SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Cross-Layer Resource Control and Scheduling for Improving Interactivity in Android

Sungju Huh[1], Jonghun Yoo[2] and Seongsoo Hong[1,2,3,*,†]

[1]*Department of Transdisciplinary Studies, Graduate School of Convergence Science and Technology, Seoul National University, Suwon-si, Gyeonggi-do, Republic of Korea*
[2]*Department of Electrical and Computer Engineering, Seoul National University, Seoul, Republic of Korea*
[3]*Advanced Institutes of Convergence Technology, Suwon-si, Gyeonggi-do, Republic of Korea*

SUMMARY

Android smartphones are often reported to suffer from sluggish user interactions due to poor interactivity. This is partly because Android and its task scheduler, the completely fair scheduler (CFS), may incur perceptibly long response time to user-interactive tasks. Particularly, the Android framework cannot systemically favor user-interactive tasks over other background tasks since it does not distinguish between them. Furthermore, user-interactive tasks can suffer from high dispatch latency due to the non-preemptive nature of CFS. To address these problems, this paper presents framework-assisted task characterization and virtual time-based CFS. The former is a cross-layer resource control mechanism between the Android framework and the underlying Linux kernel. It identifies user-interactive tasks at the framework-level, by using the notion of a user-interactive task chain. It then enables the kernel scheduler to selectively promote the priorities of worker tasks appearing in the task chain to reduce the preemption latency. The latter is a cross-layer refinement of CFS in terms of interactivity. It allows a task to be preempted at every predefined period. It also adjusts the virtual runtimes of the identified user-interactive tasks to ensure that they are always scheduled prior to the other tasks in the run-queue when they wake up. As a result, the dispatch latency of a user-interactive task is reduced to a small value. We have implemented our approach into Android 4.1.2 running with Linux kernel 3.0.31. Experimental results show that the response time of a user interaction is reduced by up to 77.35% while incurring only negligible overhead. Copyright © 2014 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Android smartphone; cross-layer resource management; interactivity enhancement; Linux task scheduling

1. INTRODUCTION

Android has been the most widely deployed software platform for smartphones partly due to its openness and rapid development cycle [1]. Since its 2.3 Gingerbread release, Android has continuously adopted many innovative features to enhance user experience. The latest release of Android 4.1 Jelly Bean has made significant progress in improving user perception through various techniques such as triple buffering, VSYNC timing, touch anticipation and CPU input boost [2]. Despite such success, there still exists room for further improvement in the Android framework particularly in use cases where user interactions are intensive and complicated. Android may demonstrate an unresponsive touchscreen and perceptible lag in a graphical user interface when

*Correspondence to: Seongsoo Hong, Real-Time Operating Systems Laboratory, Department of Electrical and Computer Engineering, Seoul National University, Seoul, Republic of Korea.
†E-mail: sshong@redwood.snu.ac.kr

Copyright © 2014 John Wiley & Sons, Ltd.

an interactive task is running with multiple CPU-bound applications such as a high definition media player or an anti-virus program. Smartphone manufacturers are still putting a great deal of effort into achieving a higher degree of interactivity in their Android products.

In Android, the interactivity can be quantitatively evaluated by the end-to-end response time taken to react to a user's interaction. In this time interval, Android's Linux kernel services an interrupt raised by an input device, schedules several system-related tasks, and then dispatches application tasks which eventually deliver a response to an output device. Users anticipate that all such processing be finished with no perceptible latency. There have been extensive research efforts to quantify requirements for desirable response time [3]. Despite slight differences between individuals, a typical user strongly prefers response time well below one second.

Unfortunately, it is becoming increasingly difficult to ensure such a quick response time due to the ever growing complexity of the Android framework. It supports multi-tasking by which a foreground application runs simultaneously with multiple background applications. In such an environment, user-interactive tasks generated by a foreground application may be interfered by dozens of batch tasks launched by background applications. For example, garbage collection performed by the Dalvik virtual machines of background applications is often pointed to as the culprit of visible lag in user response [4].

To address such a difficulty, Android framework 2.3 Gingerbread or later introduce a CPU resource control mechanism based on Linux's control group [5]. It is essentially a task grouping scheme which keeps a foreground and a background group separately [6]. In Android, an application becomes a foreground application if it comes to possess a display and thus becomes visible on the screen. The foreground group includes the user interface task of a foreground application and system-related tasks. Tasks in this group are ensured to occupy at least 90% of the total CPU bandwidth. Tasks in the background group are given the remaining share of the CPU bandwidth and thus restricted to run. In this way, an application's user interface and input-related services are expedited to serve user-interactions more quickly.

Unfortunately, there are cases where the task grouping scheme cannot effectively reduce the end-to-end response time of a user input. The reason for this is two-fold. First, Android allocates worker tasks generated by a foreground application into the background group so as to minimize their interference with the user interface task of the foreground application. Since such worker tasks are treated equally with other tasks in the background group, they have to compete for one tenth of CPU resources with non-user-interactive tasks. They may be preempted multiple times whenever they run out of insufficient time slices before completing their interactive work. We refer to the amount of time during which a user-interactive task is preempted by other tasks as *preemption latency*. It can be one of main factors in long response time.

Second, Android's underlying task scheduler, the completely fair scheduler (CFS), runs tasks in a non-preemptive manner for their time slices. It sorts runnable tasks in a run-queue in the non-decreasing order of their virtual runtimes and schedules them in a weighted round-robin fashion. Thus, even if an urgent user-interactive task is woken up, it must wait until all of the runnable tasks with smaller virtual runtimes run out of their time slices. We call such waiting time *dispatch latency*. Its length varies depending on both the number of preceding tasks in the run-queue and their weights. In our experiment with a target smartphone, we observed that user-interactive tasks often suffered from dispatch latency longer than 150 milliseconds.

In this paper, we propose framework-assisted task characterization (FTC) and virtual time-based CFS (VT-CFS) to solve the two problems. FTC is a cross-layer resource control mechanism between the Android framework and the underlying Linux kernel. At the framework-level, a chain of user-interactive tasks is identified via carefully selected framework APIs. In our approach, the framework is modified so that it conveys to the kernel scheduler the task identifiers of user-interactive tasks and their interaction types while servicing a given I/O request. At the kernel-level, the kernel is also redesigned so that it dynamically and selectively adjusts the priorities of worker tasks among the identified user-interactive tasks. As a result, worker tasks in a user-interactive task chain can get larger time slices than others in the background group and run longer without being preempted. This helps reduce the preemption latency of worker tasks. We further extend CFS to VT-CFS in

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

a cross-layer fashion. It forces a task to be preempted at any predefined time tick. Since the tick period is much smaller than a typical time slice of a task in Linux CFS, it renders user-interactive tasks quicker at reacting to a user input. In addition to this, VT-CFS selectively adjusts the virtual runtime of the woken-up user-interactive task using a hint provided by FTC. This ensures that the user-interactive task can be inserted as the first node in a run-queue and thus shorten its dispatch latency to a small time interval.

The proposed approach is an extensive extension to our earlier work in [7], which had two limitations. First, the former FTC considered only those tasks that were directly generated by a foreground application. It did not take into account system servers and kernel tasks even though they also contributed to delays in end-to-end response time. Second, the former VT-CFS could not differentiate user-interactive tasks from general woken up tasks in the system even though not all woken up tasks need special treatment. To rectify these limitations, we first introduce the notion of a user-interactive task chain. It is a sequence of task executions from a user input to output rendering on a screen. It includes the executions of system servers and kernel tasks as well as application tasks. The extended FTC mechanism identifies the user-interactive task chain of a foreground application and notifies the Linux kernel of their task identifiers. We also revise VT-CFS, using the extended FTC such that it can adjust the virtual runtimes of only user-interactive tasks among woken up tasks. Finally, we elaborately redesign both FTC and VT-CFS such that they can maintain compatibility with the latest version of Android. In doing so, we analyze its system model, runtime behavior and task grouping scheme in terms of interactivity.

We have implemented the FTC and VT-CFS into Android 4.1.2 Jelly Bean running on Linux kernel 3.0.31. We have empirically evaluated the proposed mechanisms with well-known benchmark programs as well as real-world applications. Our experimental results prove the effectiveness of our approach. The end-to-end response time was reduced by up to 77.35% compared to the legacy system. This improvement comes from the facts that the modified framework achieved 80.23% shorter preemption latency than the legacy framework and that the VT-CFS yielded 78.42% shorter dispatch latency for user-interactive tasks than CFS. Our approach incurs only negligible run-time overhead compared to the legacy Android.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Android framework and the CFS scheduler of the Linux kernel. Section 3 describes our problem at hand and gives a solution overview. Sections 4 and 5 technically treat the FTC and VT-CFS, respectively. Section 6 reports on our experimental evaluation. Section 7 discusses the related work and Section 8 concludes the paper.

2. BACKGROUND

In this section, we explain the overall architecture and internal operations of Android 4.1.2 Jelly Bean. We then present the technical details of CFS by summarizing its design goal, run-queue data structure and runtime algorithm. This discussion provides a background for readers in understanding the proposed approach.

2.1. Android Framework

The Android framework can be better understood via its generic layered architecture and key internal operations such as input event handling and rendering. We present them in order.

2.1.1. System Architecture Android is a software platform for mobile devices such as smartphones and tablets. It consists of the Linux kernel, the Android runtime, native libraries and the application framework [8].

- ***Linux Kernel***: Android relies on a customized version of the Linux 3.0 kernel. Different from the mainline Linux kernel, several enhancements are included into Android's kernel to better address the needs of mobile devices. They are an aggressive power management mechanism

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

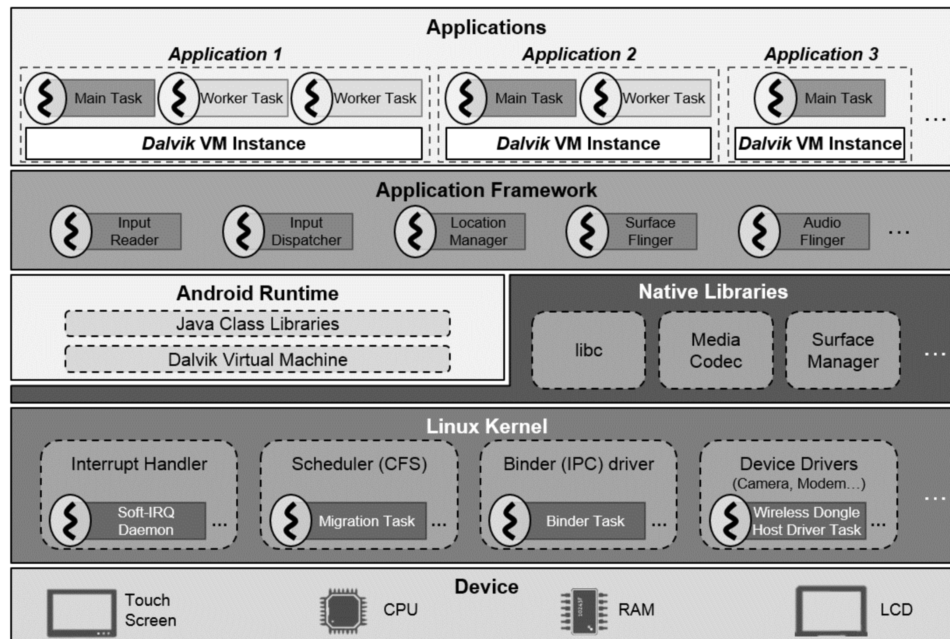


Figure 1. Layered system architecture of the Android platform.

called wake locks, an efficient IPC mechanism called a binder and the prioritized out-of-memory killer, to name a few.

- ***Android Runtime***: Android provides an application with a run-time environment which consists of the Dalvik virtual machine and standard Java class libraries. An Android application is written in the Java language and compiled into Dalvik Executable (DEX) byte code. It is executed on top of a Dalvik VM instance at run-time.
- ***Native Libraries***: Android provides a set of libraries written in C/C++ for performance. They include the standard C library such as bionic `libc`, a graphics engine and a multimedia codec. These functionalities are exposed to applications through the application framework.
- ***Application Framework***: The application framework consists of a set of system servers through which an application can get Android services. It also includes services that manage devices through underlying Linux device drivers. Examples of servers are the window manager, content provider, view system and telephony manager.

The layered architecture of the Android platform along with various types of involved task instances is illustrated in Figure 1. In the remainder of this paper, we use the term "task" instead of a process or a thread to avoid unnecessary confusion. In Linux, processes and threads at user-level are created via the `clone()` system call and associated with tasks at kernel-level. Thus, a task is the kernel-level incarnation of a process or a thread and is also an execution entity subject to scheduling and resource allocation. As shown in the figure, three types of tasks run on the platform: system servers, kernel tasks and applications. Particularly, system servers are implemented as a set of ordinary Linux tasks, each of which is responsible for a certain dedicated system resource administration. Each device is exclusively accessed by a dedicated system server and this relationship is specified by the application framework. For example, *Surface Flinger* is a server in charge of the frame buffer.

Kernel tasks are native Linux tasks executing for core system services. Such services include interrupt handling, task management and IPC. Examples of kernel tasks are soft-IRQ daemons for deferring interrupt work, migration tasks for load balancing among CPU cores, watchdog tasks for detecting lockups and binder tasks for IPC.

An application runs on top of a Dalvik VM instance and thus is not allowed to access hardware devices directly via native system calls. Instead, it must use a different interface provided by a

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

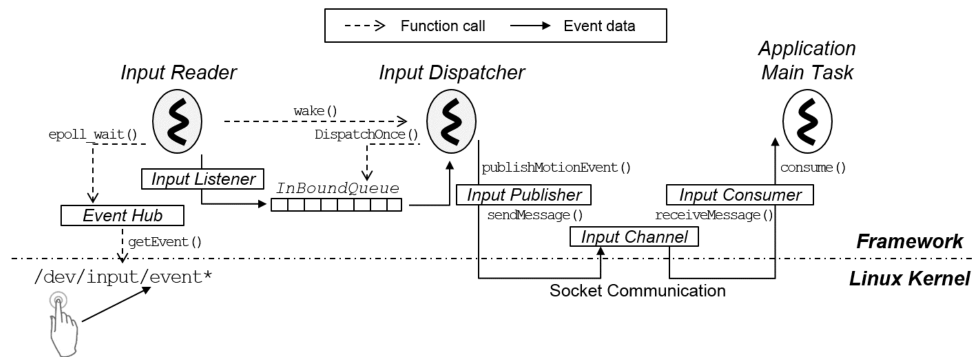


Figure 2. Input event handling in Android: from a user input to the execution of the corresponding user task.

dedicated system server. Since it is vital to guarantee an instantaneous response of an application's user interface, an Android application is split into one main task and several worker tasks when long-running operations are needed. The main task is also called a user interface task which has direct impact on responsiveness perceived by a user. It is in charge of dispatching events to appropriate user interface widgets. Worker tasks do most of the real work for an application. For example, it performs potentially long running jobs such as network and database operations and conducts computationally intensive calculations such as resizing bitmaps. An application runs with an aid of the native multi-threading of the Linux kernel since Dalvik does not offer its own multi-threading [9].

Communication between a system server and an application is done by either a binder IPC mechanism or a UNIX native socket called *Input Channel*. Communication through *Input Channel* is selectively used in time sensitive cases. For example, Android Jelly Bean uses *Input Channel* for communication between an input event handling server and the corresponding application task to enhance user responsiveness.

2.1.2. Runtime Behavior In processing a user-interactive task in Android, input event handling and rendering are the two most important operations. We describe in detail how these operations are performed by the Android Jelly Bean framework.

- ***Input Event Handling*** Figure 2 depicts an event handling process from a user input to the execution of its corresponding application task in Android Jelly Bean. As shown in the figure, two system servers, *Input Reader* and *Input Dispatcher* are used for handling input events. Suppose that a user contacts the touch screen device of an Android smartphone to draw dots or lines on the display. Then the touch screen device issues an interrupt and the kernel immediately stops a currently running task and jumps to the entry point of an interrupt handler which writes the input event data into a corresponding device file in directory `/dev/input/`. For instance, the interrupt handler of a touch screen device writes raw input events into file `/dev/input/event3`.

  Input Reader is responsible for retrieving raw input events and processing them. Specifically, it continuously performs a polling operation to retrieve a raw input event from a device file via `eventHub::getEvent()`. It then converts the raw input event into metadata. For the case of a touch event, the metadata include event occurrence time and contact coordinates. In turn, the *Input Reader* task en-queues the metadata into `InBoundQueue` using the *Input Listener* interface. It finally awakes the *Input Dispatcher* task.

  Input Dispatcher is responsible for determining a valid input target and dispatching an input event to it. After the *Input Dispatcher* task gets woken up by the *Input Reader*, it periodically reads in an input event from `InBoundQueue`. It then determines a foreground application task with the touched viewable window. In turn, it sends the event to the application task using `sendMessage()`. For communication between the *Input Dispatcher*

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

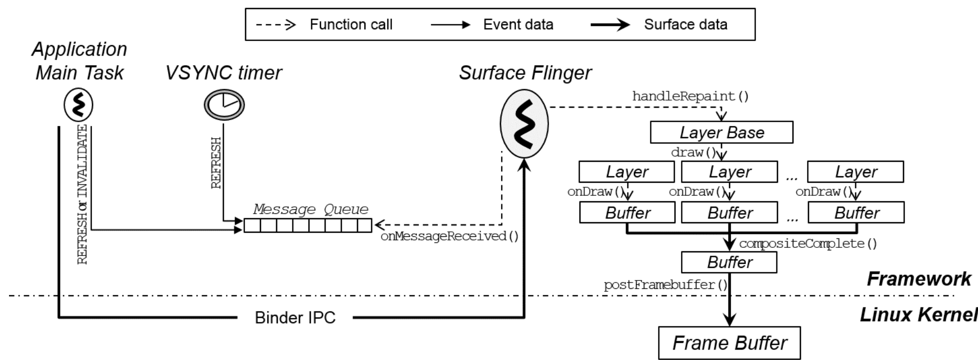


Figure 3. Rendering in Android: from an application to the frame buffer.

task and an application task, *Input Channel* is used. *Input Publisher* and *Input Consumer* are interfaces which respectively handle each end-point of the *Input Channel*. The *Input Dispatcher* uses *Input Publisher* to notify an application task on the other side of the *Input Channel*. In response to a touch event, for example, it calls `publishMotionEvent()` provided by the *Input Publisher* to send the event to the application task. On the other hand, an application task uses the *Input Consumer* to receive an event from the *Input Dispatcher*.

When an application task finally receives an event through *Input Channel*, an event listener of the application task handles it. Note that an Android application is comprised of multiple views and a couple of event listeners may be attached to each view. Application programmers can specify a listener such that a view reacts appropriately according to the type of a user input. For example, `onTouchListner` and `onKeyListener` are event listeners for the touch event and key press event, respectively.

- ***Rendering*** Figure 3 illustrates a rendering process from an application task's request to the kernel's frame buffer. *Surface Flinger* is a system server which performs rendering to the frame buffer. A rendering operation is triggered by two different sources: one originated by an application task and the other by the VSYNC timing. The former is a usual rendering operation in response to an application task's explicit request. The latter is a new type of rendering operation introduced in 4.1 Jelly Bean. It is a periodic rendering operation for ensuring a consistent frame rate. Since difference between the two is just timing, we discuss only on the former case.

  In Android, a screen is composed of separate memory regions called *surfaces*. When an application changes the contents of certain surfaces, information on the modified surfaces is transmitted to *Surface Flinger* by a binder IPC and then stored in a dedicate memory area called a *layer*. At the same time, the application task inserts a message into a message queue. There are two types of messages: `REFRESH` for refreshing a whole screen and `INVALIDATE` for invalidating a specific region of the screen. *Surface Flinger* reads in a message by polling via calling `onMessageReceived()`. If any message exists in the message queue, *Surface Flinger* calls `handleRepaint()`. This function draws the contents of layers onto a temporary image, synthesizes all the temporary images in the system and constructs a single image. It then conveys the consolidated image to the kernel's frame buffer. As a result of such a complex interplay from an application to the kernel, a user can see the rendering outcome on the device's display.

2.2. Linux Completely Fair Scheduler

CFS is a symmetric multiprocessor scheduling algorithm which maintains a dedicated run-queue for each CPU and makes scheduling decisions independently of each other. Its primary goal is to provide fair share scheduling by giving each runnable task CPU time proportional to its weight. A task's weight is specified by a nice value, which is an integer ranging [−20, 19]. A smaller nice value corresponds to a higher weight and vice versa.

Copyright © 2014 John Wiley & Sons, Ltd.

In its basic form, CFS is a weighted round-robin algorithm where tasks have time slices proportional to their weights in a round-robin queue [10]. A task is de-queued for execution from the head of the queue and is running non-preemptively for its time slice. Unlike a general weighted round-robin algorithm, however, CFS en-queues a task into a run-queue such that runnable tasks are sorted in the non-decreasing order of their virtual runtimes. A task's virtual runtime is defined as the task's actual cumulative runtime inversely scaled by its weight. Intuitively, it represents the amount of a task's progress normalized by its weight. Since the task that has not received a sufficient amount of CPU share tends to have a smaller virtual runtime than others, it is placed near the head of a run-queue and thus will be selected for execution quickly. Suppose that task τ_i is assigned weight W_i. Let ω_0 denote the weight of nice value 0 and let $C_i(t)$ be the amount of CPU time that task τ_i has received for time t. Then, the virtual runtime of τ_i is represented as below.

$$V_i(t) = \frac{\omega_0}{W_i} \times C_i(t) \tag{1}$$

As Linux introduced the notion of a control group [5], it is necessary for CFS to group tasks and provide fair share scheduling among tasks and task groups altogether. To do so, CFS assigns virtual runtime to a task group, which is defined as the sum of the cumulative runtimes of tasks in the task group inversely scaled by the group's total weight. After that, CFS treats tasks and tasks groups equally. If a task group has the smallest virtual runtime in a run-queue at a given scheduling decision point, CFS dispatches the task with the smallest virtual runtime in that task group.

To implement a run-queue, CFS uses a balanced binary tree called a red-black tree. Clearly, it has an advantage over a plain list structure since the task with the smallest virtual runtime can be found at the leftmost node of a red-black tree in $O(1)$ time and a task can be inserted into its sorted position in $O(\log n)$ time where n is the number of tasks in the tree. If a node in a red-black tree represents a task group, it contains a pointer to another red-black tree for the task group. In that case, CFS continues to search the tree until it eventually locates a single task to run.

As an instance of a weighted round-robin algorithm, CFS relies on the notion of a time slice. A time slice is associated with a task and defined as a time interval for which the task is allowed to run without being preempted. In CFS, the length of a task's time slice is proportional to its weight. The time slice of task τ_i is computed by

$$T_i = \frac{W_i}{\sum_{\tau_j \in T} W_j} \times P \tag{2}$$

where T is the set of runnable tasks in the run-queue, W_i is the weights of task τ_i and P is a round-robin interval.

A value for P has to be carefully selected depending on given workload. Tasks in a run-queue are meant to run at least once during each round-robin interval in CFS. If P has too short a value compared to the number of runnable tasks, a time slice for each task gets too small, which incurs too much context switching overhead; otherwise, fairness among tasks is very much compromised since context switches occur too infrequently. The Linux kernel uses a predefined threshold called `sched_nr_latency` to characterize the current workload. P gets a small constant value, `sched_latency` if the number of runnable tasks is smaller than the threshold. Otherwise, it is assigned a value proportional to the number of runnable tasks. Let n be the number of tasks in T. The value of P is defined as below:

$$P = \begin{cases} \texttt{sched_latency} & n > \texttt{sched_nr_latency} \\ \texttt{sysctl_sched_min_granularity} \times n & \text{otherwise} \end{cases} \tag{3}$$

`sched_latency`, `sched_nr_latency` and `sysctl_sched_min_granularity` are system-wide tunable parameters which are configured during kernel compilation time. In the latest version of Android, they are set to 6, 8 and 0.75 milliseconds, respectively.

Inside the Linux kernel, CFS is invoked on each timer interrupt called a scheduling tick. The tick period is also a tunable parameter determined by a system administrator. For modern smartphones,

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

it is often set to five milliseconds. At each scheduling tick, CFS updates the virtual runtimes of a currently running task and task group using Equation (1). It then decreases the time slice of the current task by the tick period. If the remaining time slice becomes zero, CFS preempts the current task and dispatches the task with the smallest virtual runtime in the task group with the smallest virtual runtime. Finally, it replenishes the time slice of the preempted task and puts it back to the run-queue.

It is worthwhile to understand how CFS manipulates virtual runtime when it de-queues or en-queues a task. In executing runnable tasks in rounds, CFS needs to maintain execution order among runnable tasks within each round. For the sake of performance, it keeps track of only relative order among runnable tasks in a run-queue, instead of maintaining cumulative virtual runtime for each task. This complicates the implementation of CFS to a certain degree since CFS needs to adjust virtual runtime when a task is inserted into or removed from a run-queue. Specifically, it clears the virtual runtime of a task when it is selected for running, i.e., when it is de-queued from the run-queue. When task τ_i is de-queued from run-queue at time t_1, the virtual runtime value is re-calculated as below.

$$V_i'(t_1) = V_i(t_1) - V_{min}(t_1) = 0 \tag{4}$$

where $V_{min}(t_1)$ is the minimum virtual runtime of the run-queue at time t_1. In fact, $V_{min}(t_1)$ is $V_i(t_1)$ since τ_i had the smallest virtual runtime at the time of scheduling.

Similarly, the virtual runtime is reversely adjusted when τ_i is en-queued back to the run-queue at time t_2. The virtual runtime value of τ_i is re-calculated as below.

$$V_i(t_2) = V_i'(t_2) + V_{min}(t_2) \tag{5}$$

Note that $V_i'(t_2)$ is some positive value since τ_i must have been executed for some portion of interval (t_1, t_2).

Unfortunately, the virtual runtime adjustment according to Equation (5) may incur an intolerable amount of dispatch latency to a user-interactive task. If the woken up task τ_i had spent a large portion of its time slice before it went to sleep, it has large $V_i'(t_2)$ and hence gets relatively large $V_i(t_2)$. In this case, τ_i must wait until all the runnable tasks with smaller virtual runtimes than $V_i(t_2)$ use up their time slices.

3. PROBLEM DESCRIPTION AND SOLUTION OVERVIEW

In this section, we state our problem to solve after defining key concepts appearing in the formulation of the end-to-end response time of a user-interactive application in Android. We then give an overview of the proposed solution approach.

3.1. Problem Definition

We begin with defining a user-interactive task by introducing the notion of a user-interactive task chain. We further define two types of scheduling latency as the objective metrics of our approach. We then describe our problem with a couple of motivating examples.

3.1.1. Problem Definition Tasks in an Android system fall into three types: application tasks, system server tasks and kernel tasks. The sets of system server tasks and kernel tasks are denoted by S and K, respectively. Let A^i be an Android application. It is composed of one main task and multiple worker tasks such that $A^i = \{m^i, w_1^i, \ldots, w_{n^i}^i\}$ where m^i and w_j^i denote the main task and the jth worker task of A^i, respectively. The main task is responsible for executing an input event listener whereas the worker tasks perform actual computational work asynchronously to the main task. For a given time point, there exists only one foreground application which possesses a display and thus is visible on screen. We denote it by A^c.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

The Android framework tries to make sure that the user interface task of A^c and the system-related tasks receive sufficient CPU time regardless of the amount of background load. To do so, it classifies tasks into one of two separate task groups: a foreground group and a background group [6]. They are defined as below.

DEFINITION 1. (FOREGROUND GROUP) *Foreground group F is a set of tasks consisting of the main task of foreground application A^c, system server tasks and kernel tasks such that $F = \{m^c\} \cup S \cup K$.*

DEFINITION 2. (BACKGROUND GROUP) *Background group B is a set of tasks consisting of the worker tasks of foreground application A^c and all the tasks of background applications such that $B = \left(\bigcup_{k \neq c} A^k\right) \cup (A^c - \{m^c\})$.*

In order to assign the two groups different aggregate shares of CPU time, Android makes use of a Linux kernel facility called a *control group*. In Linux, a control group is a collection of tasks which are subject to a given specific resource allocation policy. Android favors the foreground group over the background group such that the former gets a CPU weight which is about ten times larger than the latter. Specifically, Android assigns the foreground group a weight value which corresponds to the default priority of typical application tasks [11]. The background group receives a smaller weight value which corresponds to the default priority of background tasks [12]. These values are 1024 and 110, respectively. As a result, CFS allocates CPU times to the task groups proportionally to their weights as described in Section 2.2.

As described in Section 2.1.2, the Android framework executes a sequence of tasks in response to a given user input. We refer to such a task execution sequence as a *user-interactive task chain*.

DEFINITION 3. (USER-INTERACTIVE TASK CHAIN) *A user-interactive task chain is a sequence of task executions which begins with a task handling an input event and ends with a task rendering the outcome of that input. In the Android framework, it starts from the Input Reader task and ends with the Surface Flinger task.*

DEFINITION 4. (USER-INTERACTIVE TASK) *A user-interactive task is a task which appears in a user-interactive task chain.*

Figure 4 illustrates a user-interactive task chain in an Android smartphone. Figure 4(a) depicts the three types of tasks appearing in the task chain and their dependencies. Figure 4(b) gives a Gantt chart of the task chain. Note that the Gantt chart is vastly simplified for the sake of presentation. A user tabs on the screen at time t_0 and sees a response on the display at t_9. As shown in the Gantt chart, the user-interactive task chain begins with the execution of *Input Reader* at t_1 and ends with that of *Surface Flinger* at t_9.

As a metric for evaluating the interactivity of a system, we use the end-to-end response time of a user input. It is shown as interval $[t_1, t_9]$ in Figure 4(b). In order to shorten end-to-end response time in Android, it is critical for a user-interactive task to reduce two types of latency: preemption latency and dispatch latency. They are defined as below.

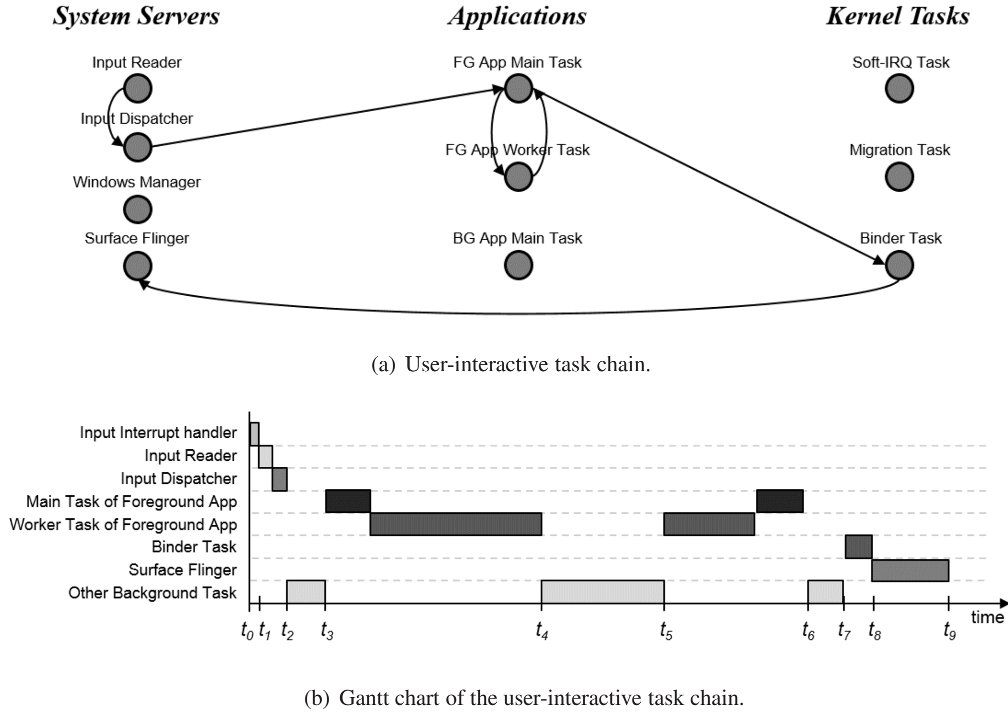
DEFINITION 5. (PREEMPTION LATENCY) *Preemption latency is the accumulated amount of time during which a user-interactive task is preempted by other tasks until the completion of its execution.*

DEFINITION 6. (DISPATCH LATENCY) *Dispatch latency is the delay between the time when a user-interactive task is inserted into a run-queue and the time when it begins to execute its first instruction.*

In the example depicted in Figure 4, the preemption latency for the worker task is $t_5 - t_4$ while the

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe



(a) User-interactive task chain.

(b) Gantt chart of the user-interactive task chain.

Figure 4. Example scenario of user interaction and the user-interactive task chain in Android.

dispatch latency for the main task is $t_3 - t_2$ and the dispatch latency for a binder task is $t_7 - t_6$. Cleary, the goal of our problem is to reduce the preemption latency and dispatch latency of user-interactive tasks so as to shorten the end-to-end response time.

3.1.2. Motivating Examples Android often fails to provide a desired level of interactivity due to the latencies defined above. The long preemption latency is ascribed to the fact that the worker tasks of a foreground application are assigned to the background group even though they are user-interactive tasks. Since they compete for CPU with other tasks in the same group, their executions are subject to preemption whenever they run out of their time slices before completing their interactive work. They may be preempted even multiple times. This clearly incurs long preemption latency to an entire user-interactive task chain. The example below illustrates this problem.

EXAMPLE 1: Consider an example in which foreground application A^c is running with background applications. Suppose that the background applications generate ten CPU-intensive tasks whose nice values are all 0's. When a user interacts with A^c, the main task of A^c generates one worker task w_1^c and becomes blocked until w_1^c finishes its work. We also assume that w_1^c takes 3 milliseconds to complete its interactive work and its nice value is also set to 0. As explained in Section 3.1.1, w_1^c runs within the background group and competes with other background tasks for occupying CPU. Since both w_1^c and other background tasks are assigned an identical time slice of 0.75 milliseconds by Equation (2), w_1^c is preempted whenever it executes for 0.75 milliseconds. In order to complete its response, w_1^c is preempted three times. After each of the preemptions, w_1^c has to wait for 7.5 milliseconds during which other ten tasks are executed. As a result, the preemption latency of w_1^c in this example is 22.5 milliseconds.

The non-preemptive nature of CFS may cause the long dispatch latency of a user-interactive task. CFS runs a runnable task for its entire time slice without preemption once the task gets started. When a user taps on the screen, the CFS wakes up a corresponding task and inserts it into the run-queue after adjusting its virtual runtime according to Equation (5). This task will get relatively small

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

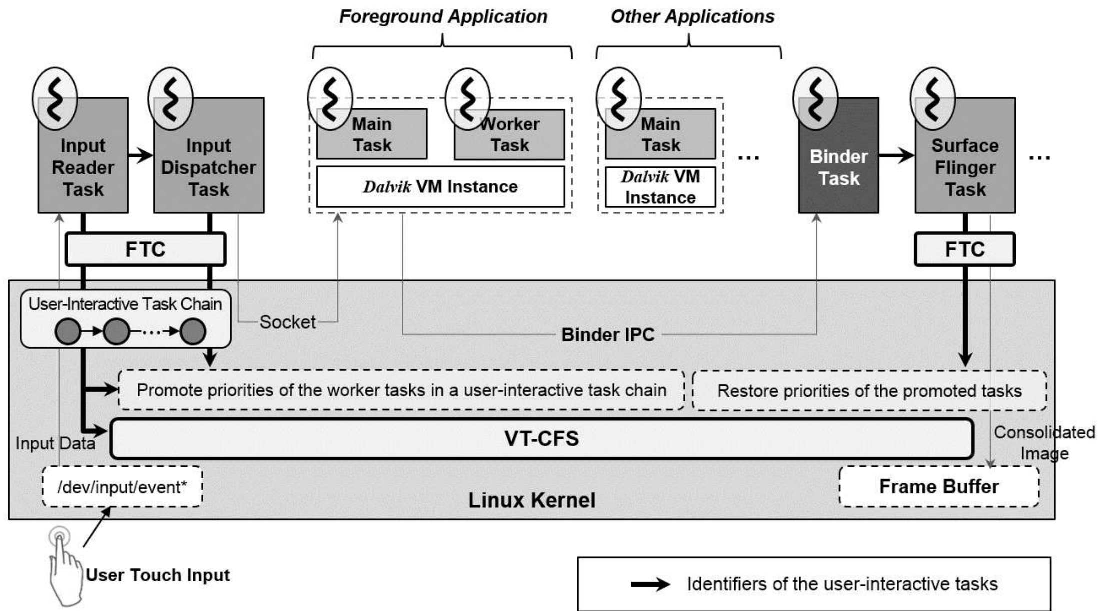


Figure 5. Overview of the proposed solution approach.

virtual runtime at the time of wakeup because its virtual runtime stopped advancing while sleeping. As a result, it is inserted into a node near the leftmost one in the run-queue and thus can be scheduled quickly as soon as a few tasks preceding it in the run-queue run out of their time slices. However, an excessive delay may occur if these tasks are CPU-intensive ones with high priorities. In that case, the user-interactive task must wait in the run-queue for a long time until all the time slices of those tasks are exhausted. This problem is illustrated by the following example.

EXAMPLE 2: Consider an example where foreground application A^c is running with 30 background tasks. Among these tasks, one is an audio decoding system server running with nice value -16 as in the current implementation of Android while the others are running with a normal priority which is nice value 0. We denote the former task by τ_a. Since the weights of nice values 0 and -16 are 1024 and 36291, respectively, the time slices of the tasks with nice value 0 become 0.34 milliseconds and that of τ_a does 12.37 milliseconds according to Equation (2). In response to a user input, main task m^c of A^c is woken up and en-queued into a run-queue. We assume that τ_a is in the run-queue and has the smallest virtual runtime at this moment. When m^c is en-queued, it is assigned virtual runtime adjusted by Equation (5). Since this value is always larger than the smallest virtual runtime in the run-queue, τ_a will be selected to run next and m^c will have to wait until τ_a runs out its time slice. In this example, m^c will suffer from large dispatch latency which exceeds 12 milliseconds.

3.2. Solution Overview

In order to solve the problems described above, we present two interactivity enhancing mechanisms suitable for Android smartphones: (1) Framework-assisted task characterization (FTC) to reduce the preemption latency and (2) Virtual Time-based CFS (VT-CFS) to reduce the dispatch latency. Figure 5 depicts the overall structure of the proposed solution approach.

In order to shorten the preemption latency, FTC selectively promotes the priorities of the worker tasks in a user-interactive task chain. Specifically, it performs three steps: identification, promotion and restoration. First, it identifies user-interactive tasks using the notion of a user-interactive task chain. In doing so, FTC accepts help directly from the Android framework rather than relying on uncertain heuristics as in dynamic priority boosting adopted by the Linux $O(1)$ scheduler. Second, FTC selectively promotes the priorities of the worker tasks among the identified user-interactive

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

tasks. Such priority promotion ensures that the worker tasks can get larger CPU times than other tasks in the background group and consequently reduces the preemption latency incurred by the background tasks. Third, it restores the priorities of the promoted worker tasks to their original values when the entire task chain ends.

The VT-CFS achieves smaller dispatch latencies of the tasks in a user-interactive task chain. When a task gets woken up, VT-CFS checks if it is a user-interactive task using a hint from FTC; if so, it inserts the task into the run-queue as the first node. It also becomes more preemptible than CFS since it uses preemption ticks. In the following two sections, we present the two mechanisms in detail.

4. FRAMEWORK-ASSISTED TASK CHARACTERIZATION

This section presents the framework-assisted task characterization (FTC) mechanism. The key idea behind it is to selectively promote the priorities of the user-interactive tasks in the background group so that they can get larger time slices under CFS. This effectively reduces the preemption latencies of user-interactive tasks by carefully sacrificing non-interactive background tasks. Clearly, some of those background tasks used to be user-interactive ones. Nevertheless, it is generally acceptable to slightly slow down their responses since they were detached from user-interactive devices such as a touch screen and their responses are not directly perceivable to a user. Our mechanism is composed of three steps: (1) one for identifying user-interactive tasks, (2) another for promoting the priorities of the worker tasks among the identified user-interactive tasks and (3) the other for restoring the priorities of the promoted worker tasks. The first step is performed at the framework-level and the other two at the kernel-level; hence it is a cross-layer resource control mechanism.

In fact, it is not easy for the kernel alone to identify user-interactive tasks since many tasks are running concurrently in a system and their CPU usage patterns differ from each other. We take advantage of a certain run-time behavior of the Android framework. As explained in Section 2.1, it is known that an event triggered by user interaction is first delivered to and handled by a dedicated system server, *Input Reader*. Another system server, *Input Dispatcher* in turn dispatches the event to the main task of the foreground application. Since *Input Reader* is the starting point of the current user-interactive task chain and *Input Dispatcher* is able to identify the current foreground application, we modify them so that they can forward to the kernel the identifiers of tasks appearing in the user-interactive task chain. Specifically, *Input Reader* informs the kernel of *Input Dispatcher*'s task identifier when it retrieves an input event from a touch screen device. In turn, *Input Dispatcher* delivers to the kernel the identifiers of the main task and worker tasks of the foreground application. It also forwards the task identifiers of *Surface Flinger* and the binder task in the same way. In doing so, we have them use the *Input Channel* mechanism such that a task identifier is delivered to the kernel as an extra argument of *Input Channel*. We make such a design decision because this incurs a lower additional overhead than raising the priorities of user-interactive tasks using an extra system call such as `setpriority()`. Tasks in a user-interactive task chain have to interact with each other via *Input Channel* anyway.

We elaborate on FTC's mechanism for obtaining a work task's identifier. In Android, an application can create a worker task in either of two ways: (1) via the `start` method of Android's `Thread` class or (2) via the `doInBackground` method of the `AsyncTask` class. In the former case, it is relatively easy for FTC to obtain a worker task's identifier since the `start` method invokes the kernel's `fork` system call through Java Native Interface (JNI). When a task is forked, FTC checks whether the task's parent is a user-interactive task. If so, the forked task is treated as a user-interactive worker task and its identifier is forwarded to the kernel. On the other hand, a worker task spawned by the `AsyncTask` class is subject to a unique threading model [13] which is distinct from the generic `fork` framework. A spawned task has a task identifier irrelevant to the main task of an application. Thus, we have modified the `AsyncTask` class such that FTC can obtain the worker task's identifier from the constructor of the class and forward it to the kernel if a caller main task is a user-interactive task.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

After a user-interactive task chain is known to the kernel, it selectively adjusts the priorities of the worker tasks. When a task gets woken up from `wait()` of *Input Channel*, the kernel compares its task identifier with the user-interactive tasks' identifiers. If a match is found, the woken-up task is marked as a user-interactive one. Furthermore, if it is one of the worker tasks, the kernel immediately promotes its priority. Later, the kernel demotes the priority back to the original value and clears the marking when *Surface Flinger* becomes blocked after finishing its rendering work.

The kernel promotes the priority of a worker task to such a degree that the task can get a time slice larger than a system-wide constant `target_slice`. This tunable constant must be large enough for a worker task to complete a given request without preemption at most of times. It varies depending on the computational power of the underlying hardware as well as user interaction patterns. We identify two interaction types: a continuous interaction pattern such as dragging and scrolling and an instant interaction pattern such as tapping and double tapping. Since continuous interactions require larger execution time than instant interactions, it is desirable to assign a relatively large time slice to a task with a continuous interaction pattern. Determining an optimal value for `target_slice` involves a tradeoff analysis. Too small a value causes excessive preemptions to worker tasks whereas too large a value may lead to interference with the execution of time-sensitive system tasks. Note that a watchdog task in Linux initiates a series of time-consuming recovery actions if time-sensitive system tasks are excessively delayed; this surely slows down the execution of user-interactive tasks. Since such a tradeoff analysis is a case of an extremely complex optimization problem, we resort to an engineering approach. We continuously increment the `target_slice` value by the size of Linux's scheduling tick period of 5 milliseconds until we reach a point where end-to-end response time starts to increase. In our experiments, the end result is 20 milliseconds and 30 milliseconds for instant and continuous interaction patterns, respectively.

In order to assign worker task w_i^c a time slice larger than `target_slice`, our mechanism adjusts its nice value. To do so, it first computes a new weight value for w_i^c as follows.

$$W_i' = \texttt{target_slice} \times \frac{\sum_{\tau_j \in T} W_j}{P} \tag{6}$$

where T is a set of runnable tasks in the run-queue and P is a constant computed by Equation (3). Our mechanism then assigns w_i^c the greatest nice value whose corresponding weight is equal to or larger than W_i'. Since a task's nice value cannot be lower than -20 by definition, we assigns w_i^c nice value -20 if computed W_i' is larger than 88761 which is the weight of nice value -20.

We revisit EXAMPLE 1. The proposed mechanism identifies A^c's worker task w_1^c as a user-interactive task and temporarily promotes its priority. Specifically, it assigns w_1^c a new nice value of -15 according to Equation (6). In this particular example, the adjusted time slice of w_1^c is computed to be 5.69 milliseconds by Equation (2). Since it is larger than the required execution time of w_1^c which is 3 milliseconds, it can complete its interactive job without any preemption. In reality, however, user-interactive tasks can still experience a small amount of preemption latency since several urgent real-time tasks such as a dynamic voltage frequency scaling task need to run periodically with very high priority.

5. VIRTUAL TIME-BASED COMPLETELY FAIR SCHEDULER

In addition to the FTC, we present a revised scheduling algorithm called virtual time-based CFS (VT-CFS). As explained in Section 3.1.2, CFS incurs dispatch latencies of variable length depending on the number of runnable tasks and their weight distribution. This deteriorates the responsiveness of a user-interactive task. We attempt to reduce the dispatch latency. In fact, we even intend to reduce it down to a small time value by modifying the runtime algorithm of CFS.

The key ideas behind VT-CFS are two-fold. First, we introduce the notion of a preemption tick which triggers the run-time scheduling more frequently. Since the tick period is much smaller than a time slice, a task in VT-CFS becomes more preemptible than in CFS. As a result, VT-CFS schedules tasks in a weighted fair queuing [14] manner rather than in a weighted round-robin [10] fashion. Second, user-interactive tasks identified by the FTC are treated differently such that their virtual

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
Algorithm 1: Virtual Time-based Completely Fair Scheduler
Input:
        τcurr - the task currently running
        gcurr  - the task group currently running
        rqcurr - task-level run-queue of gcurr
        rq    - a group-level run-queue in a system
Output:
        None

  1: update_curr(τcurr, gcurr)
  2: if rq->nr_running > 1 then
  3:     task group I = __pick_first_entity(rq)
  4:     task τnext  = __pick_first_entity(rq_of(I))
  5:     if τcurr != τnext then
  6:         put_prev_task(rqcurr, τcurr)
  7:         τnext  = pick_next_task(rq_of(I))
  8:         context_switch(τcurr, τnext)
  9:     endif
 10: endif
```

Figure 6. Pseudo code for the run-time algorithm of VT-CFS.

runtimes become the smallest one in the run-queue when they wake up. This ensures that the woken-up user-interactive tasks are always scheduled prior to the other tasks in the run-queue.

The run-time algorithm of VT-CFS is given in Figure 6. It takes as inputs the currently running task (τ_{curr}), the task group (g_{curr}) of τ_{curr}, a task-level run-queue of g_{curr} (rq_{curr}) and a system-wide group-level run-queue (rq). The algorithm is executed at every preemption tick whose interval is denoted by λ. VT-CFS maintains the identical data structure to CFS; it uses a red-black tree as a run-queue and maintains a task's virtual runtime. The algorithm makes use of the following subroutines:

- `update_curr(x,y)`: updates the runtime statistics of current task `x` and task group `y`;
- `rq->nr_running`: the number of runnable tasks in run-queue `rq`;
- `rq_of(x)`: the run-queue of scheduling entity `x`;
- `pick_first_entity(x)`: returns a descriptor of a scheduling entity stored at the leftmost node in run-queue `x`;
- `put_prev_task(x,y)`: en-queues task `y` into run-queue `x`;
- `pick_next_task(x)`: de-queues a task stored at the leftmost node in run-queue `x` and returns its task descriptor;
- `context_switch(x,y)`: performs context switching between tasks `x` and `y`.

At every λ, VT-CFS updates the virtual runtime of τ_{curr} and g_{curr} using `update_curr()` as specified in line 1. In line 2, it counts the number of tasks in rq. If it is non-empty, VT-CFS selects the task with the smallest virtual runtime in the task group with the smallest virtual runtime, through lines 3 and 4. That task is denoted by τ_{next}. Then, it checks whether the virtual runtime of τ_{curr} is still the smallest in line 5. If so, VT-CFS lets it run for another λ; otherwise, it preempts τ_{curr} and inserts it back to rq_{curr} in line 6. In turn, VT-CFS removes τ_{next} from its run-queue for execution in line 7. Finally, in line 8, actual context switching between τ_{curr} and τ_{next} is performed.

Unlike CFS, VT-CFS keeps track of cumulative virtual runtimes since runnable tasks compete for CPU only with their virtual runtimes without relying on the notion of a round. Thus, VT-CFS does not use Equation (4) or (5) when it removes a task from or inserts it into the run-queue. Instead, VT-CFS treats a task in a user-interactive task chain differently when it gets woken up. Since a task's virtual runtime does not march on during its sleep, VT-CFS uses the following equation to make user-interactive task τ_i catch up with other tasks.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

Table I. Hardware and Software Configuration of the Target System

Hardware	Systme on Chip	Texas Instruments OMAP 4460
	CPU	1.2 GHz dual-core ARM Cortex-A9
	Main Memory	1 GB
	Storage	16 GB NAND Flash
	Display	4.65 in diagonal HD Super AMOLED
Software	Kernel	Linux kernel version 3.0.31
	Android Framework	Android 4.1.2 Jelly Bean
	Build (build number)	JZO54K (485486)
	ROM	yakju

$$V_i'(t) = V_{curr}^*(t) + \frac{\omega_0}{(\omega_{-20} + 1)} \times \lambda \tag{7}$$

where $V_{curr}^*(t)$ is the stored virtual runtime of τ_{curr} and ω_0 and ω_{-20} are the weights of nice values 0 and -20, respectively. Since VT-CFS makes a scheduling decision at every preemption tick, a virtual runtime difference between any pair of tasks in the run-queue is no lower than the virtual runtime increment of a task with nice value -20. If τ_i is the only user-interactive task in the run-queue during λ, $V_i'(t)$ guarantees that τ_i is at the first node and will be scheduled at the next preemption tick. Hence, its dispatch latency is reduced to λ. If multiple user-interactive tasks are woken up and en-queued during a given λ, they are assigned the same virtual runtime value by Equation (7) since $V_{curr}^*(t)$ remains the same during that λ. In such a case, the user-interactive tasks are scheduled in FIFO order.

A trade-off exists between the degree of interactivity and run-time overhead in our algorithm. The preemption tick period λ is a tunable parameter that can control the trade-off. A smaller period leads to shorter dispatch latency while incurring larger overhead due to frequent context switches. We demonstrate such a trade-off through our experiments in the next section.

To illustrate the effectiveness of VT-CFS, we revisit EXAMPLE 2. Recall that the audio decoding task τ_a with nice value -16 is in the run-queue and has the smallest virtual runtime when m^c is just woken up. Under CFS, m^c suffers from large dispatch latency longer than 12 milliseconds. We run the same task set under VT-CFS with λ being 5 milliseconds. Assume that $V_a^*(t)$ is 0.01 and τ_a has executed for 2 milliseconds since the previous preemption tick. When m^c is en-queued, it is assigned virtual runtime of 0.0672823 by Equation (7). At the next preemption tick, τ_a's virtual runtime becomes 0.1510818. Since m^c's virtual runtime is smaller than τ_a's, m^c is scheduled to run next. This leads to dispatch latency smaller than λ. In this particular example, the dispatch latency is 3 milliseconds.

6. EXPERIMENTAL EVALUATION

In this section, we present a series of experiments we have conducted to show the degree of interactivity enhanced by the proposed approach. We first describe the experimental setup and then report on the experimental results along with the analysis of run-time overhead incurred by our approach.

6.1. *Experimental Setup*

We have performed our experiments on Google's Galaxy Nexus GSM/HSPA+ smartphone where Android 4.1.2 Jelly Bean is running with Linux kernel 3.0.31. We have modified and recompiled the Android framework so that *Input Reader* and *Input Dispatcher* could deliver to the kernel the task identifiers of user-interactive tasks as well as their interaction types. *Surface Flinger* was also modified so that it could notify the kernel of the end of the current user-interactive task chain. Finally, we added to the Linux kernel new code implementing the priority adjustment and VT-CFS. The detailed hardware and software configuration of the target system is summarized in Table I.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

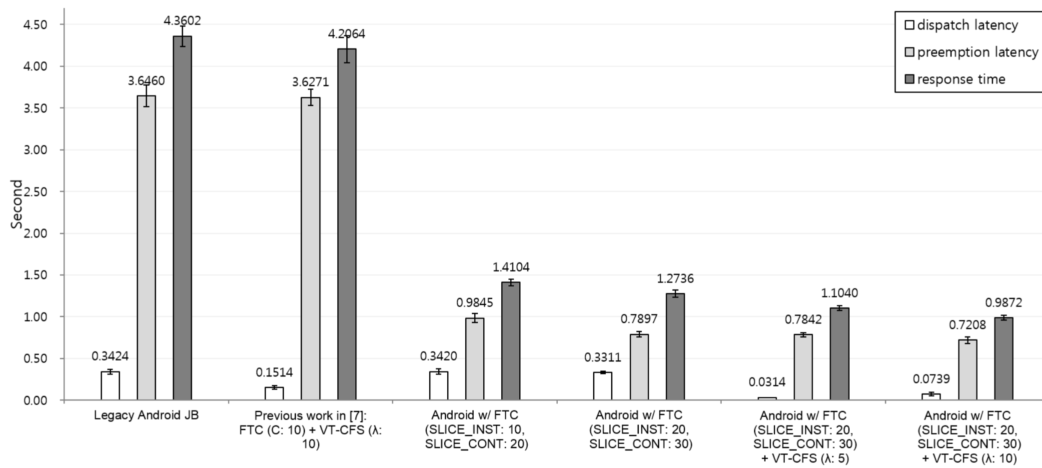


Figure 7. Response times of Aviary under different Android Jelly Bean configurations.

As a metric for interactivity, we have measured the end-to-end response time of a user input along with two latency components of the accompanying user-interactive tasks: preemption latency and dispatch latency. We have also measured the extra run-time overhead of the proposed approach. To obtain the end-to-end response time, we ran popular Android applications which we had downloaded from Google Play. They were Aviary [15], an anti-virus program [16], a media encoder [17] and a PI benchmark [18]. Aviary is an Android photo editing application which requires close interaction with a user. We also ran the `hackbench` [19] and `lmbench` [20] benchmark to evaluate the run-time overhead incurred by the proposed approach.

6.2. *Evaluating Interactivity Enhanced by the Proposed Approach*

To evaluate improved interactivity, we ran the Aviary application in the foreground along with three other CPU-intensive applications in the background. The background applications included an anti-virus program, media encoder and PI benchmark. They collectively generated nine tasks which were assigned nice value 0 according to the default weight assignment rule of Android. We then measured the end-to-end response time of Aviary. In order to better understand the interworkings of Aviary, we used kernel trace tools called `trace-cmd` and `KernelShark`. Specifically, we monitored and collected the scheduling events of tasks in Aviary via `trace-cmd` and visualized them using `KernelShark`. As a result of our analysis, we found out that Aviary worked as follows. When a user taps on the image enhancement icon of Aviary, it creates one worker task which is put into the background group to perform a series of image processing operations on the loaded image. The main task of Aviary then asks *Surface Flinger* to draw the result onto the screen. For our experiment, we measured the time interval between the time when *Input Reader* started to poll a user input and the time when *Surface Flinger* finished its rendering job. To do so, we read the time stamp values of kernel scheduling activities using `trace-cmd`, Dalvik Debug Monitor Server (DDMS) and `systrace`. We repeated this measurement 50 times and took the average response time. We then compared the results under various target system configurations: the legacy Android Jelly Bean with the standard Linux CFS as a baseline, our previous work presented in [7], the modified Android with the FTC, and the modified Android with both the FTC and VT-CFS. Figure 7 plots them. The horizontal axis denotes such configurations with various parameter settings and the vertical axis does latencies and response times explained in Section 3.1.1.

Obviously, the response times are significantly reduced by adopting our mechanisms. Observe that the response time measured under the legacy Android Jelly Bean was 4.36 seconds, 7.85% and 83.62% of which were dispatch latency and preemption latency caused by other background applications, respectively. For comparison, we have also ported the previous FTC with VT-CFS [7] to the target hardware. The response time was reduced to 4.206 seconds, which was only 3.53%

Copyright © 2014 John Wiley & Sons, Ltd.

speedup compared to the legacy system. Note that preemption latency was reduced merely by 0.5%. This is because the previous FTC did not consider a user-interactive task chain and was designed to reduce the preemption latency of only the main task of Aviary. Thus, its worker task was stalled by other background tasks just as in the legacy Android.

Figure 7 clearly shows that response time is significantly shortened by adopting the proposed FTC. For this experiment, we varied the values of two tunable parameters, `SLICE_INST` and `SLICE_CONT` to create two different Android configurations. As explained in Section 4, for each user-interactive task chain, the FTC chooses either `SLICE_INST` or `SLICE_CONT` as the `target_slice` value depending on a given interaction pattern. The response time was reduced to 1.4104 seconds when they were set to 10 and 20, respectively and 1.2736 seconds when set to 20 and 30, respectively. The latter yielded a better result; it was 70.79% shorter than that of the legacy Android Jelly Bean and its measured preemption latency was only 0.7897 seconds on average. We further set the parameters to larger values. However, we could not get any further improvement on interactivity since the Linux kernel did not allow a task's weight to exceed 88761.

As the final experiment, we integrated the FTC and VT-CFS together. As demonstrated in Figure 7, we could achieve even shorter response time due to shorter dispatch latency. Clearly, a smaller preemption tick period leads to shorter dispatch latency. The dispatch latencies were measured by 31.4 and 73.9 milliseconds when we set λ to 5 and 10 milliseconds, respectively. However, the shorter end-to-end response time were achieved when we set λ to 10 rather than 5. The response times were 1.104 and 0.987 seconds when we set λ to 5 and 10 milliseconds, respectively. This was because too short a preemption tick period in VT-CFS introduced too much run-time overhead incurred by frequent context switches and increased the end-to-end response time.

6.3. Evaluating the Run-time overhead

As demonstrated in Section 6.2, there exists a trade-off between response time and context switching overhead in VT-CFS and λ serves as a variable controlling it. In order to meticulously investigate how λ affects the trade-off analysis, we measured both per-context-switch overhead and the number of context switches by running a well-known benchmark under VT-CFS. At the same time, we also measured the overall run-time overheads of VT-CFS and CFS and compared them.

In order to make a comparison between VT-CFS with CFS in terms of overall context switching overhead, we used the `hackbench` benchmark program. It created a given number of task pairs with nice value 0 and let each pair send and receive 100-byte data via either a pipe or a socket. The benchmark measured the time taken for all pairs to send data back and forth. We generated three sets which consisted of 200, 400 and 800 task pairs, respectively. For each task pair, we computed accumulated response time by repeating the test 50 times.

Table II gives the results. The VT-CFS achieved nearly identical results compared to the standard Linux CFS when we set λ to 10; the extra run-time overhead of the VT-CFS was only 1.13% and 5.06% on average when the benchmark tasks communicated with each other via a pipe and a socket, respectively. We also set λ to 5. As expected, the run-time overhead was increased due to more frequent context switches. The extra run-time overhead for the pipe and socket IPC model were 3.63% and 8.87%, respectively.

In order to measure per-context-switch overhead and the number of context switches, we ran the `lat_ctx` benchmark from the `lmbench` 3.0 benchmark suite. The benchmark creates a given number of tasks, each of which has an independent memory area with a specific size. The tasks are connected with each other into a pipe ring. The `lat_ctx` benchmark lets each task pass a token to

Table II. Performance Overhead of the VT-CFS Measured via the hackbench Benchmark

IPC model	Legacy CFS (Baseline)			Former VT-CFS [7] ($\lambda = 10$)			Modified VT-CFS ($\lambda = 10$)			Modified VT-CFS ($\lambda = 5$)		
(# of pairs)	200	400	800	200	400	800	200	400	800	200	400	800
Pipes	2.76	5.25	10.34	2.79	5.12	10.79	2.77	5.12	10.91	2.81	5.35	11.08
Sockets	2.15	5.05	18.53	2.48	5.06	18.62	2.41	5.19	18.61	2.47	5.21	20.12

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

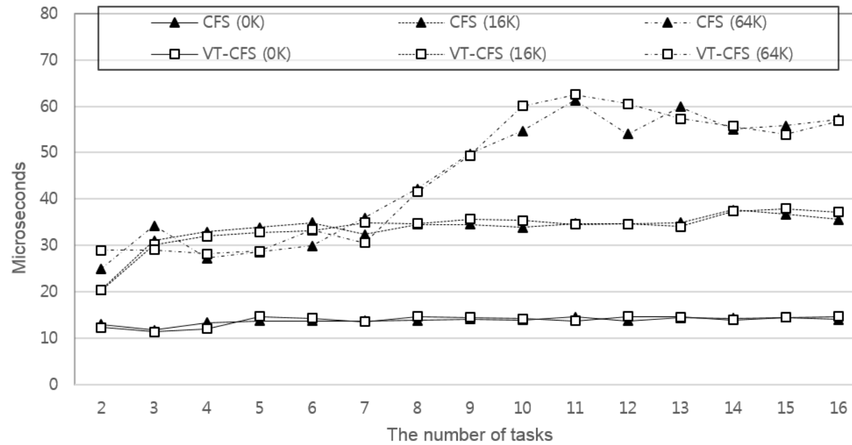


Figure 8. Average time per context switch measured via lat_ctx benchmark.

Table III. The Number of Context Switches during the lat_ctx Benchmark

	Legacy CFS (Baseline)	Former VT-CFS [7] ($\lambda = 10$)	Modified VT-CFS ($\lambda = 10$)	Modified VT-CFS ($\lambda = 5$)
Average # of context switches	11113.52	11354.92	11324.52	12539.02
Standard deviation	1131.36	599.67	528.02	422.95

its next task and then triggers a context switch between them. It repeats these operations 100 times and measures the average elapsed time per context switch. Figure 8 plots the results varying the size and number of tasks. The horizontal axis denotes the number of benchmark tasks while the vertical axis does time for one context switch measured in microseconds. Context switching times of CFS and VT-CFS are marked with black triangles and hollow squares, respectively.

Figure 8 shows that VT-CFS incurred nearly identical per-context-switch overhead compared to CFS. The extra overhead per context switch was only 0.62% on average. This is because VT-CFS has both a strength and a weakness over CFS with respect to per-context-switch overhead: VT-CFS need not calculate a task's time slice during context switching while CFS has to, but VT-CFS maintains more tasks in a run-queue than CFS on average. In Figure 8, VT-CFS incurs slightly smaller overhead than CFS where the number of tasks is 3, 6, 13 and 15 while it is the other way around for other cases; however, this is rather accidental.

We also measured the number of context switches while running the `lat_ctx` benchmark via the Linux `perf stat` command. This command displays a program's performance counter statistics such as the number of context switches, page faults, cache misses and so on. Table III shows the average number of context switches and its standard deviations measured under various kernel schedulers. VT-CFS incurred 12.82% and 1.89% more context switches than CFS when we set λ to 5 and 10, respectively. As is clear from this experiment, VT-CFS with a proper λ value of 10 yielded only slightly more run-time overhead than CFS while achieving a significant reduction in response time.

7. RELATED WORK

This section discusses related work on improving interactivity. Since task scheduling is one of critical factors in determining the interactivity of a Linux-based system, we begin with a survey of fair share task scheduling in Linux. We then describe existing techniques for improving interactivity in Android.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

7.1. Fair Share Task Scheduling in Linux

Fair share task scheduling algorithms studied for the Linux kernel can be classified into two categories: (1) one based on Weighted Round-Robin (WRR) and (2) the other based on Weighted Fair Queuing (WFQ). The former is usually fast and offers a weak fairness guarantee and poor interactivity. On the other hand, the latter exhibits strong fairness and enhanced interactivity and incurs high scheduling overhead.

7.1.1. WRR-based Scheduling In WRR-based scheduling, tasks are executed for weighted time slices with the same frequency. This type of scheduling has a constant time complexity in task selection and leads to poor fairness since its fairness lag from GPS [21] is bounded by the largest weight in the system. It also contributes to poor interactivity since its dispatch latency can be as large as the largest weight in the system. Despite such disadvantages, many variants of the WRR scheduler have been proposed for the Linux kernel due to run-time efficiency.

Prior to kernel 2.6.23, Linux used an $O(1)$ scheduler as its task scheduler [22]. It maintains a pair of priority arrays as a per-CPU run-queue and schedules the highest priority task in a system. If multiple tasks exist at the same priority level, they are scheduled in a round-robin manner. The length of a time slice is proportional to the task's priority. In order to improve interactivity, it employs a number of heuristics to determine whether a task is I/O-bound or CPU-bound. The scheduler dynamically promotes the priority of an identified I/O-bound task. Unfortunately, such heuristics often misclassify a task's characteristics. Thus, they may lead to the starvation and unfair allocation of CPU resources under certain workload [23]. In [24], an MLFQ scheduler was implemented to enhance the interactivity of the $O(1)$ scheduler. It eliminates the heuristics for task characterization and uses an inverse relationship between a task's priority and time slice length. Their experiments showed improved interactivity over the $O(1)$ scheduler. However, it demonstrated poor throughput when it scheduled CPU-intensive workload. Linux CFS [25] also attempted to provide fair share task scheduling in a weighted round-robin fashion where a scheduling decision is made when the previous task runs out of its time slice. Note that CFS uses virtual runtime only when selecting a task to be scheduled. The Distributed Weighted Round-Robin (DWRR) was proposed as a WRR-based scheduling algorithm for a scalable multiprocessor [26]. It schedules tasks in a trivial WRR manner on each processor. Across processors, it performs load balancing to ensure that all tasks go through the same number of rounds where the number represents the progress of virtual time.

7.1.2. WFQ-based Scheduling In WFQ-based scheduling, a task is assigned virtual time and allowed to execute for a constant time period called a time quantum. After executing for one time quantum, a task's virtual time is computed and the task with the earliest virtual time is scheduled. Since tasks are ordered by their virtual times, this approach has $O(\log n)$ run-time complexity where n is the number of tasks. It has been proved that the fairness lag in WFQ-based scheduling is bounded by a constant in usual workload [21]. It can also provide short response time and improved interactivity for an interactive task by adjusting the virtual time of a newly woken-up task. Stride scheduling was the first WFQ-based scheduling algorithm implemented for the Linux kernel [27]. It employed a fair queuing algorithm originally designed for packet scheduling [28]. Their experiments demonstrated that it could achieve enhanced fairness as well as lower response time. Nieh and Lam presented a WFQ-based scheduler for multimedia and real-time applications [29]. They additionally considered real-time guarantees. Their algorithm uses biased virtual finish time in selecting candidates for scheduling and then chooses a task among the candidates by considering real-time constraints. Similarly, Zhang et al. proposed Virtual Deadline Scheduling (VDS) for multimedia and weakly hard real-time systems [30]. In VDS, virtual deadlines are assigned to tasks such that they can share the CPU resource proportionally to their weights and task scheduling is done in the earliest virtual deadline first manner. More recently, Kolivas developed a WFQ-based kernel scheduler called BFS, as an alternative to CFS [31]. Its main goal was to improve desktop interactivity by using a simple run-time algorithm based on tasks' virtual deadlines [32]. Since BFS works with a centralized run-queue, it can make a global scheduling decision and does not require a sophisticated load balancing mechanism. It was once added to the Android repository in the Éclair

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

release thanks to the improved interactivity reported by Linux magazine [33], but later excluded from the Froyo release. The improvement contributed by the BFS was rarely distinguished by a human user in practice.

7.2. Improving Interactivity in Android

In the literature, there have been active research activities to enhance the interactivity of Android systems. They can be divided into two categories: one at the kernel-level and the other at the framework-level.

7.2.1. Kernel-level Techniques Android has employed three kernel-level techniques to enhance interactivity: First, it keeps background and default priorities separately in order to control the amount of time during which background tasks interfere foreground tasks. As described in Section 3.1.1, the Linux kernel gathers all background tasks into a special control group and restricts them to occupy no more than 10% of the total CPU bandwidth. Second, Android 4.1 Jelly Bean adopted a mechanism called the CPU input boost scheme [2]. It utilizes a dynamic voltage and frequency scaling (DVFS) technique of the Linux kernel such that the CPU clock frequency is temporarily scaled up to the maximum when a user touches the screen. This ensures that all work related to user interaction can be processed at the fastest possible speed. Third, Android Jelly Bean introduced a feature called triple buffering in the rendering pipelines in order to maintain a consistent frame rate [34]. It is an extension to a double buffering technique which uses a back buffer to store an intermediately rendered image and a front buffer to store a completely rendered image seen on a display. In the double buffering, an application can stall before it starts the next drawing if the current image in the back buffer is being copied into the front buffer. The waiting time could be several milliseconds and seriously affect user experience in a smartphone environment. To overcome this problem, Android introduced the additional back buffer. With two back buffers, an application is allowed to immediately use either one of the back buffers which is not involved in image copying.

7.2.2. Framework-level Techniques Android has adopted three techniques at the framework-level for improved interactivity. First, it makes use of a unique rendering strategy in order to reduce time to draw a screen [35]. As explained in Section 2.1.2, a screen in Android is composed of separate pieces of region called surfaces. Android draws each surface independently so that only modified surfaces are re-drawn. Since individual surfaces are merged into a screen and rendered by *Surface Flinger*, total rendering time is reduced and a user can perceive improved interactivity. Second, Android Jelly Bean extends the VSYNC timing across all interactive work done by the Android framework [34]. Such work includes operations performed by *Input Reader*, *Input Dispatcher* and *Surface Flinger*. Android intentionally synchronizes application rendering, layer composition and display refreshing as well as touch event handling at every 16 milliseconds when the VSYNC timer is triggered. By doing so, it can ensure a consistent frame rate of 60 FPS. Third, Android Jelly Bean runs a touch anticipation algorithm which predicts where a touch will be made in the next display refresh [36]. The algorithm calculates the coordinates of the next touch using measured information such as a velocity vector. This contributes to reactive and uniform user touch response.

8. CONCLUSIONS

We have presented two cross-layer interactivity enhancing mechanisms for Android smartphones. Android systems often incurred palpably long preemption and dispatch latency due to the lack of the appropriate task-specific prioritization and the non-preemptive nature of CFS. Being motivated by this observation, we came up with framework-assisted task characterization and virtual time-based CFS. The former is a cross-layer resource control mechanism which selectively favors user-interactive tasks. It first identifies a user-interactive task chain at the framework-level, and then enables the kernel scheduler to selectively promote the priorities of worker tasks appearing in the identified task chain. The latter is a cross-layer refinement of CFS which makes user-interactive

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

tasks quicker at reacting to a user interaction. It allows a task to be preempted at every predefined period. It also adjusts the virtual runtimes of the identified user-interactive tasks to ensure that they are always scheduled prior to other tasks in the run-queue when they wake up.

We have implemented the proposed mechanisms into the latest version of the Android framework on commercially available smartphone hardware. Our experiments with diverse applications and benchmarks indicated that the proposed approach effectively improved interactivity while incurring only negligible run-time overhead. It was shown that the end-to-end response time of a user input was reduced by up to 77.35% compared to the legacy Android. This is indeed a significant improvement in interactivity that can be easily perceived by a user in everyday use. Interested readers are advised to refer to a video clip at [37] we have uploaded for an effective demonstration of our approach.

There are future research directions along which the proposed mechanisms can be extended. We are looking to extend the VT-CFS to integrate it with a virtual runtime-based task migration algorithm [38] which we have developed as an alternative to CFS's load balancing. We expect that the mixture of the two virtual runtime-based algorithms can achieve synergistic benefits in terms of fairness as well as interactivity.

ACKNOWLEDGEMENT

This work was partly supported by the IT R&D program of MSIP/KEIT [10041753 , Development of the Core Technologies of a General Purpose OS for Reducing 30% of Energy Consumption in IT Equipments], by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) [No. 2012-0000348] and by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program [NIPA-2013-H0301-13-4006] supervised by the NIPA (National IT Industry Promotion Agency).

REFERENCES

1. Fingas J. Android has a heady 59 percent of world smartphone share, iphone still on the way up 2012. URL http://www.engadget.com/2012/05/24/idc-q1-2012-world-smartphone-share/.
2. Google. Android 4.1 jelly bean: Faster, smoother, more responsive 2012. URL http://developer.android.com/about/versions/jelly-bean.html.
3. Tolia N, Andersen DG, Satyanarayanan M. Quantifying interactive user experience on thin clients. *Computer* 2006; **39**(3):46–52.
4. Rieder A. Dealing with garbage collection 2011. URL http://code.google.com/p/libgdx-users/wiki/ForceGarbageCollection.
5. Menage P. Cgroups 2008. URL http://www.mjmwired.net/kernel/Documentation/cgroups.txt.
6. Hackborn D. About fundamental design decisions in android 2011. URL https://plus.google.com/105051985738280261832/posts/XAZ4CeVP6DC.
7. Huh S, Yoo J, Hong S. Improving interactivity via vt-cfs and framework-assisted task characterization for linux/android smartphones. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, IEEE, 2012; 250–259.
8. Brady P. Anatomy & physiology of an android. *Google I/O Developer Conference*, 2008.
9. Google. The basic terminology of the android platform 2013. URL http://developer.android.com/guide/appendix/glossary.html.
10. Nagle J. On packet switches with infinite storage 1985; .
11. Google. Default priority of typical applications 2014. URL http://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_DEFAULT.
12. Google. Default priority of background applications 2014. URL http://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_BACKGROUND.
13. Google. Asynctask 2014. URL http://developer.android.com/reference/android/os/AsyncTask.html.
14. Demers A, Keshav S, Shenker S. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, vol. 19, ACM, 1989; 1–12.
15. Aviary. Aviary photo editing application 2012. URL https://play.google.com/store/apps/details?id=com.aviary.android.feather.
16. AVG. Avg antivirus 2012. URL https://play.google.com/store/apps/details?id=com.antivirus.
17. SilentLexx. Ffmpeg media encoder 2012. URL https://play.google.com/store/apps/details?id=com.silentlexx.ffmpeggui.
18. Markovic SD. Pi benchmark 2012. URL https://play.google.com/store/apps/details?id=rs.in.luka.android.pi.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

19. Russell R. Hackbench: A new multiqueue scheduler benchmark. *Message to Linux Kernel Mailinglist: http://www. lkml. org/archive/2001/12/11/19/index. html* 2001; .
20. McVoy LW, Staelin C, *et al.*. lmbench: Portable tools for performance analysis. *USENIX annual technical conference*, San Diego, CA, USA, 1996; 279–294.
21. Parekh AK, Gallager RG. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking (TON)* 1993; **1**(3):344–357.
22. Aas J. Understanding the linux 2.6.8.1 cpu scheduler. *Retrieved Oct* 2005; **16**:1–38.
23. Salah K, Manea A, Zeadally S, Alcaraz Calero JM. On linux starvation of cpu-bound processes in the presence of network i/o. *Computers & Electrical Engineering* 2011; **37**(6):1090–1105.
24. Torrey LA, Coleman J, Miller BP. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Software: Practice and Experience* 2007; **37**(4):347–364.
25. Molnar I. The cfs scheduler 2007. URL http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.
26. Li T, Baumberger D, Hahn S. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *ACM Sigplan Notices* 2009; **44**(4):65.
27. Waldspurger CA, Weihl WE. Stride scheduling: Deterministic proportional-share resource management. *Technical Report*, Technical Memo MIT/LCS/TM-528, MIT Laboratory for Computer Science 1995.
28. Zhang L. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM SIGCOMM Computer Communication Review*, vol. 20, ACM, 1990; 19–29.
29. Nieh J, Lam MS. A smart scheduler for multimedia applications. *ACM Transactions on Computer Systems (TOCS)* 2003; **21**(2):117–163.
30. Zhang Y, West R, Qi X. A virtual deadline scheduler for window-constrained service guarantees. *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, IEEE, 2004; 151–160.
31. Kolivas C. Brain fuck scheduler 2010. URL http://ck.kolivas.org/patches/bfs/bfs-faq.txt.
32. Stoica I, Abdel-Wahab H, Jeffay K, Baruah SK, Gehrke JE, Plaxton CG. A proportional share resource allocation algorithm for real-time, time-shared systems. *Real-Time Systems Symposium, 1996., 17th IEEE*, IEEE, 1996; 288–299.
33. Hilzinger M. Con kolivas introduces new bfs scheduler 2009. URL http://www.linux-magazine.com/Online/News/Con-Kolivas-Introduces-New-BFS-Scheduler.
34. Guy R, Haase C. For butter or worse: Smoothing out performance in android uis 2012. URL https://developers.google.com/events/io/sessions/gooio2012/109/.
35. Google. Optimizing the view 2012. URL http://developer.android.com/training/custom-views/optimizing-view.html.
36. Guy R, Haase C. What's new in android? 2012. URL https://developers.google.com/events/io/sessions/gooio2012/105/.
37. Huh S. Video demonstration of cross-layer resource control and scheduling for improving interactivity in android 2013. URL http://www.youtube.com/watch?v=0bj7196uBl0&feature=youtu.be.
38. Huh S, Yoo J, Kim M, Hong S. Providing fair share scheduling on multicore cloud servers via virtual runtime-based task migration algorithm. *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, IEEE, 2012; 606–614.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls