# Improving Interactivity via VT-CFS and Framework-assisted Task Characterization for Linux/Android Smartphones

Sungju Huh[1], Jonghun Yoo[2] and Seongsoo Hong[1, 2]
[1] Department of Intelligent Convergence Systems, Graduate School of Convergence Science and Technology
[2] School of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
{sjhuh, jhyoo, sshong}@redwood.snu.ac.kr

*Abstract*—**Android smartphones are often reported to suffer from sluggish user interactions due to poor interactivity. This is because the Linux kernel may incur perceptibly long response time to user interactive tasks. Particularly, the completely fair scheduler (CFS) of Linux cannot systematically favor a user interactive task over background tasks since it fails to effectively distinguish between them. Even if a user interactive task is successfully identified, such a task can suffer from a high scheduling latency due to the non-preemptive nature of CFS. This paper presents a framework-assisted task characterization and virtual runtime-based CFS (VT-CFS) to address these problems. The former is a cooperative mechanism between the Android application framework and the kernel. It identifies a user interactive task at the framework level and then enables the task scheduler to selectively promote the priority of the identified task at the kernel level. VT-CFS is an extension of the CFS. It allows a task to be preempted at any preemption tick so that the scheduling latency of a user interactive task is bounded by the tick interval. We have implemented our approach into Android 2.2 running with Linux kernel 2.6.32. Experimental results show that the response time of a user interactive task is reduced by up to 31.4% while incurring only 0.9% more run-time overhead than the legacy system.**

***Keywords-interactivity; responsiveness; Android smartphones; Linux; task scheduling***

## I. INTRODUCTION

Android is a mobile software platform that runs on top of the Linux kernel and provides applications with Java-based run-time environments. It has been the most widely deployed software platform for smartphones since 2010 [1] partly due to its openness and rapid development cycles. While many innovative features have been constantly introduced into the recent Android releases, Android smartphones are still reported to suffer from sluggish user interactions due to poor interactivity [2]. UI-related complaints on Android include an unresponsive touchscreen and a perceptible lag in the GUI. Manufacturers are putting a great deal of effort into achieving a high degree of interactivity in their Android smartphone products.

The interactivity can be quantitatively evaluated by the end-to-end response time taken to react to a user's action. In this time interval, the kernel services an interrupt raised by an input device and wakes up an application task which eventually delivers a response to an output device. Users anticipate that all such processing is finished within no perceptible delay. There have been extensive research efforts to quantify requirements on satisfactory response time [3]. Despite slight differences between individuals, a typical user gets satisfied when response time is less than 150 ㎳.

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

In Android, ordinary tasks are scheduled by the completely fair scheduler (CFS) which is the default scheduler of the mainline Linux kernel. The objective of CFS is to achieve fairness in the distribution of CPU times among tasks. To do so, it gives each task a time slice proportional to its weight. At run-time, CFS keeps track of the currently running task's virtual runtime, defined by the task's cumulative runtime inversely scaled by its weight. When a task runs for its time slice, CFS preempts it and dispatches another task with the smallest virtual runtime. This approach is particularly effective in a server environment where a number of batch tasks owned by different users share computing resources in a fair manner.

However, CFS falls short of expectations when it comes to scheduling interactive tasks [5]. The reasons for this are two-folds. First, CFS does not identify a user interactive task to favor it over background tasks. Thus, a time slice given to a user interactive task may not be long enough to complete given user interaction. In such a case, the user interactive task is preempted by other runnable tasks whenever it runs out of its time slice. Second, CFS often leads to a long scheduling latency even if a user interactive task can be identified. CFS forces a task to run non-preemptively for its entire time slice. Thus, when a 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. The scheduling latency varies depending on both the number of tasks and their weights.

In this paper, we propose framework-assisted task characterization and virtual runtime-based CFS (VT-CFS) to solve the above problems. Framework-assisted task characterization is a cooperative mechanism between the Android application framework and the kernel. At the framework level, a user interactive task is easily identified since it must invoke a specific framework API to access I/O devices. While servicing a given I/O request, the framework conveys the task identifier of a user interactive task to CFS. At the kernel level, CFS temporarily promotes the priority of such a task so that it gets a larger time slice than others. This effectively reduces the preemption latency of a user interactive task since it can run longer without preemption than it does under CFS. VT-CFS extends CFS to allow a task to be preempted at any predefined tick called preemption tick. This guarantees that a scheduling latency of a user interactive task is bounded by the tick period. It is a constant value smaller than a typical time slice of a task in Linux.

We have implemented the proposed mechanisms into Android 2.2 running on the Linux kernel 2.6.32. We measured the response times of a set of user interactive tasks which ran with background batch tasks. Experimental results prove the effectiveness of our approach. The response time is reduced by up to 31.4% compared to the legacy system. This improvement comes from the facts that the modified framework achieves 59.7% shorter preemption latency than the legacy framework and that VT-CFS yields 6.07% shorter scheduling latency for user interactive tasks than the CFS. The extra run-time overhead of our approach was only 0.9% of the legacy Android system.

The rest of this paper is organized as follows. Section 2 discusses the background and existing work for improving interactivity. Section 3 formulates the problem and presents the solution overview. Sections 4 and 5 describe the framework-assisted task characterization mechanism and VT-CFS, respectively. Section 6 reports on the experimental evaluation. We conclude this paper in Section 7.

## II. BACKGROUND AND RELATED WORK

In this section, we provide background information on the Android framework and the Linux kernel's task scheduler in the context of interactivity. We then discuss related work on improving interactivity.

### A. Android Framework

Android is a software platform for mobile devices such as smartphones and tablets. It consists of a kernel, Android runtime, native libraries and application framework [6].

- ***Kernel***: Android relies on a customized version of the Linux 2.6 kernel. Different from the mainline Linux kernel, several enhancements are included in Android's kernel to better address the needs for mobile devices: aggressive power management called *wake locks*, efficient IPC mechanism called *binder*, prioritized out of memory killer, and so on.
- ***Android runtime***: Android provides an application with a run-time environment which consists of a Dalvik virtual machine instance and the standard Java class libraries. Android applications are written in the Java language and compiled into Dalvik Executable (*DEX*) byte code. At run-time, an application is executed on top of an independent Dalvik VM instance.
- ***Native libraries***: Android provides a set of libraries written in C/C++ for performance. These include the standard C library, graphic engine, multimedia codec, surface manager, and so on. 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 make use of Android services. Examples of servers are the window manager, content provider and view system, to name a few.

### B. Completely Fair Scheduler

CFS is a symmetric multiprocessor scheduling algorithm [7] which maintains a dedicated run-queue for each core and makes scheduling decisions independently of each other. Its primary goal is to provide fair share scheduling by giving each 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.

Within a run-queue, CFS successfully achieves the goal by using the notion of a task's virtual runtime. A task's virtual runtime is defined as the task's cumulative runtime inversely scaled by its weight. Assume that task $\tau_i$ is assigned weight $W(\tau_i)$. Let $\omega_0$ denote the weight of nice value 0 and let $A(\tau_i,t)$ be the amount of CPU time that the task $\tau_i$ has received in time duration $[0,t)$. The virtual runtime of task $\tau_i$ at time *t* is represented as below.

$$VR(\tau_i,t) = \frac{\omega_0}{W(\tau_i)} \times A(\tau_i,t) \quad (1)$$

Perfect fairness is achieved if virtual runtimes are the same among all the tasks at any given time. CFS approximates this by dispatching the task with the smallest virtual runtime at its scheduling decision point. In order to reduce the run-time complexity of run-queue manipulation, CFS uses a balanced binary tree called a red-black tree in maintaining a list of tasks sorted by their virtual runtimes. For a given red-black tree, the task with the smallest virtual runtime can be found at the leftmost node of the tree in $O(1)$ time.

In order to enforce fair share scheduling at a reasonable run-time cost, CFS makes use of 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 $\tau_i$ is computed by

$$TS_{\tau_i} = \frac{W(\tau_i)}{\sum_{\tau_j \in S} W(\tau_j)} \times P \quad (2)$$

where $S$ is the set of runnable tasks in the run-queue, $W(\tau_i)$ is the weight of $\tau_i$ and $P$ is the constant for given workload. Let $n$ be the number of tasks in a system. $P$ is then defined as below in the latest version of Android [8].

$$P = \begin{cases} 5\,\text{ms} & \text{if } n < 5 \\ 1\,\text{ms} \times n & \text{otherwise} \end{cases} \quad (3)$$

At each scheduling tick, CFS updates the virtual runtime of the currently running task and checks if it ran out of its time slice. If so, CFS preempts the running task and dispatches the task stored at the leftmost node in the run-queue. It replenishes the time slice of the preempted task and puts it back to the run-queue at the same time.

If a task is woken up after a long sleep, it would have a very small virtual runtime compared to other tasks. In order to prevent such a task from monopolizing the CPU until it catches up other tasks' virtual runtimes, CFS adjusts the virtual runtime of a newly woken-up task $\tau_i$ at time *t* as below.

$$VR(\tau_i, t) = VR(\tau_i, t) + \min_{\tau_j \in S}(VR(\tau_j, t)) \quad (4)$$

It ensures that the virtual runtime of $\tau_i$ is slightly larger than the minimum virtual runtime at the run-queue when it is inserted back to the run-queue.

### C. Related Work

There have been several research efforts to improve the interactivity of an Android system. They can be divided into two categories: one performed at the kernel level and the other at the framework level.

In the literature, the Linux kernel developers have attempted to improve the interactivity of Linux by refining task scheduling algorithms. Prior to kernel 2.6.23, Linux used an *O(1)* scheduler as its task scheduler [9]. It addressed interactivity with a number of heuristics to determine if tasks are I/O-bound or CPU-bound. Once it characterizes tasks, it promotes the priorities of I/O-bound tasks. In general, the *O(1)* scheduler generally shows better interactivity performance than CFS [10, 11]. Unfortunately, the heuristics often misclassify a task's characteristics. Thus, it may lead to starvation and unfair allocation of CPU resource under certain workload [12]. Torrey *et al.* implemented an MLFQ scheduler into Linux 2.6 kernel to enhance the interactivity of the *O(1)* scheduler [13]. Its differences from the *O(1)* scheduler are in the elimination of the heuristics for interactivity and the use of the opposite relationship between a task's priority and time slice. Their experiments showed that interactivity was improved. However, it demonstrated poor throughput when it worked with CPU-intensive workload. Kolivas proposed a Brain Fuck Scheduler (BFS) [5] as an alternative to CFS. The main goal of BFS is to improve desktop interactivity by providing a simpler run-time algorithm without relying on any heuristics. Unlike CFS, it keeps track of a task's virtual deadline [14] for fair allocation of CPU resource. It keeps the virtual deadline and remainder of the time slice of a task when it becomes blocked so that it maintains an earlier virtual deadline when it gets woken up. This improves the interactivity of the system. Since the improved performance of BFS was reported by Linux magazines [15], it was added to the Android repository in the *Éclair* release. However, it was later excluded from the *Froyo* release since such an improvement was rarely distinguished by user experience.

Android also employs various techniques for enhancing interactivity as follows. (1) It maintains background and default priorities separately in order to reduce how much background tasks interrupt tasks which interact with a user [16]. To do so, it makes use of a Linux kernel facility called *control group*. This aggregates all background tasks into a special scheduling group and this group is restricted to use no more than 10% of the CPU. As a result, background tasks do not seriously affect a foreground task which interacts with a user. (2) Android also achieves improved interactivity via enabling the fast startup of an application task [17]. As described in Section 2-A, an Android application task runs in its own Dalvik VM instance. However, it notoriously takes a long time to cold-start a VM instance. Since it leads poor responsiveness, Android uses *Zygote*, a pre-initialized and pre-loaded task which includes the common library classes that will be used by most of applications. By spawning an application task from *Zygote*, an application task can be launched in short startup time. (3) Android performs rendering work separately to reduce time to draw a screen [18]. In Android, a screen is composed of separate pieces of region called *surface*. Android draws each surface independently of each other so that only an updated surface is newly rendered. Each individual surface is then composited with others to the screen by a dedicated system server called *surface flinger*. By doing so, total rendering time is reduced and a user can perceive improved interactivity.

## III. Problem Formulation and Solution Overview

In this section, we model the target system, define our metric for evaluating the degree of interactivity and present the overview of the proposed solution approach.

### A. System Modeling

Our target system architecture is depicted in Fig. 1. Its software consists of the Linux kernel, the application framework and applications. Particularly, the application framework contains system servers, each of which is implemented as a set of Linux tasks. Since a system server is responsible for a certain dedicated system resource administration, each device is exclusively accessed by a dedicated system server. The system server types are specified by the Android framework. For example, *Surface Flinger* is a server which is in charge of the frame buffer. System servers are implemented in C/C++ to efficiently handle I/O requests. They run as ordinary Linux tasks.

In contrast, an Android application is executed on top of an independent Dalvik VM instance which in turn runs in an

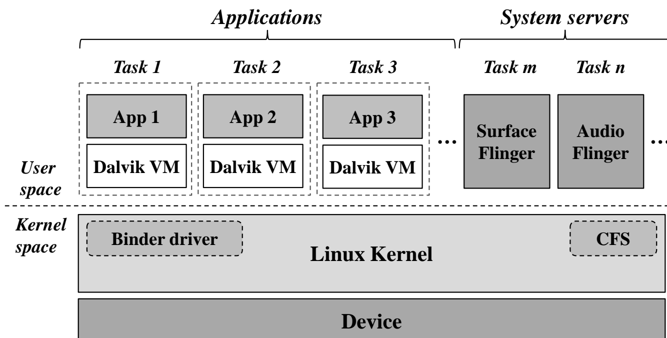


Figure 1. Target Android smartphone architecture

ordinary Linux task. Android applications are not allowed to access hardware devices directly via native system calls. It must use a different interface provided by a dedicated system server. Communication between an application and a system server is done by a special IPC mechanism called *binder*.

From the perspective of task scheduling, CFS does not distinguish system servers from application tasks except their weights. It allocates CPU times to tasks proportionally to their weights. In order to favor system servers over application tasks, the Android runtime assigns system servers with relatively higher weights. For example, the weight of *SurfaceFlinger* is 6100 (nice -8) whereas an application task is assigned weight 1024 (nice 0) by default [16].

### B. *Problem Definition*

We define the interactivity of a given system and formulate our problem at hand. We then explain why Android smartphones fail to demonstrate adequate interactivity by analyzing motivating examples.

As a metric for evaluating the interactivity of a system, we use the response time of a user input. It represents the amount of time span from a user input to the end of the response to that input. It consists of the execution time of a user interactive task and latencies caused by the kernel, an interrupt handler and other tasks. Suppose that a user input triggers application task $\tau_a$ whose execution time for processing the input is $C_a$. Then the response time of that user input is defined as below.

$$RT = C_a + L_I + L_S + L_P \qquad (5)$$

Interrupt handling latency $L_I$ is the delay from the generation of a hardware interrupt to the completion of the interrupt handler. Since the interrupt dispatching mechanism of Linux has been heavily optimized since its 2.5 kernel, the interrupt handling latency in the current Linux on a modern machine may well be less than 100 microseconds. Scheduling latency $L_S$ is the amount of delay between a task becoming runnable and executing the first instruction of the task. Preemption latency $L_P$ is the accumulated amount of time for which a task is disturbed by other tasks until completion of its execution. Clearly, our goal is to reduce $L_S$ and $L_P$ so as to decrease the response time.

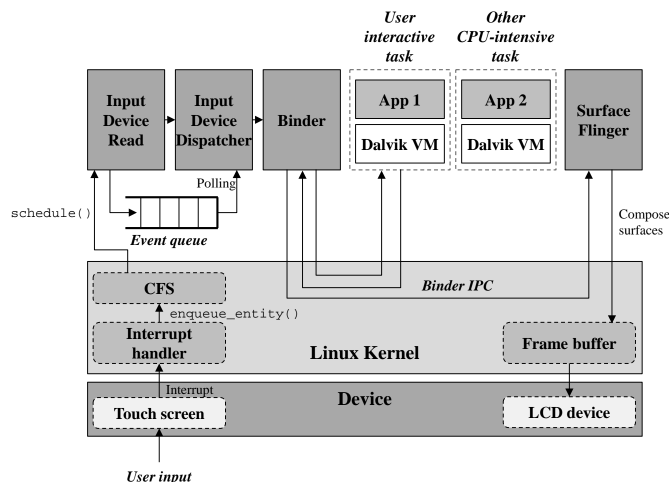


Figure 2. Event delivery path when the user interacts with the system

Fig. 2 shows the event delivery path from a user input to its response in Android smartphones. Suppose that a user touches the touch screen of an Android smartphone to draw a dot. The touchscreen device issues an interrupt and the kernel immediately stops the currently running task and jumps to the entry point of the interrupt handler which wakes up the *InputDeviceRead* task. It receives an incoming event data and stores its metadata into the event queue. For the case of a touching event, the metadata includes event occurrence time, contact coordinates, and so on. The *InputDeviceDispatcher* task takes out the stored metadata from the event queue and transmits it to a user interactive task via a *binder* IPC. The user interactive task then attempts to draw a dot at the designated coordinate but it does not draw the rendering result directly to the kernel's frame buffer. Rather, it stores the result to a dedicated memory area called *surface*. In turn, the updated *surface* is transmitted to *Surface Flinger* through a *binder* IPC. *Surface Flinger* synthesizes all the *surface*s in the system into one image and delivers it to the frame buffer. As a result of such complex interactions between the kernel and the framework, the user can finally see the dot on the screen.

Fig. 3 shows a Gantt chart which illustrates a sequence of task executions for the user interaction explained in Fig. 2. Note that the Gantt chart is vastly simplified for the sake of presentation. A user touches the screen at time $t_0$. At $t_1$, the interrupt handler completes its execution. Some internal tasks are then executed to convey the input event to the user

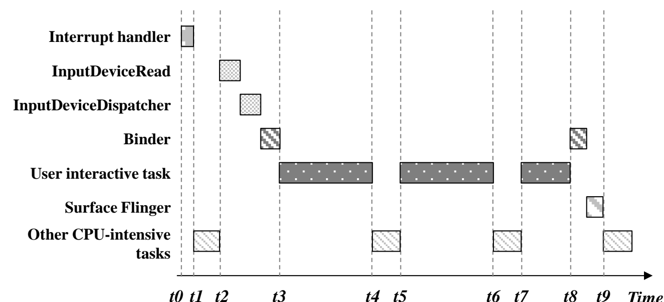


Figure 3. Gantt chart for user interaction described in Fig. 2

interactive task from $t_2$ to $t_3$. The task finally runs to handle the input event from $t_3$ to $t_8$. In this particular example, latency components $L_I$, $L_S$ and $L_P$ are equal to $t_1 - t_0$, $t_3 - t_1$ and $(t_5 - t_4) + (t_7 - t_6)$, respectively.

CFS does not identify a user interactive task to favor it over other tasks as explained in Section 2-B. A time slice given to a task is dependent on both the number of runnable tasks and their weights. Because every application task in Android is assigned the same weight specified by nice value 0, interactive tasks and CPU-intensive tasks are given the same amount of time slices. Often, the allotted time slice is not long enough to process a given user interaction. Thus, a user interactive task may be preempted by other tasks multiple times whenever it runs out of its time slice before completing the interactive work. This leads to high preemption latency. The example below illustrates this problem.

**Example 1** Consider an example in which ten CPU-intensive tasks with nice value 0 are running. Suppose that user interactive task $\tau_a$ takes 3 ㎳ to complete interactive work. Since all the tasks have an identical time slice of 1 ㎳ by (2), $\tau_a$ is preempted when it executes for 1 ㎳. In order to complete the response, $\tau_a$ is preempted more than twice and the total preemption latency of $\tau_a$ in this example is 20 ㎳.

Even though a user interactive task is identified and then favored, CFS may also lead to a long scheduling latency due to its non-preemptive nature. When a user interacts with an interactive task, CFS wakes up the slept user interactive task and inserts it into the run-queue after adjusting its virtual runtime by (4). In general, the user interactive task commonly uses the CPU for a small amount of time to set up an I/O and then sleeps awaiting the completion of the I/O. This nature makes the interactive task have relatively small virtual runtime and be inserted into the node nearby the leftmost node. Therefore, the interactive task can be rapidly scheduled as soon as a few task with smaller virtual runtimes run out of their time slices. However, the problem may occur when such tasks are CPU-intensive tasks with high priorities. The interactive task should wait in the run-queue until the time slices of all the preceding tasks run out. Note that a task is assigned a time slice proportional to its priority and CFS allows a task to be preempted only if it runs out of time slice. This problem is illustrated by the following example.

**Example 2** Consider an example in which nine tasks with nice value 0 (weight 1024) and one task with nice value -12 (weight 14949) are running in the foreground. Assume that they are all CPU-intensive tasks. According to (2), the time slice of the task with nice 0 is 0.42 and that of the task with nice -12 is 6.18. Suppose that user interactive task $\tau_a$ is woken up at time *t* and the task with nice value -12 has the smallest virtual runtime at that time. When $\tau_a$ becomes runnable at time *t*, it is assigned an adjusted virtual runtime by (4) and continues to execute until it runs out its time slice. In this example, $\tau_a$ will suffer from relatively large scheduling latency longer than 6 ㎳.

## C. *Solution Overview*

We have shown that CFS may lead to long preemption and scheduling latencies in Android smartphones. Two key factors behind the latencies are described as bellows.

- In Android smartphone, all application tasks running in the foreground are assigned the identical nice value of 0 and are scheduled by the CFS. In order to quickly respond to a user input, it ordinarily requires longer time than a time slice of a task with nice value 0. Therefore, the execution of a user interactive task should be disturbed by other tasks simultaneously running in the foreground. The higher preemption latency can be observed as the number of tasks increases.

- CFS is known as one implementation of a proportional fair-share scheduling algorithm which has its roots in weighted fair queuing [19]. However, contrary to intuitions, CFS schedules runnable tasks in a weighted round robin manner rather than in a weighted fair queuing manner. Unfortunately, this type of scheduling policy usually shows poor responsiveness since a task is given a time slice proportional to its weight and runs in a non-preemptive fashion until running out of its time slice. In particular, the higher scheduling latency can be observed when higher priority tasks exist in the run-queue and their virtual runtimes are smaller than that of a user interactive task.

In order to solve the problems described above, we present two interactivity enhancing mechanisms suitable for Android smartphones: (1) framework-assisted task characterization to reduce the preemption latency and (2) VT-CFS to reduce the scheduling latency. Fig. 4 depicts an overview of the proposed solution approaches. In order to shorten the preemption latency, we temporarily promote the priority of a user interactive task. Different from dynamic priority boosting adopted by the *O(1)* scheduler, our solution is assisted by the Android framework in classifying task's characteristic rather than relying on uncertain heuristics. In doing so, our modified framework detects a task which will get a user interaction event at run-time. It then informs the underlying scheduler of the task's identifier so that the priority of the user interactive task becomes temporarily promoted. Compared with the legacy CFS, the VT-CFS achieves the smaller amount of scheduling latency by reducing the time interval for which other tasks execute prior to the user interactive task. This interval is shown as $t_2 - t_1$ in Fig. 3. In the following two sections, each solution approach will be

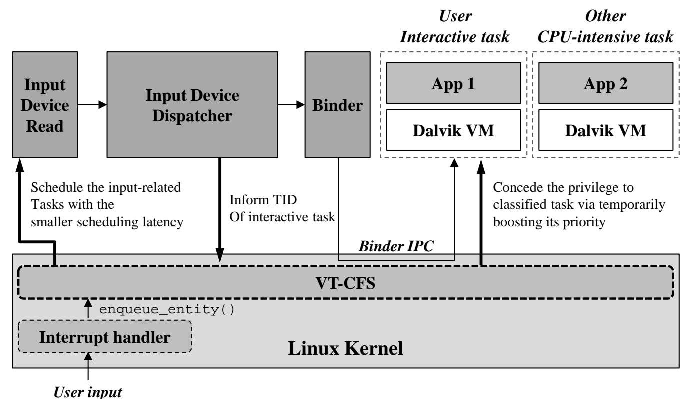

Figure 4. Overview of the proposed solution approaches: VT-CFS and framework-assisted task characterization

consecutively presented in detail.

## IV. Framework-assisted Task Characterization

This section presents a framework-assisted user interactive task characterization mechanism we additionally propose to enhance the interactivity of Android smartphones. The key idea behind this mechanism is to selectively promote the priority of a user interactive task so that it can get a larger time slice under CFS. Our mechanism is composed of two components: (1) one for identifying a user interactive task at the framework level and (2) the other for promoting the priority of the identified user interactive task at the kernel level.

Our mechanism is not the first attempt to improve the interactivity via boosting the priority of a user interactive task. As mentioned in Section 2, the Linux *O(1)* scheduler tracks tasks' run-time information such as sleeping time and then uses heuristics to identify interactive tasks. As shown in [12], it has been proved that such heuristics often misclassify the characteristics of tasks.

In fact, it is not easy for a kernel alone to identify a user interactive task since there are multiple tasks running in the foreground. We take advantage of a certain run-time behavior of the Android framework rather than relying on heuristics. It is known that in the Android framework, an event triggered by user interaction is first delivered and handled by a dedicated system server called *InputDeviceDispatcher*. The event is later dispatched to an application task through a *binder* IPC. In our approach, we modify the system server such that it can forward the task identifier (*TID*) of a user interactive task to the kernel during the *binder* IPC invocation. We make such a design decision because this incurs a minimal additional overhead since the system server has to interact with the kernel anyway by invoking the kernel's *binder* driver.

Once the Android framework identifies a user interactive task, the kernel scheduler promotes its priority. When the scheduler executes a wakeup routine, it compares *TID* of the woken-up task with the given *TID*. If the two identifiers match, the kernel treats the woken-up task as user interactive and then temporarily promotes its priority by calling the *setpriority()* function. The task is demoted to its original priority when it runs out of its time slice or becomes blocked again.

The motivation behind such priority boosting is to get a user interactive task a time slice larger than system-wide constant *C*. It represents the average time for a task to complete a user request. It varies depending on the computational power of the underlying hardware. We thus let *C* be a tunable parameter. The system administrator is responsible for carefully setting *C*. In our target system, we observe that *C* for usual applications is 100 ㎳ on average.

In order to assign user interactive task $\tau_a$ a time slice larger than *C*, our mechanism readjusts the nice value of $\tau_a$. To do so, it first computes a new weight for $\tau_a$ as follows.

$$W'(\tau_a) = \begin{cases} C \times \sum_{\tau_j \in S} W(\tau_j) / P & \text{if } W'(\tau_a) < \omega_{-20} \\ \omega_{-20} & \text{otherwise} \end{cases} \qquad (6)$$

where $S$ is the set of tasks in the run-queue, $\omega_{-20}$ denotes the weight of nice value -20 and $P$ is defined by (3). Note that a task's nice value can be decreased down to -20 by definition. Our mechanism then assigns $\tau_a$ a new nice value whose corresponding weight is larger than $W'(\tau_a)$.

Consider again the example demonstrated in **Example 1**. By our framework-assisted task characterization mechanism, user interactive task $\tau_a$ is identified and its priority is temporarily promoted to $\omega_{-20}$. In this particular example, the time slice of $\tau_a$ is computed to be 9.86 ㎳ by (2); it is larger than the required execution time 3 ㎳ of the user interactive task. Thus, task $\tau_a$ can complete its interactive work without being preempted. Theoretically, the task does not experience any preemption latency. However, in a real-world system, a small amount of preemption latency still remains since some urgent system servers such as the dynamic voltage frequency scaling task periodically run with high priority.

## V. Virtual-Time Completely Fair Scheduler

This section describes our VT-CFS algorithm in detail. As shown in Section 3, scheduling latency varies depending on the number of runnable tasks and their weights under CFS. We design VT-CFS such that it allows a task to be preempted at every predefined period. To do so, we modify the run-time algorithm of the CFS so as to schedule tasks in a weighted fair queuing manner.

VT-CFS maintains the identical data structure as the CFS. It uses a red-black tree as a run-queue and maintains a task's virtual runtime in the run-queue to provide fair share scheduling. Unlike the CFS, VT-CFS allows a task to be preempted at every predefined tick, instead of using the time slice of a task for preemption. This predefined tick is called a preemption tick. It is an integer multiple of a scheduling tick and a constant regardless of given workload. The length of the preemption tick interval is a tunable parameter capable of controlling a tradeoff between interactivity and run-time overhead. The system administrator is responsible for making a tradeoff decision. A meaningful value should be less than 6 ㎳ since a human being is known to perceive a latency larger than 5.7 ㎳ [20].

Fig. 5 summarizes the run-time algorithm of the VT-CFS with a flowchart. The VT-CFS updates the run-time information of current task $\tau_i$ at each scheduling tick. It then checks whether $\tau_i$ has been executed for more than one preemption tick period. If so, it compares the updated virtual runtime of $\tau_i$ with the smallest virtual runtime in the run-queue. Since a task with the smallest virtual runtime is stored at the leftmost node in a red-black tree, such comparison can be done in *O(1)* time. If the virtual runtime of $\tau_i$ is still the smallest, the VT-CFS lets it run for the next scheduling tick period. Otherwise, it preempts $\tau_i$ and inserts it back to the run-queue. In turn, the task with the smallest virtual runtime is selected for execution.

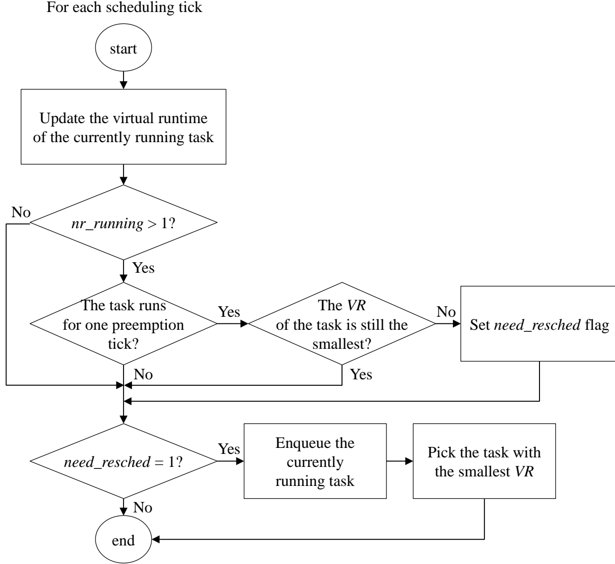


Figure 5. Flowchart of the run-time scheduling algorithm of VT-CFS

In order to bound the scheduling latency by a preemption tick period, it is critical to make a woken-up task be always inserted at the second node from the leftmost. To do so, VT-CFS adjusts the virtual runtime of woken-up task $\tau_i$ at time *t* as below.

$$VR(\tau_i, t) = \min_{\tau_j \in S}(VR(\tau_j, t)) + \frac{\omega_0}{\omega_{-20}} \times T \quad (7)$$

where $T$ is the scheduling tick period, $\omega_0$ and $\omega_{-20}$ denote the weights of nice value 0 and -20, respectively. Note that the VT-CFS makes a scheduling decision at every scheduling tick. Thus, a virtual runtime difference of any pair of tasks cannot be smaller than a virtual runtime increment of a task with nice value -20 running for one scheduling tick period.

To illustrate the effectiveness of the VT-CFS, we consider the same example as **Example 2**. Assume again that user interactive task $\tau_a$ is woken up at time *t* and the task with nice value *-12* has the smallest virtual runtime at that time. As defined in (7), the VT-CFS gives an adjusted virtual runtime to $\tau_a$ when it gets awaken. It lets the task with nice *-12* execute for one preemption tick period. The VT-CFS then preempts the running task since its virtual runtime becomes larger than that of $\tau_a$. As a result, the execution of $\tau_a$ is postponed for one preemption tick period. By definition, the length of the preemption tick period is smaller than 6 ㎳, which is a relatively small scheduling latency compared to the legacy CFS.

## VI. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the interactivity provided by the proposed framework-assisted task characterization mechanism and VT-CFS. We also show the run-time overhead incurred by our approach.

TABLE I. HARDWARE AND SOFTWARE COMPONENTS OF THE TARGET SYSTEM

| | | |
|---|---|---|
| **Hardware** | ***CPU*** | Dual core ARM Cortex A9 Clock: 1.2GHz |
| | ***Main memory*** | 1-GB LP-DDR2 |
| **Software** | ***Kernel*** | Linux kernel version: 2.6.32 |
| | ***Android Framework*** | Android 2.2 Froyo |
| | ***File System*** | YAFFS2 |

### A. Experimental Setup

We have implemented our framework-assisted task characterization and VT-CFS into Android 2.2 Froyo running with Linux 2.6.32. on the NVIDIA® Tegra™ 250 harmony board. We modified and recompiled the Android framework so that *InputDeviceDispatcher* could deliver the identifier of a user interactive task to the kernel scheduler. We also modified the Linux kernel with the new scheduling code implementing VT-CFS. The detailed hardware and software components of the target system are shown in Table I.

During our experiments, we relied on well-known tools and benchmark programs. In order to evaluate the enhanced interactivity achieved by the framework-assisted task characterization mechanism, we ran the Android painting application [21] with two CPU-intensive tasks; and then we measured the response time of the painting application using the *trace-cmd* tool. We ran *Interbench* [22] in order to evaluate the scheduling latency of VT-CFS. We ran *Quadrant* [23] and *Hackbench* [24] to evaluate the run-time overhead of framework-assisted task characterization and VT-CFS, respectively.

### B. Experimental Results

#### 1) Evaluating the interactivity of framework-assisted task characterization

In order to show the interactivity enhanced by the proposed framework-assisted task characterization mechanism, we ran the Android painting application [21] with two CPU-intensive tasks. They simply executed infinite loops and were assigned nice value 0. We measured the response time of the painting application by reading the temporal information of task scheduling with the *trace-cmd* tool. We repeated the test ten times and took the average response time. We then compared the results with and without the framework-assisted task characterization. We also measured the response time under the new Android framework with the integrated VT-CFS and framework-assisted task characterization.

Fig. 6 plots the average response times of the user interactive task with and without the mechanism. The horizontal axis denotes various target system configurations and the vertical axis does the three types of latencies and the response times explained in Section 3-B. Clearly, the response times are effectively reduced by adopting our mechanism. The response time measured under the legacy system is 275.9 ㎳

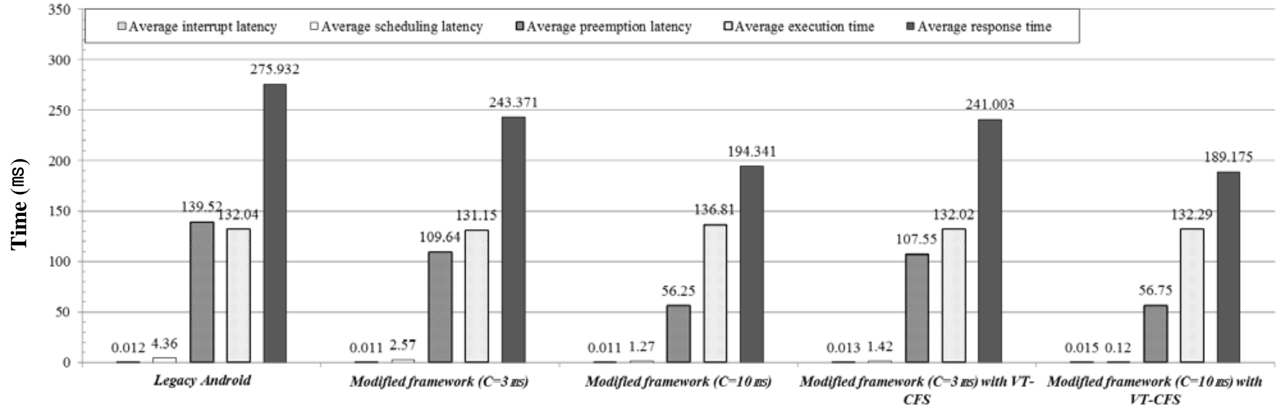


Figure 6. Bar chart of average response time of the painting application on the legacy Android system and the modified one with the different tunable option

and 50.56% of the response time is the preemption latency caused by the other tasks.

Note that *C* is a tunable value in the framework-assisted task characterization mechanism. When we set *C* equal to 3 and 10 ms, the response time was reduced to 243.37 and 194.34 ms, respectively. The average preemption latency in these configurations is 109.64 and 56.25 ms, which is 21.4% and 59.7% smaller than in the legacy system, respectively. We also set *C* to a larger value; however, it does not affect the interactivity since the Linux kernel does not allow a task's weight to be promoted larger than 88761. The modified framework integrated with the VT-CFS provides slightly shorter response time. The achieved response time is reduced by 12.6% and 31.4% when we set *C* to 3 and 10 ms, respectively.

*2) Evaluating the interactivity of VT-CFS*

In order to evaluate the scheduling latency of VT-CFS, we ran the *Interbench* benchmark program [22]. It runs a real time priority task that wakes up interactive tasks and measures scheduling latency under various background tasks. The types of simulated and background tasks used in our experiments are shown in Table II. Each task ran as a separate Linux task with nice value 0. Each simulation has run for 30 seconds and repeated ten times. We took the average scheduling latency values under both CFS and VT-CFS.

TABLE II. TYPES OF THE SIMULATED INTERACTIVE TASKS AND BACKGROUND TASKS

| | | |
|---|---|---|
| **Simulated interactive tasks** | ***X*** | Task simulating a GUI where a window is grabbed and dragged across the screen and requiring 0~100% of the CPU |
| | ***Audio*** | Real-time priority task running at every 50 ms and requiring 5% of the CPU |
| | ***Video*** | Real-time priority task trying to receive the CPU 60 times per second and requiring 40% of CPU |
| | ***Gaming*** | Task using 100% of the CPU without being blocked |
| **Background tasks** | ***Video*** | The video simulation task as a background load but running in non-real-time priority |
| | ***X*** | The X simulation task as a background load |
| | ***Burn*** | Four fully CPU-bound tasks |
| | ***Memload*** | Tasks simulating heavy memory and swap pressure |

The results of this test are shown in Fig. 7. The horizontal axis denotes types of background loads while the vertical axis does scheduling latencies measured in milliseconds. Each bar represents the average scheduling latency under a specific scheduler and the number above a bar indicates the maximally observed scheduling latency. In the case of CFS, a relatively high latency is observed when the interactive task runs with CPU- or memory-intensive background loads. This is because even if the benchmarked interactive task is woken up, it must wait until a background task with a smaller virtual runtime yields the CPU. Unfortunately, a CPU- or memory-intensive

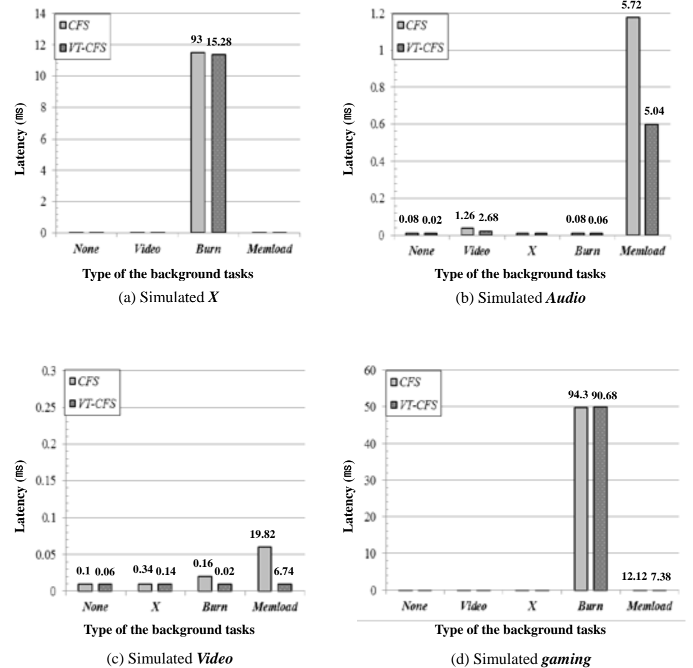


Figure 7. Bar chart of average latency of the interactive tasks on the legacy CFS and VT-CFS under various background tasks

task usually yields the CPU only when its time slice is expired or memory transaction is completed. Note that the *X* task requires a variable amount of the CPU ranging from 0 to 100%; the *Burn* task demands 100% of the CPU; and the *Memload* task demands almost 100% of the CPU. We also observed a huge amount of scheduling latency of 94.3 ㎳ with the *Gaming* interactive task. This is because the task did not get its virtual runtime adjusted by (4) since it never slept.

VT-CFS incurred a lower scheduling latency than the CFS as shown in Fig. 7. We observed small, bounded scheduling latencies in most of the benchmarks except *Gaming*. The maximum latency was 15.28 ㎳ when we set the preemption tick to 3 ㎳. As seen from the experiments, the latency was higher than the preemption tick in a few cases. This is because real-time priority tasks ran with the benchmarked tasks for the system service in the Linux operating system. Since they were scheduled earlier than the benchmark tasks, the scheduling latency could be higher than the theoretical value. In the *Gaming* benchmark, VT-CFS caused the maximum scheduling latency of 90.68 ㎳, which was comparable to that of the CFS.

*3) Evaluating the run-time overhead*

In order to measure the run-time overhead of the proposed approach, we used two benchmarks. First, we ran the *Quadrant* benchmark program [23] to evaluate the run-time overhead of the framework-assisted task characterization. It measures and scores the performance of the CPU, memory, I/O and 3D graphics processor. We focus only on evaluating the total score, CPU and I/O performance since memory and 3D graphics performance is highly hardware-dependent. We compared the results from these benchmarks with the legacy and the modified system, respectively. The run-time overhead of the framework-assisted task characterization is given in Table III. The results show that the modified framework achieves slightly better performance than the legacy framework. This is because the promoted priority of the benchmark task mitigates the disturbance caused by other background tasks. We also measured the performance of the modified framework integrated with VT-CFS. As shown, the extra run-time overhead incurred by our approach is only 0.91% of the legacy Android framework.

Second, we ran the *Hackbench* benchmark program [24] to examine the run-time overhead of VT-CFS. It creates a specified number of pairs of tasks with nice value 0 and lets each pair send and receive 100-byte data via either a pipe or a socket. The benchmark measures the time taken for all pairs to send data back and forth. We generated three sets which consisted of 100, 200 and 400 task pairs, respectively. For each task pair, we computed an accumulated response time by repeating the test 1000 times. Table IV gives the result which demonstrates that the VT-CFS achieves nearly identical results compared to the CFS. The extra overhead is only 0.99% and 0.46% on average when the *Hackbench* tasks communicate with each other via pipes and sockets, respectively.

TABLE III. OVERALL PERFORMANCE OF THE MODIFIED ANDROID FRAMEWORK MEASURED BY QUADRANT BENCHMARK

| | ***Legacy framework*** | ***The modified framework*** | ***The modified framework with VT-CFS*** |
|---|---|---|---|
| *CPU* | 4254.6 | 4403.1 | 4217.0 |
| *I/O* | 1006.8 | 1013.4 | 1005.6 |
| *Total score* | 1768.8 | 1798.9 | 1752.6 |

TABLE IV. OVERALL PERFORMANCE OF VT-CFS MEASURED BY HACKBENCH BENCHMARK

| ***# of pairs*** / ***Model*** | ***Legacy CFS*** | | | ***VT-CFS (T=5 ㎳)*** | | |
|---|---|---|---|---|---|---|
| | ***100*** | ***200*** | ***400*** | ***100*** | ***200*** | ***400*** |
| *Pipes* | 4.051 | 5.390 | 9.782 | 4.084 | 5.450 | 9.886 |
| *Sockets* | 3.244 | 6.846 | 13.99 | 3.272 | 6.848 | 14.06 |

Total elapsed time (*s*)

## VII. CONCLUSION

In this paper, we have presented two mechanisms for enhancing the interactivity of an Android smartphone. To do so, we have carefully analyzed the Android application framework and the task scheduler of the Linux kernel. Based on our analysis, we have proposed a framework-assisted task characterization and VT-CFS to reduce the preemption and scheduling latency of a user interactive task. The framework-assisted task characterization mechanism is a cooperative mechanism between the Android framework and the underlying kernel. It first identifies a user interactive task by receiving identifier information from Android's dedicated system server and then selectively promotes a priority of the identified task to favor it over other tasks. VT-CFS is an extension of the CFS which allows a task to be preempted at any preemption tick. It guarantees that the scheduling latency of a user interactive task is bounded by the tick interval. We have implemented the framework-assisted task characterization and VT-CFS into the Android framework and Linux kernel, respectively. Experimental results indicate that our approaches significantly improve the interactivity while incurring the negligible run-time overhead.

As future research, we are currently migrating our approach from Android version 2.2 Froyo to version 4.0 ICS by analyzing the new event delivery path which does not rely on the binder IPC for handling a user input. Also, we are extending VT-CFS to incorporate a virtual runtime-based task migration algorithm [25] which we have developed as an alternative to CFS's load balancing. We expect that the combination of the two can achieve synergistic effects in terms of improving interactivity as well as fairness.

ACKNOWLEDGMENT

The work reported in this paper is supported in part by the Technology Innovation Program (No. 10036495) funded by the Ministry of Knowledge Economy (MKE, Korea), by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) (No. 2012-0000348), by Advanced Institutes of Convergence Technology (AICT),

Institute for Smart Systems and by Automation and Systems Research Institute (ASRI).

## REFERENCES


[1] http://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform

[2] http://techland.time.com/2011/12/07/is-android-doomed-to-lag-more-than-ios/

[3] Tolia, N., Andersen, D.G., and Satyanarayanan, M. "Quantifying interactive user experience on thin clients", Computer, 2006, 39, (3), pp. 46-52.

[4] http://code.google.com/p/libgdx-users/wiki/ForceGarbageCollection

[5] http://ck.kolivas.org/patches/bfs/sched-BFS.txt

[6] http://sites.google.com/site/io/anatomy--physiology-of-an-android

[7] http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[8] http://developer.android.com/sdk/android-4.0-highlights.html

[9] Aas, J. "Understanding the Linux 2.6. 8.1 CPU scheduler", Retrieved Oct, 2005, 16, pp. 1-38.

[10] Wang, S., Chen, Y., Jiang, W., Li, P., Dai, T., and Cui, Y. "Fairness and interactivity of three CPU schedulers in Linux". Proc. 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009, pp. 172-177.

[11] Wong, C., Tan, I., Kumari, R., Lam, J., and Fun, W. "Fairness and interactive performance of O (1) and CFS Linux kernel schedulers". Proc. International Symposium on Information Technology, 2008, pp. 1-8.

[12] Salah, K., Manea, A., Zeadally, S., and Alcaraz Calero, J.M. "On Linux starvation of CPU-bound processes in the presence of network I/O", Computers & Electrical Engineering, 2011, 37, (6), pp. 1090-1105.

[13] Torrey, A., Cleman, J., and Miller, P. "Comparing interactive scheduling in Linux", Software- Practices & Experience, 2007, 34, (4), pp. 347-364.

[14] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S.K., Gehrke, J.E., and Plaxton, C.G. "A proportional share resource allocation algorithm for real-time, time-shared systems". Proc. 17th IEEE Real-Time Systems Symposium, 1996, pp. 288-299.

[15] http://www.linux-magazine.com/Online/News/Con-Kolivas-Introduces-New-BFS-Scheduler

[16] http://developer.android.com/reference/android/os/Process.html

[17] Bornstein, D. 'Google i/o 2008-dalvik virtual machine internals', 2008

[18] https://plus.google.com/105051985738280261832/posts/XAZ4CeVP6DC

[19] Li, T., Baumberger, D., and Hahn, S. "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin", ACM Sigplan Notices, 2009, 44, (4), pp. 65-75.

[20] Kelly, D.H. Visual science and engineering: models and applications. CRC, 1994, edn. 1994

[21] http://pixle.pl/bord/

[22] http://ck.kolivas.org/apps/interbench/interbench-0.31/readme.interactivity

[23] http://www.aurorasoftworks.com/products/quadrant

[24] http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

[25] Huh, S., Yoo, J., Kim, M., and Hong, S. "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm". Accepted to 32nd International Conference on Distributed Computing Systems, 2012.