Reducing Event Latency and Power Consumption in Mobile Devices by Using a Kernel-Level Display Server

Stephen Marz, *Member, IEEE* and Brad Vander Zanden and Wei Gao, *Member, IEEE*
E-mail: stephen.marz@utk.edu, bvanderz@utk.edu, wgao@utk.edu

Abstract—Mobile devices differ from desktop computers in that they have a limited power source, a battery, and they tend to spend more CPU time on the graphical user interface (GUI). These two facts force us to consider different software approaches in the mobile device kernel that can conserve battery life and reduce latency, which is the duration of time between the inception of an event and the reaction to the event. One area to consider is a software package called the display server. The display server is middleware that handles all GUI activities between an application and the operating system, such as event handling and drawing to the screen. In both desktop and mobile devices, the display server is located in the application layer. However, the kernel layer contains most of the information needed for handling events and drawing graphics, which forces the application-level display server to make a series of system calls in order to coordinate events and to draw graphics. These calls interrupt the CPU which can increase both latency and power consumption, and also require the kernel to maintain event queues that duplicate event queues in the display server. A further drawback of placing the display server in the application layer is that the display server contains most of the information required to efficiently schedule the application and this information is not communicated to existing kernels, meaning that GUI-oriented applications are scheduled less efficiently than they might be, which further increases power consumption. We propose moving the display server to the kernel layer, so that it has direct access to many of the event queues and hardware rendering systems without having to interrupt the CPU. This adjustment has allowed us to implement two power saving strategies, discussed in other papers, that streamline the event system and improve the scheduler. The combination of these two techniques reduces power consumption by an average of 30% and latency by an average of 17ms. Even without the implementation of these power saving techniques, the KDS increases battery life by 4.35% or on average about ten extra minutes for a typical mobile phone or thirty extra minutes for a typical tablet computer. It also reduces latency by 1.1 milliseconds.

Index Terms—Graphical User Interfaces, Kernel Display Server, Event Handling

1 Introduction

Display servers are used by graphical user interfaces (GUIs) to coordinate event handling and draw graphics to the screen. They have traditionally served as "middleware" between the application and the OS and have been implemented in user space. However, running the display server in user space on a mobile device has two important drawbacks: 1) most of the event handling and rendering routines historically implemented in display servers are now implemented in the OS, which means that the display server duplicates much of the kernel's GUI functionality, and 2) it prevents the recently introduced power saving hardware architectures from fully realizing their ability to reduce power consumption.

The reason display servers still run in user space is a historical artifact. When graphical user interfaces (GUIs) became widely adopted in the 1980s, display servers were implemented in user space because the GUI was not considered part of the OS and because developers wanted the flexibility to choose their own display server. In the 1990s this situation changed as both Windows and Mac OS adopted graphical interfaces for their look-and-feel and tightly integrated their display servers with the OS, even though their display servers continued to run in user space (the tight integration precluded developers from choosing their own display server). Mobile devices, such as tablets and smart phones, have continued this custom of using the OS as a graphical interface, tightly integrating the display server with the OS, and running the display server in user space.

What has changed over time is that many of the services that used to reside in the display server, such as event handling and rendering, have migrated to both desktop and mobile OS kernels, because certain real-time applications, such as games, demand fast interaction and rendering. Section 2 describes these efforts in greater detail. The movement of event handling and rendering routines into the kernel means that the display server increasingly duplicates kernel activity. This duplicative overhead has ramifications for both the performance and power consumption of mobile devices.

First, modern kernels contain the event handling routines that deal with inputs received via hardware, such as a mouse, keyboard, or stylus. However, existing display

• *S. Marz and B. Vander Zanden are with the Department of Electrical Engineering and Computer Science, 1520 Middle Drive, Min H. Kao Building, University of Tennessee, Knoxville, TN 37996. W. Gao is with the Department of Electrical and Computer Engineering, 3700 O'Hara Street, Benedum Hall, University of Pittsburgh, Pittsburgh, PA 15261.*

Manuscript received XXX YYY, ZZZZ; revised XXX YY, ZZZZ.

servers still maintain event queues that duplicate the kernel event queues and these display servers must use costly system calls to obtain this information from the kernel. In particular, existing display servers use polling loops that constantly poll the kernel's event queues to determine whether an event has occurred. This constant polling keeps rousing the CPU and prevents it from being placed in a lower power consuming sleep state. In a previous article, we described an Event Stream Model (ESM) that pushes events from input devices to the CPU rather than forcing the CPU to constantly poll the input devices for events, as is done by existing mobile device OS's [1]. This scheme should allow the CPU to sleep until events arrive because it eliminates polling loops in the kernel. However, in order for this ESM model to fully eliminate polling loops, the display server must be moved to the kernel; otherwise the display server continues to use a polling loop and the CPU cannot be placed in a lower power sleep state.

A second drawback of this duplicative overhead is that the display server must coordinate with the kernel to draw to the screen. For example, existing Android implementations use four layers to render an app, thus slowing the CPU considerably. First the app sends a surface to Surface Flinger, Android's display server, in user space. Then, Surface Flinger composes the surface using a software package called HWComposer. The surface is then passed to the direct rendering manager (DRM), which is Android's link between user space and kernel space. DRM is handled by the DRI driver in the kernel. This driver translates the surface into something that the display driver can understand and then sends it to the driver to be written to the display device, which is typically the screen. The slowdown associated with the communication through these multiple layers has given rise to many "direct rendering" systems, such as DirectFB [3], that bypass the HWComposer and directly utilize the DRM, which allows the display server to write directly to the graphics system in the kernel. However, this still leaves the communication layer between the display server and the kernel's DRM, which our experimental results show moderately increases power consumption and the amount of time the operating system takes to handle events.

A third drawback of existing display servers is that they do not provide the kernel with important scheduling information that it could use to optimize the CPU's performance. For example, desktop computer users may place application "windows" on top of each other, beside each other, or overlapping each other. By contrast, a mobile device may have several applications running at the same time, but either only one application is displayed, or on some devices with larger displays, a couple apps might be displayed in side-by-side windows. A display server on a mobile device knows which apps are hidden and hence are in the background. If this information were known to the kernel, it could stop scheduling hidden apps which would increase the opportunities to place the CPU in lower power consuming sleep modes. However, this information is not known to the kernel and hence it will keep scheduling the app, even though the app is invisible and any drawing it does will be discarded by the display server.

The first two drawbacks can be eliminated if the display server is moved into the kernel for mobile devices and the third drawback can be eliminated if the display server is redesigned so that it can provide scheduling information to the kernel's scheduler. In this paper, we describe 1) how we re-designed the Android display server to provide scheduling information, 2) how we moved it from the application layer to the kernel, and 3) how applications and middleware interact with this new display server. Our display server, which we have named the Kernel Display Server (KDS) offers the following benefits:

1) It eliminates the need for the application layer to continuously poll the kernel for events.
2) It has direct access to the kernel's event queues and the kernel's graphics package rather than having to access them through various system calls. While many improvements have been made to the CPU interrupts caused by system calls, they are by no means a trivial expense in terms of power consumption and latency [2].
3) It removes the need for multiple hardware drivers for the GUI system. In a traditional display server, a system programmer would write a driver to allow the display server to communicate with the kernel, and a second driver to allow the kernel to communicate with the hardware. Since the KDS is already in the kernel, it can directly interact with the kernel-level driver, thus reducing the system programmer's burden. In particular the KDS has only 2 rendering layers as opposed to Android's 4 rendering layers. With our KDS, the user space layer generates a "surface" using Android's Surface Flinger. This surface is then passed directly to the KDS, which composes it with the OS graphics and then uses DirectFB [3] to draw it to the screen. By directly incorporating DirectFB into the display server, the KDS eliminates the previous communication layer between the application and DirectFB. The KDS also combines compositing and drawing into a single layer in the kernel, which essentially collapses the last three layers of the existing Android rendering pipeline into a single layer.
4) It splits a GUI into four threads–an event handling thread, a drawing thread, a background thread, and a foreground thread. These threads provide the scheduler with important information about an app and allow it to more intelligently schedule apps. Since the KDS is in the kernel, the kernel's scheduler has direct access to these threads, rather than having to rely on a constant stream of system calls that would be required to notify it of this scheduling information if the display server were still in user space.

This KDS implementation has enabled us to implement two other power saving strategies, an Event Stream Model (ESM) that pushes events from input devices to the CPU and eliminates kernel polling loops [1], and a scheduling algorithm that takes advantage of the thread information to de-schedule apps when they are in the background [4]. These power saving strategies result in almost a 30% improvement to battery life and a 17ms reduction in latency for some apps, where we define latency as the duration

of time between the inception of an event and the reaction to the event. Although moving the display server into the kernel was primarily done to enable other power saving strategies, the elimination of duplicative overhead in our KDS implementation also realizes a 4.35% improvement to battery life and a 1.1 millisecond reduction in latency when implemented by itself, without any other power saving strategies.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 further makes the case for why it is important to move the display server to the kernel. Section 4 describes the kernel implementation and Section 5 describes the API the KDS provides for applications or middleware, such as Android. Section 6 describes the experiments we used to measure the power savings that could be achieved with the KDS. Section 7 discusses the potential drawbacks of moving the display server to the kernel and Section 8 provides a summary of our conclusions and results.

2 Related Work

This section starts by reviewing the most commonly used application-layer display servers. To our knowledge, there are no kernel-level display servers. It then describes some prior direct rendering solutions that bypass the application-layer display server and that we have incorporated into our KDS. This section concludes by discussing alternative power management approaches that have been tried previously and alternative ways to measure power consumption by mobile devices.

2.1 Display Servers

The most common display server for Unix type environments is the X11 display server, now commonly called X.org [5], [6]. Graphical user interface applications connect to the X11 display server in order to draw to the screen and to handle events. The X.org server serves as "middleware" between the application and the kernel. This display server is implemented in user space and communicates through character devices and system calls to and from the kernel [7]. The X windows system requires its own driver for each of the input devices, graphics devices, and other devices that work with the display server. This approach works with a wide range of desktop architectures and Unix-flavored OS's. However, in mobile devices, it represents an additional layer between the kernel and application that tends to lead to increased power consumption and latency since the generic display server is unable to take full advantage of the app-centric mobile device approach.

Wayland is an open-source display server meant as a more modern, simpler X11 replacement for Unix-flavored applications but it also suffers from its inability to take advantage of the app-centric mobile device approach [8]. Furthermore, Wayland uses many of the drivers and technologies from the X11 system, such as the event polling loop in the kernel, and hence, some of the inefficiencies of the X11 system are inherited by the Wayland system.

The Android operating system for mobile devices contains a software package called Surface Flinger which is ultimately responsible for drawing graphics to the screen [9]. Surface Flinger is given a set of GUI components, such as a menu bar, a tool bar, or a status bar, and then it merges all of the components into a single object that the hardware can draw. Typically, Surface Flinger utilizes OpenGL ES for drawing most of its objects. This is important since OpenGL ES maintains a client and server relationship, where the client is in the application layer and the server is in the kernel layer, thus necessitating costly system calls between the two layers even when using direct rendering systems (described in the next subsection).

MacOS uses the Quartz system to draw to the screen. Since MacOS uses Darwin, which is a microkernel [10], the drawing layer sits directly above the kernel layer and is not directly in the kernel layer. The Quartz drawing system is split into three sections: the compositor, the graphics library called Quartz2D, and the window server, which is responsible for routing hardware events, such as a mouse click or keyboard input [11]. Like other existing display servers, Quartz uses polling loops that needlessly consume power on idle, mobile devices.

Although not a display server, Ping-Peng, et al. developed an interaction model for cooperative, event-based systems, such as distributed and multi-processor workloads that influenced the design of our KDS [12]. Ping-peng created primitives that combine complex events into "composite" events that cover the life cycle of an event, which mirrors the approach we take in the KDS and the ESM. For example, Ping-pen's language shows the multiple layers through which an event may pass, such as originating in the kernel, being sent to a socket, and being picked up by an application. Their paper is purely theoretical, but their figures do a nice job of documenting how an event "works". Their event semantics and event detection methods helped us understand an event life cycle and lend credibility to combining a display server with a kernel-level event handling model.

Netlink sockets appear to be an alternate way to eliminate event polling loops from user space since they can directly dispatch an event from the kernel to a user application [13]. However, Netlink adds itself to a BSD socket, and the BSD socket still requires polling [1]. Hence, the polling is not eliminated using Netlink but simply pushed into the kernel. In this respect it is much like epoll, which also appears to eliminate polling in user space, but in fact transfers the polling into the kernel [1].

2.2 Direct Rendering Solutions

In order to accelerate the rendering process, a number of researchers have developed systems that allow graphical applications to bypass the display server and directly communicate with an OS's rendering engine. Direct framebuffer (DirectFB) is a library and a Linux kernel module that allows GUI applications to draw directly to the screen through the kernel's graphics driver [3]. Our KDS uses DirectFB as a starting point, and in fact uses many of the same kernel routines as DirectFB to draw to the screen. DirectFB is a desirable starting point since it has been used extensively in multiple consumer products, such as Roku.

Direct rendering manager (DRM) is a kernel-level graphics system that allows user applications to more or less

directly render to the graphics processing unit (GPU) [14]. We use the term "more or less" since the kernel still lies between the application and the GPU. However, the DRM is positioned one layer above the graphics driver in the kernel, and so it bypasses some of the middleware layers associated with a display server. OpenGL and other hardware rendering systems use DRM to directly interact with the GPU's driver. It is important to note that this exposes the GPU's driver to a user application and not necessarily the GPU itself.

Singhai and Bose explored the power consumption associated with graphics intensive applications [15]. Their approach centered on the surface model present in Android operating systems. The surface view system in Android is a software layer compositing system which Java and other systems use. Singhai and Bose decided to implement a Windows-style GUI system where all drawing is performed on the main window. This reduces the mathematical complexities of flattening a layer. Our KDS implements this direct compositing approach for applications, but does not do so with adornments, such as the battery status icons and toolbars.

2.3 Alternative Power Management Strategies

Our power management approach centers on improving event handling and scheduling in the kernel. The most common alternative software approach to power management is sleep management policies that attempt to aggressively put applications to sleep when they are not being used [16]. Unfortunately, these policies have two drawbacks. First, they can make a device seem less responsive. For example, when one of the authors picks up his iPhone, it will show the clock, and then the screen will turn black. A double press then causes the screen to come back on, and then it goes off again, before a final double press brings the screen on. This is an example of the user battling an aggressive sleep policy. To stop this lack of responsiveness, many applications must use wake locks to force the mobile device to stay awake; however, this comes at the cost of increased power consumption since the mobile device cannot go to sleep while locked awake [17]. 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. Hence wake locks and their associated problems are a second drawback of aggressive sleep policies. Our approach, which involves eliminating polling loops in the kernel and having the display server communicate knowledge about apps to the scheduler to improve scheduling should help obviate the need for aggressive sleep policies since our approaches should provide more intelligent ways to put apps to sleep.

A number of researchers have developed energy-aware programming languages to assist with power management. The Eon programming language is an energy-aware programming language that is structured to automatically adapt programs to a mobile devices energy state [18]. Eon is designed to be portable among hardware platforms and to maximize the performance of an application under several energy conditions. The ET programming language is another energy-aware programming language that specifically targets the Android mobile operating system [19]. 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. At present very few actual applications are written in either language.

Most power management approaches are more hardware oriented. The most obvious one to users is the dimming of the LCD screen after a certain period of disuse because the LCD screen is typically the greatest consumer of power [20]. Unfortunately the power manager is often too aggressive about dimming the display, which leads to applications using the wake locks mentioned previously [16].

More subtle power management approaches react when power is being used by apps and, like the LCD dimming strategy, attempt to take advantage of idle pauses [21], [22]. They suffer the same drawbacks of potentially being too aggressive about putting the CPU in a lower power state and then suffering latency issues when the CPU must respond to an event more quickly than was anticipated.

Our approach attempts to avoid these problems in two ways. First, it eliminates polling loops so idle applications by default do not consume power. Hence reactive power management strategies are not required to put these apps to sleep. Second, the KDS obtains greater information about an application by having an application allocate its functions to four separate threads–event, display, background, and foreground–which gives the scheduler better information about how to intelligently schedule apps, and in particular, when it is possible to completely put apps to sleep to conserve power. For example, if an app is not in the foreground and has no background tasks, it can be de-scheduled and if all open apps can be de-scheduled, the CPU can be moved to a lower power state.

2.4 Measuring Power Consumption

It is a rather daunting task to identify the specific areas where a mobile device is consuming power. There are many components in modern mobile devices that may interact to increase power consumption. However, several tools have been developed to accurately measure power consumption in mobile devices and to show how each mobile device component contributes to power consumption.

Power Tutor is an open source power measuring app based on the Power Booter power model which is specifically designed for mobile devices [23]. This app aggregates usage statistics from the kernel and applies a battery consumption model to scale the figures. AppScope and DevScope are a pair of cooperating tools used to measure power consumption in mobile devices [24]. One runs DevScope to get a model of the system, and then runs AppScope with DevScope's result to get power measurements for a device.

Both of these approaches use formulas to model power consumption. Unfortunately, this software solution does not function with our experimental NVIDIA TK1 board. Instead, our experiments use a Nvidia TK1 board that has a power measuring system in the hardware that allowed us to measure power consumption directly, and hence, we did not use these formulas to model the power consumption of our KDS.

Rice and Hay [25] and Murmuria [26] used actual kernel statistics to measure power. However, this approach proved rather inaccurate and not-repeatable using the TK1, and as noted earlier, we discovered a direct way to measure power consumption in the TK1, thus eliminating the need to use formulas or models.

Xiao and his associates developed an approach to measuring power consumption in mobile devices that focused on power consumption at the system level rather than the individual component level [27]. We wanted to get power consumption figures directly related to the KDS however so this approach was not applicable to our research.

Kindelsberger and his associates have identified ways to record long-term power consumption in mobile devices [28]. They found that the GUI, LCD, and WiFi systems consume most of the power in a mobile device. The KDS is aimed at reducing the power consumption of GUIs.

3 The Case for a Kernel Display Server

The introduction briefly made the case for a kernel display server and in this section we more fully explore the limitations of a display server implemented in the application layer. 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 1 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 an event push-model called the Event Stream Model (ESM) to push events to the application (see Figure 2) [1]. The ESM alerts the CPU whenever an input device receives an event and takes advantage of the fact that newer input devices are able to interrupt the CPU when an event arrives. The ESM registers interrupt handlers with the CPU that collect information about the event from the input device and then forward the event to any event handlers that are registered with the KDS and which the KDS has "switched on" (the KDS switches off event handlers associated with background apps). Older input devices required the CPU to periodically poll each devices' event queue and existing mobile kernels still cater to the restrictions of these older devices. Hence our ESM/KDS model is aimed at exploring how kernels can be modified to take advantage of the ability of input devices to directly notify the CPU that an event has arrived.

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 application's 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 re-registers 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 [29].

4 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 thread system that coordinates with the ESM and scheduler (b) the compositor system, and (c) the drawing system.

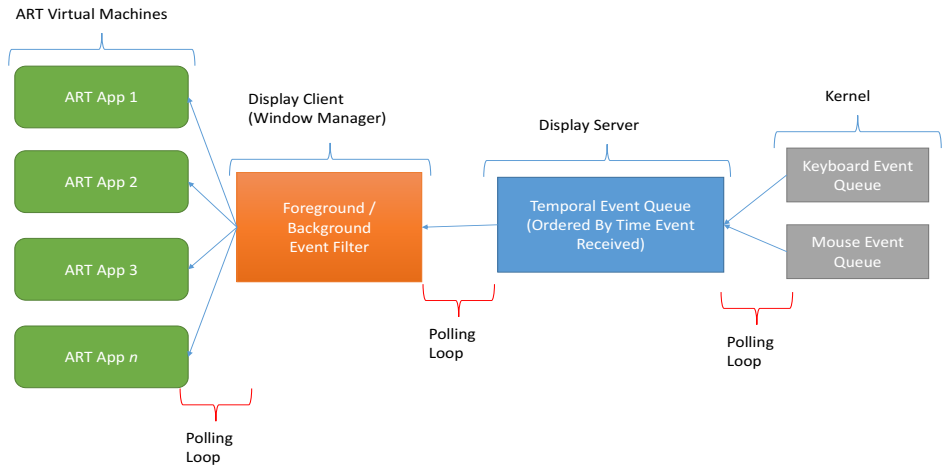

Fig. 1. The existing pull model used by display servers store events into several queues where they are propagated through multiple layers to the applications that will handle them. This figure illustrates the architecture of a typical event hub, which is a combination of the display server's temporal ordering and the display client's event filter, for GUI applications. As an example of an event filter, Java requires applications to add event listeners. If there are no event listeners for an event, then the event will be discarded (i.e., filtered out).

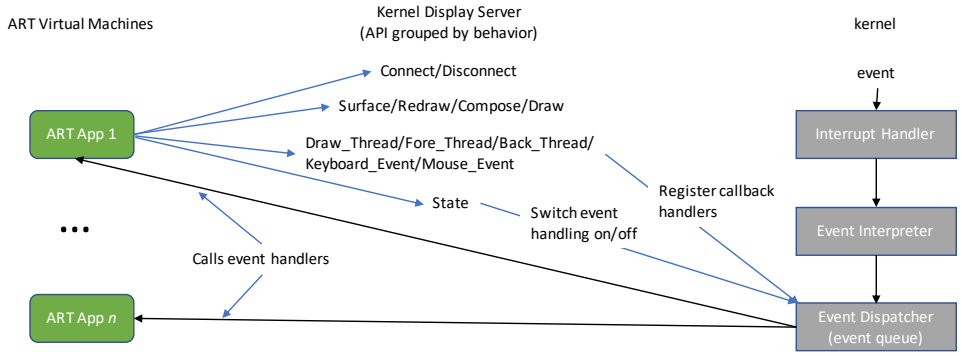

Fig. 2. This figure shows how the KDS interacts with apps and the ESM via its API and also shows how the KDS simplifies event routing and eliminates event polling. The Connect/Disconnect commands allow an app to register with the KDS on start-up and de-register with the KDS when quitting. The draw command group allows an app to obtain a drawing surface, notify the KDS that a redraw must occur, compose the app's graphics with the OS graphics, and draw the resulting image to the frame buffer. The thread and event group allows an app to register tasks and event handlers with the KDS. The KDS in turn registers mouse and keyboard event handlers with the ESM. When an event occurs, the interrupt handler receives the initial notification of the event and collects relevant information about the event from the input device. The event is then propagated through the event interpreter and dispatcher routines which check to see if an event handler(s) has been registered for that event. If so the event handler(s) for that event is called. If an event handler is already executing, the Event Dispatcher stores the event in an event queue for an app. This is the only event queue required for an app. Events are dispatched directly from the event queue to the app as opposed to the more circuitous route taken in Figure 1. Moving the display server to the kernel eliminates the need for the duplicate event queues previously maintained by display servers, thus eliminating the rightmost polling loop in Figure 1. Finally the State command tells the KDS whether the app is in the foreground or the background. The KDS re-registers event handlers if the app has moved into the foreground and de-registers event handlers if the app has moved into the background. The KDS's API is described in more detail in Section 5.1. The State command eliminates the need for the Foreground/Background filter shown in Figure 1 and eliminates the remaining two polling loops.

4.1 KDS Thread System

A GUI has three primary functions: accept input in the form of events, process the events to update the application's state, and draw the updated application state to the display. The KDS allocates four threads in which the GUI application may place its code to handle these three activities: (a) an event handling thread that executes event handlers that are responding to hardware inputs, (b) a display thread for drawing to the screen, (c) a background thread for handling constant activity, such as decompressing video frames, and (d) a foreground thread which runs both when the app is the topmost layer and accepting input from the user and when the screen is turned off and the app needs to perform certain activities, such as audio decoding and playing.

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.

Fortunately, middleware can almost completely hide this thread messiness from the application programmer. First, the KDS itself manages the event thread and allows the programmer to attach multiple event handlers to the thread through commands described in Section 5. Second, middleware such as Java's virtual machine will typically handle interactions with the event handling and display threads. For example, in Java a programmer overrides the `paint_component` method to draw to the screen. With our implementation the programmer would continue to use this routine to draw to the screen. The Java middleware would be responsible for ensuring that the code in the `paint_component` method executes on the KDS drawing thread. Finally, the middleware can be tasked with marshalling all the background and foreground tasks that the programmer wishes to execute and implementing the functions that fork multiple threads for each of these different foreground and background tasks. Hence the programmer only needs to indicate to the middleware whether a callback procedure, such as one for decoding compressed video frames, should run in the background or the foreground. Thus, the cognitive load on an app programmer is not significantly increased by the KDS.

A customized scheduler can use knowledge of these four threads to more intelligently schedule an app. For example the event handling threads might be placed on a lower power shadow core that spins up quickly from a sleep state and can respond more quickly to input events while the remaining threads might be placed on faster, more power hungry cores. Additionally, when an app is not being displayed, only its background thread might be scheduled for execution.

4.2 The Compositor System

A typical Android application has three layers displayed on the screen comprised of 1) the application's GUI, 2) the system's navigation bar, and 3) the system's status bar (however some apps may have more or fewer layers– for example a game may hide the status and navigation bars) [9]. These three layers get flattened into a single surface by the KDS compositor. The compositor makes sure that the GUI buttons, menus, and other decorations look like they are on top of a window pane.

The KDS compositor uses a simple ordered list to determine how to layer objects into a single image (see Figure 3). 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 "lie" on top of lower z-index surfaces.

The KDS compositor is rather simple in its design since 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 the graphics for each running application.

Fig. 3. The KDS compositor is a simple layer-flattening algorithm which flattens multiple-layered surfaces into a single layer surface (bitmap). Surfaces are allocated for different GUI decorations. For example, one surface is allocated for an application to draw its graphics and 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.

```
1: function KDS_COMPOSE(surface)
2:     flattened_surface ← CREATE_BLANK_SURFACE( )
3:     for all layer in surface do
4:         flattened_surface.DRAW(layer)
5:     end for
6: end function
```

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 run in the drawing thread since it only needs to run when the results can be seen by the user.

4.3 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 4 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 uses many of the DirectFB routines that are already written in the Linux/Android kernel. As previously mentioned, DirectFB 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 will not 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.

The KDS does not automatically place 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 predict every instance where the programmer wishes to use the KDS' drawing system.

5 Connecting the KDS to Android and the Application

This section describes the set of commands supported by the KDS, how applications connect to the KDS, and how the KDS coordinates with Android. It also presents a complete example that shows how an application would use the various KDS subsystems presented in the previous section to draw a pair of overlapping rectangles to the screen when the user presses a key on the keyboard.

Fig. 4. The KDS 2-dimensional drawing system draws a composed and flattened surface to the framebuffer of the device that is passed in as a parameter. The `dev` parameter would normally be a C-style struct containing information about the device to be drawn to, such as a GPU. 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 is allocated to the application. 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 fails to ensure that the surface drawing area will not intrude on another application.

```
1: function KDS_DRAW_2D(dev, flat_surface)
2:     framebuffer ← GETFBFORAPP(dev)
3:     for all pixel in flat_surface do
4:         framebuffer[pixel.x][pixel.y] ← pixel.rgb
5:     end for
6: end function
```

5.1 The KDS Application Programmer Interface

The application programmer communicates with the KDS by obtaining a private "communication" device and then writing commands to the communication device by passing it a command block (see Figure 5). The means through which the programmer obtains this communication device are described in the next section. The KDS system supports the set of commands shown in Table 1.

Request	Inputs
CONNECT	Application
COMPOSE	Layered Surface
DRAW	Device and Flattened Surface
SURFACE	Device to allocate surface on
DRAW_THREAD	Pointer to drawing function
FORE_THREAD	Pointer to foreground function
BACK_THREAD	Pointer to background function
REDRAW	
STATE	State of window
DISCONNECT	Application
Events	
KEYBOARD_EVENT	A keyboard handler
MOUSE_EVENT	A mouse handler

TABLE 1
This table lists the KDS's API. The request is a simple C++-style, enumeration constant. Device for both the DRAW and SURFACE commands would be a device structure containing information about the device to be drawn to, such as a GPU.

The COMPOSE command takes a layered surface and returns a flattened surface using the `kds_compose` function shown in Figure 3. The DRAW command takes the flattened surface and writes it to the framebuffer using the `kds_draw_2d` function shown in Figure 4. The DRAW_THREAD, BACK_THREAD, and FORE_THREAD requests allow the programmer to attach tasks to each of these three threads. Notice that only the foreground, background, and drawing threads are exposed through the API. This is because the KDS automatically manages the event thread via the KEYBOARD_EVENT and MOUSE_EVENT commands, which allow the programmer to attach keyboard and mouse handlers to the event thread. The SURFACE command allocates a new surface on a display device and returns it to the application programmer. The REDRAW command manually forces a rerun of the drawing thread. The STATE command is used by the window manager to notify the KDS that the application changed states (from background to foreground and so forth). The KDS system uses this command to de-register an app's event handlers when the application changes state to the background and to re-register the application's event handlers when the application changes state back to the foreground. Finally, the CONNECT command connects the application to the KDS and the DISCONNECT command disconnects an application from the KDS and frees the resources.

```
1: Struct kds_command {
2:     Integer request_type
3:     Integer data_length
4:     Blob data
5: };
```

Fig. 5. 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.

```
1: function KDS_CONNECT(application)
2:     struct kds_command cmd
3:     kds ← OPEN("/dev/kds/control")
4:     cdev ← kds.CREATE_COMMUNICATION_DEVICE( )
5:     cmd.request_type ← CONNECT
6:     cmd.data ← application
7:     cmd.data_length ← SIZEOF(application)
8:     cdev.WRITE(cmd)
9:     kds.CLOSE( )
10:    return cdev
11: end function
```

Fig. 6. An application 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.

5.2 Attaching an Application to the KDS

Applications begin interacting with the KDS by calling the `kds_connect` function shown in Figure 6. `kds_connect` returns the communication device described in the previous section that the application subsequently uses to communicate with the KDS.

The `kds_connect` function attaches an application to the KDS through the device file system common in UNIX-style operating systems, including Android. As shown in Figure 6, `kds_connect` initiates this process by opening "/dev/kds/control", which returns a "kds_control_device". `kds_connect` then asks the control device to return a new, private "communication" device specifically for that application. This communication device ensures that a malicious application cannot commandeer other applications' requests to the KDS. Finally `kds_connect` attaches the application to the communication device by sending the CONNECT command to this device.

5.3 Linking Android with the KDS

Android's GUI system contains a software service called Surface Flinger which is the starting point for drawing surfaces to the screen [9]. In existing Android systems, SurfaceFlinger renders the application to the screen in a number of steps. First the application obtains a drawing surface from Android's Surface Flinger and draws its GUI to this surface. It then returns this surface to Surface Flinger, which employs a sub-service called the Hardware Compositor (HWC) to flatten the application layer and the system-generated navigation bar and status bar layers into a single surface. Finally, the composed layer is drawn to the display device using the Hardware Abstraction Layer's (HAL) interface to the drawing device. This process is illustrated in Figure 7.a.

The KDS modifies this rendering process in three ways:

1) It intercepts the application's surface request to SurfaceFlinger, gets a surface from SurfaceFlinger and returns it to the application. The application draws to this surface in exactly the same manner that it would draw to a SurfaceFlinger surface.
2) When the application is ready to have the surface composited, the surface is given to the KDS compositor, which flattens the application and system-provided surfaces.
3) The flattened surface is directly rendered to the framebuffer using the KDS-provided drawing function.

The updated rendering pipeline is shown in Figure 7.b. Our KDS implementation uses its own compositing/drawing routines for increased flexibility and performance on our NVIDIA TK1 testing platform. The TK1 does not have an actual hardware composer so our routines make use of the GPU instead. As an added benefit, a programmer might be able to perform offscreen drawing and buffer swapping with these routines. However, Android has incentivized the development of actual hardware devices to perform composition so implementors on other mobile devices that provide such a hardware device might find it equally feasible to simply use the device's hardware composer and eliminate the KDS's composition and drawing routines. In this case the implementor could revert to the rendering pipeline shown in Figure 7.a (the KDS would no longer need to intercept the surface request since the interception is done for the benefit of the KDS compositor and drawing routines).

When an application wishes to draw, it performs the same actions that it would with Android's current drawing system. Since we modified Surface Flinger to work with the KDS, the modified Surface Flinger code 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 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 do not need to be modified 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.

5.4 Example Application Interaction with the KDS

Figure 8 shows how an application would use the commands enumerated in the previous section to interact with the KDS compositing and drawing subsystems in order to draw graphics to the screen and Figure 9 shows how an application would use the KDS thread subsystem to set up event handling and to allocate code to different KDS threads. These two figures together illustrate how the application layer interacts with the KDS layer to draw two overlapping rectangles when the user presses a key on the keyboard.

While the interaction with the KDS communication device might seem complex and esoteric, we have developed a user API that includes higher-level functions, such as the `kds_connect` function, 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. Hence, existing Android applications can work with the KDS without any modification.

6 Experimental Results

The primary goal of the kernel display server is to conserve power and reduce latency in mobile devices. It does so in two ways: 1) by enabling other software strategies to more effectively manage power consumption, and 2) by eliminating system calls between the application layer and the kernel layer and by streamlining the movement of events from the kernel to the application. This section first describes our testing environment. Then it shows the power savings we have achieved by 1) coupling the KDS with other power management saving strategies and 2) by running the KDS along. This section concludes by discussing the battery life improvements that might be expected using the KDS with the two other power saving strategies or by itself.

6.1 Testing Environment

We designed three applications that would simulate how an actual user would interact with their mobile device. One was a program where the user traced a spiral with a stylus while randomly clicking and releasing the stylus to create mouse down/up events. The second program was a text messaging app. The third program was a video playing system which directed large amounts of data to the framebuffer with a minimal number of input events.

The spiral application tests how well the KDS handles events that are relatively evenly spaced, the text message app tests how well the KDS handles bursty events with occasional long delays between events, and the video application tests how well the KDS handles drawing GUI components to the screen. Together these three apps provide a good representation of the types of demands apps typically place on the kernel. For example, games match the regularly-spaced event profile of the spiral app, many social

Application → Surface Flinger → HWC → HAL → GPU
(a)
Application → KDS.SURFACE → Surface Flinger → kds_compose → kds_draw_2d → HAL → GPU
(b)

Fig. 7. The Android rendering pipeline (a) and the modified KDS/Android rendering pipeline (b). The HW composer is effectively bypassed by the KDS compositing and drawing routines.

```
1: function DRAW_SCENE(kds_device) ▷ kds_device is the communication device returned from kds_connect
2:     struct kds_command cmd
3:     Surface surf ← SFLINGER.CREATE_SURFACE( )
4:     Layer lay1 ← surf.CREATE_LAYER(0)     ▷ z-index 0
5:     Layer lay1 ← surf.CREATE_LAYER(1)     ▷ z-index 1
6:     lay1.RECTANGLE(0, 0, 15, 15)
7:     lay2.RECTANGLE(10, 10, 15, 15)
8:     cmd.request_type ← COMPOSE
9:     cmd.data ← surf
10:    cmd.data_length ← surf.size
11:    kds_device.WRITE(cmd)
12:    FlatSurface flat_surface ← cmd.data
13:    cmd.request_type ← DRAW
14:    cmd.data_length ←SIZEOF(flat_surface)
15:    kds_device.WRITE(cmd)
16: end function
```

Fig. 8. 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.

```
1: Device kds_device     ▷ The kds communication device for this app
2: struct kds_command cmd
3: function KEY_CALLBACK(event)
4:                                   ▷ Redraw the scene
5:     cmd.request_type ← REDRAW
6:     kds_device.WRITE(cmd)
7: end function
8: function DRAW_THREAD
9:     DRAW_SCENE(kds_device) ▷ Drawing function from Fig 8.
10: end function
11: function BG_THREAD
12:                ▷ Perform actions, such as decoding audio.
13: end function
14: function MAIN( )
15:     kds_device ← KDS_CONNECT(this)
16:     cmd.request_type ← KEYBOARD_EVENT
17:     cmd.data ← KEY_CALLBACK
18:     cmd.data_length ← SIZEOF(KEY_CALLBACK)
19:     kds_device.WRITE(cmd)
20:     cmd.request_type ← DRAWING_THREAD
21:     cmd.data ← DRAW_THREAD
22:     cmd.data_length ← SIZEOF(DRAW_THREAD)
23:     kds_device.WRITE(cmd)
24:     cmd.request_type ← BACK_THREAD
25:     cmd.data ← BG_THREAD
26:     cmd.data_length ← SIZEOF(BG_THREAD)
27:     kds_device.WRITE(cmd)
28:     cmd.request_type ← DETACH
29:     cmd.data ← NULL
30:     cmd.data_length ← 0
31:     kds_device.WRITE(cmd)
32: end function
```

Fig. 9. 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 can automatically register and de-register events with our kernel implementation of the ESM model depending on the foreground or background state of the application.

media apps match the bursty event with long pauses profile of the text messaging app, and music-playing apps like Pandora, audio-playing apps like Sirius, and video-playing apps like YouTube and ESPN, match the computational but low event generation profile of the video app.

The applications were run on a 32-bit NVIDIA TK1 reference board [30]. Power consumption was directly determined using the TK1's onboard power consumption monitoring system. To measure latency, we designed a high-resolution timing system. We used one of the NVIDIA K1 CPU's high-resolution timers to measure latency within one millisecond resolution. The timer was set to a fixed 1 kHZ rate to provide a wall clock timer. The timer operated by automatically increasing its internal counter by one for each cycle. When the stylus made an input, the kernel recorded the wall clock time. We then placed code in the application's event handler to record the wall clock time (now presumably advanced since the event was generated). The difference between the two is the event's latency.

6.1.1 Spiral Tracing App

Figure 10 illustrates our spiral tracing application in which the user traced a spiral while randomly pressing and releasing the stylus. We recorded the events and then replayed this record to have a consistent test between our KDS and the existing Android system. The test generated 1614 events over 60 seconds, which corresponds to an event input rate of 26.9 events per second.

We ran the pre-recorded script on both systems ten times and then averaged the results.

6.1.2 Text Messaging App

The text-messaging app generates random messages using a simulated keyboard and sends them to a server using the standard Android text messaging service. The server then responds with a random message to simulate a reply. The testing program simulates a user reading a message

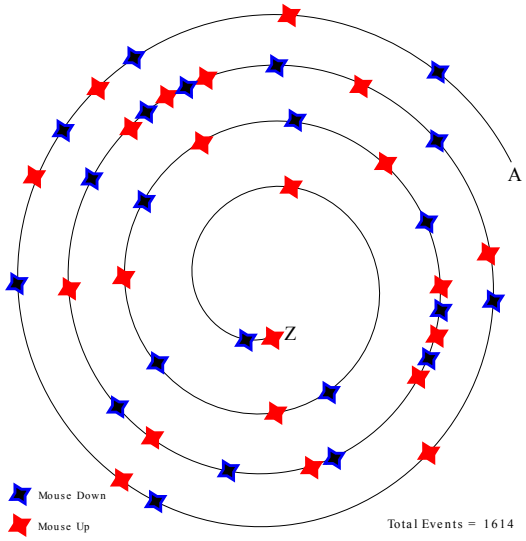

Fig. 10. In the stylus motion/click test, the user started at point A and manually traced the spiral to point Z while randomly pressing and releasing the stylus (shown as mouse down and up events). The blue stars indicate where the user pressed the stylus and the red stars indicated where the user released the stylus as the user traced the spiral.

by remaining idle for 5 seconds after receiving the server's response and then generates a return text message. The testing was conducted over 20 second intervals resulting in an average of 3 messages being sent to the server and 3 messages being received from the server.

6.1.3 *Video Playing App*

The video playing application displayed a 30-second movie clip and was designed to test if the direct, kernel control of the framebuffer would have any impact on power consumption. For the entire duration of this test, no events were handled, which allowed us to control for the event model. 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. We ran the test, with the same parameters ten times and then averaged the results.

6.2 Power Savings

As noted earlier, the KDS makes it possible for the kernel to implement two new power saving strategies. First, the kernel can switch from the existing pull event-handling model that utilizes polling loops to a push event-handling model that eliminates polling loops. Second, the kernel can more effectively schedule applications by taking advantage of the threads–event handling, display, foreground, and background–to which an app assigns its functions. Using both of these power saving strategies along with the KDS implementation, we achieved the following savings [4]:

- Spiral App: Power was reduced by up to 185 milliwatts (roughly 23%) and latency was reduced by up to 6.7 milliseconds (Figure 11).
- Text Messaging App: Power was reduced by up to 218 milliwatts (roughly 30%) and latency was reduced by up to 17 milliseconds (Figure 12).
- Video App: Power was reduced by an average of 182 milliwatts (roughly 6%) but at peak rates power was reduced by almost 400 milliwatts (roughly 11%). Because of the lack of events there was no latency for this test (Figure 13).

By itself the KDS produces less savings. For the spiral app our KDS had a slight latency reduction of 1.1 milliseconds and reduced power consumption by 11.7 milliwatts (roughly 1.5%). For the video app our KDS system showed an average power reduction of 47.4 milliwatts (roughly 1.3%) and a peak power reduction of 82.1 milliwatts (roughly 2.3%). Latency was not an experimental variable for this test because the test involved no event handling. The text messaging app was implemented after we had implemented the other power saving strategies so we do not have an independent measurement for the KDS for the text messaging app. However it is likely that the power consumption savings would be comparable to the other two apps, roughly 1 to 2%, because the power savings is derived from the streamlining of the event handling and rendering pipelines, which while helpful, is not going to lead to significant power savings by itself.

It is important to note that while the KDS by itself does not generate significant power savings, it does achieve significant power savings and latency reductions when combined with the ESM model [1] and a modified scheduler [4] because 1) it eliminates polling loops in user space that would otherwise prevent the CPU from being placed in lower power-saving modes, and 2) it introduces threads that provide important information to the scheduler about what type of GUI-related activity is being performed by each GUI task.

6.3 Battery Consumption Analysis

Based on our experimental results, we estimate that by itself the KDS would improve battery life by 4.35% and with the event handling and scheduling strategies it would improve battery life by 30%. We made this estimate by examining current battery capacities and estimating how much time they would power a mobile device. We then used the power consumption improvements from our experiment results to calculate an estimated battery life improvement. Table 2 shows standard battery capacities and the amount of time a mobile device could be powered by them. We added a column that displays the battery life that may be expected by using our KDS alone and by using our KDS with the two other power saving strategies we have devised. For example, a battery that could last 10 hours without the KDS would see its life extended by .435 hours or roughly 26 minutes with the KDS alone and by 3 hours with the KDS and the two other power saving strategies.

7 Potential Drawbacks of the KDS

Moving the display server from user space into kernel space has three potential drawbacks. First it reduces flexibility because it is "hard coded" into the kernel and hence the display server can no longer be swapped out for another display server. However, this drawback could be mitigated if the KDS were re-implemented in a commercial product.

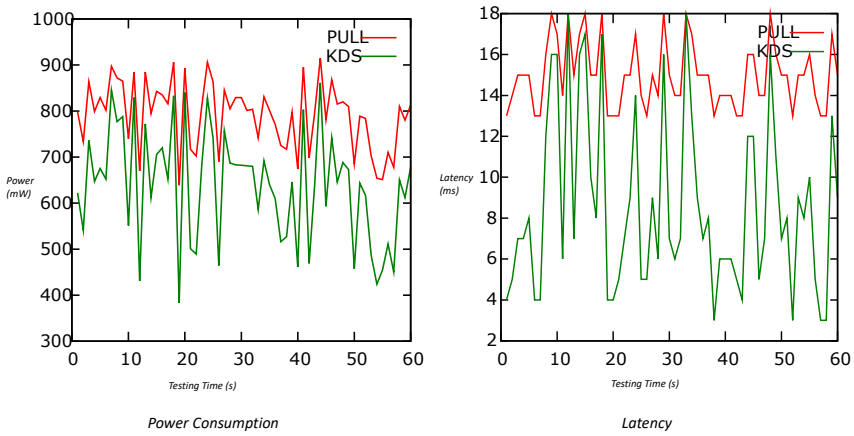

Fig. 11. Power consumption in milliwatts and latency in milliseconds for the spiral tracing program during a 60-second user interaction when run using the pull model, red, and the KDS model, green.

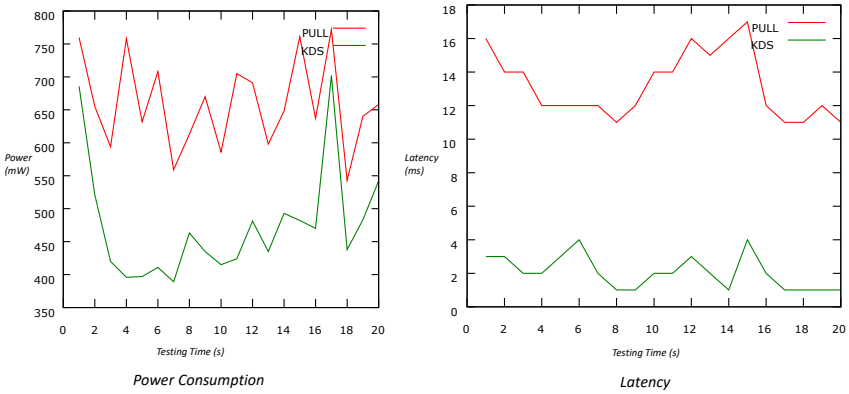

Fig. 12. Power consumption in milliwatts and latency in milliseconds for the text message program during a 20-second user interaction when run using the pull model, red, and the KDS model, green.

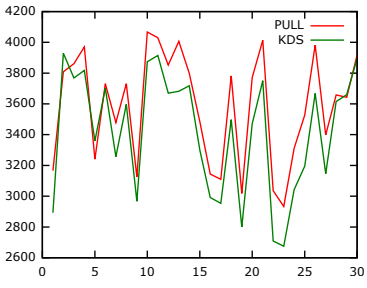

Fig. 13. Power consumption in milliwatts for a 30 second video clip when run using the pull model, red, and the KDS model, green.

Battery Capacity (mAh)	Life w/o KDS (hours)	Life w/ KDS (hours)	Life w/ KDS & other power strategies
1000	3.0	3.1	3.9
2000	6.0	6.3	7.8
4000	10.0	10.4	13.0
8000	15.0	15.7	19.5

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. NOTE: This table does not consider user habits or every device configuration and is only a summary of what could be expected using our model with varying battery capacities.

All display servers must use the same kernel functions so it is possible to design a drop-in display server module, much like drivers, file systems, and other systems are currently handled in Linux. It should also be noted that the loss of flexibility is more of a theoretical concern than a practical concern because OS developers long ago sacrificed flexibility when they chose to tightly integrate display servers with their OS's, so in practice this flexibility was already lost.

Second, the KDS could introduce security issues by introducing new attack vectors. However, many of the essential event handling and rendering routines used by the display server have already moved into the kernel. The attack vectors for our KDS would have to occur either through the registration of callback functions with events or with the calls to the rendering routines. In the former case, any function pointer handed to the registration routines will be called by our event handlers, but the kernel will not jump to the memory address unless the pointer lies in the appropriate application space. These functions also cannot crash the display server because they are restricted to user space. Hence if they crash then the application will crash but

the display server will be unaffected. Once an application crashes, it exits and our implementation cleans up after it just like a normally terminated application so the KDS does not increase the chance of a server crash if an application function crashes.

It also should not be possible to attack the KDS through its rendering routines. The KDS uses existing kernel functions to write to the drawing surface. All the KDS does is copy bits from user space into frame buffers. Even if the attacker is trying to pass the KDS malicious code, the drawing routines will simply interpret it as pixels to be drawn to the screen. Buffer overflow situations do not apply because we are not providing the application with a variable-sized storage area. Any data we receive from the application is fixed size, either a pointer or a screen area. Also as noted in Figure 4, if the application attempts to draw a pixel that exceeds the bounds of the applications window, then an error is thrown. The KDS does not make any kernel memory areas directly accessible by the application so the application cannot either inspect kernel memory nor modify it without invoking the KDS gatekeeper functions. Finally, the KDS allocates new private communication devices to each application so it is not possible for malicious applications to commandeer other applications' requests to the KDS as long as the kernel's normal protection mechanisms prevent malicious applications from accessing other applications' process space.

The final drawback of the KDS is that the kernel is bigger. Much of the KDS has to be compiled into the kernel, which grows the size of the kernel image by about 22 kilobytes. However this increase in size is not significant relative to the size of existing kernels. Additionally, because the KDS is part of a research project rather than a commercial implementation, we did not make it modular. It might be possible to reduce the size of the KDS's kernel image by using kernel modules.

In sum, the drawbacks of moving the display server to the kernel are relatively mild, and the flexibility and size issues could be further mitigated in a commercial implementation.

8 Conclusion and Future Work

Existing display servers for mobile devices run in user space largely because of outdated decisions made in the 1980s when GUIs were not an integral part of the OS. OS developers failure to migrate display servers to the kernel has had two negative impacts on OS performance: 1) display servers duplicate much of the event handling and rendering functionality that has been migrated to the OS, and 2) OS's are unable to fully take advantage of recent power saving hardware architectures because display servers are forced to use polling loops that constantly rouse the CPU and because display servers cannot provide important scheduling information that could allow the OS to de-schedule background apps.

In this paper, we have described the design and implementation of a kernel level display server optimized for mobile devices. This display server divides the workload of an app into four threads–an event handling thread, a drawing thread, a foreground thread, and a background thread–that helps improve the scheduling of apps. Middleware can handle the allocation of tasks to the event handling and drawing threads, so an app developer only needs to write callback procedures and assign them to either the foreground or background thread.

Our KDS streamlines the implementation of the OS by eliminating the polling loops required by application-layer displayer servers and by permitting the display server to directly interact with the kernel's data structures rather than communicating with them via system calls. 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 as compared with application-level display servers. The biggest drawback in moving the display server to the kernel is some loss of flexibility because the display server cannot be as easily swapped out with another display server. However, in practice OS developers long ago made the choice to tightly integrate the display server with the OS and hence this flexibility was really already lost.

Moving the display server into the kernel has allowed us to implement a stream-lined event push model [1] and an improved scheduler [4]. The event model eliminates polling loops that were required to fetch events into the application. Eliminating these polling loops leads to less frequent rousing of the CPU and allows the scheduler to more frequently place the CPU in lower power-consuming sleep states. The scheduler takes advantage of the KDS's division of an app into four threads. For example, the drawing, event handling, and foreground threads can be suspended when an app is not the foreground app. Additionally, the scheduler can use its knowledge of the threads to assign them to different cores. For example, it might assign the event handling thread to a so-called shadow core that spins up faster from a sleeping state but is not as fast as more power-hungry cores. These cores might be assigned to the drawing, foreground, and background tasks. We have published some preliminary information on this scheduler [4] and plan to report more fully on it in the future.

These two power saving strategies have achieved almost a 30% improvement in battery performance and a 17ms reduction in event handling latency. Neither power saving strategy would be possible without the movement of the display server to the kernel. Even without the implementation of these power saving strategies, the KDS by itself improves battery life by 4.35%, which translates to about ten extra minutes of battery life for a typical mobile phone or thirty extra minutes of battery life for a typical tablet computer. The KDS also reduces latency by an average of 1.1 milliseconds.

Acknowledgments

The research reported in this paper is being supported by NSF grant CNS-1617198.

References

[1] S. Marz and B. V. Zanden, "Reducing power consumption and latency in mobile devices using an event stream model," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 1, pp. 11:1–11:24, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2964203

[2] E. Vicente, R. Matias, L. Borges, and A. Macedo, "Evaluation of compound system calls in the linux kernel," *2013 III Brazilian Symposium on Computing Systems Engineering*, vol. 0, pp. 164–169, 2011.
[3] D. O. Kropp, *Direct Frame Buffer (DirectFB)*, 2014.
[4] S. G. Marz, "Reducing power consumption and latency in mobile devices using a push event stream model, kernel display server, and gui scheduler," Ph.D. dissertation, The University of Tennessee - Knoxvile, 2016.
[5] G. Lehey, "Setting up x11: A no-tears guide to xfree86 configuration." *Linux J.*, vol. 1995, no. 15es, Jul. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=324855.324859
[6] R. R. Repasky, "Easy access to remote graphical unix applications for windows users," in *Proceedings of the 32Nd Annual ACM SIGUCCS Conference on User Services*, ser. SIGUCCS '04. New York, NY, USA: ACM, 2004, pp. 357–359. [Online]. Available: http://doi.acm.org/10.1145/1027802.1027886
[7] S. Anderson, R. Mor, and A. Coopersmith, "X transport interface," 2002.
[8] K. Hogsberg, *The Wayland Protocol*, https://wayland.freedesktop.org/docs/html/, Freedesktop.org, 2012. [Online]. Available: https://wayland.freedesktop.org/docs/html/
[9] *Surface Flinger and Hardware Composer*, 1st ed., https://source.android.com/devices/graphics/arch-sf-hwc, Open Android Project, May 2017. [Online]. Available: https://source.android.com/devices/graphics/arch-sf-hwc
[10] C. Toporek, C. Stone, and J. McIntosh, *MAC OS X in a Nutshell: A Desktop Quick Reference*, 1st ed. Sebastopol, California, USA: O'Reilly Media, 2003.
[11] J. Siracusa, "Mac os x 10.4 tiger," *Ars Technica*, vol. 2005, Apr. 2005. [Online]. Available: https://arstechnica.com/apple/2005/04/macosx-10-4/13/
[12] Y. Ping-Peng, C. Gang, D. Jin-Xiang, and H. Wei-Li, "An event and service interacting model and event detection based on the borker/service model," in *The Sixth International Conferance on Computer Supported Cooperative Work in Design*, ser. CSCWD '01. Ottowa, Canada: NRC Research Press, 2001, pp. 20–24. [Online]. Available: http://www.nrcresearchpress.com/doi/book/10.1139/9780660184937#.WV0ZF4jyuUk
[13] P. NeiraAyuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and userspace in linux using netlink sockets," *Software Practice and Experience*, vol. 40, pp. 797–810, August 2010.
[14] *The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure*, 1st ed., http://dri.sourceforge.net/doc/drm_low_level.html, Precision Insight, Inc., May 1999. [Online]. Available: http://dri.sourceforge.net/doc/drm_low_level.html
[15] A. Singhai and J. Bose, "Reducing power consumption in graphic intensive Android applications," in *The Sixth International Conference on Communication Systems and Networks*, ser. COMSNETS '14. Santa Barbara, CA, USA: IEEE, 2014, pp. 1–4. [Online]. Available: https://www.ieee.org/conferences_events/conferences/conferencedetails/index.html?Conf_ID=32213
[16] W. L. Bircher and L. John, "Predictive power management for multi-core processors," in *Proceedings of the 2010 International Conference on Computer Architecture*, ser. ISCA'10. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 243–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24322-6_21
[17] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307661
[18] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. Corner, and E. Berger, "Eon: a language and runtime system for perpetual systems," in *ENSS '07*, 2007.
[19] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu, "Et programming language," in *OOPSLA '07*, 2012.
[20] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *USENIX Annual Technical Conference*, Boston, MA, USA, jun 2010, pp. 271–284.
[21] H.-C. Shih and K. Wang, "An adaptive hybrid dynamic power management algorithm for mobile devices," *Comput. Netw.*, vol. 56, no. 2, pp. 548–565, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2011.10.005
[22] U. A. Khan and B. Rinner, "Online learning of timeout policies for dynamic power management," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 96:1–96:25, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2529992
[23] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878982
[24] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: Application energy metering framework for android smartphone using kernel activity monitoring," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 387–400. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoon
[25] A. Rice and S. Hay, "Decomposing power measurements for mobile devices," in *2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2010, pp. 70–78.
[26] R. Murmuria, J. Medsger, A. Stavrou, and J. M. Voas, "Mobile application and device power usage measurements," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, June 2012, pp. 147–156.
[27] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski, "A system-level model for runtime power estimation on mobile devices," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, Dec 2010, pp. 27–34.
[28] J. Kindelsberger, F. Willnecker, and H. Krcmar, "Long-term power demand recording of running mobile applications," in *2015 IEEE 10th International Conference on Global Software Engineering Workshops*, July 2015, pp. 18–22.
[29] C. Qin and F. Rusu, "Scalable i/o-bound parallel incremental gradient descent for big data analytics in glade," in *Proceedings of the Second Workshop on Data Analytics in the Cloud*, ser. DanaC '13. New York, NY, USA: ACM, 2013, pp. 16–20. [Online]. Available: http://doi.acm.org/10.1145/2486767.2486771
[30] NVIDIA, "Nvidia tegra k1 processor specifications," http://www.nvidia.com/object/tegra-k1-processor.html, 2015.

Stephen Marz received his B.S. in Computer Science from the Illinois Institute of Technology (2004), and an M.S.I.T (2010) from Kaplan University, and Ph.D. in Computer Science (2016) from the University of Tennessee. He has been involved in operating system development (since 1996), graphical user interface development (since 2004), and Android OS and middleware development (since 2010). He is currently a Lecturer in the Electrical Engineering and Computer Science department at the University of Tennessee in Knoxville. He is a member of the IEEE and ACM.

Brad Vander Zanden received his BS degree in Accounting and Computer Science from The Ohio State University in 1982 and his MS and PhD degrees in Computer Science from Cornell in 1985 and 1989. He was a Post Doctoral researcher at CMU from 1988 to 1990. His research interests encompass graphical user interfaces, instructional technology, and Android OS and middleware development. He is currently a Professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. He is a member of the ACM.

Wei Gao received his BE degree in Electrical Engineering from the University of Science and Technology of China in 2005 and PhD degree in Computer Science from Pennsylvania State University in 2012. He is currently an Associate Professor in the Department of Electrical and Computer Engineering at the University of Pittsburgh. His research interests include wireless and mobile network systems, mobile social networks, cyber-physical systems, and pervasive and mobile computing. He is a member of the IEEE.