Reducing Power Consumption and Latency in Mobile Devices by using a GUI Scheduler

STEPHEN MARZ, The University of Tennessee
BRAD VANDER ZANDEN, The University of Tennessee
WEI GAO, The University of Pittsburgh

Mobile devices employ the same scheduling algorithms as their desktop cousins. Unfortunately, these algorithms do not take advantage of an application's heavy reliance on graphical user interfaces (GUIs). In this article, we discuss how we can partition the functionality of a GUI into 4 threads–an event handling thread, a display thread, and two threads for handling background and foreground tasks–and then use the information imparted by these threads to schedule applications in a way that reduces power consumption. We also describe how we can combine our scheduling algorithm with our previous work on push event models to obtain additional power savings and to also obtain reductions in latency, which is the elapsed time between when an input event arrives and when the app starts processing it. These combined approaches yield power savings approaching 30% in applications that alternate "bursty" events with long idle periods, and up to a 17.1 millisecond reduction in latency.

CCS Concepts: • **Human-centered computing** → **Graphical user interfaces**; **Mobile devices**; • **Computer systems organization** → *Embedded software*;

Additional Key Words and Phrases: Graphical user interface scheduler, graphical user interfaces, operating systems

ACM Reference format:
Stephen Marz, Brad Vander Zanden, and Wei Gao. 2018. Reducing Power Consumption and Latency in Mobile Devices by using a GUI Scheduler. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2018), 18 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Most process schedulers for consumer-oriented, mobile devices employ the same scheduling algorithms as their desktop cousins. For example, Android schedules applications using the completely fair scheduler (CFS) that is also used by desktop Linux machines [12]. However, mobile devices have two characteristics that make it desirable to develop more sophisticated schedulers. First, they have a limited power source, which increases the importance of ensuring that the CPU does not perform useless work, such as executing a polling loop for a graphical application that is completely obscured by other applications. Second, most applications that run on mobile devices involve graphical user interfaces that lend themselves to more nuanced scheduling. For example,

This work is supported by the National Science Foundation, under grant CNS-1617198.
Authors' addresses: S. Marz and B. Vander Zanden, Department of Electrical Engineering and Computer Science, The University of Tennessee, 1520 Middle Drive, Min H. Kao Building, Knoxville, Tennessee, USA, 37996. W. Gao, Department of Electrical and Computer Engineering, 3700 O'Hara Street, Benedum Hall, University of Pittsburgh, Pittsburgh, PA 15261.

1539-9087/2018/3-ART39 $15.00
https://doi.org/0000001.0000001

unlike desktop computers where one might have many overlapping windows, mobile devices tend to have only one application visible at a time [15]. Some newer devices with bigger screens will split the screen to display more than one app but even on these devices it is unusual to display more than 2 apps at a time. If an app is not visible, then there is no need to schedule it, or its power consuming polling loop.

The reason that existing desktop schedulers cannot more effectively schedule GUI-oriented apps is that they only consider the CPU when deciding which applications get scheduled. They do not take into account other factors that can influence scheduling, such as the graphics processing unit (GPU), the event subsystem, the I/O subsystem and the memory controller. Instead, they use a binary wait or execute model where the process is either scheduled (execute) or not scheduled (wait). A process waits until some event occurs, such as the completion of I/O, the passage of a certain amount of time, the acquisition of a lock, or the arrival of an input event. Unfortunately a GUI-oriented app "busy-waits" because the CPU executes a polling loop to check for events for that app. Since existing desktop schedulers do not consider whether the app has the input focus and can actually receive events (i.e., they ignore the state of the event subsystem), they execute this polling loop regardless of whether or not the app can actually receive any events. Application programmers can try to tell the scheduler that the app can be put to sleep, but usually they prefer not to in order to keep their code portable [2].

To address this shortcoming of existing schedulers, we have designed and implemented a new scheduler called the guiS scheduler or "GUI Scheduler" that takes advantage of the fact that almost all apps on mobile systems are graphically oriented. guiS attempts to reduce power consumption in four ways: (a) coordinate process scheduling for GUI-specific situations, such as when an application is running but is not actively visible to the user, (b) improving the efficiency of hardware timer interrupts (c) more intelligently assigning tasks an app must perform to either low or high power cores, and (c) shutting down certain CPU cores when the apps they are executing have no pending input events and are not executing background tasks.

Previously, we have implemented two other kernel subsystems with which guiS coordinates to improve its scheduling decisions. First we implemented an event push model, called the Event Stream Model (ESM), that eliminates event polling loops by pushing events to the appropriate application using vector controller interrupts [8]. By eliminating polling loops, we make it possible for a scheduler to avoid scheduling apps until they have an actual event waiting to be processed. Previously, apps had to be constantly scheduled so that their polling loop could canvas the input devices and determine if there was an event waiting for the app. Second, we have implemented a kernel display server called the KDS that divides GUI applications into four distinct threads, an event handling thread, a display thread, a foreground thread for long-running tasks that handle constant updates to an app while it is in foreground mode (an app such as Pandora or Sirius could be in the foreground without being visible), and a background thread for background tasks that handle less immediate responses such as the decoding of audio as well as activities that should proceed even when the app is neither visible nor in the foreground [9]. These four threads constitute well-defined task information that guiS can use to perform more intelligent scheduling.

Using these two subsystems, the GUI scheduler improves upon current schedulers by considering multiple factors when scheduling applications, including GUI window state, GPU contexts, I/O load, and event load. For example, the KDS communicates to the scheduler when a GUI window has been placed into the background. guiS considers this information and makes the appropriate decision about how the application is scheduled, which often involves de-scheduling the application.

guiS also incorporates several improvements that Hsiu et al. [5] made in their heterogeneous CPU core scheduler. A heterogeneous CPU has both so-called shadow cores and normal cores. Shadow cores consume less power than normal cores but can also run more slowly [16]. Hsiu's work

determined which proportion of shadow cores to normal cores is best for balancing performance and power consumption in GUI applications. We adapted this result by allowing guiS to coordinate with our ESM in order to identify which applications need to run. When the scheduler determines that no GUI applications need to run, such as 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 Section 5 for more details).

The rest of this paper is organized as follows. Section 2 describes existing scheduling strategies and existing power saving strategies for mobile devices. Section 3 presents an overview of our guiS scheduler. Section 4 describes in detail how it schedules processes, Section 5 describes how guiS manages the assignment of processes to cores, and Section 6 describes how it improves the efficiency of hardware ticks once polling loops have been eliminated. Section 7 describes experiments we have performed with our guiS scheduler on several apps and the power and latency savings it achieves, and Section 8 summarizes the work and presents our conclusions.

2 RELATED WORK

This section begins by examining the strategies used by existing schedulers, both desktop schedulers and mobile device schedulers. It then looks at work that has been done with scheduling heterogeneous cores that have different power consumption and performance capabilities. Finally it concludes with an examination of the most common alternative power saving approaches that have been utilized for mobile devices.

2.1 Scheduling Algorithms

Most scheduling algorithms used by existing operating systems use some variation of a priority-based scheme that tries to ensure that all processes are given a share of the CPU while still giving preference to higher-priority processes. The current scheduler used by the Android operating system and other Linux-based operating systems is the “Completely Fair Scheduler” (CFS) [12]. This scheduler aims to provide an equal (fair) proportion of CPU to each running process and to allow for priorities [17]. Those processes that have not received an equal portion of CPU time are prioritized over those that have. Furthermore, the scheduler balances processes with a higher priority with those with a lower priority to enable higher priority processes to have more CPU usage without starving the lower priority processes. However, CFS does not use information about GUI applications to improve its scheduling.

The scheduler used in Microsoft’s Windows operating system is a priority scheduler that assigns all processes a priority between 0 and 31. The scheduler assigns time slices in a round robin fashion to all processes with the highest existing priority level. For example, if the highest existing processes have a priority of 29, then all processes with a priority level of 29 will be scheduled in round robin fashion. If no process at the highest existing priority level is ready to run, then the Windows scheduler will start allocating time slices in a round robin fashion to processes at the next highest priority level. If a process with a higher priority level becomes ready to run, the lower priority process will be suspended and the higher priority level process will begin running. The scheduler can dynamically boost or lower priority levels to ensure that CPU starvation does not occur for any process. Windows also has a direct rendering scheme called DirectDraw that allows a GUI to bypass some drawing layers and more expeditiously render a GUI [10]. However, the scheduler itself does not distinguish between GUI and non-GUI apps.

The Mac desktop and iOS schedulers use a derivation of the Mach kernel's scheduler [1]. These schedulers use a priority queue-based scheduling system that divides processes into four priority bands with each band having multiple priority-levels. Processes can migrate between priority-levels within a band to try to prevent CPU starvation but typically will not migrate between bands. The bands do not distinguish between GUI applications and non-GUI applications. However, one of the priority bands is for real-time processes, and if a GUI has real-time constraints, it can request to be a real-time process. A real-time process can request that it receive x out of the next y CPU units. The scheduler will try to honor these requests, but may not be able to, thus making it a "soft real time" scheduling system. Hence there are ways for a GUI application to obtain scheduling preferences, but this is different from using GUI-specific information to improve the scheduling process.

The guiS scheduler described in this paper differs from these schedulers in that it takes advantage of the unique features of mobile device software, specifically their GUI-oriented focus, to improve the scheduling process. guiS is a heavily modified CFS scheduler that uses information about GUI apps to 1) avoid scheduling background GUI apps, and 2) place higher latency and lighter weight GUI tasks on lower power consuming cores and heavier weight, lower latency GUI tasks on higher power but faster cores.

2.2 Scheduling with 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. Li et al. modified the Linux kernel's load balancing algorithms for the big.LITTLE architecture [7]. ARM's big.LITTLE system is an energy optimization scheme that pairs high performing, but power consuming cores with lower performing, but lower power consuming cores. Li's work emphasized how to allocate processes efficiently among the heterogeneous cores. In our guiS scheduler we have employed their idea of placing the lightweight event dispatching and event handling tasks on the low energy, power-reducing cores and placing the more computationally expensive, longer-running application tasks on the faster, but higher energy cores.

Seehwan Yoo, et al describe several conditions which can decrease the efficiency of the big.LITTLE or shadow core processors [18]. They lay out several ways to achieve increased power efficiency without sacrificing performance. Our guiS scheduler 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. For example, when the cores have been powered down, guiS causes initial event processing to occur on the lower power cores that spin up faster while simultaneously spinning up a higher power core that will execute the drawing procedure that updates the app's window.

Hsiu, Tseng, et al. examined the current set of process schedulers and determined they are not suitable for mobile devices with heterogeneous CPU cores [5]. Their research shows that with the conventional schedulers, such as the Completely Fair Scheduler (CFS) and its variants, energy efficiency and performance are not maximized. The research presented in this paper confirms their finding by implementing a new scheduler that focuses on energy efficiency and achieving power reductions of as much as 30%. Their work helped guide the design of our guiS scheduler. For example, guiS can determine how to best balance event handling among a set of cores and will attempt to place IO-intensive apps on slower, lower power cores and CPU-intensive apps on faster, higher power cores.

Gaspar, Tanica, Tomàs, Ilic, and Sousa have proposed a framework for an application-aware task management system for mobile devices using heterogeneous CPU cores [4]. 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 studied the effects of using the larger, power hungry cores when scheduling web page loading for Chromium and Firefox on mobile devices [3]. They achieved an average of a roughly 24% energy savings using Chromium on a big.LITTLE smart phone, which nicely aligns with our own findings that we can achieve up to 30% reductions in power consumption for some apps by smartly managing GUI apps on the different cores.

2.3 Alternative Power Saving Methods

Our techniques for saving power involve modifying the kernel software so that application programmers can realize power savings without any explicit effort. By contrast, mobile OS's currently provide application-level techniques for conserving power. The most common such techniques are "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 application is performing [13]. 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 [6].

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. For example, when the LCD screen on a mobile phone dims and then turns off, it is adhering 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. A drawback of aggressive sleep policies is that they can increase application latency when they inaccurately predict that a device can be powered down because a lag occurs before a sleeping device fully awakens.

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 KDS 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, such as the downloading of updated news articles, and hence the LCD screen can be powered down when only the background thread is executing. The LCD is free to power down since this activity does not require the LCD screen. In current mobile OS implementations, the LCD screen must use an aggressive timer to determine when to power down since it does not know what the application is doing.

3 GUI SCHEDULER (GUIS) OVERVIEW

guiS is a modified version of the "Completely Fair Scheduler" (CFS) currently employed in the Android OS [12]. 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.

Our GUI scheduler differs from the CFS scheduler in its explicit recognition of and handling of GUI applications. guiS will not schedule some GUI threads if it knows that the GUI's application is in the background, and guiS will also schedule GUI threads on different power-consuming cores depending on the type of work being performed by the thread.

The original motivation for guiS derived from the observation that most mobile devices display only one app at a time. Any app whose window is covered is unable to directly receive user input events and has no need to display graphics so it can be theoretically disabled. Newer mobile devices have bigger displays that permit more than one app to be displayed at a time, but for the most part, the apps are displayed in non-overlapping windows which still makes it easy to determine whether or not an app's display is completely covered and thus not visible.

An obvious problem arises with this simple observation however. An application may want to execute instructions even though it is in the background. For example, an application may want to play music even though it may not be able to receive user inputs or display its graphics. We would like to make it as easy as possible for the scheduler to determine when an application can be completely de-scheduled while placing a minimal burden on the application programmer to tell the scheduler what the app is doing.

This motivation drove the design of the Kernel Display Server (KDS), in which we divided an application's tasks into four threads: event handling, display, foreground tasks, and background tasks. A display server is traditionally a piece of middle ware in the application layer that distributes events to an app and delivers drawing commands from the app to the kernel. Hence if we define events as GUI input and drawing commands as GUI output, then all GUI I/O is handled through the display server. Applications typically register call back procedures with event handlers and these call back procedures execute when the appropriate event is received. These call back procedures are foreground tasks as are some longer running tasks, such as video decoding. Additionally, some apps may initiate background tasks, such as audio decoding or news article downloading, that should proceed even in the absence of an input event. In both of these cases, it is appropriate for the display server to know about the tasks, and hence it is appropriate for the display server to manage an app using the aforementioned four threads. Existing display servers require the application programmer to register call back procedures so the only additional burden that our KDS imposes on an application programmer is to specify if a longer-running procedure is a foreground or background task.

We have also moved the display server into the kernel to reduce the amount of system calls that must be made between the kernel and display server and also so that the display server can share the scheduling information it obtains from the threads with the guiS scheduler. With our KDS, certain threads can be toggled on or off through the GUI scheduler. This allows for the application's event, drawing, and foreground threads to be suspended if the application is in the background, hence cutting their computational and power consumption requirements.

The important piece of the KDS for the guiS scheduler is the division of GUI tasks into four separate thread types. This classification allows the guiS scheduler to determine which tasks need to run even if the GUI is in the background (and conversely, which tasks do not need to run) and how to schedule the tasks on different cores.

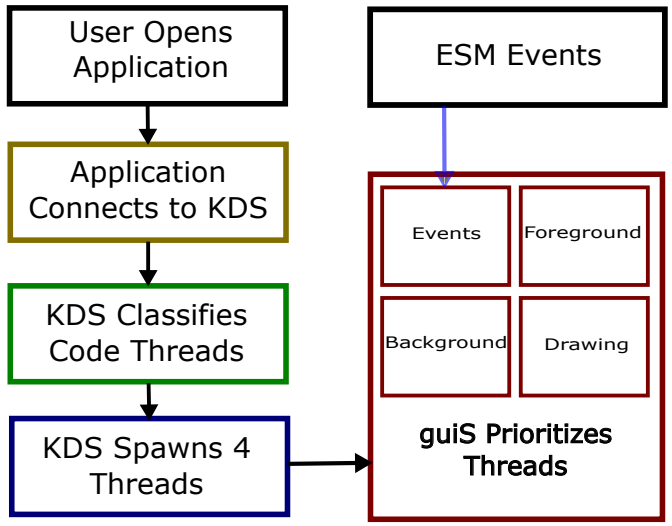

Fig. 1. The application connects through the KDS directly or through middleware. The application then tells the KDS to which of the four threads certain code belongs. From there, the GUI scheduler will prioritize the threads and schedule them to run based on their category.

The next three sections describe in greater detail how the guiS scheduler is implemented so that it can take advantage of this thread information. In the discussion that follows we will use three terms to which we want to ascribe a precise meaning:

- application: The overall GUI application. We will sometimes abbreviate application as app.
- task: A block of code that is executed, typically as a thread, by the scheduler. A task will normally be associated with an app.
- event handler: A memory address where code that will execute later in response to an event is stored. A handler will eventually become a task when it is pushed an event by the ESM. Otherwise, the handler is a dormant block of code in RAM.

We will also use three variables to which we want to ascribe a precise meaning:

- task.thread_state: Whether the app with which the task is associated is in the foreground or background.
- task.thread_type: Which of the four KDS threads a task is running on if the task is associated with a GUI app.
- task.state: The current state of the task, such as sleeping, running, or waiting for an event.

4 SCHEDULING PROCESSES FOR GUI-SPECIFIC APPLICATIONS

Figure 1 shows how an app starts up and the guiS scheduler obtains information about the app from the KDS and Table 1 shows how the scheduler schedules the threads. Algorithm 1 shows the top-level pseudo code implementation of how the guiS schedules each thread. All four threads abide by the same scheduling rules that normal processes do, namely that a sleeping process will only be set to run when the blocking 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.

Task Type	When Scheduled	Where Scheduled
Event Handling Thread	When event is received from the ESM	Low power core
Drawing Thread	When app is visible to the user	High power core
Foreground Thread	When app is in the foreground	High power core
Background Thread	Always (foreground and background)	High power core

Table 1. When a task is eligible for scheduling by the guiS scheduler and where it is typically scheduled. The rationale for the assignment of tasks to cores is discussed in Section 5.

ALGORITHM 1: guis_schedule() is called when a context switch is requested through a hardware timer "tick" or when a task 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 task is not part of a GUI process, as in the default case, the scheduler schedules the task normally in the Completely Fair Scheduler mode. If the scheduler runs through the entire task list and is unable to schedule a task, then it schedules the idle task. The idle task gradually puts cores into sleep states and eventually dims and then shuts down the LCD if the idle task keeps running for an extended period of time.

```
guis_schedule()
begin
    taskScheduled = false;
    foreach task ∈ task_list do
        switch task.thread_type do
            case EVENT_HANDLING_THREAD do
                taskScheduled = guis_schedule_event_handling(task)
            end
            case DRAWING_THREAD do
                taskScheduled = guis_schedule_drawing(task)
            end
            case FOREGROUND_THREAD do
                taskScheduled = guis_schedule_foreground(task)
            end
            case BACKGROUND_THREAD do
                taskScheduled = guis_schedule_background(task)
            end
            otherwise do
                taskScheduled = schedule(task) // revert to CFS mode
            end
        end
        if taskScheduled then
            return;
        end
    end
    schedule(idleTask);
end
```

The event handling thread executes the event handlers that the application registers with the KDS and handles user input, wi-fi traffic, and so forth. The scheduler will only run this thread when an event occurs in which the app has expressed an interest. The event dispatcher runs on this thread and it in turn pushes events to the appropriate event handlers (see Algorithm 2). When the event handler finishes, the event handling thread is put back to sleep and is left undisturbed until

ALGORITHM 2: If the app is in the EV_WAIT state, then the task is waiting for input and there is no need to schedule anything. This should rarely happen as event handlers are expected to handle only a single event and then return. However, for maximum programmer flexibility the event system was written so that an event handler could wait on another event. If the app is not in the EV_WAIT state, that means this task, either an event handler or the event dispatcher, is running, and hence the scheduler allows the task to continue to run. Also, even if the app has moved to the background, a handler that has already started to execute is allowed to complete its execution, hence the lack of a check for the app's thread state.

```
guis_schedule_event_handling(task)
begin
    if task.state != EV_WAIT then
        task.run();
        return true;
    end
    return false;
end
```

ALGORITHM 3: 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 screen 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. When an app becomes invisible, its GPU context is immediately saved, and the guiS scheduler is also notified of the context switch by the GPU and guiS immediately saves the thread's state. That is why this scheduling code does not attempt to save the thread state if the task is in the background–the state has been previously saved.

```
guis_schedule_drawing(task)
begin
    if task.thread_state = FOREGROUND then
        if CAN_SEE(task's app) then
            task.state = RUNNING;
            task.run();
            return true;
        end
    end
    return false;
end
```

another event occurs or until the application exits. If the application is placed in the background, the currently executing event handler, if there is one, is executed to completion. However, 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, thus preventing any other event handlers from starting up 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 Algorithm 3). 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.

ALGORITHM 4: 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).

guis_schedule_foreground(task)

```
begin
    if task.thread_state = FOREGROUND then
        task.state = RUNNING;
        task.run();
        return true;
    end
    return false;
end
```

ALGORITHM 5: Any task in the background thread is scheduled to run regardless of its state.

guis_schedule_background(task)

```
begin
    task.state = RUNNING;
    task.run();
    return true;
end
```

The foreground thread is scheduled to run when the application is in the foreground (see Algorithm 4). 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 Algorithm 5). 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 Facebook application would place the routines which retrieve notifications for the user in this thread. As another example, a stopwatch app that is not currently visible but which is keeping track of time would place the time keeping routine in this thread.

The KDS coordinates with the ESM and guiS 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, event, and foreground threads 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 also deregisters the application's event handlers from the ESM so that the ESM will not try to forward events to the application. When the application returns to the foreground and becomes visible, the KDS will re-register the application's event handlers with the ESM and guiS will re-activate the event handling, foreground, and drawing threads.

ALGORITHM 6: The window manager calls the kds_on_background function to inform the KDS that an application has been placed into the background. This function deregisters the event handlers that cannot run when the application is in the background and notifies the guiS scheduler that the application has been placed into the background.

```
kds_on_background(app)
begin
    foreach eventHandler ∈ app do
        esm_register(app, eventHandler.event, NULL);
    end
    guis_set_state(app, BACKGROUND);
end
```

ALGORITHM 7: The window manager calls the kds_on_foreground function when either the app moves into the foreground or when the app is in the foreground and its visibility changes. This function de-registers the event handlers if the app is no longer visible and re-registers the event handlers if the app has become visible. In both cases, it notifies the guiS scheduler that the application was placed into the foreground.

```
kds_on_foreground(app)
begin
    if CAN_SEE(app) then
        foreach eventHandler ∈ app do
            esm_register(app, eventHandler.event, eventHandler.address);
        end
    end
    else
        foreach eventHandler ∈ app do
            esm_register(app, eventHandler.event, NULL);
        end
    end
    guis_set_state(app, FOREGROUND);
end
```

The manner in which the KDS learns when an application has moved to the background or the foreground, either because of a user action or because of an API command written in the program, is via the window manager. The window manager coordinates with Android's Input Flinger, which ultimately calls kds_on_background or kds_on_foreground as shown in Algorithm 6 and Algorithm 7, respectively. These two KDS commands call the guis_set_state function, which in turn restarts the threads if the app is moving into the foreground and stops the threads if the app is moving into the background. The guis_set_state function is depicted in Algorithm 8.

5 IMPROVING POWER CONSUMPTION BY MANAGING THE CORES

The previous section discussed how tasks get scheduled but glossed over which core a task is assigned to when it executes. guiS uses the following guidelines when assigning a task to either a slower, but lower power consuming core or a faster, but higher power consuming core:

- It assigns IO-intensive tasks to lower power consuming cores and CPU-intensive tasks to higher power consuming cores with the expectation that IO-intensive tasks will be IO-bound and hence will not be unreasonably slowed down by the slower compute speed on the lower

ALGORITHM 8: guis_set_state sets the state of the GUI application. This state information is used by guiS's thread 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.

```
guis_set_state(app, state)
begin
    app.thread_state = state;
end
```

power cores. A task is deemed IO-intensive if there are IO bytes scheduled to be read or written for at least three consecutive context switches to the task (i.e, three consecutive time slices). Then, it remains an IO intensive process until three switches occur without any IO bytes scheduled. There may well be better ways to decide if a task is CPU- or IO-intensive but this rule worked well for our scheduling purposes.

- It assigns tasks on the event handling thread to the lower power consuming cores and tasks on the remaining three threads to the higher power consuming cores. The event handling thread typically executes callback procedures that normally require little computational time and can be run acceptably quickly on a low-power core. In contrast foreground, background, and drawing tasks typically are compute-intensive tasks that require the faster, higher power consuming cores.
- The event interrupt and event dispatch routines that the ESM executes to initially interpret an event, collect information about the event, and direct the event to the appropriate event handlers, are placed on the lower power consuming cores as they execute very few instructions and hence finish very quickly no matter what core they execute on.
- If an event handling task executes for more than 100 timer interrupts, then it is switched to a higher power core under the presumption that it is a more compute-intensive task that needs to be completed quickly in order to be responsive to the user. There is perhaps a better way to set this policy dynamically, but this heuristic worked well for our experiments.

One of the biggest issues we faced with our power conserving scheme is that waking the larger cores has a "spin up time" that can produce latency. Our guiS handles this problem by preemptively waking a larger core when the event dispatcher starts pushing an event with the ESM. The expectation is that if an event is handled, the GUI will have to be updated somehow, usually via the drawing thread, but sometimes also a foreground or background task will be launched. By spinning up the larger core as the event dispatcher starts pushing an event, the larger core is typically awake by the time the event handler has finished, thus eliminating the latency issue. Of course if the event does not lead to the GUI being updated, then the core may have been unnecessarily awakened. This has not been a problem in practice since 1) it is relatively uncommon for an event not to update the GUI in some way, and 2) even when an individual event does not update the GUI, events often arrive in bursts, meaning that a subsequent event is likely to update the GUI, and thus the larger core is quickly needed any way.

6 IMPROVING THE EFFICIENCY OF HARDWARE TIMER INTERRUPTS BY ELIMINATING POLLING LOOPS

Throughout this paper, 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.

ALGORITHM 9: The kernel looks at the task' sleep times to determine when to schedule the next dynamic tick. If all tasks 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 process for scheduling dynamic ticks. Rather than iterating through the entire task list every time a task 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 length of the task's sleep segment.

```
kernel_schedule_tick()
begin
    total_sleep_time = FOREVER;
    foreach task ∈ task_list do
        if task.sleep_time < total_sleep_time then
            total_sleep_time = task.sleep_time;
        end
    end
    hardware_timer.set_period(total_sleep_time);
end
```

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 [14]. A periodic tick describes a timer interrupt that occurs at a known frequency. 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.

Since the ESM uses vectored interrupts to process events, it only requires that dynamic ticks be scheduled when events occur. Algorithm 9 depicts a simplified version of the kernel's tick scheduling algorithm in pseudo code. 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.

In particular, if all ESM tasks 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.

7 EXPERIMENTS AND RESULTS

The primary goal of the guiS scheduler is to coordinate with the ESM and KDS sub-systems to allow GUI-oriented apps to be completely de-scheduled when they are incapable of receiving input events and are in the background. When all open apps are in the background, such as when the mobile device is idle and in the user's pocket, then the CPU can enter deep sleep states that conserve power. A secondary goal of the scheduler is to reduce latency by allowing an app to be immediately scheduled when an input event occurs rather than having to wait for the next iteration of the polling loop to detect the event and then schedule the app.

To test our guiS scheduler's effect on power savings and latency, we designed a spiral tracing app, a text messaging app, and a video playing app that provide a representative sample of how different apps interact with a mobile device's kernel. We then ran our three apps on a 32-bit NVIDIA TK1 reference board [11] and measured power consumption using the TK1's power consumption monitoring system. We established the baseline power consumption of our apps by running them using the out-of-the-box Android kernel provided with the reference board. This kernel included the conventional event pull model with polling loop delays set to 16 milliseconds. Then, we ran the tests again using the Android kernel modified with our KDS/ESM/guiS subsystems.

In the remainder of this section we first describe our three apps and how we performed our testing on each of them and then we describe the power savings and latency reductions we achieved with our guiS scheduler.

7.1 Testing Apps

Our test apps and the the typical category of apps they were meant to emulate are described as follows:

(1) A spiral app in which the user traces a spiral with a stylus while randomly clicking and releasing the stylus to create mouse down/up events (Figure 2). This app simulates both game playing apps and apps in which a user browses their screen by clicking on GUI interaction objects, such as buttons, images, or hyperlinks. Input events arrive regularly and at relatively evenly spaced intervals in such apps. For testing purposes we traced the spiral for 60 seconds and recorded a transcript of the events. The test generated 1614 events. 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 time frames in order to test the efficiencies (or lack thereof) of scheduling the CPU in the context of rapidly occurring events versus infrequently occurring events.

(2) The standard Android text messaging app which we instrumented to send text messages to and from a server with pauses to simulate a user reading the text message and formulating a response. This simulates an interaction style that is typical of many social media apps in which input events arrive in an irregular, "bursty" fashion. For example, a user may use the keyboard to key in a text message and then pause to read one or more text messages. Our testing program used a simulated keyboard to compose and send a random text message to the server, receive a reply, wait for 5 seconds to simulate a user reading the text message, and then send a return text message to the server. We used a 20 second test which meant on average that there were three round trips between the user and the server.

(3) A video playing app which has minimal interaction with a user but which writes large amounts of data to the frame buffer. This app simulates media playing apps, including audio-based apps such as Pandora and Sirius, and video playing apps, such as ESPN and YouTube. These apps do a good deal of foreground processing of audio or video but typically do not

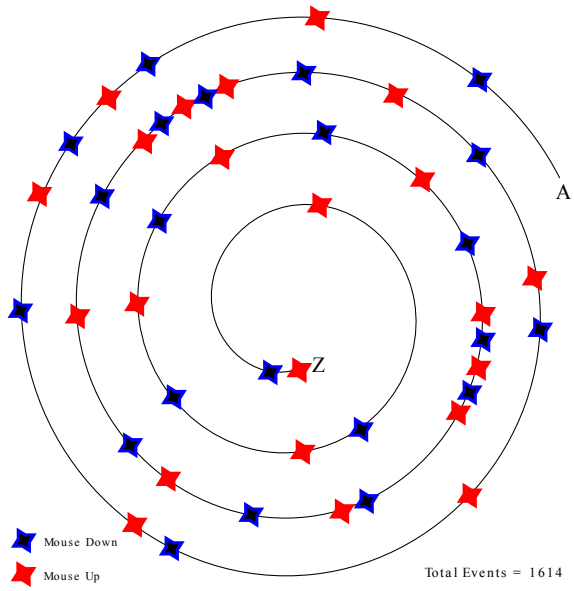

Fig. 2. A screen shot of the spiral app. The user traced a spiral path from point A to point Z while randomly pressing and releasing the stylus. The app displayed blue stars wherever the user pressed the stylus and red stars wherever the user released the stylus. The stylus events were transmitted to the kernel as mouse pressed and released events.

perform much event handling. For our tests the video playing application displayed a 30-second movie clip with no event handling so that we could observe how the guiS scheduler performed in the absence of input events. The movie clip used MPEG-4, Part 10 (AVC/H.264) encoding. The movie frames were 1920 pixels wide by 1080 pixels tall and the clip was played at a rate of 29.97 frames per second.

7.2 Power Savings

By taking advantage of the scheduling information provided by the KDS and the elimination of polling loops by the ESM, the guiS scheduler was able to obtain the power savings shown in Table 2. For the spiral and text messaging apps, almost all of the power savings comes from the elimination of polling loops and the ability of guiS to power down the cores when events are not being received. As might be expected, the guiS achieved its best performance on the text messaging app since events occurred in bursts followed by relatively long idle times.

In the spiral app, guiS achieved greater power consumption savings as the average intervals between incoming events increased. This is to be expected since for both the 10 second and 20 second tests, the average spacing between events is less than the 16ms clock rate and therefore both the push and pull event models are dealing with event queues that often have events in them and therefore the guiS scheduler cannot power down the CPU. However, even in these two tests, guiS achieved some savings because the events were not completely evenly spaced and so occasionally there were pauses where the event queues emptied and the core could be put in a lighter sleep mode. In the 60 second test, the event queue frequently emptied and guiS was able to power down the core handling the spiral app more frequently.

App	Power Reduction (milliwatts)	Power Reduction (%)
Spiral (10s)	41 mW	5%
Spiral (20s)	118 mW	15%
Spiral (60s)	185 mW	23%
Text Messaging	218 mW	30%
Video (Average Savings)	182 mW	6%
Video (Peak Savings)	400 mW	11%

Table 2. Power reductions achieved by guiS on 3 different apps.

Finally, guiS achieved its power savings in the video app via another means. In the video app, the pull model's increased power consumption is a result of the CPU cores being 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. When we probed deeper, we discovered that multitasking between decoding video and polling devices perturbs cache locality and this produces extra stress, hence extra power consumption, on the CPU. With the guiS/ESM/KDS model, there was no diversion to the event system because no events were being generated and hence guiS was able to keep the CPU focused on the task of decoding the video, thus eliminating the cache perturbations associated with the pull model.

7.3 Reductions in Latency

The guiS scheduler was also successful in reducing latency for the two event handling apps, which were the spiral and text messaging apps. It reduced latency for the spiral app 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. The better performance as the length of the test increases is due to the fact that the events arrive at larger intervals and in particular more frequently exceed the 16ms testing period of the polling loops in the pull model. If an event arrives inside the 16ms testing period of the polling loop, then there is an event waiting in the queue when the current event has finished its processing and hence there is no latency between the processing of events. However, if the next event falls outside this 16m window, then the app must wait until the next 16ms window to process the event and this creates some latency.

The guiS scheduler was even more effective at reducing latency in the text messaging app, doing so by an average of 13.6ms and up to 17.1 milliseconds in some interactions. The reason for the increased effectiveness of the guiS scheduler is twofold. First, the polling loop sleeps in 16ms chunks during idle periods and hence takes some time to detect the next event when it arrives. Second, users do not type as quickly as they move a stylus and in particular the intervals between keystrokes tends to exceed 16ms so there is almost always some delay before a polling loop detects a keyboard event. One would not expect latency to average more than 8ms for a 16ms polling loop or to exceed 16ms in any situation but there was another factor that increased latency for the event pull model which is scheduler-induced latency. The default scheduler for the pull model is the Completely Fair Scheduler (CFS) which attempts to give an equal share of CPU time to each process [17]. Therefore, an app could be starved of CPU time if the scheduler determined the app had consumed more than its fair share of CPU. Because the apps' polling loop consumed CPU time, they were sometimes scheduled less frequently than the polling loop desired. The removal of polling loops by the ESM subsystem eliminates this penalty and hence apps running in the guiS/ESM/KDS kernel do not suffer scheduler induced latency. In addition, even if the CPU is in a

power reducing sleep state, the guiS scheduler can spin up a lower power core quickly to process a pushed event as soon as it occurs, thus keeping latency quite low.

8 CONCLUSION

In this paper, we have presented a graphical user interface scheduler (guiS) that can be used in mobile devices to improve the scheduling performance of GUI-oriented apps. guiS takes advantage of knowledge that is provided by a separate kernel component we have implemented, the Kernel Display System (KDS). The KDS divides an app into four threads: event handling, display, foreground tasks, and background tasks. When an app is not currently visible and is also in the background, then all threads, except for the background thread, can be de-scheduled. When an app is not currently visible, but may be still active, such as streaming audio, then the event handling and display threads can be de-scheduled while the background and foreground threads continue to execute. The KDS tells guiS when an application changes state, and guiS responds by appropriately scheduling or de-scheduling threads based on the GUI's state. 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 30% 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 KDS and guiS make further modest power consumption and latency reductions, although it is important to note that guiS is required to achieve the ESM power consumption reductions since it is the one which de-schedules the event handling threads and handles the dynamic tick computations, thus allowing the CPU to enter its deep sleep states. In high computation environments with few or no occurring events, the KDS and guiS 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".

Finally, we developed some simple heuristics for determining how to allocate tasks among the lower and higher power consuming cores. In the future, it would be useful to look at more sophisticated strategies for dynamically allocating tasks to the appropriate core. For example, one might come up with better ways to distinguish IO-intensive from CPU-intensive tasks in the context of GUI applications or better strategies for migrating event handlers between lower and higher power consuming cores.

REFERENCES

[1] Apple, Inc. 2013. *Mach Scheduling and Thread Interfaces.* Apple, Inc. https://developer.apple.com/library/content/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html

[2] N. Audsley, I. Bate, and A. Grigg. 1999. Portable code: reducing the cost of obsolescence in embedded systems. *Computing Control Engineering Journal* 10, 3 (June 1999), 98–104. https://doi.org/10.1049/cce:19990302

[3] Duc Hoang Bui, Yunxin Liu, Hyosu Kim, Insik Shin, and Feng Zhao. 2015. Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15).* ACM, New York, NY, USA, 14–26. https://doi.org/10.1145/2789168.2790103

[4] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. 2015. A Framework for Application-Guided Task Management on Heterogeneous Embedded Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 42 (Dec. 2015), 25 pages. https://doi.org/10.1145/2835177

[5] Pi-Cheng Hsiu, Po-Hsien Tseng, Wei-Ming Chen, Chin-Chiang Pan, and Tei-Wei Kuo. 2016. User-Centric Scheduling and Governing on Mobile Devices with Big.LITTLE Processors. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 17 (Jan. 2016), 22 pages. https://doi.org/10.1145/2829946

[6] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. 2013. Hypnos: Understanding and Treating Sleep Conflicts in Smartphones. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 253–266. https://doi.org/10.1145/2465351.2465377

[7] Xianfeng Li, Guikang Chen, and Wen Wen. 2017. Energy-Efficient Execution for Repetitive App Usages on Big.LITTLE Architectures. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 44, 6 pages. https://doi.org/10.1145/3061639.3062239

[8] Stephen Marz and Brad Vander Zanden. 2016. Reducing Power Consumption and Latency in Mobile Devices Using an Event Stream Model. *ACM Trans. Embed. Comput. Syst.* 16, 1, Article 11 (Oct. 2016), 24 pages. https://doi.org/10.1145/2964203

[9] Stephen Gregory Marz. 2016. *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.

[10] Microsoft, Inc. 2017. *About DirectDraw*. Microsoft, Inc. https://docs.microsoft.com/en-us/windows-hardware/drivers/display/about-directdraw

[11] NVIDIA. 2015. NVIDIA Tegra K1 Processor Specifications. http://www.nvidia.com/object/tegra-k1-processor.html. (2015).

[12] Chandandeep Singh Pabla. 2009. Completely Fair Scheduler. *Linux Journal* 184 (August 2009), 82–83. http://www.linuxjournal.com/issue/184

[13] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. 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 (MobiSys '12)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/2307636.2307661

[14] Dan Tsafrir. 2007. The Context-switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-nothing Loops). In *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS '07)*. ACM, New York, NY, USA, Article 4. https://doi.org/10.1145/1281700.1281704

[15] Keith S. Vallerio, Lin Zhong, and Niraj K. Jha. 2006. Energy-Efficient Graphical User Interface Design. In *IEEE Transactions on Mobile Computing*. Vol. 5. Institute of Electrical and Electronics Engineers (IEEE), 846–859.

[16] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. H. Katevenis. 2017. Modeling energy-performance trade-offs in ARM big.LITTLE architectures. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 1–8. https://doi.org/10.1109/PATMOS.2017.8106950

[17] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. 2008. Towards Achieving Fairness in the Linux Scheduler. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 34–43. https://doi.org/10.1145/1400097.1400102

[18] Seehwan Yoo, YoonSeok Shim, Seunghac Lee, Sang-Ah Lee, and Joongheon Kim. 2015. A Case for Bad Big.LITTLE Switching: How to Scale Power-performance in SI-HMP. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower '15)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/2818613.2818745

Received February 2018; revised March 2018; accepted June 2018