11 display server (Anderson, Mor, & Coopersmith, 2002). Graphical user interface applications connect to the X.org display server in order to draw to the screen and handle events. The X.org server is a middleman between the application and the kernel and represents an additional hop between the kernel and the application. Unfortunately, in mobile devices, this generic, distributed approach leads to increased latency and power consumption.

Direct framebuffer or DirectFB is a library and a Linux kernel module that provides user applications access to graphical drawing commands directly through the kernel’s graphics driver (Kropp, 2014). DirectFB provides only the essential routines to implement the drawing side of a display server, however, this means that no event handling routines are provided through DirectFB. DirectFB is a useful starting point for our Kernel Display Server’s drawing system. In fact, our KDS uses many of the same kernel routines as DirectFB in order to draw directly to the screen. Furthermore, DirectFB has already been extensively tested since it is used by several popular consumer products, such as Roku.

LightDM is a display manager commonly used in the Ubuntu Linux distribution. A display manager is responsible for managing multiple connections to a single display server. However, a display manager can also manage only a single connection to a display server, which is the common configuration for mobile devices. One item to note, however, is that LightDM executes a polling loop to retrieve events from the display server, and the display server in turn executes a polling loop to retrieve events from the kernel. While the display manager is not the focus of this dissertation, any display manager that uses a polling loop could potentially benefit from our ESM.

Scheduling and Heterogeneous CPU Cores

The introduction of heterogeneous CPU cores into both mobile and desktop operating systems creates interesting scheduling questions about which processes should be placed on which cores. Kim et al. (2014) modified the Linux kernel’s load balancing algorithms for the big.LITTLE architecture, which is ARM’s energy optimization scheme that pairs high performing, but power consuming cores with lower performing, but lower power consuming cores ("big.LITTLE Technology," 2016). Kim’s work emphasized how to allocate processes efficiently among the heterogeneous cores. We used several aspects of their results to implement the ESM and scheduling models we describe in this dissertation. For example, we place the lightweight event dispatching processes on the low energy, power-reducing cores and place the more computationally expensive application event handling functions on the faster, but higher energy cores.

Seehwan Yoo, Shim, Lee, Lee, and Kim (2015) describe several conditions which can decrease the efficiency of the big.LITTLE or shadow core processors. They lay out several ways to achieve increased power efficiency without sacrificing performance. Our dissertation uses the knowledge gained from several of their test cases to minimize the negative effects that occur when switching between the performance cores and the power efficient cores.

Hsiu, Tseng, Chen, Pan, and Kuo (2016) examined the current set of process schedulers and determined they are not suitable for mobile devices with heterogeneous CPU cores. Their research shows that with the conventional schedulers, such as the Completely Fair Scheduler (CFS) which is used in most Linux-type OSes, including Android, energy efficiency and performance are not maximized. Our research, as part of this dissertation, came to the same conclusions, which provided the impetus for modifying the scheduler in order to efficiently accommodate the performance cores and power efficient cores. However, their paper helped guide our approach when we extended the ESM into a single scheduler we call the GUI scheduler (guiS). For example, our guiS can determine which set of cores is best to balance event handling on, and furthermore, the guiS can differentiate between a single shadow core and multiple, lower-power consuming cores and choose the best technique.

Gaspar, Taniça, Tomas, Ilic, and Sousa (2015) have proposed a framework for an application-aware task management system for mobile devices using heterogeneous CPU cores. Their system utilizes the new mobile device CPU technologies, such as big.LITTLE, in order to improve the performance of applications, as well as using the objective of the application to better determine how to allocate CPU resources to that application. We have taken a similar approach in implementing our GUI scheduler. Our work differs from their work in that our scheduler is specifically designed to work with mobile GUI applications. In addition, as far as we can determine, their task management system is a theoretical framework, whereas our GUI scheduler is an actual implementation.

Bui, Liu, Kim, Shin, and Zhao (2015) studied the effects of using the larger, power hungry cores when scheduling mobile GUI applications, specifically Chromium and Firefox. Their research neatly aligns with ours in that one of this dissertation’s hypotheses is that the power consumption of GUI applications can be significantly reduced by smartly managing the different cores.

The current scheduler in the Android operating system and other Linux-based operating systems is the “Completely Fair Scheduler” (CFS) (Pabla, 2009). This scheduler aims to provide an equal (fair) proportion of CPU to each running process

(Wong, Tan, Kumari, & Wey, 2008). However, the results in this dissertation show that the unique features of mobile devices might be best served by modifying this scheduler. The guiS that we designed and tested in this dissertation is a heavily modified CFS scheduler.

Power Aware Programming

Power aware programming incorporates the notion that the style of programming has an impact on the amount of power a mobile device consumes (Honig, Eibel, Kapitza, & Schroder-Preikschat, 2012). This section describes several power aware programming languages and power aware programming tools that a developer might use to examine energy inefficient parts of their code. It also summarizes several papers that examined the power consumption of GUIs in different settings. In Chapter 6 we demonstrate how our own improvements to kernel programming can lead to decreased power consumption of GUIs on mobile devices.

The Eon programming language is an energy-aware programming language that is structured to automatically adapt programs to a mobile device’s energy state (Sorber et al., 2007). Eon is designed to be portable among hardware platforms and to maximize the performance of an application under several energy conditions. While we do not know of any application written for Android, iOS, or Window Mobile that uses the Eon programming language, it does show that there are other ways of conserving power than simply powering down the device or peripherals of the device.

The ET programming language is also an energy-aware programming language that specifically targets the Android mobile operating system (Cohen, Zhu, Senem, & Liu, 2012). Their approach differs slightly from the Eon programming language in that the ET programming language identifies distinct patterns of program workload, which can then be used to determine the power state to run the program. The ET programming language allows the programmer to specify their routine's energy state or to allow the compiler to determine the most efficient energy state for their routine.

Hybrid MPI (Hybrid Message Passing Infrastructure) is a heterogeneous, distributed message passing infrastructure designed for high performance computing (HPC) applications (Wickramasinghe, Bronevetsky, Lumsdaine, & Friedley, 2014). Hybrid MPI balances high performance with energy efficiency since many HPC systems can be ravenous energy consumers. Therefore, Hybrid MPI can be useful in saving energy-related operating costs associated with HPC applications.

Trepn (developed by Qualcomm, Inc.) and.), CodeXL (developed by AMD, Inc.) are two.), Code Works (developed by NVIDIA, Inc.), and AppScope are energy profiling tools that a programmer may use to determine which parts of their programs consume the most battery power. CodeXL in particular can also determine the amount of power that the GPU is consuming in addition to the CPU. This is important in GUI applications where the GPU is used to accelerate drawing GUI components to the screen.

It would have been useful if we could use one of these energy profiling tools to help us determine how much battery power is saved by our redesigned kernel algorithms. Unfortunately, none of these energy profiling tools work intrinsically with our NVIDIA

TK1 reference board. Trepn only supports devices with the Qualcomm Snapdragon processor, and CodeXL is developed only for Linux and Microsoft Windows running on an Intel X86/64 platform. Furthermore, its source code is not open to the public, and we are unable to modify the tool to work with our testing platform. Yoon, Kim, Jung, Kang, and Cha (2012) developed AppScope, but unfortunately, it was not designed for the NVIDIA TK1. However, its power formulas could be used to develop a version of AppScope to work with the NVIDIA TK1. Finally, NVIDIA’s Code Works was designed for the Android operating system on its platforms. We were able to use Code Works to profile running code with our hardware testing platform, and we used it to confirm the accuracy of our results that we obtained using other methods.

Vallerio, Zhong, and Jha (2006) and Zhong and Jha (2003) have researched energy efficient GUIs in regard to handheld, mobile devices. Their research confirms our hypothesis about graphical user interfaces: that GUIs consume a large portion of power in mobile computers. Their paper shows that power consumption can be improved by simply designing GUIs with system energy efficiency in mind. We take this step further by redesigning several operating system algorithms in order to improve energy efficiency.

Brotherton, Dietz, McGrory, and Mtenzi (2013) examined the effect of the operating system in increasing power consumption in regard to the GUI. They examine a cascading effect where power consumption is reduced by having the operating system work in conjunction with the GUI. Unlike this dissertation, their research focused on data centers where energy consumption is compounded by the sheer size of the operating equipment. Furthermore, they suggest removing the GUI altogether from the operating system, which is something we cannot do with mobile devices. However, their research provides us insight with what can occur when the operating system is designed with an increasing number of features rather than a leaner operating system that performs simple tasks very efficiently.

Vallerio et al. (2006) and Zhong and Jha (2003) have examined the power consumption of GUIs on handheld, mobile devices. Their research confirms our hypothesis about graphical user interfaces: that GUIs consume a significant amount of power on mobile computers. We take their work one step further by redesigning several operating system algorithms in order to improve energy efficiency.

Brotherton et al. (2013) examined how operating systems can increase the power consumption of GUIs. Their research focuses on a data-centric server system, rather than mobile OSes, and hence was not directly applicable to our research. However, they do present the GUI’s overhead in terms of power consumption and performance in server applications.

Chapter 3

The Event Stream Model

This chapter first describes a number of problems with the pull model used by existing mobile operating systems. We then provide an overview of the event stream model (ESM). Finally, we provide a detailed description of the implementation of the ESM, which includes a description of its data structures, a description of the API it provides the application so that the application may interact with it, and a description of the kernel routines required to implement it.

Problems with the Current Pull Model

As noted in the introduction, the biggest drawback associated with the current pull model is its need to constantly execute polling loops to retrieve events from the input devices. However, it has other drawbacks as well. First, the current event pull model separates and duplicates storage for events by maintaining event queues in both the kernel and the application. The kernel first polls hardware devices and populates a kernel event queue with these events. The application then retrieves the events from the kernel's queue and stores the events in its own queue. Figure 3 graphically depicts how an event propagates through the current pull model.

In practice, the propensity for the pull model to block the application, meaning that the application spends all of its time waiting for events rather than executing useful instructions, has given rise to many middleware event hubs that assist in handling events (see Figure 4). Furthermore, in GUI applications, the middleware package is the display server which typically handles aggregating events for the application. However, many of these middleware packages include an additional polling loop and event queue. These middleware packages poll the kernel and then store the event into one of their own event queues. In turn, the application polls the event hub with yet another polling loop, and the events are either handled by the application immediately or stored in yet another event queue inside of the application itself.

An additional drawback of the pull model is that it requires a driver at the application level to translate a raw event into something that the application can understand (Mogul & Ramakrishnan, 1997). In contrast, the ESM translates an event before pushing it to an application. The additional routines and logic that the pull model requires an application programmer to write to translate an event can add a large amount of latency and power consumption if not efficiently written. Fortunately, Android provides this event-translation functionality in its VM, which eliminates the need for the application programmer to write these translation routines. However, Android's VM does not eliminate polling loops. In fact, Android's event hub uses Linux's *epoll* system to pull events from the kernel, and as we've stated in Chapter 2, *epoll* does not eliminate the polling loop. For clarity, Figure 5 shows a side-by-side comparison of select, epoll, and our ESM. Many mobile operating systems, including Android, mitigate a small portion of latency by setting the event polling interval equal to the refresh interval for the LCD screen (Vallerio et al., 2006). The idea is that a user generates events in response to what

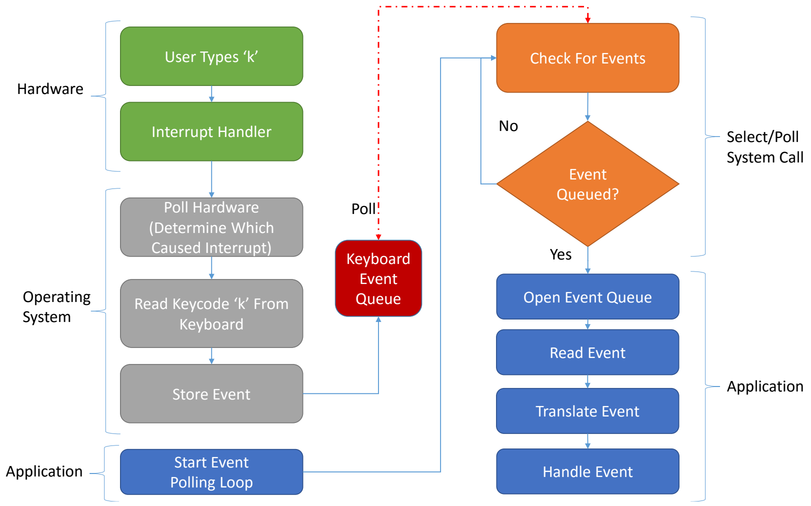


Figure 3: An event's path in the current pull model. The kernel-level routines take an event from inception and store it in an event queue for the keyboard (there are queues for each input device). The application starts a polling loop, which continually checks the event queue for events. When events are queued, the application is responsible for reading, translating, and handling the event.

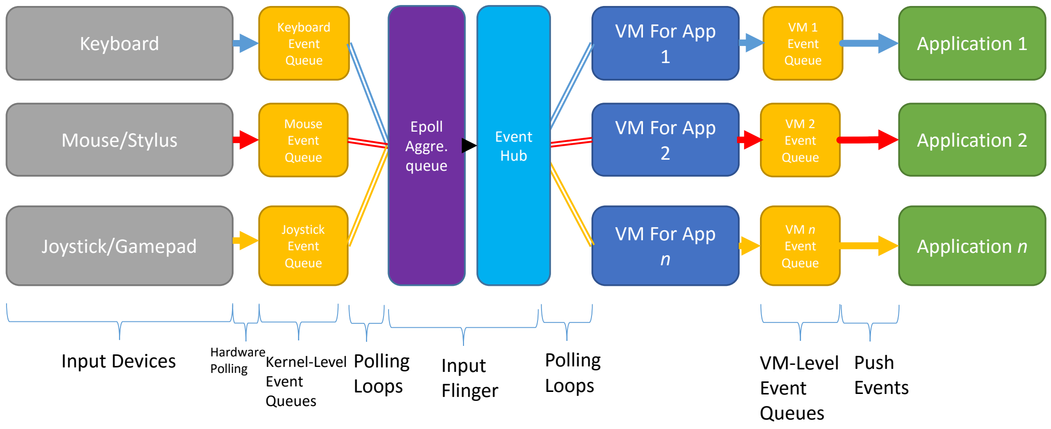


Figure 4: The event propagation between the kernel and the application in the current pull model. With virtual machines, such as Dalvik or ART in Android, an instance of the VM runs per instance of the application. An event hub is used to marshal events in temporal order and to decide how each of the connected virtual machines receives the events. In the GUI context, the event hub is typically integrated with the GUI display server. Finally, the Android VM pushes events to the application's event handlers, much like the ESM does.

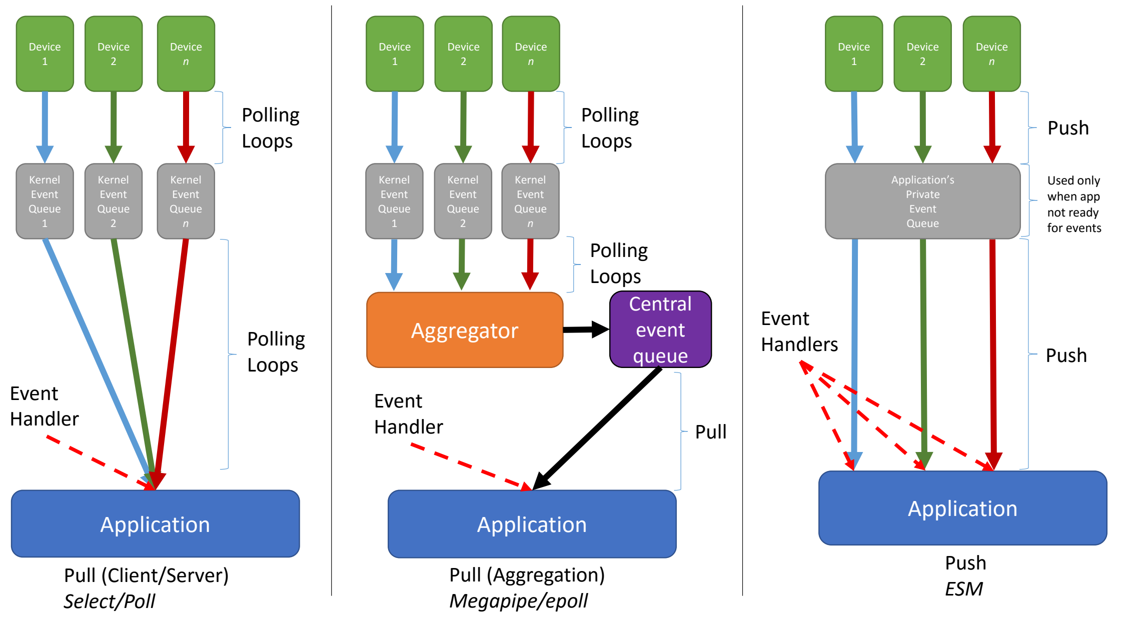


Figure 5: A pictorial comparison of the older pull model (client/server), the new pull model (megapipe/epoll), and our push model (ESM). The older pull model requires polling loops from the hardware to the queue and from the queue to the application. The newer models still require polling loops, but are much more efficient by bundling, hence aggregating, the polling into one loop and storing into a central queue. The application then pulls events one-by-one from the central queue. The ESM is a direct push model from the hardware to the queue to the application. We show all polling loops converging to a single point for the pull model as this is typical for normal applications where a “message pump” pumps messages (events) to a single function to handle all events. The aggregation model may or may not use a single function to pull or to push messages (events) to. Finally, our push model pushes to several different handler functions, allowing for near-instantaneous detection of which type of event was pushed, or at least from which device.

they see on the screen, and so events only need to be polled when the screen is refreshed. Typically, the screen refreshes 60 times every second (60 HZ), which requires a polling loop delay of 16.6ms, meaning that the screen refreshes every 16.6 milliseconds. Therefore, events will be retrieved by Android's event hub approximately every 16.6 milliseconds. However, this does not completely eliminate unnecessary polling since the polling loop will poll every 16.6 milliseconds even when no events have occurred. Furthermore, the polling loop could still incur latency when the event occurs while the polling loop sleeps. In that case, the application will not be able to respond before the next screen refresh interval, which means that the response to the event may be delayed by a complete screen refresh interval, thus further aggravating the latency issue.

Event Stream Model Overview

The goals of the event stream model (ESM) are to reduce latency and to reduce power consumption in mobile devices. The ESM is modeled after streaming graphics processors, which are based on the principle that events are more efficiently handled when they are not passed through several layers of event queues (Erez, Ahn, Gummaraju, Rosenblum, & Dally, 2007). This principle is embodied in the ESM, which significantly reduces the required number of event queues. Furthermore, the polling loop in the traditional pull model is eliminated with our ESM. This allows events to "stream" immediately from their creation to their handling, thereby reducing the power consumption and the latency associated with having multiple polling loops for a number of event queues.

The ESM kernel implementation consists of three routines: (a) device interrupt handlers and drivers that initially handle the event and collect information about the event, (b) 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 (c) an event dispatcher that calls the application's event handler. Figure 6 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: (a) notifying the application only if it is ready to receive events, and (b) packaging an event into an event structure that gives the application's event handler enough information to handle the event.

Data Structures

We needed to add several new data values and data structures to the kernel in order to implement the ESM model, including a new process state that indicates that an application is waiting for events, a new list that keeps track of the events that applications are interested in handling and the event handling functions that should be called when these events occur, and new kernel-level, private event queues for each application. The private event queues eliminate the need for applications to compete for access to a shared, centralized queue, which inevitably leads to complicated locking and exclusion schemes (Bhatt & Huang, 2010).

The Event Waiting State

A new process state called *EV_WAIT* was created to allow the application process to indicate that it is waiting for events. When the application is in this state, the scheduler can allow the application to sleep until a signal or event occurs. When the application is not in the EV_WAIT state, then the ESM scheduler will queue events in the application's private event queue. The EV_WAIT state prevents the kernel from pushing events to an application that is not ready to receive them, such as when the application is handling another event or is executing other instructions.

The Global Application List

A global application list was added to the kernel and is responsible for keeping track of which events an application is interested in receiving and for storing the address of the application's event handler. This is a persistent list that is created when the kernel is first initialized. The application or delegate is provided with a registration function that allows it to add or remove event handlers from this list. When an event occurs, the kernel checks the application list for those applications interested in the event, and then the event is forwarded to either the appropriate event handler or an application's event queue,

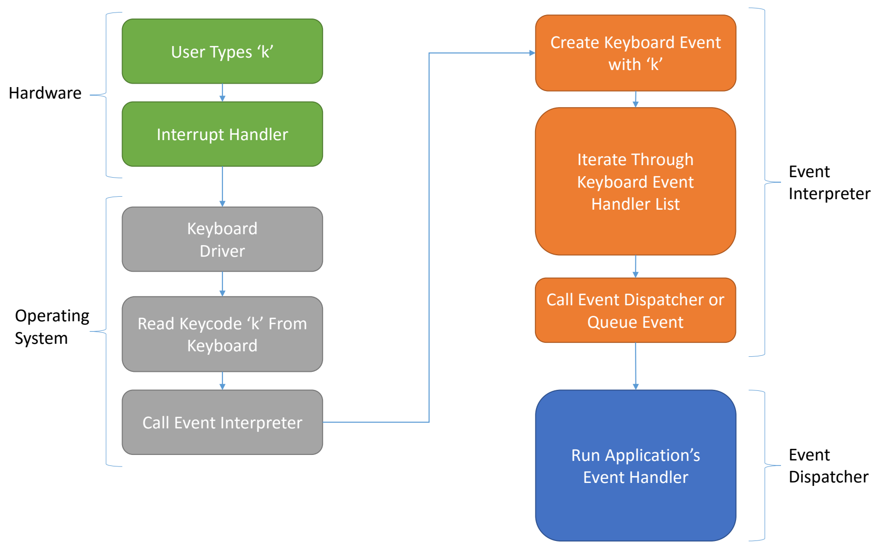

Hardware
User Types ‘k’
Interrupt Handler
Operating System
Keyboard Driver
Read Keycode ‘k’ From Keyboard
Call Event Interpreter
Create Keyboard Event with ‘k’
Iterate Through Keyboard Event Handler List
Call Event Dispatcher or Queue Event
Event Interpreter
Run Application’s Event Handler
Event Dispatcher

Figure 6: An event’s path in the event stream model. An event originates from hardware in the form of an interrupt. The kernel executes an interrupt routine to respond to the event. The ESM begins by using an event interpreter to package the event into a data structure that can be handled by the application and finding the set of applications interested in the event. The event interpreter forwards this set of applications to the event dispatcher, which calls the event handlers for each application.

depending on the application's state. In practice, this list is implemented as a look up table keyed on events, with each event having a list of pointers to objects containing information about the application handling the event (e.g., the application's process block) and the event handler for that function.

The Application's Private Event Queue

In addition to a global application list, each application contains its own private event queue which stores events in a first-in, first-out (FIFO) fashion. The private event queue only stores those events that occur when the application is not ready to receive events (i.e., when the application is not in the EV_WAIT state). The private event queue ensures that an application does not miss events that could not be immediately handled. The FIFO nature of the private event queue guarantees that events will be processed in the order that they occurred.

The event queue is stored in kernel space since there is a strict separation between kernel space and application space. If the queue were stored in application space, the esm_dispatcher would not have access to it unless it copied the queue from application space into kernel space, modified the queue, and then copied it back to application space. Placing the queue in the kernel avoids this costly step.

Application API

An application must use two system calls to coordinate with the operating system when using the ESM. One system call registers event handling routines with the kernel and one tells the kernel that the application is ready to receive events. The application may interact directly with the kernel or indirectly through a virtual machine or other such 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 ESM registration calls to the kernel).

The application will first execute any code that is needed to establish the initial program state. As it does so, it will make one or more kernel calls to register events in which it is interested and to associate callback procedures with these events. When the application finishes the initialization of its graphics and determines it is ready to receive events, it will notify the kernel though another system call. The kernel will then put the application into an event waiting state (called EV_WAIT), which indicates that events may be pushed directly to the application’s event handlers. The next two sections describe the registration and notification system calls in further detail.

Registering ESM Event Handlers

The esm_register function adds 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 call when that event occurs. If the application passes NULL as the event handler, the esm_register call removes the application from the event’s list. Figure 7 provides a pseudo-code implementation of the esm_register function.

Waiting for Events

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 first checks the application's private event queue in case events were queued while the application was performing other tasks. If events are queued, the next event is immediately pushed to the application's event handler. Otherwise, esm_wait 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. Figure 8 shows the pseudo code for the esm_wait function.

Kernel Implementation

The majority of the event stream model is implemented in kernel space, and by placing the ESM in the kernel, it has immediate access to many privileged sections of the CPU and hardware devices. The ESM's close relationship with the hardware shortens the path an event must take before being handled by an application. As with graphic processing units (GPUs), events that are in motion, meaning not stuck in an event queue,

```
event – the event to register
event_handler -- a pointer to the function that handles the event
application_list – the kernel's application event handler list

esm_register (application, event, handler) {
        if handler = NULL then
                delete application_list[event][application];
        else
                application_list[event][application] = handler;
        end
}
```

Figure 7: esm_register links an event to an event handler. If the event handler is NULL, then the event is removed from the application list.

are most efficiently processed when they are pushed directly to the application listening for the event. The ESM routines are split into three separate functions: (a) the interrupt handler/driver, (b) the event interpreter, and (c) the event dispatcher. We previously showed where these routines fit along the event pipeline in Figure 6.

Interrupt Handler or Driver

An interrupt handler is a routine that is called by the CPU to respond to an interrupt. The CPU registers an interrupt handler by storing an interrupt handler's memory address into special memory locations and is responsible for gathering information about events as they are generated by devices. In older generations of hardware, an interrupt only told the CPU that some hardware device generated

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

Figure 8: 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.

an event. It would then be the responsibility of the operating system and CPU to poll each hardware device to determine which one caused the interrupt. With the newer generation of vectored interrupt controllers, the initial interrupt tells the operating system which device generated the event, so the interrupt handler knows from which device to retrieve a hardware event. Typically, the interrupt routine hands off further processing of an event to a driver, which translates the event into some standard data structure that is understood by the OS. For example, a keyboard driver would convert the hardware signal into a key code to identify which key was pressed. Since hardware manufacturers are free to design their hardware as they see fit, a driver for the specific hardware is necessary to convert the data stream generated by the device to a data stream that the OS can understand. In other words, a driver allows the kernel to understand what the hardware is saying. Figure 9 provides a pseudo code implementation of the interrupt handler.

The Event Interpreter

The event interpreter determines where to push an event (i.e., which application is listening for the event). After the event is translated by either the interrupt routine or driver, the event interpreter uses the kernel’s application event handler list to determine which applications want to process the 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. This ensures that an application is not waiting for another application to finish before being pushed an event (e.g., if the applications are running on separate cores, then each application could process the event in parallel).

The interpreter does as little work as possible because it runs while the CPU is in the interrupt state where any additional interrupts are disabled. It is good operating system design practice to limit the amount of time the CPU is in this state (J. Lee & Park, 2010). Therefore, the event interpreter hands off the majority of the work to be done to the event dispatcher. Pseudo code for the event interpreter is presented in Figure 10.

The Event Dispatcher

The event dispatcher pushes events from the event interpreter to the applications. The event dispatcher is distinctly separate from the event interpreter since the CPU disables further interrupts while it is in the initial event interrupt handler and the event interpreter. This allows us to “free” the CPU to once again receive important interrupts at

```
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);
}
```

Figure 9: The interrupt routine calls the event interpreter. If a driver is attached to the interrupt vector, the event interpreter is called after the driver adds the event's details. A raw translation with minimal event details from an interrupt vector (for those without a driver handler) is available through a table look-up.

the same time that events are being dispatched to the applications.

The event dispatcher only pushes events to applications ready to receive them. If an application is busy handling events or other routines, the event dispatcher stores the event on the application's private event queue in the order that it was received. The application is then notified of the event when it signals to the kernel that it is again ready to receive events. Figure 11 provides a pseudo-code implementation for the event dispatcher.

Connecting to the ESM

The ESM interacts with Android through the *Input Flinger* service in Android. Input Flinger is Android's central event hub ("Android Input Pipeline," 2015), and it uses epoll to retrieve events from the kernel (Strebelow & Prehofer, 2012). We specifically

```
event – the event that was just received
application_list – the kernel's application event handler list

esm_interpret(event) {
    app_event = convert_event_into_application_readable_event(event)
    foreach application in application_list[event] do
        esm_dispatch(app_event, application)
    end
}
```

Figure 10: The event interpreter is responsible for repacking a kernel event into an application event and 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 disabled-interrupt state. For this reason, the dispatcher is called in a separate kernel thread to perform the majority of the work.

modified Input Flinger to use our ESM so that applications using the Android API could use the new ESM model without being modified. Our modifications make Input Flinger coordinate with our new Kernel Display Server (KDS—see Chapter 4), which then calls esm_wait and esm_register on the application's behalf. This is beneficial since application programmers are not required to have any knowledge of the underlying event system in order to write their mobile applications for the Android OS.

When writing "native" applications that bypass the Android API, the programmer must register the events they are interested in and the event handlers that will handle them with the esm_register system call. The programmer must then call esm_wait to put the application into the event waiting state. Any events that are subsequently received would be pushed to the application's event handlers until the application either de-registers the event handler or the application exits.

```
event – the event that was just received
application – the application to which to push the event

esm_dispatch(event, application) {
    if application.state = EV_WAIT then
        application.state = RUNNING;
        handler = application_list[event][application];
        handler(event);
    else
        application.private_event_queue.enqueue(event);
    end
}
```

Figure 11: 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 outside the interrupt state and is scheduled like any other kernel thread.

How The ESM Improves Power Consumption and Latency

The ESM improves power consumption and latency by: (a) streamlining event propagation and (b) removing costly polling loops. The polling loops used by the traditional pull model keep rousing the CPU and thus prevent the CPU from entering a lower power consuming state. The ESM eliminates these polling loops and hence does not rouse the CPU when the application is idle. This allows the CPU to more frequently enter lower power consuming states and reduces the mobile device's power consumption.

The traditional pull model incurs latency since multiple event queues are polled at certain frequencies. If the event occurs while a polling loop is delaying, then the event incurs latency until the polling loop detects its presence. In contrast, the ESM model dispatches the event to the appropriate applications as soon as the event occurs, thus reducing the latency associated with polling loops.

Chapter 4

The Kernel Display Server

The Kernel Display Server (KDS) is a GUI display server implemented in the kernel. A display server is a central GUI component responsible for passing drawing commands and input events from the application to the kernel, and vice-versa (Lehey, 1995). One very popular display server is the X11 display server (now known as X.org), which is used in many desktop Linux distributions (Repasky, 2004). The X11 display server contains its own event queues and own event handling protocol, such as using proprietary commands like *XNextEvent* or *XPeekEvent* inside of a polling loop, which is how an application would retrieve or poll for input events from the X11 display server.

The placement of the KDS into the kernel differs significantly from the current practice which implements display servers in the application layer as middleware between an application and the kernel. By moving the implementation of the display server to the kernel, we have managed to shorten the path for an event from the hardware to the application and to improve the scheduling of GUI applications by taking advantage of the knowledge we gain about a GUI, such as whether it is visible to the user.

The original impetus for the KDS came from our desire to reduce the number of polling loops that are required by middleware display servers. With current display servers, applications must retrieve their events directly from the display server, which in turn retrieves events directly from the kernel’s event system. This means that an event must first be stored in the kernel, where it is then polled and retrieved by the display server. The display server then stores the same event in its own event queue. Finally, the application polls the display server for the event, and the display server supplies the application with the event. Figure 12 shows the different polling loops and event queues that are required by existing middleware display servers. In mobile devices, these additional queues and polling loops are costly in terms of both increased power consumption and increased latency.

The Kernel Display Server removes these additional queues and polling loops by using the ESM to push events to the application (see Figure 13). The KDS mediates the interaction between the ESM and the application (or application middleware such as Android ART) by controlling which event handlers are active. For example, if the user minimizes an application or puts it into the background, the KDS will automatically unregister the applications' input event handlers since the application can no longer feasibly receive input events. When the application is restored into the foreground, and hence starts interacting with the user, the KDS reregisters the event handlers with the ESM and normal event operations resume.

In addition to better event coordination and the elimination of event queues and polling loops, two additional gains are realized by implementing the display server in the kernel. First, it eliminates the need to make system calls in order to acquire data from the kernel's numerous data structures, such as device event queues. A system call is a special CPU instruction that causes the CPU to unconditionally jump to a specific *system call* kernel routine. To make a system call, the CPU must generate an interrupt to itself by using a specific *service call* CPU instruction, context switch from the currently running application to the interrupt handler, and then change the privilege level from application mode (typically referred to as *user mode*) to kernel mode. While many architectures and operating systems have improved the efficiency of this process, it is by no means an efficient procedure. Our KDS removes many steps in this procedure since it has *direct* access to several data structures within the kernel, and hence it improves the efficiency of handling events from the kernel to the application. In turn, this improved process reduces power consumption and latency.

A second gain afforded by our kernel implementation of the display manager is that the KDS is aware of the various roles played by GUI code, some of which is I/O bound and some of which is CPU bound. For example, a GUI application must be able to handle inputs, such as a finger tap, which are I/O bound, while simultaneously displaying feedback to the screen or vibrating the device in response to that input, which are CPU bound actions. Depending on the application, there could be several more pieces of code that will traditionally fall into either CPU-bound code, I/O-bound code, or some percentage of both. For example, a video player will need to decode incoming video, which will require both constant network access and constant CPU access for decoding. These types of code can be prioritized in the scheduler for more efficient processing (Qin & Rusu, 2013).

The KDS fulfills the different needs of each segment of GUI code by allocating four threads in which the GUI application may place its code: (a) an event handling thread for handling events pushed by the ESM, (b) a display thread for drawing to the screen, (c) a background thread for handling constant activity, such as decompressing

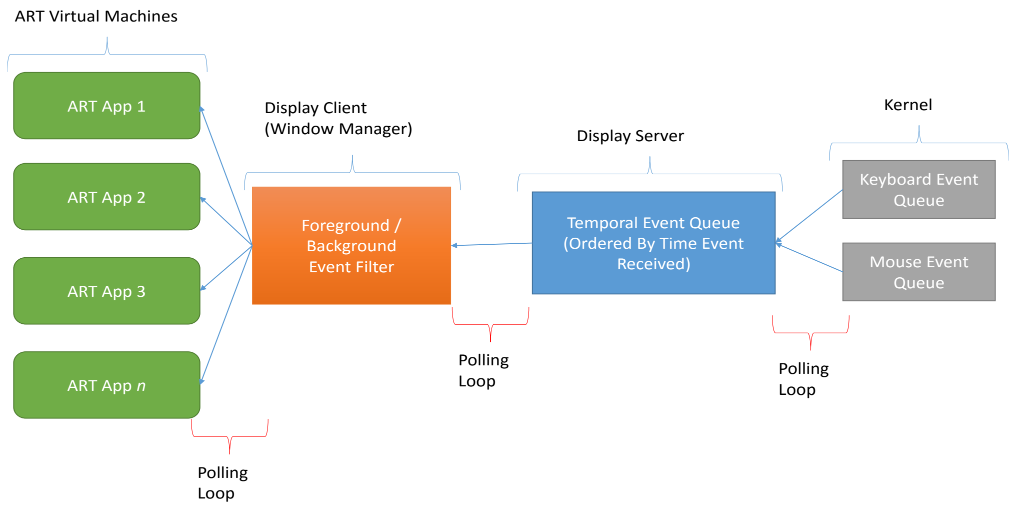


Figure 12: The current pull model stores events into several queues where it is propagated to the application which will handle the event. With the current GUI model, the event is filtered at several layers. This figure shows a breakdown of a typical event hub, which is a combination of the display server's temporal ordering and the display client's event filter, for GUI applications.

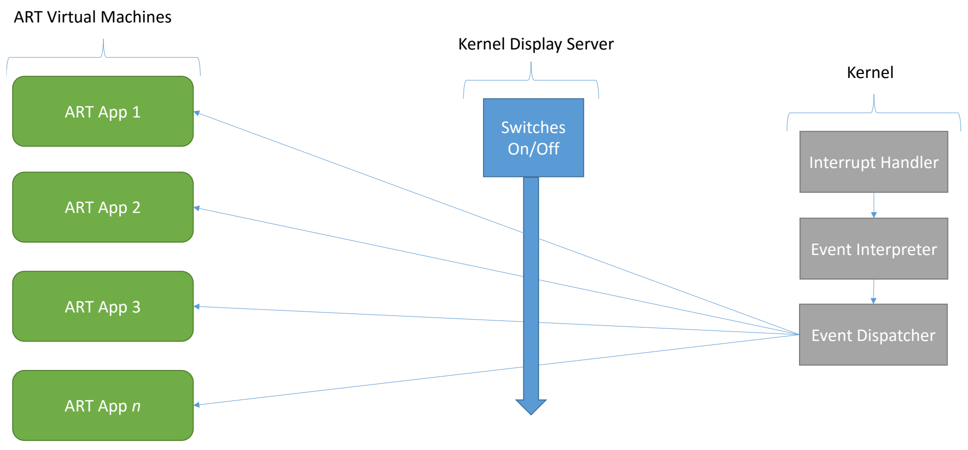


Figure 13: The KDS takes care of any type of event filtering by registering and deregistering event handlers with the ESM and provides a much more direct route for an event between the kernel and the application. Also, note that all polling loops are eliminated.

video frames, and (d) a foreground thread which runs even when the screen is turned off and which, for example, could allow audio decoding and playing to continue even when the screen is turned off. The event handling thread should execute event handlers that are primarily responsible for responding to hardware inputs and hence spend their life waiting for events. On the other hand, the screen drawing code will be constantly writing to the display, but it will not have to wait for events. It will occupy both CPU and I/O time. Typically, there is some sort of main program that will use the foreground and background threads to process input or output (e.g., code that decodes compressed video frames) and is typically CPU bound since it requires the processor to accomplish a certain task. The KDS prioritizes these types of processes by coordinating with the GUI scheduler to achieve a better use of a CPU's cores (see Chapter 5). Finally, the threads that the KDS creates are empty until the programmer specifies the code that should run in those threads. In other words, the programmer can decide to put whatever code they wish into whatever thread they wish. The names that we gave the threads are only recommendations.

By contrast, the traditional middleware display server has no control over scheduling an application based on the application's GUI state. Therefore, the scheduler must schedule the application to run even when it does not need to, such as when it is in the background. This performance hit is typically mitigated by having the operating system enforce aggressive sleeping policies, but such policies do not completely eliminate unnecessary scheduling and can also have a negative impact on latency. Aggressive sleeping policies contribute to latency because the policy might start powering down the device before the user has a chance to make further inputs. For example, if a user is reading a text message, the screen will dim. Depending on the sleep policy, if the user doesn’t react quickly, the screen will shut off. Then, in order to resume reading the text message, the user will have to awaken the entire device. However, the increased latency is apparent even with faster mobile processors and hardware devices when waking the device (Chauhan, Sammakia, & Ghose, 2015).

The shorter event path, improved event coordination, reduction in system calls, and improved scheduling afforded by the KDS provide a reduction in power consumption and a reduction in latency at the cost of reduced flexibility, as compared with middleware display servers. However, even though the KDS is slightly less flexible than the traditional display server, it still uses portable kernel routines, which allows it to be reasonably portable between architectures.

The rest of this chapter describes how applications connect to the KDS, how the KDS coordinates with the ESM and scheduler, and how the KDS is implemented.

Applications Running on the KDS

Generally, an application attaches itself to the kernel display server through the device file system common in UNIX-style operating systems, including Android. An application that wishes to communicate with the KDS first opens a control device called the “kds_control_device” and attaches to it (see Figure 14). If the KDS is able to accommodate the request, it returns a new, private “communication” device specifically for that application, which then becomes the main communication pathway between the

KDS and the application. This ensures that a malicious application cannot commandeer other applications' requests to the KDS. While the communication protocol between the control device and private device are complex and esoteric (see Figure 15), we have developed a user-level API that handles encoding and decoding these messages to the control device. Our API is built into Android's virtual machine, so application programmers do not have to initiate any of this communication, but instead, it is completely handled by Android.

Linking Android with the KDS

Android's GUI system contains a software service called *Surface Flinger* which is responsible for drawing surfaces to the screen ("Android's Surface Flinger," 2015). A sub-service in the Surface Flinger service is called the *hardware composer*, which is responsible for allocating and layering the many GUI components into a single surface as well as overlaying window decorations, such as the battery status icon and clock. The KDS embeds itself between Surface Flinger and the hardware composer and between Surface Flinger and the kernel. Therefore, there are two separate places where the KDS coordinates the actions of Android's drawing system.

When an application wishes to draw, it performs the same actions that it would with Android's current drawing system. Since the KDS is embedded in Surface Flinger, it intercepts the screen drawing commands, coordinates with the kernel, and ultimately writes to the screen's framebuffer. In other words, Surface Flinger is drawing to a virtual screen (called a *surface*), which the KDS then draws to the actual screen after making

```
kds_connect(application) {
        KDS = open(kds_control_device)
        cdev = KDS.create_communication_device()
        cdev.attach(application)
        close(KDS)
        return cdev
}
```

Figure 14: An application first connects to the KDS through the kds_connect routine, which opens a central KDS control device located in the device filesystem. The KDS then creates a private communication device specifically for the application connecting to the KDS. All further communication between the KDS and the application is through the new, private communication device.

```
KDS_COMMAND
{
        Integer request_type
        Integer data_length
        Blob data
}
```

Figure 15: The C-style structure that an application sends to its private KDS control device to get the KDS to execute a command on its behalf. The request_type is what command the application wishes to use. The "blob" data is simply a memory pointer that contains up to "data_length" bytes.

certain adjustments.

Putting the KDS between Surface Flinger and the hardware composer adds an additional layer between the application and the screen. However, by using the existing Android drawing system we allow the end-user applications to ignore which display system is being used by the mobile OS. In other words, current mobile applications would not need to be modified in order to incorporate the KDS' benefits. This allows for much more flexibility and a much shorter adoption period when mobile devices are upgraded to the KDS.

KDS Implementation

The KDS has several subsystems that are used to draw graphical objects to the screen and to coordinate with the ESM and scheduling routines. At the kernel level, the KDS includes three subsystems: (a) the compositor system, (b) the ESM/scheduler coordination routines, and (c) the drawing system.

The Compositor System

The KDS flattens layers of drawing objects through the use of a compositor. The compositor makes sure that the GUI buttons, menus, and so forth look like they are on top of a window pane. Furthermore, the KDS compositor uses a simple ordered list to determine how to layer objects into a single image (see Figure 16). The list stores all of the objects that need to be drawn and is sorted by an increasing "z-index" which is set by the applications programmer. This means the objects with a lower z-index are drawn first and the objects with a higher z-index are drawn last. Therefore, higher z-index surfaces "lay" on top of lower z-index surfaces.

The KDS compositor is rather simple in its design due to the fact that most GUI graphics packages, such as Android's Surface Flinger handle much of the compositing. However, the difference is that the KDS' compositor flattens the entire screen, including all GUI attachments, whereas Surface Flinger composes for each running application. In other words, the KDS determines how applications are layered on top of each other, and Surface Flinger determines how objects are layered on a single application. The KDS compositor should be set to run in the drawing thread since it is only necessary when the results can be seen by the user, however the programmer or middleware system must explicitly place the call to the KDS compositor in the drawing thread.

The Drawing System

The KDS drawing system is a low-level drawing mechanism that is called by the middleware drawing routine, such as Android's Surface Flinger, and is responsible for drawing GUI objects to the graphics framebuffer and runs after the compositor system has finished executing. Figure 17 depicts the KDS drawing routine. The routine sweeps through the flattened surface created by the KDS compositor and copies the bits to the framebuffer.

The KDS makes no automatic placement of the drawing code in the drawing thread. Instead, the programmer or middleware must ensure that they place the call to the KDS drawing system in the drawing thread. This is desirable since the KDS cannot

```
kds_compose(surface) {
    flattened_surface = create_blank_surface()
    //layers in the surface are sorted back to front
    foreach (layer in surface) {
        flattened_surface.draw(layer)
    }
    return flattened_surface
}
```

Figure 16: The KDS composer is a simple layer-flattening algorithm which flattens multiple-layered surfaces into a single layer surface (bitmap). Surfaces are allocated for each GUI decoration. For example, one surface is allocated for an application to draw to whereas another surface is allocated for the system to draw icons, such as the battery status icons. The composer ensures that the GUI decorations are appropriately placed so that it looks like a single contiguous picture.

```
kds_draw_2d(dev, flat_surface) {
    foreach (pixel in flat_surface) {
        framebuffer = GetFrameBufferForApp(dev)
        framebuffer[pixel.x][pixel.y] = pixel.rgb
    }
}
```

Figure 17: The KDS 2-dimensional drawing system draws a composed and flattened surface to the framebuffer. The drawing system uses the already existing framebuffer utilities in the Android kernel. The GetFrameBufferForApp is merely a helper function which returns the framebuffer that the application is allocated. In the actual kds_draw_2d code, if the pixel being written to (on the last line of the code) exceeds the bounds of the application’s window, then an error is thrown. Since the KDS coordinates with the SurfaceFlinger, such an out-of-bounds write should never happen. However, it is an additional check in case someone uses their own display manager on top of the KDS and does not provide due diligence to ensure that the surface drawing area will not intrude on another application.

predict every instance where the programmer wishes to use the KDS’ drawing system.

Example Application Interaction with the KDS

Figure 18 shows how an application would use the commands enumerated in the previous section to interact with the KDS in order to draw graphics to the screen. Figure 19 shows how an application would use ESM and KDS commands to set up event handling and to allocate code to different KDS threads. This figure should help the reader understand how the application layer interacts with the KDS layer to complete a drawing interaction. In practice, middleware would handle all of the interaction with the KDS and the application programmer would use the middleware’s drawing commands. Hence, existing Android applications can work with the KDS without any modification.

The KDS uses many of the DirectFB routines that are already written in the Linux/Android kernel. As previously mentioned, DirectFB merely provides helper functions to draw to the framebuffer using the hardware to improve the drawing speed. This allowed us to implement the KDS without having to duplicate DirectFB’s functionality.

The KDS is initialized after the framebuffer system and uses the first enumerated framebuffer as its drawing surface. This presents a drawback if the device is connected to an external display, since the KDS won’t recognize it. However, most mobile devices are not typically used in this manner, and therefore, the KDS is relatively safe in assuming the first framebuffer is the desired drawing target.

Figure 19 shows an example of how an application programmer might separate their code into the individual threads that the KDS allocates. When the programmer directs the KDS to run a function on a thread, the KDS first clears the thread of any existing code and then replaces it with the code specified by the function pointer. The KDS is then responsible for scheduling and executing the code on the threads. The four threads that the KDS automatically allocates may only be used by one task at a time. However, the application programmer might want to run multiple tasks on a single thread, such as decoding both audio and video on the background thread. In this example, the programmer would create a single function that forks two threads, one for the audio task and one for the video task. The programmer would then pass this function to the background thread via a KDS command. Since the function executes on the background thread, any threads that it forks would run on the background thread as well. Luckily, middleware can hide this messiness from the application programmer. For example, the programmer could register certain background tasks with the middleware and have the middleware marshal and fork a thread for each task. The middleware would in turn use the underlying KDS system and register this function with the background thread.

```
Main()
{
        Device kds_device // This app’s communication channel with the KDS
        KDS_COMMAND cmd

        //This block of code is kds_connect() manually performed by the app
        Device kds_control_device = open(“/dev/kds/control”)
        cmd.request_type = ATTACH
        kds_control_device.write(cmd)
        kds_device = cmd.data
        kds_control_device.close()

        Surface surf = Android.SurfaceFlinger.create_new_surface()
        Layer lay1 = surf.create_new_layer(0); // z-index 0
        Layer lay2 = surf.create_new_layer(1); // z-index 1
        lay1.rectangle(0, 0, 15, 15)
        lay2.rectangle(10, 10, 15, 15)

        cmd.request_type = COMPOSE
        cmd.data = surf
        cmd.data_length = surf.size
        kds_device.write(cmd) //calls kds_compose on the main thread

        //We now have two rectangles on a single surface,
        //lay2 on top of lay 1 (higher z-indices are closer to the user)
        FlatSurface flat_surface = cmd.data
        cmd.request_type = DRAW
        cmd.data = flat_surface
        cmd.data_length = sizeof(flat_surface)
        kds_device.write(cmd) //calls kds_draw_2d on the main thread

        cmd.request_type = DETACH
        cmd.data = NULL
        cmd.data_length = 0
        kds_device.write(cmd)

}
```

Figure 18: This example pseudocode shows an application attaching, interacting, and detaching with the KDS by drawing two 15x15 squares. There is a 5x5 pixel overlap of layer2 on top of layer1. Therefore, during composition, a bottom-right, 5-pixel square of layer1 will be obscured and overwritten by the pixel values of layer2. This figure is meant to help the reader understand how the application layer interacts with the KDS. In practice these commands would be in a middleware package, such as Android’s Surface Flinger and the application programmer would not have to worry about writing these commands.

```
Device kds_device // This app's communication channel with the KDS
KDS_COMMAND cmd

Keyboard_Callback(event)
{
        Log("Got keyboard input " + event.keyCode)
}
Drawing_Thread()
{
        //Insert code like Figure 18 here
}
Background_Thread()
{
        //Perform actions, such as decoding audio
}
Main()
{
        //This block of code is kds_connect() manually performed by the app
        Device kds_control_device = open("/dev/kds/control")
        cmd.request_type = ATTACH
        kds_control_device.write(cmd)
        kds_device = cmd.data
        kds_control_device.close()

        //Register keyboard events with the ESM
        //This is registered to the event handling thread, so all events
        //are pushed there.
        cmd.request_type = ESM_KEYBOARD_EVENT
        cmd.data = Keyboard_Callback //Pointer to the event handling function
        cmd.data_length = sizeof(Keyboard_Callback)
        kds_device.write(cmd) //Calls esm_register on the event handling thread

        //Tell the KDS to run function "Drawing_Thread" on the drawing thread
        cmd.request_type = DRAWING_THREAD
        cmd.data = Drawing_Thread //Pointer to the function to run
        cmd.data_length = sizeof(Drawing_Thread)
        kds_device.write(cmd) //Runs Drawing_Thread on the drawing thread

        //Tell the KDS to run function "Background_Thread" on the background thread
        cmd.request_type = BACKGROUND_THREAD
        cmd.data = Background_Thread //Pointer to the function to run
        cmd.data_length = sizeof(Background_Thread)
        kds_device.write(cmd) //Runs Background_Thread on the drawing thread

        //Close the connection to the KDS
        cmd.request_type = DETACH
        cmd.data = NULL
        cmd.data_length = 0
        kds_device.write(cmd)

}
```

Figure 19: This example pseudocode shows an application separating its code into three of the individual KDS threads (event handling, background, and drawing are shown above). For event handling, the KDS duplicates the esm_register routine so that it can automatically register and de-register ESM events depending on the foreground or background state of the application.

Chapter 5

The GUI Scheduler

Most desktop OS schedulers are unaware of what an application is doing, and thus are unable to intelligently schedule how that application runs. This is desirable for desktop schedulers because it simplifies the process of running applications and because of the heterogeneity of applications that run on desktop computers. However, mobile devices have two factors that make it desirable to develop more sophisticated schedulers. First, they have a limited power source, which make it less appropriate to run a power-consuming application if it cannot perform useful work (e.g., a polling loop for a GUI application that is completely obscured by other applications). Second, most of the applications that run on mobile devices involve GUIs that lend themselves to more nuanced scheduling (e.g., we would rather not schedule applications that are not visible to the user).

Because the KDS separates the code for GUI applications into four distinct threads with well-defined task information and because the KDS is implemented in the kernel, it is particularly well-suited for assisting the scheduler with performing more intelligent scheduling. We have therefore developed a more sophisticated scheduler called guiS that takes advantage of this application-specific information. guiS has three goals: (a) coordinate process scheduling for GUI-specific situations, such as when an application is running but is not actively visible to the user, (b) reduce power consumption by improving the efficiency of hardware timer interrupts, and (c) relay information to the ESM so that it can make smarter power policy decisions, such as when to shut down the CPU cores. In particular, if all ESM processes are sleeping, the guiS artificially sets a timer that monitors the amount of time since the last user input in the dynamic power scaling subsystem to the timeout value. Setting the timer to this value effectively forces the dynamic scaling subsystem to think that no user input has occurred, which then starts the process of shutting down the mobile devices hardware elements, such as the CPU cores and LCD.

guiS is a modified version of the current “Completely Fair Scheduler” (CFS) system currently employed in the Android OS (Pabla, 2009). Much of guiS looks exactly like the CFS. In fact, when a running thread is not associated with a GUI application (e.g., a console application), then the thread is scheduled in accordance with CFS policy, meaning that the scheduler looks and acts like the current Android OS with non-GUI applications. Furthermore, if real-time scheduling is desired, guiS’ modified scheduling algorithm can be completely disabled, in which case scheduling reverts to the original CFS behavior. Of course, the performance benefits associated with the modified algorithm are also lost when guiS is reverted back to its compatibility mode.

guiS incorporates several improvements that Hsiu et al. (2016) made in their heterogeneous CPU core scheduler. Their work determined which proportion of shadow cores to normal cores are best for balancing performance and power consumption in GUI applications. We take this result from their scheduler and advance it by allowing guiS to coordinate with the ESM in order to identify which applications need to run. When the scheduler determines that no GUI applications need to run, like when the user places their phone in their pocket, it triggers the power subsystem in the Android kernel to power down the CPU and other peripherals. It also causes the ESM to reprogram the vectored interrupt controller to target the shadow core for future events. Since the shadow core is the first line of event processing, the ESM (running on the shadow core) can then make the decision whether or not the power-hungry CPU cores need to be awoken (see Chapter 3).

Coordinating Processes for GUI-Specific Situations

The KDS divides an application into four threads: the event handler thread, the drawing thread, the foreground thread, and the background thread. Based on its knowledge of what each thread accomplishes, the KDS provides hints to the scheduler (see Figure 20), and then the scheduler schedules the threads as listed in
Table 1. Figure 21 shows a pseudocode example of how the guiS schedules each thread. All four threads abide by the same scheduling rules that normal process do, namely that a sleeping process will only be set to run when the sleeping condition is resolved. For example, if the programmer explicitly calls sleep() in the drawing thread, the drawing thread will sleep for the desired amount of time, regardless of whether or not the screen needs to be redrawn.

The *event handling thread* contains the event handlers registered by the application via *esm_register* calls and handles user input, wi-fi traffic, and so forth. The scheduler will only run this thread when an event occurs which is pushed to an event handler by the ESM (see Figure 22). When the event handler finishes, the thread is put back to sleep and is left undisturbed until another event occurs or until the application exits. If the application is placed in the background, then the KDS deregisters the ESM event handlers that handle input events or other events that should be ignored while the application is in the background. For example, if there is a mouse handling routine, the KDS knows that if this application is in the background, it cannot receive mouse inputs, and hence it deregisters any event handlers that are listening for mouse events.

The *drawing thread* is only scheduled to run when the application is visible to the user (see Figure 23). For example, if the LCD is turned off for any reason or the application is minimized, the drawing thread is suspended. Most of an application’s drawing routines should be placed in the drawing thread for efficient power management.

The *foreground thread* is scheduled to run when the application is in the foreground (see Figure 24). It runs regardless of whether or not the application is visible to the user. For example, if the LCD screen is turned off, but the application is in the foreground (i.e., the active application), the foreground thread is scheduled to run. For example, a video player app, where the programmer wishes to play the audio when the screen is turned off but wants any output suspended when other apps are activated, would place the audio decoding and output code in this thread.

The *background thread* is scheduled to run regardless of the application’s status (see Figure 25). Programmers should place in the background thread any routines that must run regardless of whether or not the application is currently interacting with the user. Typically, non GUI related code would be placed in this thread. For example, the

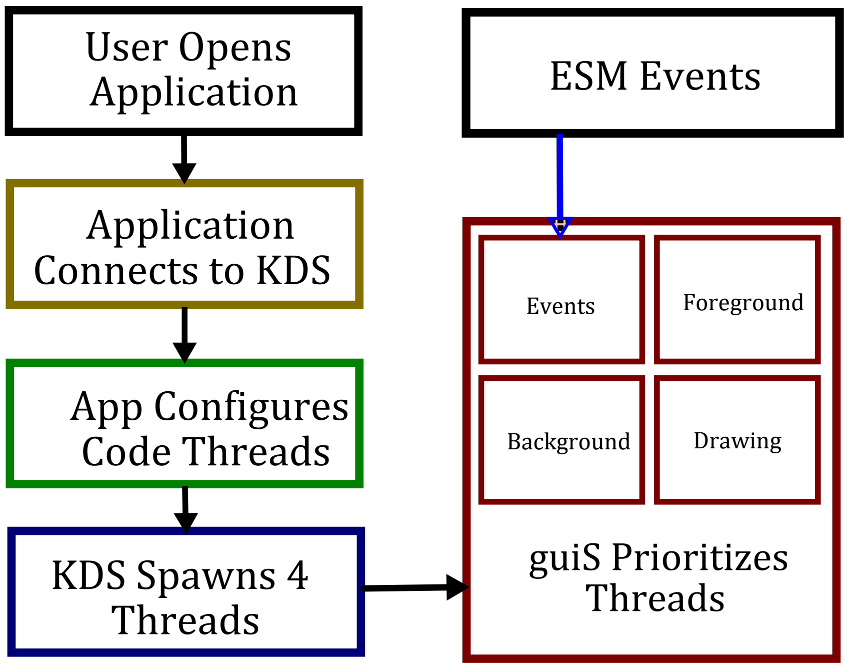


Figure 20: The application connects through the KDS directly or through middleware. The application then tells the KDS to which of the four threads certain code belongs. From there, the GUI scheduler will prioritize the threads and schedule them to run based on their category.

Table 1: How threads are prioritized based on the category of the running thread. We make a distinction between a visible app and an app in the foreground. For instance, if the LCD is turned off, the app is still in the foreground, but is not visible. Furthermore, when a window decoration covers the top-level application (e.g., the settings scroll), the app is not visible, but is still in the foreground.

Process Category	***Scheduler Runs This Thread***
Event Handling Thread	When event is received from the ESM
Drawing Thread	When app is visible to the user
Foreground Thread	When app is in the foreground
Background Thread	Always (foreground and background)

```
guis_schedule() {
    process = process_list.next()
    switch (process.thread_type) {
        case EVENT_HANDLING_THREAD:
            guis_schedule_event_handling(process)
        case DRAWING_THREAD:
            guis_schedule_drawing(process)
        case FOREGROUND_THREAD:
            guis_schedule_foreground(process)
        case BACKGROUND_THREAD:
            guis_schedule_background(process)
        default:
            schedule(process) //revert to non-categorical, CFS mode
    }
}
```

Figure 21: guis_schedule() is called when a context switch is requested through a hardware timer "tick" or when a process yields to the scheduler. The scheduling algorithm determines which type of thread is running (e.g., event handling, drawing, foreground, or background thread) and schedules it accordingly. If the process is not a GUI process as in the default case, the scheduler schedules the process normally in the Completely Fair Scheduler mode.

```
guis_schedule_event_handling(process) {
    if (process.state != EV_WAIT)
    {
        process.run()
    }
}
```

Figure 22: When scheduling the event handling thread, this function ensures that events need to be processed, which the ESM automatically does by setting the process' state to RUNNING. If the event handling thread is not in the EV_WAIT state, that means that the event handlers are running, thus the scheduler allows the process to continue to run.

```
guis_schedule_drawing(process) {
    if (process.thread_state = FOREGROUND and CAN_SEE(process)) {
        process.state = RUNNING
        process.run()
    }
}
```

Figure 23: The drawing thread only runs if the application is in the foreground and can be seen by the user. The macro CAN_SEE() is used to check if the LCD screen is on or off. If the LCD is on, then it stands to reason that any drawing could be seen by the user, and hence it is necessary to draw to the screen. Otherwise, this scheduling algorithm keeps the thread in a sleeping state.

```
guis_schedule_foreground(process) {
    if (process.thread_state = FOREGROUND) {
        process.state = RUNNING
        process.run()
    }
}
```

Figure 24: An application in the foreground thread will only be scheduled to run if the application is in the foreground. It does not check whether or not the results would be visible to the user (i.e., if the LCD is turned on or off).

Facebook application would place the routines which retrieve notifications for the user in this thread.

The KDS coordinates with the ESM and the process scheduler by updating the status of each GUI application. For example, the KDS is notified whenever an application moves from the foreground (visible to the user) into the background (not visible to the user). In this case, the KDS will relay to the scheduler that the application's drawing thread and foreground thread should be suspended since an application in the background cannot possibly draw to the screen. These threads will not execute until the application's GUI state changes, which means that they will not require the CPU, and hence reduce power consumption.

The KDS will also deregister the application's event handlers from the ESM so that it will not try to forward events to the application. When the application returns to the foreground, the KDS will re-register the application's event handlers with the ESM.

Lastly, when an application is moved to the background or the foreground, either because of a user action or because of an API command written in the program, the window manager coordinates with Android's Input Flinger, which ultimately calls kds_on_background or kds_on_foreground as shown in Figure 26 and Figure 27, respectively. Through these two KDS commands, the threads are either started or stopped based on the new state of the application. The guis_set_state function that kds_on_background and kds_on_foreground refer to are depicted in Figure 28. This function is necessary to update the current state of the GUI application (i.e., if the application is in the background or in the foreground).

```
guis_schedule_background(process) {
        process.state = RUNNING
        process.run()
}
```

Figure 25: Any process in the background thread is scheduled to run regardless of its state. This is helpful for situations where a programmer still wants to use the CPU even if the results are not visible by the user.

```
kds_on_background(application) {
        foreach (handler in application.[event|foreground|drawing]_thread) {
                esm_register(application, handler.type, NULL)
        }
        guis_set_state(application, BACKGROUND)
}
```

Figure 26: The KDS is notified when an application is placed into the background when the window manager explicitly calls kds_on_background when the application is moved to the background. The KDS automatically deregisters the event handlers that cannot run when the application is in the background. Finally, the KDS notifies the GUI scheduler that the application was placed into the background.

```
kds_on_foreground(application) {
        foreach (handler in application. [event|foreground|drawing]_thread) {
                esm_register(application, handler.type, handler.address)
        }
        guis_set_state(application, FOREGROUND)
}
```

Figure 27: The KDS is notified when an application is placed into the foreground when the window manager explicitly calls kds_on_foreground, and hence the window becomes visible to the user. The KDS automatically re-registers the event handlers that were disabled when the application was in the background. Finally, the KDS notifies the GUI scheduler that the application was placed into the foreground.

<u>Shadow Core/Heterogeneous Core Computing</u>

Another benefit to adding the GUI scheduler to the KDS and ESM combination is that the event system can now take advantage of the shadow core technology. As we have previously mentioned, the shadow core is a lower-performing CPU core that shares cache and memory with the other CPUs, but only uses a fraction of the power.

The KDS can identify which applications are actively being used by the GUI. Therefore, it can send hints to the scheduler in order to force all processing to the shadow core. When no GUI applications are being used, such as when the device is idle in the user’s pocket, the KDS/guiS will move all event handling to the shadow core. This allows the main, power hungry CPU cores to maximize their sleep while the shadow core marshals any events.

When an event occurs, the vectored interrupt controller only awakens the shadow core. The shadow core then discriminates between those events an application is waiting for and those that no application is waiting for. If the shadow core determines that an event needs to be handled by an application, it will awaken the other CPU cores and resume the normal operating system of the mobile device. However, if no applications want to handle the given event, the shadow core silently discards the event, and puts the device back to sleep, thus using only a minimal amount of power.

<u>How Categorized GUI Threads Enable Smarter Power Policy Decisions</u>

Since guiS prioritizes threads based on their category, it allows the scheduler to allot CPU and I/O time based on what type of code is running on that thread. For

```
guis_set_state(process, state) {
        process.thread_state = state
}
```

Figure 28: guis_set_state simply sets the state of the GUI application. This is used by the individual scheduling algorithms when determining which threads to schedule. For example, if the state is set to BACKGROUND, meaning that the application is in the background, then only the background thread will be scheduled to execute. Note that thread_state and thread_type describe two different elements of the process. The thread_state indicates the thread's active state, such as being in the background or foreground. On the other hand, thread_type describes the job that the thread is performing, such as drawing or handling events.

example, the event handling thread only needs to run when an event is present, and thus the event handling thread is only scheduled when an event is pushed to the application.

The existing windowing system in the Android operating system cannot make use of the KDS's knowledge of what is running on each thread and hence uses a complicated system of "wake locks" and aggressive sleeping policies to reduce power consumption. A wake lock is a lock that an application programmer can set which prevents the CPU from entering a sleeping state regardless of what activities the applications are performing (Pathak, Jindal, Hu, & Midkiff, 2012). However, when used improperly, wake locks can unnecessarily keep the CPU at the highest power state, thus consuming an inordinate amount of power. While the guiS scheduler cannot completely eliminate wake locks, programmers can move the code requiring a wake lock into one of the prioritized threads, such as the background thread, rather than using a wake lock. Theoretically, this should eliminate the need to keep the CPU awake once the code finishes executing and the CPU can be automatically powered down thus reducing power consumption. In contrast, wake locks must be manually deactivated, and an inexperienced application programmer may forget to remove the lock, thus preventing the mobile device from ever entering a sleeping state, and hence needlessly consuming the mobile device's battery (Jindal, Pathak, Hu, & Midkiff, 2013).

Aggressive sleeping policies are the consequence of attempting to maximize the limited power source in mobile devices. All mobile operating systems must use some sort of sleeping policy in order to reduce power consumption by powering down several peripherals, such as the LCD screen or Wi-Fi card (Pathak et al., 2012). For example, when the LCD screen on a mobile phone dims and then turns off, that is due to a sleeping policy that prescribes that the screen will dim after a certain duration, and then turn off after a longer duration. Unfortunately, sleeping policies are reactive since they involve some sort of measurement of the last user interaction. The term "aggressive" applied to sleeping policies means that with mobile devices, the duration between stepping down a fully awake device to a sleeping device is significantly reduced. Furthermore, since a lag occurs before a sleeping device fully awakens, aggressive sleeping policies can increase application latency.

Our guiS solves many of the problems caused by aggressive sleeping policies by removing several of the situations that require them. By categorizing the type of work that is performed in each thread, the operating system knows what type of computation is occurring and hence can eliminate many of the guesses that are made by current sleeping policies. For example, activities on the background thread should not affect the display, and hence the LCD screen can be powered down when only the background thread is executing. For example, the background thread might start downloading updated news articles. The LCD is free to power down since this activity does not require the LCD screen. In the current implementation, the LCD screen must use an aggressive timer to determine when to power down since it does not know what the application is doing.

Improving the Efficiency of Hardware Timer Interrupts by Eliminating Polling Loops

Throughout this dissertation, we have indicated that the elimination of polling loops can reduce the power consumption of a mobile device. The specific mechanism by which this power reduction is achieved is by the improved scheduling of hardware timer interrupts. In this section, we discuss how we achieved this improved scheduling.

A kernel *tick* is a hardware timer interrupt that is used to perform many kernel-related routines, such as context switching, application timing, and updating the system clock (Tsafrir, 2007). A *periodic tick* describes a timer interrupt that occurs at a known frequency (Love, 2010). This frequency is set when the Android kernel is compiled and is typically set by the device manufacturer between 250 Hz and 1000 Hz. While periodic ticks are simple, they have a serious drawback: periodic ticks occur even when all of the applications are idle (i.e., when the tick would be unnecessary). Hence, periodic ticks unnecessarily use the CPU and prevent the CPU from ever entering a deep sleep since it is servicing the periodic ticks.

In order to mitigate the problem with periodic ticks, the Linux kernel can be configured as a *tickless kernel*, which means that rather than having ticks occur at a regular frequency as with periodic ticks, they occur at a variable frequency and they occur only when they are needed. Variable frequency timer interrupts are known as *dynamic ticks*, since the interrupt is dynamically scheduled to meet the demand of the operating system. Unlike periodic ticks, dynamic ticks use a programmable timer in order to dynamically schedule the next timed interrupt to the CPU. For example, when all applications are idle, the timer is programmed so that it never interrupts the CPU. Instead, the CPU is awakened only when a useful interrupt or event occurs, such as a finger tap or when the power button is pressed. This allows the CPU to sleep for much longer periods of time, and hence, the CPU consumes only a minimal amount of power (Pathak et al., 2012).

Since the ESM uses vectored interrupts to process events, it only requires that dynamic ticks be scheduled when events occur. Figure 29 depicts a simplified version of the kernel's tick scheduling algorithm in pseudocode. In contrast, existing pull models force a dynamic tick to be scheduled for each iteration of a polling loop. For example, if the polling loop delays for 16 milliseconds, the next dynamic tick must be scheduled at most 16 milliseconds in the future. By eliminating the polling loop, the CPU sleeps for a longer period of time, thus reducing the amount of power consumed by the CPU.

```
kernel_schedule_tick() {
    total_sleep_time = FOREVER
    foreach (process in process_list) {
        if (process.sleep_time < total_sleep_time) {
            total_sleep_time = process.sleep_time
        }
    }
    hardware_timer.set_period(total_sleep_time)
}
```

Figure 29: The kernel looks at the process' sleep times to determine when to schedule the next dynamic tick. If all processes have an indefinite sleep, then no hardware tick is scheduled. In this case, the only interrupt from the hardware would be an event, such as the user clicking the power button or finger tapping the LCD screen. This figure simplifies the processes for scheduling dynamic ticks. Rather than iterating through the entire process list every time a process changes state, the Linux tick scheduler incrementally updates the dynamic tick time each time an application goes into the sleeping state with the lesser of the current tick time or the length of the application's sleep segment.

Chapter 6

Testing and Results

This chapter presents the results we have obtained by testing our event stream model with the Android Lollipop 5.1 operating system. We used the NVIDIA Tegra TK1 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 hardware features that 1) make the event stream model, kernel display server, and GUI scheduler feasible, and 2) are increasingly appearing on other mobile hardware platforms. This board uses an ARM Cortex-A15 processor with the 4+1, shadow core technology. Our tests compared the power consumption and latency between our ESM/KDS/guiS model and the event pull model.

We created three applications in order to contrast power consumption and latency between the pull model and the push model. The first application is a gesture tracking application where we manually recorded and replayed the events that a stylus generated. This application tests the power consumption and latency in situations where the user is interacting continuously with their device. The second application is the Android built-in text-messaging program in which we generate random messages to and from a simulated respondent. The text messaging application contrasts the pull model and push model in a “bursty” event situation where a user occasionally engages in a flurry of activity while preparing a text message, but is spending the majority of his or her time reading text messages, thus leaving the device in an idle state. The third application is a video playing application. It plays a real video while statistics about power consumption and latency are recorded. This application tests the pull model and the push model in a situation where no events are being generated. In other words, this tests the event models when the event system is idle.

Methodology

In this section, we describe our methodology for measuring the power consumption and latency of applications. We begin by discussing how we measured power consumption and latency. We then describe the configuration we used for the pull model and the configuration we used for the TK1 board's CPU cores.

Measuring Power Consumption

We measured the NVIDIA TK1's power consumption device with a software voltage monitor using a similar to the one that Carroll and Heiser (2010) used in their power consumption monitoring apparatus. The voltage monitor recorded the voltage values from a particular section of the NVIDIA board referred to as "R5C11" in the TK1 schematic, which is a 0.005-ohm resistor across the input power. We then converted the results into milliwatts using Ohm's law:

$$mW = \left(\frac{V^2}{R}\right) \times 1{,}000$$

After setting up the power measuring equipment, we performed our tests with both Android's existing epoll pull model and with our new event stream model.

Measuring 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 (Salah, Manea, Zeadally, & Alcaraz Calero, 2011). The default scheduler for the pull model is called the Completely Fair Scheduler (CFS) which attempts to give an equal share of CPU time to each process (Wong et al., 2008). 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. 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 millisecond resolution. For our purposes, this provided us with sufficient 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 one for each cycle, which gave us the one 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.

Pull Model Configuration

We used Android's existing event model for our pull model tests. Android uses both the select and epoll methods for querying and for handling events. Our tests were confined to two event queues, the mouse and the keyboard, so the tests provide an accurate measure of power savings and latency reduction that can be achieved by eliminating the polling loop. It is possible that interrupt vectors could provide an improvement over epoll's event aggregation mechanism, since interrupt vectors deliver an event directly to the kernel without any querying. However, since we used only two queues, any advantage gained by interrupt vectors over epoll's event aggregation techniques should be minimal, and the primary power and latency savings should result from the elimination of the polling loop.

CPU Configuration

All of the power consumption and latency tests that we performed used the NVIDIA TK1 at its full configuration, meaning that we did not restrict the CPU cores or restrict the use of the shadow core. This allows us to observe what occurs when the pull model and ESM have full access to all of the device's facilities without imposing any artificial limitations.

Results

This section reports our power consumption and latency results using the testing configurations described above.

Tracking Program Overview

Our stylus tracking program displayed a spiral for the user to trace with their stylus (see Figure 30). The user moves the stylus along the spiral and alternately presses and lifts the stylus at random intervals. When the stylus was pressed, a blue star was drawn to the screen, and when the stylus was released, a red star was drawn to the screen. The purpose of this test was to simulate a user navigating their screen and clicking GUI interaction objects, such as buttons, images, or hyperlinks.

We recorded events for the test by manually tracing the spiral while randomly pressing the stylus and releasing the stylus. Then, we scaled all of the events to fit into a 10 second test (161.4 events per second), a 20 second test (80.7 events per second), and a 60 second test (26.9 events per second). We chose these timeframes in order to illustrate the efficiencies (or lack thereof) in the event model when handling rapidly occurring events versus infrequently occurring events. The tests were performed by first using the traditional pull model where all polling loop delays were set to 16 milliseconds. Then, we ran the tests again with the KDS/ESM/guiS model. Finally, the results were plotted together on a line graph.

Figure 31 and Figure 32 show the power and latency profiles for the tracking test, which was scaled so that all 1,614 events occurred within 10 seconds. This is important since the event queue is more likely to remain full, and hence the polling loop should not introduce delays. This shows the inherent latency within the polling loop and event system in a scenario where the polling loop is not likely to delay.

Figure 33 and Figure 34 show the power and latency profiles for the tracking test, which was scaled so that all 1,614 events occurred within 20 seconds. This test shows what happens when events occur with moderate frequency. When events are moderately frequent, the likelihood that the event queue becomes empty increases, and hence the likelihood that the polling loop introduces delays is increased, but is still not likely. In this case events are being generated at roughly 81 events a second and the polling loop is set to a frequency of 60 events a second, so occasionally an event may miss one of the 16 millisecond polling intervals since the events are not evenly spaced and hence the event queues will be occasionally empty.

Figure 35 and Figure 36 show the power and latency profiles for the tracking test, which was scaled so that all 1,614 events occurred within 60 seconds. This test shows what happens when events occur at a relatively slow rate thus causing the event queue to

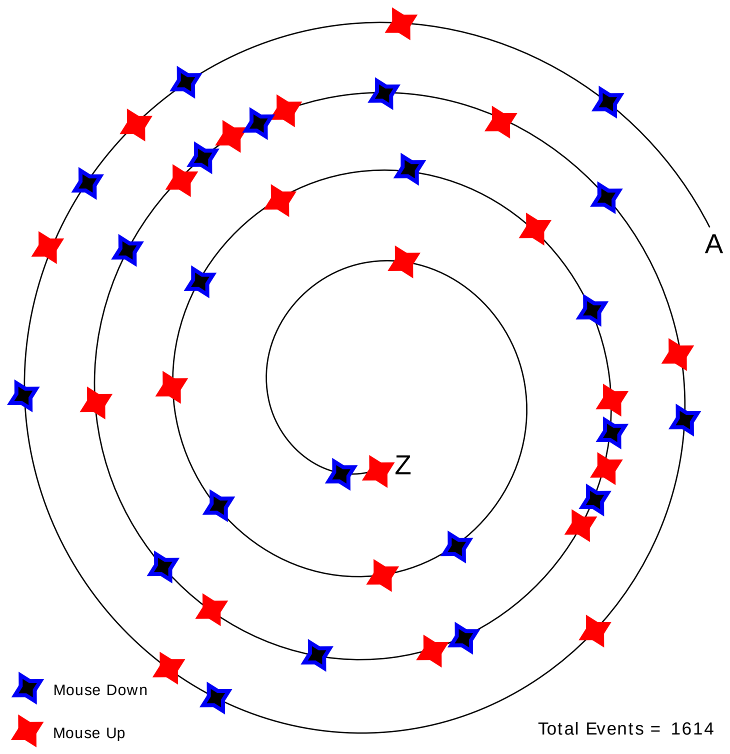


Figure 30: In the stylus motion/click test, the user started at point A and manually traced the spiral with random stylus presses (mouse clicks). From the starting point (A) to the ending point (Z), the entire test created 1,614 separate motion and click events. The blue stars represent when the stylus was pressed down, and the red stars represent when the stylus was released. The recorded events were then scaled to different time intervals to automatically generate events.

be frequently empty. In this scenario, the polling loop is likely to check the event queue when no events exist, which unnecessarily awakens the CPU, and hence, increases power consumption.

Tracking Program Analysis

The tracking program produced results that agreed with our expectations for the polling loop and its associated event pull model. When using a push model, our ESM/KDS/guiS system reduced power consumption over the pull event model in all three experiments. The 10-second test showed an average of 41.3 milliwatts (5.2%) reduction, the 20-second test showed an average of 117.9 milliwatts (15.3%) reduction, and the 60-second test showed an average of 184.7 milliwatts (22.9%) reduction. Furthermore, latency was also reduced by an average of 0.3 milliseconds for the 10-second test, 1.9 milliseconds for the 20-second test, and 6.7 milliseconds for the 60-second test.

Text Messaging Program Overview

Our second test involved Android’s text messaging application. Text messaging is a very popular way to communicate with mobile phones (Heyer, Brereton, & Viller, 2008). However, text messaging generates events slowly, and in fact, when the user is reading messages rather than writing them, no input events are being generated. Therefore, this test shows the amount of power that the pull model might use to service

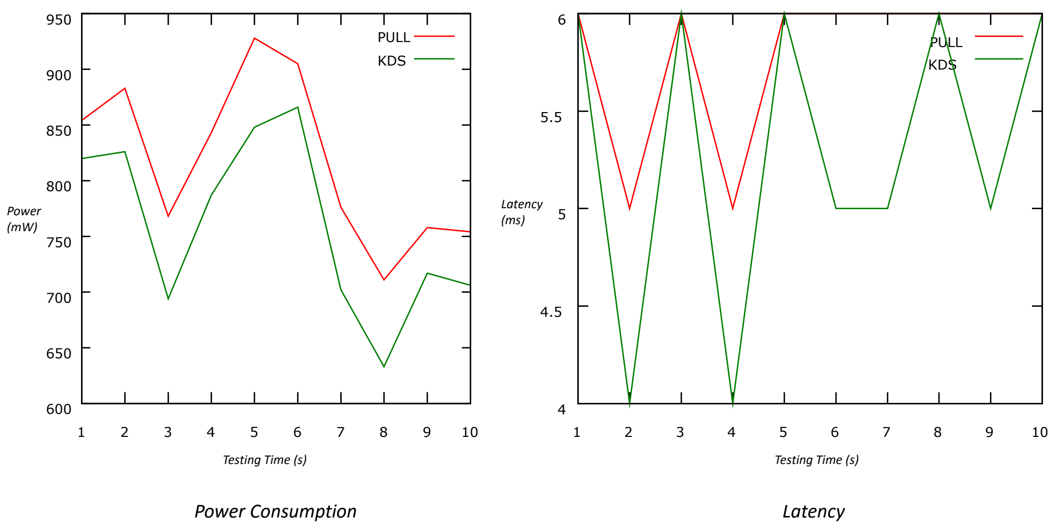


Figure 31: Power consumption and latency profile when the events for the spiral tracing program are condensed into a 10-second user interaction. This experiment models a situation in which events arrive frequently and the event queue is likely to remain full.

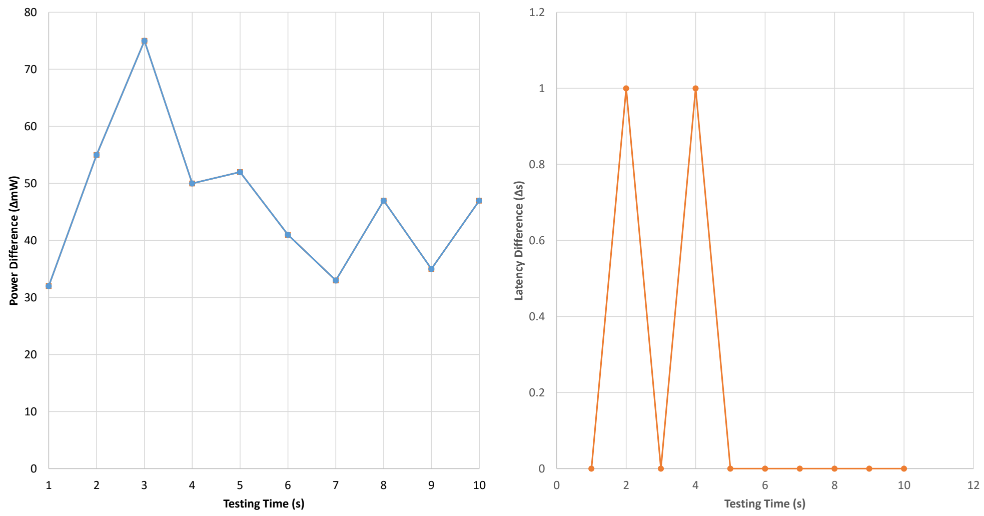


Figure 32: Power consumption and latency differences between the ESM/push model and the pull model for the 10-second spiral test. Positive numbers favor the ESM/push model.

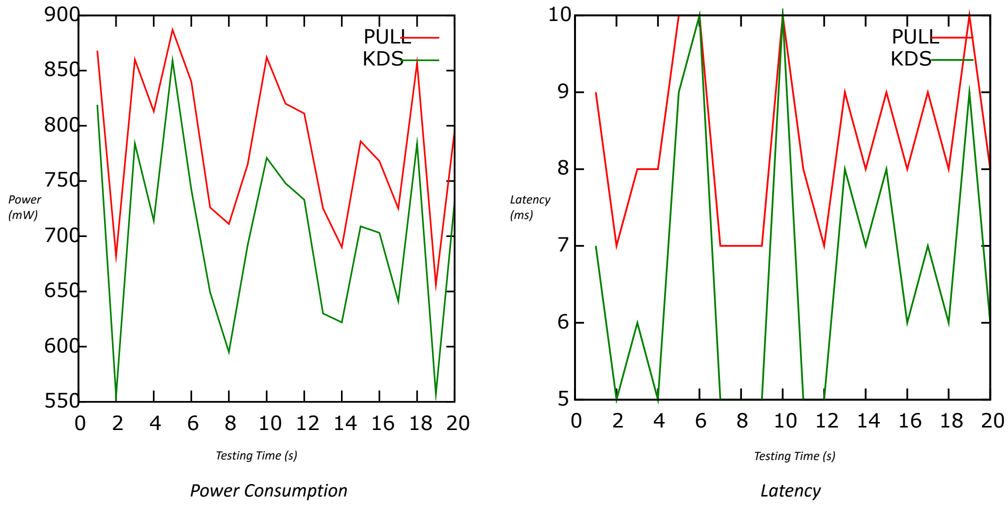


Figure 33: Power consumption and latency profile when the events for the spiral tracing program are condensed into a 20-second user interaction. This experiment models a situation in which events arrive moderately frequently and the event queue is likely to remain full but may occasionally become empty.

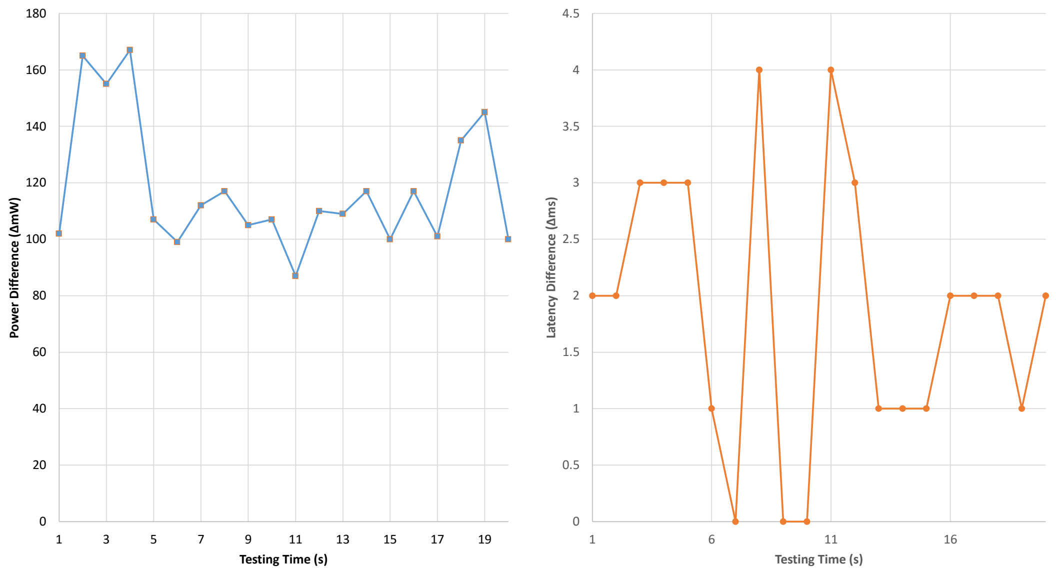
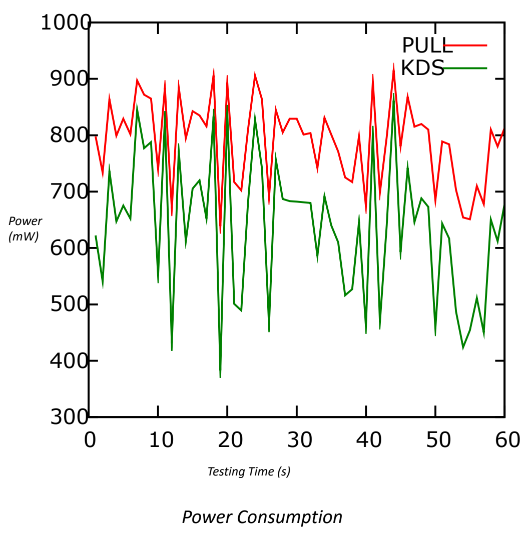


Figure 34: Power consumption and latency differences between the ESM/push model and pull model in the 20-second spiral test. Positive values favor the ESM/push model.



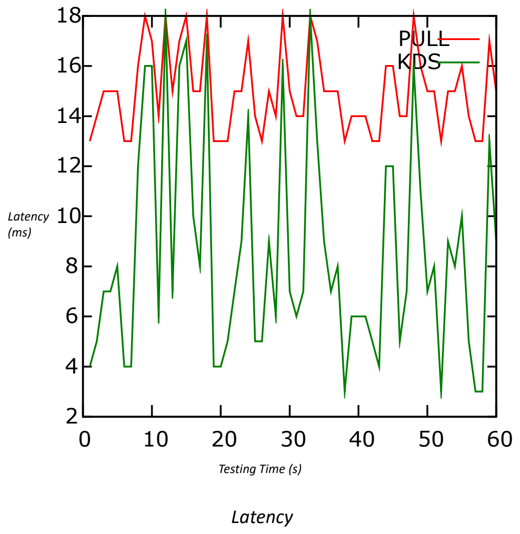
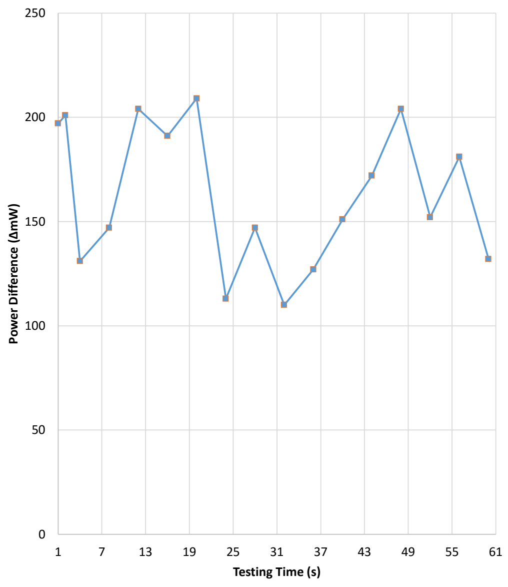


Figure 35: Power consumption and latency profile when the events for the spiral tracing program are condensed into a 60-second user interaction. This experiment models a situation in which events arrive do not arrive very frequently and the event queue is likely to become empty on many occasions



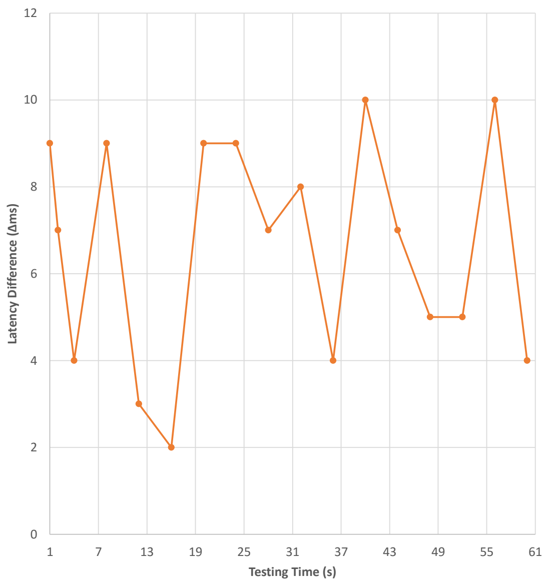


Figure 36: Power consumption differences between the ESM/push model and pull model in the 60-second spiral test. Positive values favor the ESM/push model.

its many event polling loops when an application is largely idle, and it also shows how our model eliminates this waste of power.

The text-messaging test uses a simulated keyboard and the standard Android text messaging service to generate random messages and send them to a server which simulates a respondent. The server then responds with a random message to simulate a reply. After the test program receives the server's response, it delays for 5 seconds without any interaction in order to simulate the user reading the text message that was just received. This process continues until 20 seconds of testing time has elapsed. With the five-second delay, we see an average of three messages being sent to and from the server for six total messages. See Figure 37 for a depiction of the testing program's process.

Text Messaging Program Analysis

Figure 38 and Figure 39 show that the polling loop causes the pull model to continually use the CPU in order to check the event queue. Our ESM event model's power consumption drops significantly by 218.4 milliwatts when the program enters the 5 second delay. Furthermore, our ESM/KDS/guiS also shows a reduction in power consumption compared to the traditional pull model during the periods when the application is handling events.

In addition to a significant reduction in power consumption, latency is also significantly reduced in this test by up to 17.1 milliseconds. The latency is significantly higher in the pull model due to the fact that the polling loop is very likely to sleep

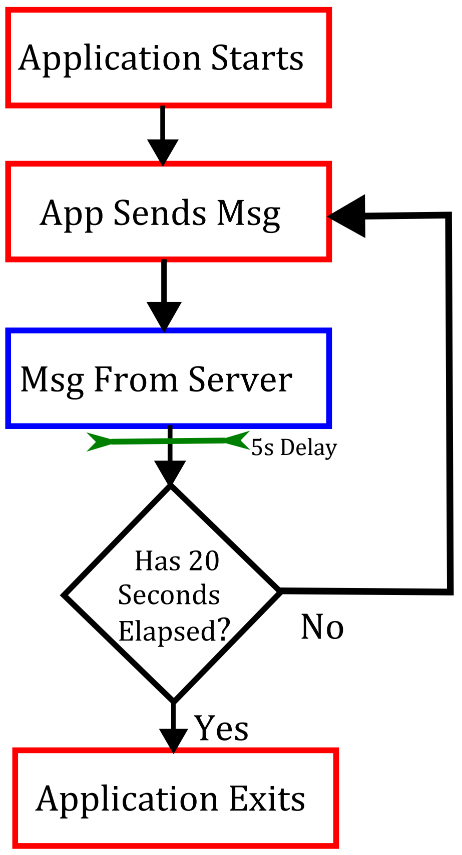


Figure 37: The text messaging program generates a random message and sends it to a server. The server immediately generates a response. Afterward, there is a 5 second delay to simulate the user reading the screen or where the user is not in a position to read their text message. Then the test is repeated until 20 seconds of testing time has elapsed.

because events seldom occur. Hence, many of the events occur while the polling loop is sleeping. Furthermore, the scheduling algorithms, display server, and other factors can also increase latency (as explained earlier in this chapter), which is why some latency savings exceed the 16ms frequency of the polling loop.

Video Program Overview

The final test we performed to compare the pull model and the ESM model was a video program that displayed a 30-second movie clip. This program was designed to test the ancillary tasks that are not necessarily GUI related, such as decoding the video in the background (in our push model, the decoding was done on the background thread). For the entire duration of this test, no events were handled, which allowed us to control for the event model and only compare our KDS and GUI scheduler with the current display server and scheduler. The movie clip was encoded using MPEG-4, Part 10 (AVC/H.264). It was 1920 pixels wide by 1080 pixels tall, 29.97 frames per second, and with a start to finish running time of 30 seconds.

Video Program Analysis

The power consumption results for the video program are shown in Figure 40 and Figure 41. We see an average reduction in power consumption of about 182.4 milliwatts using our display server and GUI scheduler, but there are several peak data points where our model reduced power consumption by nearly 400 milliwatts. The polling loop still shows an influence over power consumption, albeit to a much lesser extent when the

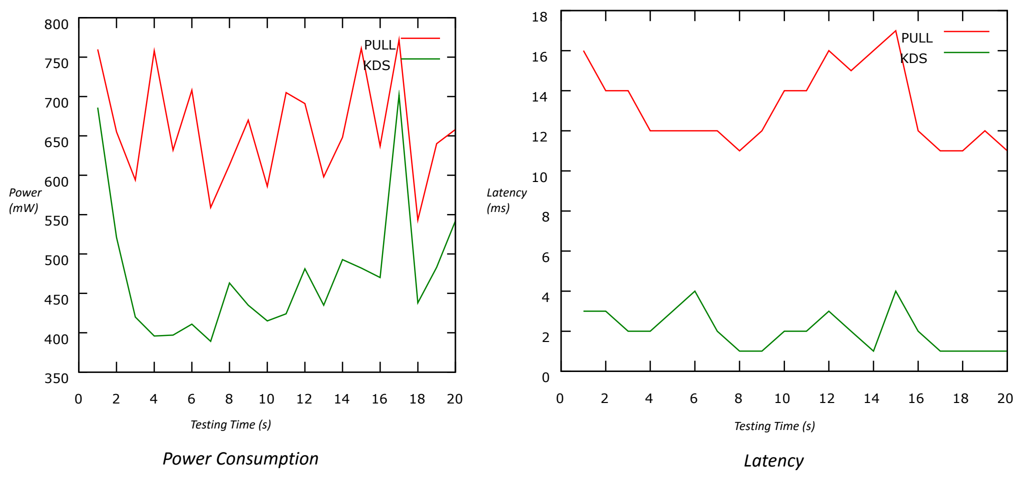


Figure 38: The power consumption and latency results of the text messaging service. The power numbers show the polling loop maintaining a relatively high power usage. The ESM shows a significant drop in power consumption when the testing program simulates the user reading the screen.

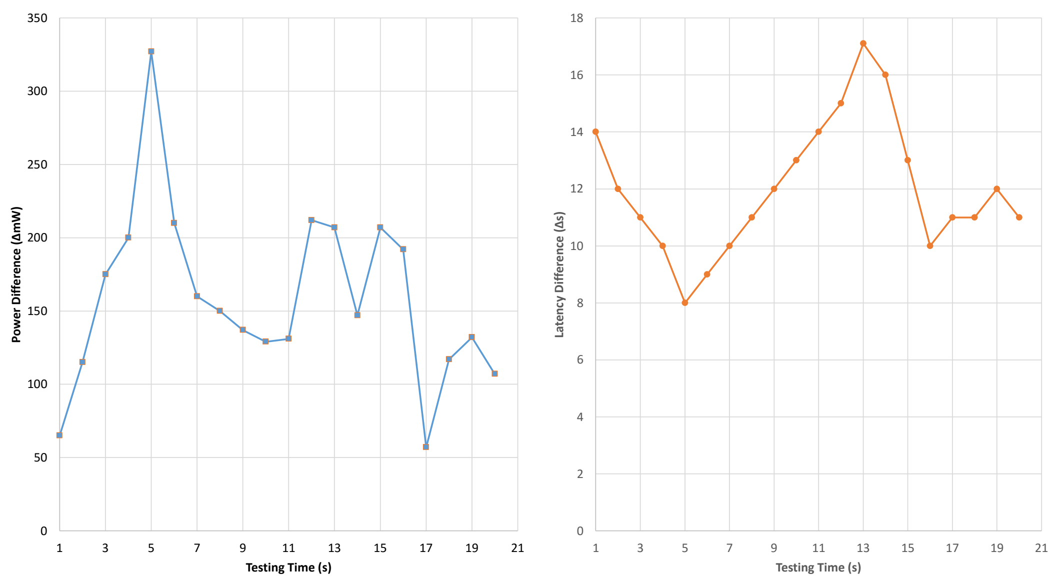


Figure 39: Power consumption and latency differences between the ESM/push model and pull model in the text messaging test. Positive values favor the ESM/push model over the pull model.

CPU cores are busy decoding and drawing video to the screen. The reason for the pull model's increased power consumption is because the CPU cores were taken off task from decoding video in order to service the event polling loops. The CPU must then catch up by decoding the video frames that were delayed while the CPU was diverting attention to the pull model event system. Upon further analysis, our testing shows that this perturbs cache locality since the program is required to compute two disjoint tasks (e.g., decoding video and not test latency since no events were generated during the test. Our latency testing apparatus tests the time between the generation of an event and the handling of and event, hence it was unusable when no events were being generated.

Summary

Our experiments show that our ESM/KDS/guiS combination reaps increasing power savings and latency reductions as the event frequency decreases and hence idle time increases. Figure 42 shows this relationship by plotting the reduction in latency and the reduction in power consumption achieved by each of the applications in our experiments. This chart shows that there is nearly a linear improvement in both power consumption and latency as event frequencies decrease and idle times increase in our experiments. Therefore, we are confident in concluding that our model should markedly improve those applications that spend an inordinate amount of time in an idle state and that have events arriving at irregular intervals.

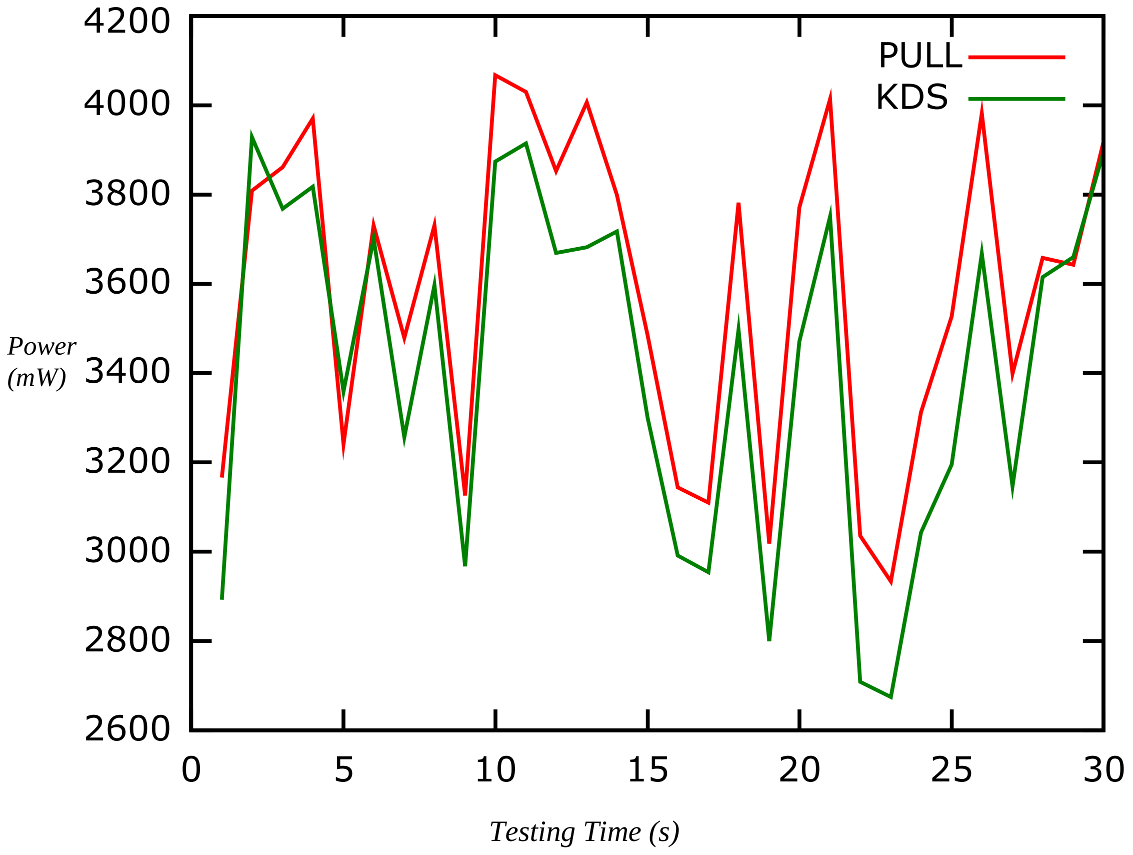


Figure 40: Power consumption results with the video program playing a 30 second video clip.

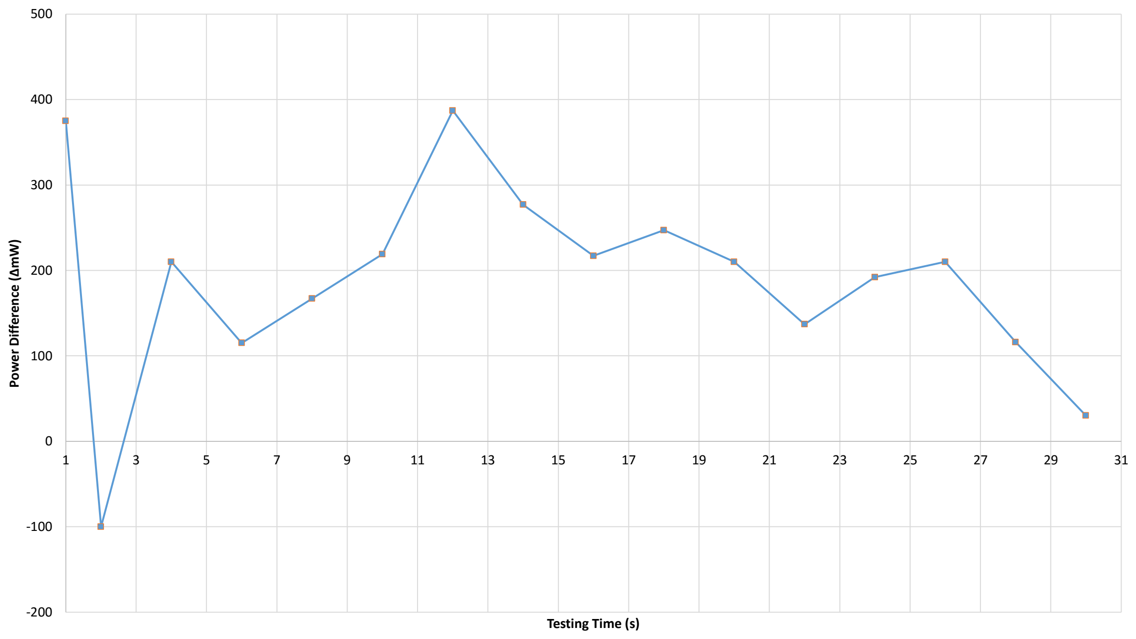


Figure 41: Power consumption differences between ESM/push model over traditional pull model for the 30 second video test. Positive values favor the ESM/push model and negative values favor the pull model.

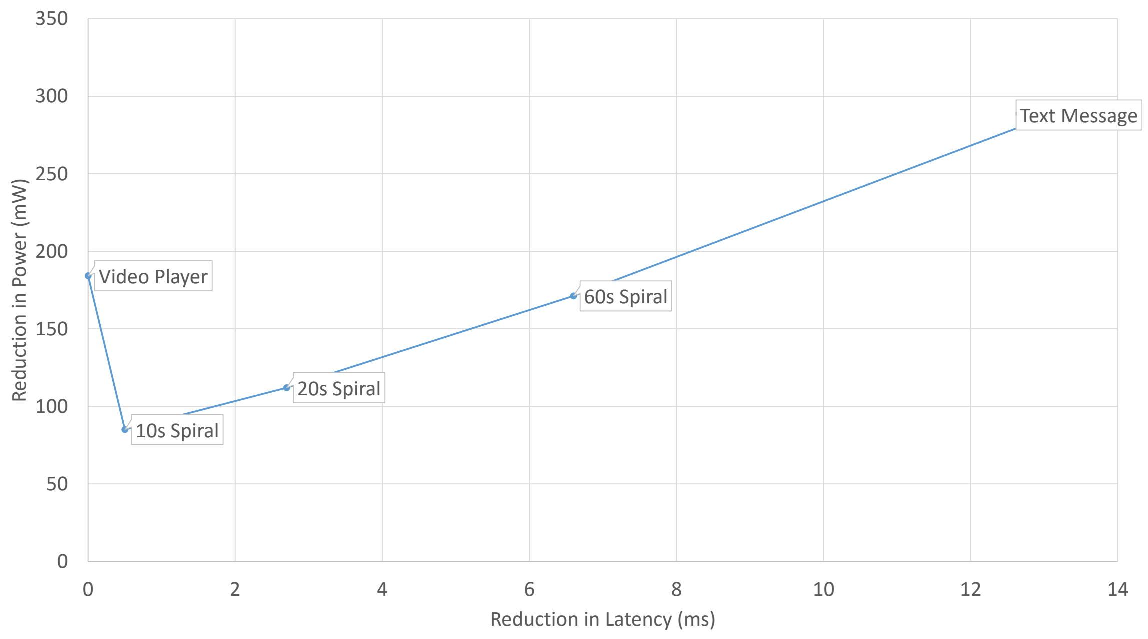


Figure 42: A 2-dimensional chart showing the power consumption reduction and latency reduction of the five testing programs used in our experiments. This chart shows the scenarios in which the ESM outperforms the current pull model. Since latency could not be tested with the video player, we set its latency reduction to zero.

In the two test cases that are event driven and require relatively little use of the CPU for computations (i.e., the stylus tracking and text messaging applications), our ESM/KDS/guiS combination significantly reduced power consumption and latency. Our ESM/KDS/guiS showed a maximal improvement of 184.7 milliwatts in power consumption reduction over the pull event model when the device was primarily idle. This confirms our hypothesis that polling loops will constantly rouse the CPU and hence prevent a mobile device from entering a deep sleep, thus consuming power unnecessarily. The ESM/KDS/guiS system also provides up to a 17.1 millisecond reduction in latency when the device is primarily running idle applications. This also confirms our hypothesis that the polling loop causes increased event handling latency when events occur while the polling loop sleeps. Furthermore, we see that even when the polling loop does not sleep, it still incurs a small amount of latency, while the ESM does not. We attribute this to the number of event queues in the pull model, to the number of polling loops, and to the overall inefficiencies in event "hopping" that occur in the pull model from the kernel to the display server, to the window manager, and to the application.

Lastly, our ESM/KDS/guiS system is optimized for situations in which the input events occur at irregular intervals or where input events are "bursty" (i.e., where input events cluster in a short period of time followed by extended idle periods). This phenomenon explains why the text messaging system shows the greatest improvement in power consumption and latency. Our push model is least likely to improve power consumption and latency when events occur at regular intervals, such as events from a temperature or velocity sensor. In these circumstances, the periodicity of the polling loop can be tuned to match the time intervals between events and the simplicity of the pull model might make it more attractive than our push model. However, as soon as irregular events are introduced into the mix, which happens in just about any mobile device that is not dedicated to sensor-like input, the push model is likely to provide the best performance.

Power Consumption and Latency Reductions Achieved by ESM Alone

When we first completed the ESM, and had not yet implemented the KDS and guiS processes, we tested it against two apps that were contrived, as opposed to the real world apps described above. This testing was done with a single CPU core since the benefits of multi-core scheduling could be achieved only when we had implemented the KDS and were able to make use of its GUI-specific knowledge. In this more restricted configuration, the ESM alone reduced power consumption by 23.8% and reduced latency by an average of 13.6 milliseconds over the current pull model on a gesture tracking app that was similar to the stylus tracking app described above (Marz & Vander Zanden, 2015). This result, combined with the result from our video decoding application which showcased the ability of the KDS and guiS processes to reduce power consumption and latency show that:

1) in low computation applications, most of the power consumption and latency reductions are attributable to the ESM, with further, albeit more moderate reductions achieved when the ESM was combined with the KDS and guiS, and
2) in high computation applications with no events, the KDS and guiS combination can achieve significant power savings.

Battery Life with the ESM/KDS/guiS Model

The results that we obtained show benefits in terms of power consumption and latency; however, mobile device users may be interested in how our solution improves the battery life of their devices. Using simple conversions and typical battery capacities, Table 2 shows the typical amount of extended battery time. Since the ESM/KDS/guiS model realizes about a 30% battery life improvement, we added the same percentage to the 1000 milliamp-hour (mAh) battery capacity. Due to increased internal resistances inherent in some batteries with higher capacities (Rong & Pedram, 2003), we used the findings by Hoque and Tarkoma (2016) and linearly scaled the ESM's benefit down to 25% for the 8000 mAh battery capacity.

Human Noticeable Latency Reduction

We achieved latency reductions of up to 17.1 milliseconds in our experimental applications. Dragging and scribbling allow the human brain to perceive latency on the order of one to two milliseconds and seven versus forty milliseconds while drawing to the screen (Ng, Annett, Dietz, Gupta, & Bischof, 2014). Therefore, our latency reductions could have a profound affect when the user is using a stylus and dragging across the screen or entering keyboard input.

In other contexts, a reduction of up to 17.1 milliseconds in latency is not by itself sufficient to be perceptible to a human. However, if this latency reduction decreased the overall latency of a response below a certain threshold perceivable to humans, then it could make a difference. For example, latency exceeding 150 milliseconds is noticeable to human vision (Jensen, 2006). If our ESM/KDS/guiS system reduces latency from 160 to 143 milliseconds (160 minus 17), then what was previously perceived as latency by a human might no longer be perceived as latency with our system's reductions.

Table 2: The battery capacity is a sampling of typical lithium ion batteries in both smartphones and tablet devices. However, since the TK1 is a development board, it does not use batteries. Therefore, we used the power consumption numbers and converted them to battery capacity. The battery life numbers are split between *talk / standby* (i.e., busy/idle) and are represented in hours for a typical 4G setup. **NOTE**: This table does not take into account user habits or every device configuration and is only a brief summary of what could be expected using our model with varying battery capacities.

Battery Capacity (mAh)	***Life w/ Pull Model (hours)***	***Life w/ ESM/KDS/guiS (hours)***
1000	3 / 100	4 / 150
2000	6 / 250	8 / 300
4000	10 / 500	13 / 700
8000	15 / 750	19 / 975

Chapter 7

Conclusions and Future Work

Mobile operating systems employ a myriad of techniques to ensure that an idle device does not drain the battery. 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 and prevent the CPU from entering lower power states. 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.

The event stream model (ESM) described and implemented in this dissertation takes a novel approach to event handling by pushing events from hardware devices to the application and thus eliminating the polling loops used by the event pull model. The elimination of these polling loops reduces the device's power consumption while the device is idle, and allows it to enter a deep sleep state that cannot be attained when polling loops are present. 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 kernel display server (KDS) extends the benefits of the ESM by moving the display server from the application layer to the kernel and thus making available to the scheduler information that can improve the scheduling of GUI applications. The KDS serves as a “gatekeeper” that controls when the ESM event model pushes GUI events to the applications and it also logically separates GUI code into four threads (events, drawing, foreground, background). These four threads can be scheduled to run or not to run depending on the state of the GUI application (i.e., if the application is visible, in the foreground, or in the background).

The graphical user interface scheduler (guiS) is a sub-component of the KDS and is responsible for scheduling or de-scheduling the KDS’s four threads. The KDS tells the guiS when an application changes state, and the guiS responds by changing how and when the GUI application is scheduled to run. For backward compatibility, if the application is not a GUI application, the guiS reverts to “compatibility mode”, where applications are scheduled according to the current completely fair scheduler (CFS) algorithms.

Our experiments have shown that the combination of the ESM, KDS, and guiS reduces the power consumption of certain apps by up to 31.2% and reduces their latency by up to 17.1 milliseconds when compared with the current pull event model. In low computation environments with irregularly occurring events, the Event Stream Model’s removal of polling loops is the main contribution to this power consumption and latency reduction. The Kernel Display Server makes further modest power consumption and latency reductions by coordinating with the GUI Scheduler. In high computation environments with few or no occurring events, the KDS and GUI Scheduler make the main contributions to power consumption reduction by allowing the CPU to remain focused on one task rather than having to context switch to polling tasks. The ability to stay on task both reduces memory cache misses and keeps the CPU from having to go to higher power states to make up for “lost time”.

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

Future Work

Mobile device developers could potentially enhance the ESM/KDS/guiS system by modifying several aspects of their virtual machines, such as Java's VM or frameworks such as .NET to use such a push model. Applications written on top of these VM's or frameworks would not need to be changed in order to benefit from the new event model and display server.

There are a number of interesting extensions of this work. First, the ESM/KDS/guiS system does not currently handle direct rendering (DRM) or 3-dimensional graphics since it does not implement any Embedded Graphics Library (eGL) routines, and it does not intercept DRM memory map requests; thus, the benefits of the KDS are not available to these types of applications. Future research in this area could potentially extend the efficiency of the KDS to 3D applications or applications that utilize direct rendering.

Second, our entire research was based on the Android operating system, primarily since it is open source and readily available to us at no cost. However, other mobile operating systems, such as iOS and Windows Mobile, may benefit just as much as Android does with the ESM. We further hypothesize that other such improvements might be made around the ESM model. For example, I/O scheduling or memory scheduling algorithms could be examined to determine if there is any improvement to be made based on the precepts of the ESM. For example, when a kernel driver makes an I/O request or memory allocation request, the kernel currently can put the driver into a “spin locked” mode, where the lock is continually polled to determine if it has been released or not. However, in these circumstances, an ESM-like notification model that pushes a lock release notification to code that is locked might prove more useful.

Third, hardware device manufacturers might design their hardware to incorporate our ESM in order to reduce power consumption and latency. For example, cell phone (GSM/4G) and wi-fi peripherals consume an inordinate amount of power, but also the amount of data processed by these devices is a large contributor to power consumption. Our push model could remove the network polling loops (i.e., those loops that search the socket buffer queues for data on a network data socket) and replace them with a push model, like our ESM. In other words, our ESM would reduce the power that an application would otherwise use polling the cell phone and wi-fi devices.

Fourth, many mobile devices include some type of cellular device for voice communications and data communications. A cellular device is forced to search for the strongest signal from the available cellular towers. To perform this operation, the cellular device “pings” the cellular tower and decides which one has the strongest signal. This in essence is a “pull” model where the cellular device “pulls” information from the tower. However, future research in this area might be able to remove this constant “pinging” and implement a push model of some sort.

Lastly, our ESM implementation was limited to mouse, gesture, and keyboard events. However, there are many other input peripherals with which mobile devices might be equipped. One obvious example is voice input. Since the underlying hardware implementation for many such peripherals, including voice, is not trivial or standard, and since mouse, keyboard, and gesture events are still the workhorses of most GUI apps, we chose to focus on those three types of events for this dissertation. However other peripherals such as voice would be an interesting focus for future work.

Since reducing power consumption will be an endless crusade for mobile devices, it is important to look at both software and hardware innovations that could lead to power savings. This dissertation has shown how a kernel implementation of a push event model for GUIs can save power and reduce latency. Its effectiveness at reducing power consumption suggests that both hardware and software developers might wish to consider using it both in future kernels and in future hardware designs.

References

Alagöz, I., Löffler, C., Schneider, V., & German, R. (2014). *Simulating the Energy Management on Smartphones Using Hybrid Modeling Techniques.* Paper presented at the 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.

Anderson, S., Mor, R., & Coopersmith, A. (2002). X Transport Interface *X Consortium Standard* (pp. 22). Dayton, OH: NCR Corporation.

Android's Surface Flinger. (2015). Retrieved from http://source.android.com/devices/graphics/architecture.html#SurfaceFlinger

Android Input Pipeline. (2015). Retrieved from http://source.android.com/devices/input/overview.html

Bhadauria, M., & McKee, S. A. (2008). *Optimizing Thread Throughput for Multithreaded Workloads on Memory Constrained CMPs.* Paper presented at the Proceedings of the 5th Conference on Computing Frontiers, New York, NY, USA.

Bhatt, V., & Huang, C.-C. (2010). *Group mutual exclusion in <i>O</i>(log <i>n</i>) RMR*. Paper presented at the Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, Zurich, Switzerland.

big.LITTLE Technology. (2016). Retrieved from http://www.arm.com/products/processors/technologies/biglittleprocessing.php

Brotherton, H., Dietz, J. E., McGrory, J., & Mtenzi, F. (2013). *Energy overhead of the graphical user interface in server operating systems*. Paper presented at the

Proceedings of the 41st annual ACM SIGUCCS conference on User services, Chicago, Illinois, USA.

Bui, D. H., Liu, Y., Kim, H., Shin, I., & Zhao, F. (2015). *Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading*. Paper presented at the Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, Paris, France.

Carroll, A., & Heiser, G. (2010, jun). *An Analysis of Power Consumption in a Smartphone.* Paper presented at the USENIX Annual Technical Conference, Boston, MA, USA.

Chauhan, A., Sammakia, B., & Ghose, K. (2015, 15-19 March 2015). *Transient power analysis to estimate the thermal time lag of a microprocessor hot spot.* Paper presented at the Thermal Measurement, Modeling & Management Symposium (SEMI-THERM), 2015 31st.

Chen, J. (2002). *Formal Modelling of Java GUI Event Handling.* Paper presented at the Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, London, UK, UK.

Cohen, M., Zhu, H. S., Senem, E. E., & Liu, Y. D. (2012). *Energy types*. Paper presented at the Proceedings of the ACM international conference on Object oriented programming systems languages and applications, Tucson, Arizona, USA.

Cugola, G., Margara, A., Pezzè, M., & Pradella, M. (2015). *Efficient Analysis of Event Processing Applications.* Paper presented at the Proceedings of the 9th ACM

International Conference on Distributed Event-Based Systems, New York, NY, USA.

Erez, M., Ahn, J. H., Gummaraju, J., Rosenblum, M., & Dally, W. J. (2007). *Executing irregular scientific applications on stream architectures*. Paper presented at the Proceedings of the 21st annual international conference on Supercomputing, Seattle, Washington.

Frias-Martinez, V., & Virseda, J. (2012). *On the relationship between socio-economic factors and cell phone usage*. Paper presented at the Proceedings of the Fifth International Conference on Information and Communication Technologies and Development, Atlanta, Georgia, USA.

Gaspar, F., Taniça, L., Tomas, P., Ilic, A., & Sousa, L. (2015). A Framework for Application-Guided Task Management on Heterogeneous Embedded Systems. *ACM Trans. Archit. Code Optim., 12*(4), 1-25. doi:10.1145/2835177

Han, S., Marshall, S., Chun, B.-G., & Ratnasamy, S. (2012). *MegaPipe: A New Programming Interface for Scalable Network I/O.* Paper presented at the Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), Hollywood, CA.

Heyer, C., Brereton, M., & Viller, S. (2008). *Cross-channel mobile social software: an empirical study*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Florence, Italy.

Holtsnider, B., & Jaffe, B. D. (2012). *IT Manager?s Handbook, Third Edition: Getting Your New Job Done* (3rd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Honig, T., Eibel, C., Kapitza, R., & Schroder-Preikschat, W. (2012). SEEP: exploiting symbolic execution for energy-aware programming. *SIGOPS Oper. Syst. Rev., 45*(3), 58-62. doi:10.1145/2094091.2094106

Hoque, M. A., & Tarkoma, S. (2016). Sudden Drop in the Battery Level?: Understanding Smartphone State of Charge Anomaly. *SIGOPS Oper. Syst. Rev., 49*(2), 70-74. doi:10.1145/2883591.2883606

Hsiu, P.-C., Tseng, P.-H., Chen, W.-M., Pan, C.-C., & Kuo, T.-W. (2016). User-Centric Scheduling and Governing on Mobile Devices with big.LITTLE Processors. *ACM Trans. Embed. Comput. Syst., 15*(1), 1-22. doi:10.1145/2829946

Jensen, A. R. (2006). *Clocking the mind*: Elsevier Ltd.

Jindal, A., Pathak, A., Hu, Y. C., & Midkiff, S. (2013). *Hypnos: Understanding and Treating Sleep Conflicts in Smartphones.* Paper presented at the Proceedings of the 8th ACM European Conference on Computer Systems, New York, NY, USA.

Kerrisk, M., & Project, L. D. (2015). POSIX Signals Manual (3.82 ed.).

Kim, M., Kim, K., Geraci, J. R., & Hong, S. (2014). *Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor*. Paper presented at the Proceedings of the conference on Design, Automation & Test in Europe, Dresden, Germany.

Kroah-Hartman, G. (2007). Everything you never wanted to know about kobjects, ksets, and ktypes: LWN.net.

Kropp, D. O. (2014). Direct Frame Buffer (DirectFB).

Lee, H.-c., Kim, C. H., & Yi, J. H. (2011). *Experimenting with system and Libc call interception attacks on ARM-based Linux kernel*. Paper presented at the Proceedings of the 2011 ACM Symposium on Applied Computing, TaiChung, Taiwan.

Lee, J., & Park, K. H. (2010). Interrupt handler migration and direct interrupt scheduling for rapid scheduling of interrupt-driven tasks. *ACM Trans. Embed. Comput. Syst., 9*(4), 1-34. doi:10.1145/1721695.1721708

Lehey, G. (1995). Setting up X11: A no-tears guide to XFree86 configuration. *Linux J., 1995*(15es), 4.

Lemon, J., & Manual, F. S. C. (2013). KQUEUE.

Love, R. (2010). *Linux Kernel Development*. Upper Saddle River, NJ: Pearson Education.

Love, R., & Zhen, Z. (2015). Inotify - A Powerful Yet Simple File Change Notification System.

Marz, S., & Vander Zanden, B. (2015). *Reducing Power Consumption and Latency in Mobile Devices Using an Event Stream Model*. Retrieved from Department of Electrical Engineering and Computer Science: https://www.eecs.utk.edu/resources/library/595

Maurer, M.-E., Hausen, D., Luca, A. D., & Hussmann, H. (2010). *Mobile or desktop websites?: website usage on multitouch devices*. Paper presented at the

Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, Reykjavik, Iceland.

Maximizing Battery Life and Lifespan. (2016). Retrieved from https://www.apple.com/batteries/maximizing-performance/

Mogul, J. C., & Ramakrishnan, K. K. (1997). Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst., 15*(3), 217-252. doi:10.1145/263326.263335

Ng, A., Annett, M., Dietz, P., Gupta, A., & Bischof, W. F. (2014). *In the blink of an eye: investigating latency perception during stylus interaction*. Paper presented at the Proceedings of the 32nd annual ACM conference on Human factors in computing systems, Toronto, Ontario, Canada.

Nilsson, A. (2015). GIMP Drawing Kit. Retrieved from

NVIDIA. (2015). NVIDIA Tegra K1 Processor Specifications.

Öquist, G., & Lundin, K. (2007). *Eye Movement Study of Reading Text on a Mobile Phone Using Paging, Scrolling, Leading, and RSVP*. Paper presented at the Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia, New York, NY, USA.

Pabla, C. S. (2009). Completely Fair Scheduler. *Linux Journal, 184*, 82-83.

Parthasarathy, R. (2006). Energy-Aware User Interfaces and Energy-Adaptive Displays, *39*, 31-38.

Pathak, A., Jindal, A., Hu, Y. C., & Midkiff, S. P. (2012). *What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone*

Apps. Paper presented at the Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, New York, NY, USA.

Pavlik, V. (2001). Linux Input Drivers. Retrieved from

Qin, C., & Rusu, F. (2013). *Scalable I/O-bound parallel incremental gradient descent for big data analytics in GLADE*. Paper presented at the Proceedings of the Second Workshop on Data Analytics in the Cloud, New York, New York. http://dl.acm.org/citation.cfm?doid=2486767.2486771

Raman, B., & Chakraborty, S. (2008). Application-specific workload shaping in multimedia-enabled personal mobile devices. *ACM Trans. Embed. Comput. Syst., 7*(2), 1-22. doi:10.1145/1331331.1331334

Repasky, R. R. (2004). *Easy access to remote graphical UNIX applications for windows users*. Paper presented at the Proceedings of the 32nd annual ACM SIGUCCS conference on User services, Baltimore, MD, USA.

Rong, P., & Pedram, M. (2003). *An Analytical Model for Predicting the Remaining Battery Capacity of Lithium-Ion Batteries*. Paper presented at the Proceedings of the conference on Design, Automation and Test in Europe - Volume 1.

Rossi, F. (2003). An Event Mechanism for Linux. *Linux J., 2003*(111), 7.

Salah, K., Manea, A., Zeadally, S., & Alcaraz Calero, J. M. (2011). On Linux Starvation of CPU-bound Processes in the Presence of Network I/O. *Comput. Electr. Eng., 37*(6), 1090-1105.

Searls, D. (2010). Eof: is Android the "top" Linux? *Linux J., 2010*(192), 10.

Singh, A. (2014). Google introduces ART(Android Runtime) in KitKat: AndroidAIO.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., & Berger, E. D. (2007). *Eon: a language and runtime system for perpetual systems*. Paper presented at the Proceedings of the 5th international conference on Embedded networked sensor systems, Sydney, Australia.

Stewart, B. (2011). Is your Android app getting enough sleep? *O?Reilly*.

Strebelow, R., & Prehofer, C. (2012). *Analysis of Event Processing Design Patterns and Their Performance Dependency on I/O Notification Mechanisms*. Paper presented at the Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools, Berlin, Heidelberg.

Tsafrir, D. (2007). *The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)*. Paper presented at the Proceedings of the 2007 workshop on Experimental computer science, San Diego, California. http://dl.acm.org/citation.cfm?doid=1281700.1281704

Vallerio, K. S., Zhong, L., & Jha, N. K. (2006). Energy-Efficient Graphical User Interface Design *IEEE Transactions on Mobile Computing* (Vol. 5, pp. 846-859): Institute of Electrical and Electronics Engineers (IEEE).

Wickramasinghe, U. S., Bronevetsky, G., Lumsdaine, A., & Friedley, A. (2014). *Hybrid MPI: a case study on the Xeon Phi platform*. Paper presented at the Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers, Munich, Germany.

Windows 10 Experience. (2016). Retrieved from https://www.microsoft.com/en/mobile/windows10/

Wong, C. S., Tan, I., Kumari, R. D., & Wey, F. (2008). Towards Achieving Fairness in the Linux Scheduler. *SIGOPS Oper. Syst. Rev., 42*(5), 34-43.

Yoo, S., & Jerraya, A. A. (2003). *Introduction to Hardware Abstraction Layers for SoC*. Paper presented at the Proceedings of the conference on Design, Automation and Test in Europe - Volume 1.

Yoo, S., Shim, Y., Lee, S., Lee, S.-A., & Kim, J. (2015). *A case for bad big.LITTLE switching: how to scale power-performance in SI-HMP*. Paper presented at the Proceedings of the Workshop on Power-Aware Computing and Systems, Monterey, California.

Yoon, C., Kim, D., Jung, W., Kang, C., & Cha, H. (2012). *AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring.* Paper presented at the Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA.

Zhong, L., & Jha, N. K. (2003). *Graphical user interface energy characterization for handheld computers*. Paper presented at the Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, San Jose, California, USA.

Appendices

Appendix A

```
//----------------SNIP-------------------------------------
bool softwareSync = mUseSoftwareVSync;
nsecs_t timeout = softwareSync ? ms2ns(16) : ms2ns(1000);
if (mCondition.waitRelative(mLock, timeout) == TIMED_OUT) {
       if (!softwareSync) {
              ALOGW("Timed out waiting for hw vsync; faking it");
       }
       // FIXME: how do we decide which display id the fake
       // vsync came from ?
       mVSyncEvent[0].header.type =
              DisplayEventReceiver::DISPLAY_EVENT_VSYNC;
       mVSyncEvent[0].header.id =
              DisplayDevice::DISPLAY_PRIMARY;
       mVSyncEvent[0].header.timestamp =
              systemTime(SYSTEM_TIME_MONOTONIC);
       mVSyncEvent[0].vsync.count++;
}
//----------------SNIP-------------------------------------
```

Event polling inside of Android's InputFlinger. If a hardware refresh (vsync) signal is present, it is used to trigger an event poll. Otherwise, Android uses a 16 millisecond loop delay.

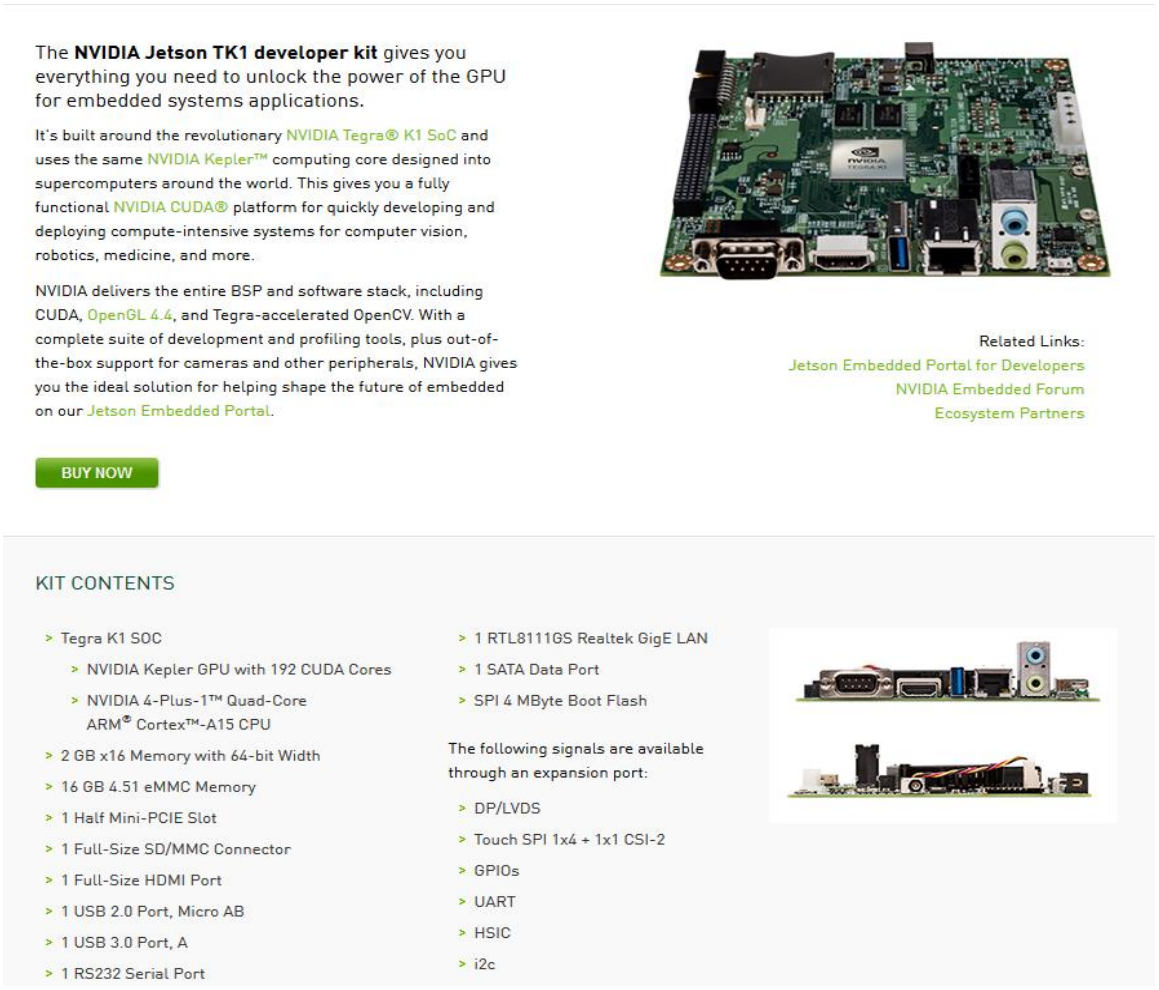
```
for (;;) {
//----------------SNIP-------------------------------------
             for (j = 0; j < BITS_PER_LONG; ++j, ++i, bit <<= 1) {
                    struct fd f;
                    f = fdget(i);
                    if (f.file) {
                           const struct file_operations *f_op;
                           f_op = f.file->f_op;
                           mask = DEFAULT_POLLMASK;
                           if (f_op->poll) {
                                  wait_key_set(wait, in, out,
                                                     bit, busy_flag);
                                  mask = (*f_op->poll)(f.file, wait);
                           }
                           fdput(f);
                           /* got something, stop busy polling */
                           if (retval) {
                                  can_busy_loop = false;
                                  busy_flag = 0;

                                  /*
                                  * only remember a returned
                                  * POLL_BUSY_LOOP if we asked for it
                                  */
                           }
                           else if (busy_flag & mask)
                                  can_busy_loop = true;

                    }
             }
       cond_resched();
}
wait->_qproc = NULL;
if (retval || timed_out || signal_pending(current))
break;
...
/* only if found POLL_BUSY_LOOP sockets && not out of time */
if (can_busy_loop && !need_resched()) {
       if (!busy_end) {
             busy_end = busy_loop_end_time();
             continue;
       }
       if (!busy_loop_timeout(busy_end))
             continue;
}
busy_flag = 0;
...
```

The select system call inside of the Linux/Android kernel uses a polling loop to discover events on a file descriptor or socket.

Appendix B

	VDD_CORE	VDD_CPU	MC/EMC Power	PLL	MC/EMC Clock	DRAM State
LP0 (deep sleep)	off	off	off	off	off	self-refresh
LP1 (suspend)	on	off	on	off	off	self-refresh
LP2 (idle)	on	off	on	on	on	active

This table shows the three power states of the NVIDIA TK1 development board. Furthermore, it depicts the several systems that are activated or deactivated depending on the power state.


The **NVIDIA Jetson TK1 developer kit** gives you everything you need to unlock the power of the GPU for embedded systems applications.

It's built around the revolutionary NVIDIA Tegra® K1 SoC and uses the same NVIDIA Kepler™ computing core designed into supercomputers around the world. This gives you a fully functional NVIDIA CUDA® platform for quickly developing and deploying compute-intensive systems for computer vision, robotics, medicine, and more.

NVIDIA delivers the entire BSP and software stack, including CUDA, OpenGL 4.4, and Tegra-accelerated OpenCV. With a complete suite of development and profiling tools, plus out-of-the-box support for cameras and other peripherals, NVIDIA gives you the ideal solution for helping shape the future of embedded on our Jetson Embedded Portal.

BUY NOW

Related Links:
Jetson Embedded Portal for Developers
NVIDIA Embedded Forum
Ecosystem Partners

KIT CONTENTS

- Tegra K1 SOC
  - NVIDIA Kepler GPU with 192 CUDA Cores
  - NVIDIA 4-Plus-1™ Quad-Core ARM® Cortex™-A15 CPU
- 2 GB x16 Memory with 64-bit Width
- 16 GB 4.51 eMMC Memory
- 1 Half Mini-PCIE Slot
- 1 Full-Size SD/MMC Connector
- 1 Full-Size HDMI Port
- 1 USB 2.0 Port, Micro AB
- 1 USB 3.0 Port, A
- 1 RS232 Serial Port
- 1 RTL8111GS Realtek GigE LAN
- 1 SATA Data Port
- SPI 4 MByte Boot Flash

The following signals are available through an expansion port:

- DP/LVDS
- Touch SPI 1x4 + 1x1 CSI-2
- GPIOs
- UART
- HSIC
- i2c

This figure depicts the NVIDIA TK1 specification and advertisement sheet. The NVIDIA is a fully-equipped mobile platform used in real-world, mobile devices.

Vita

Stephen Gregory Marz was born on July 6, 1982 in Dunkirk, New York. He earned a Bachelor's degree in Computer Science from the Illinois Institute of Technology in Chicago, Illinois on May, 2004. He entered the Air Force on August, 2000, commissioned as an Air Force officer on May, 2004, and became a combat rescue helicopter pilot on November, 2006. After several combat deployments, he retired on October 8th, 2012. After accepting a position as a graduate teaching assistant, he began his pursuit toward his Ph.D. in Computer Science at the University of Tennessee on August, 2012. In the future, he will pursue a job teaching at military universities or perhaps consulting in regards to national security in the field of computer science.