Multi-Layer Real-Time Support for JVM-based Smart Phone Systems

Youngjoo WOO[1], Donghyouk LIM[2], YungJoon JUNG[2], Euiseong SEO[1]
[1]*Sungkyunkwan University, Republic of Korea*
[2]*Electronics and Telecommunications Research Institute, Republic of Korea*
**Corresponding author: euiseong@skku.edu*

***Abstract*—Employing the Java virtual machine (JVM) architecture provides smart phone systems stability and security by sandboxing third-party applications and controlling their behavior. However, the JVM layer hinders applications from notifying the operating system scheduler about their timeliness requirements; therefore, applications sometimes fail to respond on time. In order to improve the responsiveness of smart phone applications, this paper proposes two schemes. First, for existing applications that cannot be rebuilt, we modify the kernel scheduler to value task priorities over fairness. Second, we propose cross-layer real-time support APIs to deliver applications' priorities to the kernel scheduler, which will help developers to add real-time scheduling support to their applications. Our prototype demonstrates that the suggested schemes dramatically improve response times and throughputs of prioritized applications.**

***Index Terms*—real-time schedulers, scheduling algorithm, smart phones, Java, virtual machines.**

I. INTRODUCTION

The third-party smartphone app market is dramatically expanding. Currently, there are over a million mobile apps in various app marketplaces. Users rely on these apps for a variety of tasks, from posting mildly amusing comments on social network services to online banking [1].

Thus, most modern smart phone systems are equipped with multitasking capability, and the degree of concurrency in smart phone systems is steadily increasing. According to a research group, tens of applications are running concurrently on today's smart phones [2].

As the degree of multitasking in a smart phone increases, so does the importance of maintaining the security and stability of third-party applications. For example, if the system is poorly protected, even a single malicious or improperly designed application may crash the whole system or steal sensitive information from other applications. Therefore, smart phone operating systems (OSs) require an execution architecture that is able to sandbox and control the behavior of third-party applications.

By confining the activities of Java applications in their virtual machines, the JVM architecture fundamentally prevents applications from directly accessing to the system components [3]. Thus, smartphones that may run uncertified third-party applications can protect the system from unstable or malicious third-party applications by employing the JVM architecture.

Therefore, several commercially successful smartphone operating systems including Google's Android, Nokia's Symbian, RIM's BlackBerry OS and Maemo currently employ the JVM or a derivative structure, and the use of JVM or other similar VM architectures is expected to increase in future smartphone operating systems.

Many smartphone applications such as games, media players and web browsers are interactive or involve multimedia. These applications are soft real-time tasks, of which scheduling latency directly impacts user experience and satisfaction. Therefore, smart phone OSs must fulfill the timeliness requirements of applications as well as protecting their security.

Most embedded OS kernels offer real-time schedulers of one kind or another in order to conform to timeliness requirements [4-7]. The real-time schedulers conduct scheduling decisions based on information about the timeliness requirements of applications, which are provided by applications to the kernel through system calls. However, because the JVM layer prohibits applications from directly accessing kernel interfaces, applications running inside JVMs cannot deliver their timeliness requirements to the kernel. Consequently, they cannot benefit from the real-time schedulers of embedded OS kernels [8].

This paper presents a modified kernel scheduler scheme that improves the responsiveness of applications running inside JVMs under heavy system load. Our approach instructs the scheduler to weight applications' priorities more heavily than the conventional schedulers so that it prioritizes responsiveness rather than fairness in scheduling decisions. Although this approach requires no modification or rebuilding of existing applications, however, it does not fully utilize the existing real-time scheduling features of modern embedded OSs.

Thus, for applications that require rigorous real-time characteristics and can be modified at the source code level, this paper also proposes a set of cross-layer real-time support application programming interfaces (APIs) [9]. This API set, which is implemented as a Java API library, delivers the scheduling requirements of applications to the kernel scheduler via JVMs, thereby allowing the kernel scheduler to provide real-time scheduling services to applications.

We implement the prototypes of the suggested schemes by modifying Google Android. In addition, we port the prototype implementations to an off-the-shelf smart phone

This research was supported by Basic Science Research Program (2012R1A1A2A10038823) through the National Research Foundation of Korea (NRF), and also by the IT R&D Program (10041244, Smart TV 2.0 Software Platform) through KEIT funded by the Ministry of Science, ICT and Future Planning.

Digital Object Identifier 10.4316/AECE.2013.03001

device for evaluation. We evaluate the suggested schemes in terms of scheduling latency as well as system throughput with both real application and benchmark workloads.

The remainder of this paper is organized as follows. The background and related work are introduced in Section II. In Section III, we propose the real-time support schemes. We evaluate the prototype implementations of our schemes in Section IV. Finally, Section V concludes our research and discusses directions for future research.

II. Background and Related Work

A. JVM-based Smart Phone OSs

JVM-based smart phone OSs execute third-party Java applications through JVMs in order to obtain security and hardware independence. This architecture was long used on feature phones fitting the Mobile Information Device Profile (MIDP) before the smart phone era began [10].

In the MIDP, an application is implemented as a MIDlet, which is similar to a Java applet, and it is executed in a lightweight JVM. Applications thereby become platform-independent, running regardless of the underlying hardware. However, the insufficient hardware performance at that time significantly limited the use of MIDlets on mobile phones.

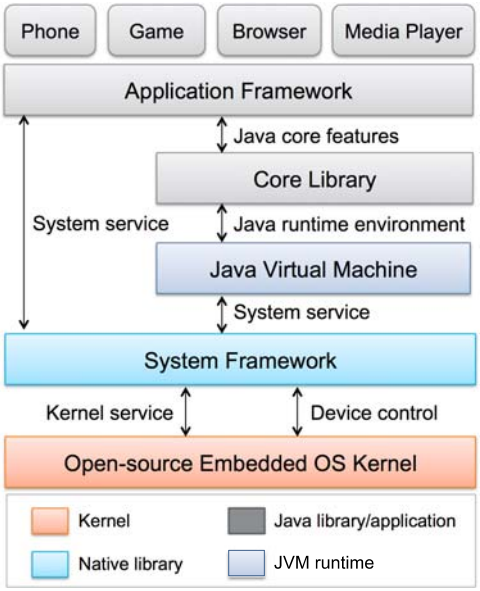

Figure 1. Diagram of typical JVM-based smart phone architecture.

The rapid improvement of mobile processor performance revived the MIDP architecture in smart phones. Figure 1 shows the architecture of a typical JVM-based smart phone software stack. Many smart phone OSs employ existing general-purpose embedded OS kernels because of their powerful features and abundant support tools. The system framework usually comprises system interface libraries, system software tools, graphics and multimedia support libraries and system support daemons, all of which run as native, not JVM-based, tasks.

Above the kernel and system framework, a JVM runs as a normal task. On top of the JVM layer, sits the core Java library layer and the application framework layers, supporting Java applications that run inside the JVMs. The application framework and core libraries provide developers with useful common methods such as GUI manipulation or networking. Requests related to system services are also delivered to the system framework or kernel through the application framework or core libraries.

While the JVM-based smart phone systems have many desirable features, such as hardware-independent executable files and system and data protection, they also display significant weaknesses, especially in response time. The kernel essentially schedules JVMs as non-real-time tasks, regardless of the importance of the applications inside the JVMs. Currently there are no means for applications inside JVMs to request real-time scheduling to the kernel even though the kernel equips real-time scheduling features. When combined with the reduced performance due to the existence of the additional JVM layer, this drawback may significantly harm the quality of service (QoS) and in turn the user experience of applications.

Although the performance overhead due to the JVM layer has been continually improved using cutting-edge technologies such as the just-in-time (JIT) compilation, scheduling latency remains a critical issue. The long scheduling delay of JVMs can induce slow or unpredictable response time of Java applications.

The specifications for a real-time JVM and Java APIs have been suggested so that Java can be used for hard real-time embedded systems [11]. Also, a JVM with enhanced thread scheduling schemes has been suggested [12] Although these approaches guarantee real-time scheduling or improve scheduling latency, they also require the underlying OSs to schedule the JVMs on time and to provide sufficient amount of processor time to them. Since most smart phone applications do not demand rigorous timeliness requirements and the hard real-time JVMs are complicated, currently most smart phones or consumer electronics are not using hard real-time JVMs. According to our evaluation, which is introduced in Section IV, an ordinary JVM reacts sufficiently fast even under extremely heavy CPU load if it is properly scheduled by the OS-level real-time scheduler.

B. Real-Time Scheduling Supports

To achieve the short response delay required by smart phone systems, OSs must limit the scheduling latency of applications below a certain threshold. Thus, a lot of research has been directed towards the design and implementation of preemptible kernels, in which prioritized applications are scheduled preferentially even to the kernel.

In addition to preemptible scheduling, many research groups have focused on improving the scheduling latency, and a few schemes such as priority inheritance mutex lock [13-14] and interrupt threading [15] are now frequently used by many embedded systems.

However, the improvement in the response delay resulting from employing such scheduling schemes also degraded performance. To remedy this, the adaptive scheduling scheme [16] was suggested, which uses real-time scheduling features only when necessary.

Applications for JVM-based smart phone OSs hardly benefit from these research achievements, since the JVM layer that lies between applications and the kernel prohibits applications from directly interacting with the kernel to deliver their timeliness requirements, which is essential

information for the kernel to properly schedule applications according to their timeliness requirements. For satisfactory user experience, it is especially crucial to schedule game or multimedia applications through strict real-time schedulers, which are provided by most embedded OSs.

As a result of this limitation, the current JVM-based smart phone OSs run applications through normal fair-share schedulers instead of real-time schedulers. In a fair-share scheduler, the priorities of tasks are not absolute criteria for determining the next task to schedule. Therefore, under heavy load or conditions of high concurrency, the execution of applications with higher priorities may be delayed in favor of the execution of applications with lower priorities. Therefore, even prioritized applications may fail to respond on time given adverse circumstances.

The background computation load is steadily increasing due to the background network packet encryption, hash value computation for memory deduplication, use of encrypted file systems and so on. In addition, growing popularity of large-sized multimedia data such as 3D or hologram media files induces extremely high computation load generally with low priorities.

Like many other smart phone OSs, Android and its kernel, which are used in our research, employ a variant of the fair-share scheduler [17], named Completely Fair Scheduler (CFS) [18-19]. The general Linux kernel as its default scheduler employs this scheduler. Thus, it is also used in OSs for PCs and even for servers. Similar to other fair-share scheduler variants, it compromises the response times of prioritized tasks in favor of fair distribution of processor time. However, considering the characteristics of smart phone applications, which are mostly interactive and multimedia-related, priorities of applications in smart phone systems should be taken more into consideration than fairness.

To overcome this limitation, the applications running inside JVMs require certain means to deliver their timeliness requirements for real-time scheduling to the kernel. Although such cross-layer optimization approach may harm the modularity in design and incur hidden dependency between layers, it is being frequently used for improving network systems, where the lower layers are unaware of the information defined in the upper layers like our target environment, due to its effectiveness.

In the bare-metal virtualization architecture, VMs with real-time applications schedule their application tasks with the real-time schedulers. However, VM schedulers that run inside the hypervisor are not aware of the existence of the real-time applications inside VM. Therefore, some researchers proposed scheduling schemes that inform the timeliness requirements of the real-time applications inside a VM to the hypervisor and schedule the VM with real-time VM schedulers [20-21].

III. Proposed Cross Layer Framework

A. *Intensified Priority Scheduling*

Because the current JVMs do not offer programming interfaces to manipulate the kernel-level scheduling policy, the priorities of applications are translated into normal-level priorities only for the CFS, not for the real-time scheduler. For an existing application to utilize the real-time scheduler, the application must be modified beforehand to notify the underlying layers of its intention to have real-time priority. Therefore, to improve the responsiveness of existing applications without modifying or rebuilding them, we have to revise the CFS.

As explained in Section II.B, the scheduler in a smart phone OS must emphasize task priorities over fairness in making scheduling decisions. We promote the influence of task priorities by modifying the existing kernel. This approach will improve the responsiveness of higher-priority tasks at the expense of some degree of fairness.

The fundamental algorithm of the CFS is as follows. The CFS periodically allocates time slices to tasks according to their priorities. The higher the priority of a task, the more time slices it receives. The time slices of a task are deducted proportionally to the execution time of the task. Basically, when choosing the next task to schedule, the CFS selects the task with the largest number of remaining time slices. When every task has used up its allotted time slices, or when a predefined length of time has passed, the CFS again allocates time slices to tasks. By repeating these steps again and again, the CFS respects tasks' priorities while maintaining fairness of processor time among them.

The base or static priority of a task in the Linux kernel is an integer number between 0 to 139. The priority number from 0 to 99 are reserved for real-time class tasks. Tasks with the priority values between 100 and 139 are scheduled by the CFS. The priority value in this range is one-to-one mapped to the *nice* value, which ranges -20 to 19. For example, if the nice value of a task is set to -19, the priority of the task will be changed to 101. By default, a normal task has a priority value of 120, which corresponds to a nice value of 0. The dynamic priority, which is actually used for scheduling basis by the CFS, is determined based on this static priority. The CFS temporarily promotes the dynamic priority of a task when the task sleeps for significantly long time to boost the performance of interactive or I/O bound tasks.

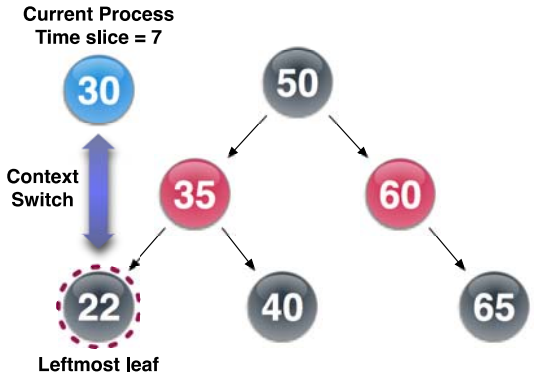

Figure 2. A red-black tree is used in the CFS to determine the next task to schedule.

The CFS uses a red-black tree data structure [22], as shown in Figure 2, to enable fast retrieval. Each node in the tree denotes a task, and the key value of each node holds the amount of virtual runtime belonging to the corresponding task.

The virtual runtime of a task increases proportionally to the cumulative execution time of the task. In addition, the

priority of the task affects the speed of the increment in virtual runtime. The value increases 10% faster with an increase in priority value by 1. For example, if the priority of a task is 120 and it was executed for 1 s, the virtual runtime of the task would be increased by 1. If the priority of a task was 121, demoted from the default priority value, and it executed for 1 s, the virtual runtime of the task would be increased by 1.1.

The leftmost leaf of a red-black tree always has the smallest key value. Therefore, the leftmost leaf represents the task for which cumulative execution time is the shortest among all tasks. At the end of every time slice, which is the unit time of task execution, the CFS compares the virtual runtime of the current task with that of the leftmost leaf of the tree. It conducts a context switch from the current task to that of the leftmost leaf when the difference between the virtual runtime values becomes larger than the length of a time slice. For example, if the virtual runtime of the currently executed task were updated to 30 after a time slice of 7 s, and if the virtual runtime of the leftmost leaf were 22, the CFS would conduct a context switch to the task in the leftmost leaf because the difference between their virtual runtimes, 8, would be greater than the time-slice length of 7 s. After the context switch, the leftmost leaf task will be popped out of the tree and the current task will be pushed up to the tree.

This scheduling policy generates frequent context switches. However, fairness among tasks is accurately controlled. Also, since the virtual runtimes of lower-priority tasks increase faster than those of higher-priority tasks, higher-priority tasks are supposedly scheduled more frequently and preferentially.

If the speed at which the virtual runtimes are increased is cut by half, the number of context switches, in accordance with intuition, will also be reduced by half. As a result, fairness will be slightly reduced, while priority will have greater influence on scheduling.

Based on this observation, we propose a modified CFS (MCFS) that intensifies the influence of task priorities. By adjusting the speed at which virtual runtimes increase, we defer context switching and favor higher-priority tasks more than does the CFS. The MCFS is easy to apply to any system using the CFS, and all applications will benefit without requiring any modifications or rebuilding.

Figure 3 shows the procedures for virtual runtime update operations in the (A) CFS and (B) MCFS. The CFS manages two values, the cumulative executed time and the virtual runtime. At every time slice, the scheduler calculates the delta value and the delta_fair value. The delta value is the actual amount of elapsed time, and the delta_fair value is the amount by which the virtual runtime is incremented, cut by a predefined rate. The cut-rate is defined system-wide; all tasks share a single cut-rate value. The delta value is added to the variable for cumulative executed time, and the delta_fair value is added to the virtual runtime of the task.

If we decrease the cut-rate, fairness will be increased accordingly. On the other hand, if we increase the cut-rate, fairness will be diminished and priority will have more influence. In addition, a high cut-rate will boost up the throughput of the whole system by reducing the number of context switches. For example, in Figure 3, the virtual runtime increases as fast as one-fourth of that under the CFS when the cut-rate of MCFS is four. Assume that the smallest virtual runtime in the run-queue is 66. The current task will keep running in the next time slice under the MCFS while it will be preempted and the next task in the runqueue will be scheduled under the CFS.

When the cut-rate is excessively high, high-priority processes behave like real-time class processes. Under this circumstance, high-priority processes can monopolize CPU resource and starve low-priority processes. The applications that are supposed to run as real-time processes were usually designed to avoid the CPU monopoly in one way or another.

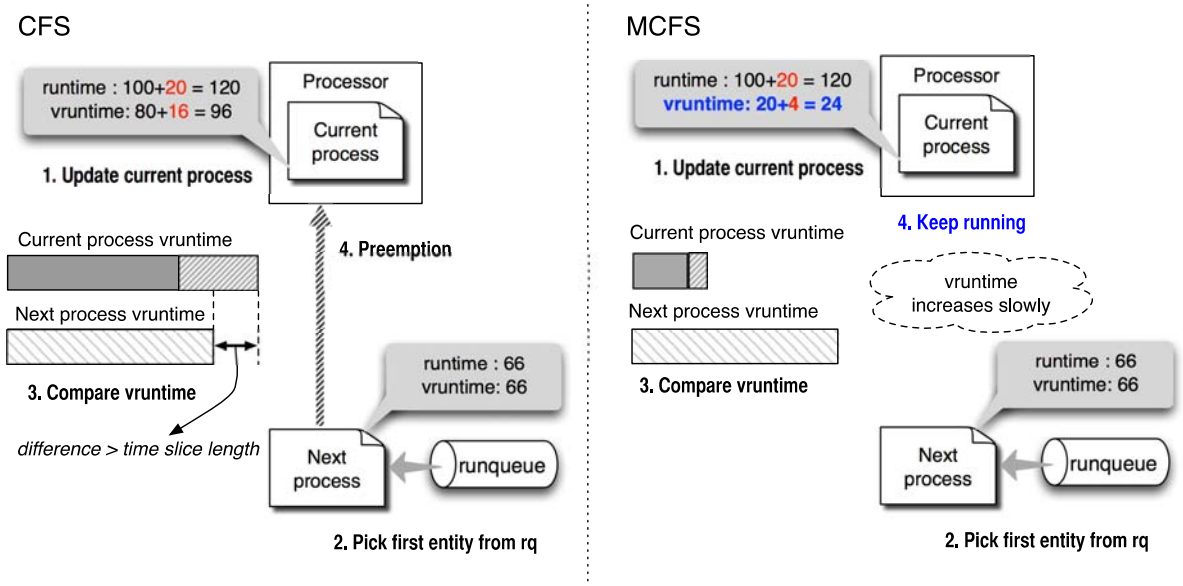

Figure 3. In the MCFS, decreasing the factor of delta_fair slows the increment of virtual runtimes.

In addition, the users are aware of the risk when they promote the priority of the processes to the real-time class. However, users cannot expect and anticipate the sudden system freeze or sluggish response from critically unfair scheduling under the MCFS with an excessively high cut-rate. Therefore, the cut-rate should be chosen carefully considering the trade-off between responsiveness of prioritized applications and fairness of scheduling.

B. Cross-Layer Real-Time Support APIs

Running JVMs through the real-time scheduler is the fundamental solution to applications' timely responses [11],[23]. However, the Dalvik JVM of Android and many other JVMs running inside smart phones prioritize applications only with the nice values or priority values of non-real-time normal schedulers.

The Java run-time framework categorizes application tasks into two groups: foreground tasks and background tasks.

Foreground tasks are applications that are being shown on the display, or that have high priorities explicitly set by users or developers. The JVMs for foreground tasks are placed in the *Normal queue*, in which normal system tasks are also placed. In order to obtain fast response times given the many competing tasks, applications change their priority values through the application framework.

Applications that are not shown on the display are categorized as background tasks. Foreground tasks are also recategorized as background tasks when they leave the screen. Background tasks are placed in the *Batch queue*. Both the Normal Batch queues are provided and managed by the CFS.

Tasks in the Normal queue are scheduled preferentially to tasks in the Batch queue. Therefore, multimedia or interactive applications, which are usually displayed on the screen, tend to have short scheduling latency by virtue of being scheduled before any background tasks.

However, despite this approach, the scheduling delays of foreground tasks increase proportionally to those of competing tasks in the Normal queue when there are extremely many concurrent tasks or when some tasks are heavily using processor resources. Under heavy load, even a task's high priority does not guarantee sufficiently short scheduling delay.

We resolve this issue by proposing a cross-layer real-time support framework. By providing Java API methods, the proposed framework enables applications inside JVMs to place their JVMs in the real-time scheduling queue so that their JVMs run as real-time tasks. Also, applications may change their JVMs' priority in the real-time queue on the fly through the API methods.

The proposed framework is designed to be a Java library. Applications invoke the framework methods like other methods offered by the Java application framework. As shown in Figure 4, through the Java methods of the suggested framework, applications can notify the kernel of their desired scheduling policies and priorities. When applications invoke a method of the framework, the framework calls the corresponding system call through the Java native interface (JNI).

In many cases, applications rely on other applications or on system services [24]. For example, a music player depend on the media server, which is a system service provided by the system framework, for decoding music files and playing them. In such cases, prioritizing an application alone cannot guarantee the quality of music services. Therefore, the framework should prioritize both applications and their dependent services at once. However, it is difficult to identify these dependency relationships among applications and services. Therefore, our scheme requires applications to notify the kernel explicitly of their dependent services or applications.

Tasks in the real-time class are scheduled purely based on their priorities. Therefore, it is possible for a malicious or malfunctioning task in the real-time class to crash or freeze the entire system by monopolizing the processor resource. In addition, if many JVMs claim that they want real-time priority at the same time, the effectiveness of the real-time scheduling will significantly diminish.

Therefore, in a production system, the cross-layer real-time support framework should require user approval for any scheduling policy or priority changes of an application through GUI during application runtime or through the privilege configuration schemes such the Android privilege manifest during application installation.

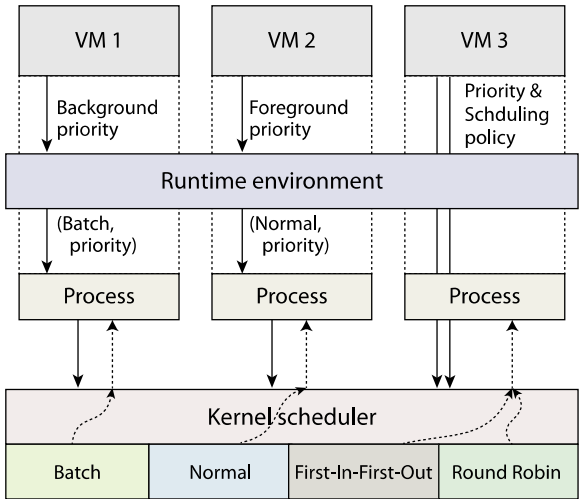

Figure 4. The proposed real-time support framework passes the timeliness requirements of applications directly to the kernel-level scheduler.

IV. EVALUATION

As mentioned, the prototype of the suggested schemes was implemented in Android OS. Android employs the Linux kernel as its kernel, like many other current smartphone OSs. The Linux kernel, by default, uses the CFS as its scheduler. The implemented prototype was ported to an off-the-shelf smart phone device, Google Nexus One, which is built for the Android OS. The Android and Linux kernel versions for the prototype implementation were 2.2.1 and 2.6.35, respectively.

In our evaluation, an open-source benchmark tool, Hackbench, was used to put the system under load as done in the previous research [16]. Besides Hackbench tasks, we executed a music player and a hypothetical periodic alarm program as the evaluative programs. The QoS of the music player and alarm program were monitored and analyzed in our evaluation.

Hackbench creates a process group, which usually

consists of 40 independent chatting processes. The processes in a process group simulate a chatting service by sending and receiving messages to each other. A process group terminates itself after a predefined number of messages. To maintain the system load, we kept Hackbench processes repeating throughout the experiments.

The load on the system can be minutely controlled through the number of concurrently running process groups. Although a larger number of process groups lead to higher system load, the relationship between the system load and the number of concurrent process groups is non-linear. This property has to be considered when analyzing experimental results.

First, we measured the scheduling delay of the alarm program. This application creates a thread that enters the sleep state for 100ms and then wakes up. This sleep and wake cycle repeats continually. After each sleep, the application prints out the current time on the screen. By calculating the interval between time stamps, we can measure how long the scheduler delayed the thread's execution from its scheduled wake time. Without any workload, which means zero Hackbench process groups on the system, the scheduling delay of the alarm clock application did not exceed 3ms in any case.

When many tasks are competing for the processor, the scheduling of the alarm clock task may be deferred in favor of the other tasks. We ran the alarm clock application with different numbers of Hackbench process groups under the CFS, MCFS, and CFS with the real-time support framework, respectively. The priority of the alarm clock program was set to the highest priority, priority value 0, under both CFS and MCFS and to real-time priority under the real-time support framework. The Hackbench tasks were set to be scheduled under CFS with the nice value of 10, which is the default configuration for background processes in the Android framework.

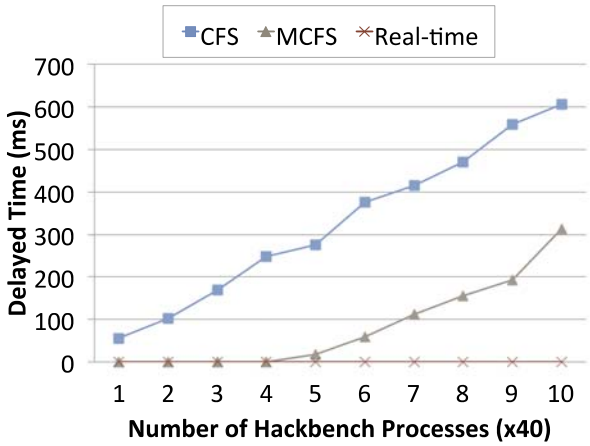

Figure 5. Maximum delayed scheduling time of the alarm clock task under different scheduling schemes while varying the number of Hackbench groups.

Given the same load, the scheduling delay of the alarm clock task varies greatly depending on the scheduler. Because the user experience is determined by the maximum rather than the average delay, we measured the maximum scheduling delay given different system loads as shown in Figure 5.

The CFS generated scheduling delay even with only a single Hackbench process group, while the MCFS generates noticeable delay when over five Hackbench process groups are running concurrently. With the CFS, the scheduling delay leapt up to 600ms when 10 Hackbench process groups were running. 600 ms is significantly long, a delay that humans can easily perceive. After five Hackbench process groups were running concurrently, the increase of the scheduling delay for each additional Hackbench process group under the MCFS became similar to that under the CFS. However, even with 10 concurrent Hackbench process groups, the scheduling delay under the MCFS was only half that of the CFS.

While the exact value of the perception threshold is dependent on the user and the type of task being accomplished, a value of 50ms is commonly used [25]. Thus, under the CFS, users would recognize jitters and delays from applications even when there is background workload as heavy as only a single hackbench group. The same amount of jitters or delays will be experienced under the MCFS when there is computational load as much as six hackbench groups, which is a rare situation in mobile devices.

The scheduling delay with the real-time support framework remained below 10ms. The alarm task was scheduled right after it woke up.

The cut-rate of the MCFS used in our experiments was empirically determined to be 16. In order to investigate the effects from the cut-rate value, we conducted the same experiments with different cut-rates. In these experiments, we increased the cut-rate by doubling it. If the cut-rate is chosen to be a power of two, applying the cut-rate can be implemented with a shift instruction instead of multiplication and thus the computation overhead can be saved.

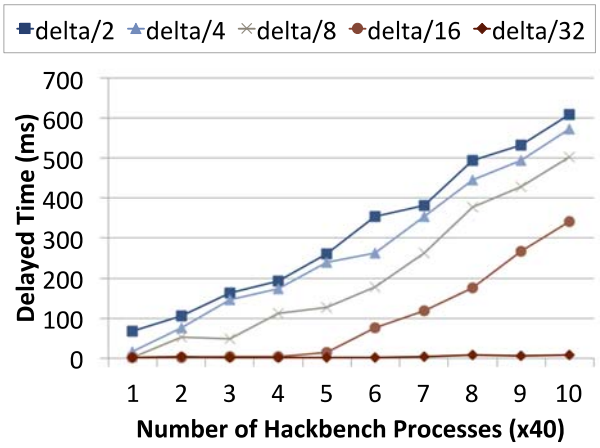

Figure 6. Maximum delayed scheduling time of the alarm clock task under MCFS with different cut-rates while varying the number of Hackbench groups.

Figure 6 shows the experiment results. As easily expected, the scheduling latency decreased as the cut-rate rose. However, in comparison to that under the CFS, the scheduling latency differed notably only when the cut-rate was higher than 8. This means that the degree of the fairness degradation due to the improved scheduling latency under MCFS is substantial. Especially, when the cut-rate was 32, the alarm clock process was scheduled like a real-time process. This means that even little difference in priority enables CPU monopoly by a process. Consequently, the responsiveness of other processes as well as the stability of

the whole system can be harmed critically. Considering this, the cut-rate should be chosen carefully based on the expected maximum load and required response time of the prioritized applications.

Next, we evaluated the proposed schemes with a multimedia player. The experiment's configurations were similar to the previous experiment. We ran multiple numbers of Hackbench process groups together with the music player and measured the QoS of the music player. The priority of the music player was set to the highest priority under both CFS and MCFS and to real-time priority under the real-time support framework, while that of the Hackbench tasks was set to normal.

Under the CFS, the perceptive QoS of the music player got worse with larger number of Hackbench process groups running on the system. The experimenters easily recognized frequent jitters and suspension of music playback under heavy load. Also, the music took significantly longer to play.

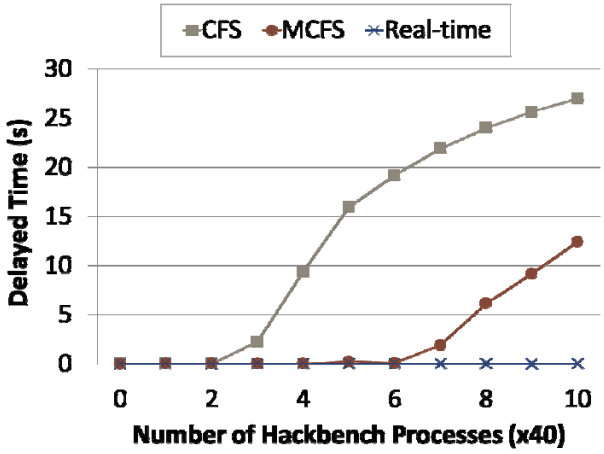

Figure 7. Delayed playback time of music player given a varying number of Hackbench process groups.

Figure 7 shows the prolonged playback time of the same music file, originally 3min 59s long, given a varying number of concurrent Hackbench process groups and under different scheduling schemes. If the music play is prolonged by 30s, the total music playtime will become 4min 30s.

Under the CFS, music playback was noticeably prolonged with only three Hackbench process groups running concurrently. The prolongation increased linearly with an increase in the number of Hackbench process groups, increasing up to 30 s with 15 concurrent Hackbench process groups.

Under the MCFS, significant prolongation of playback occurred only when there were more than seven concurrently running Hackbench process groups. Also, in the worst case, the prolongation of playback was only two-thirds of that under the CFS. We could not find any noticeable or perceived delay, jitters, or suspension of playback under the real-time support framework.

We further investigated the QoS of the music player by examining its decoding and buffer-writing activity. The music player periodically decodes a block of a music file and writes the decoded data to the sound buffer. When there is long delay in decoding of a single block, the quality of music playback may not degrade if there is remaining decoded data in the buffer. When the buffer is emptied before replenishment as a result of delayed decoding, jitters or suspension of music play occur. Thus, the single block writing delay is not an appropriate metric to judge the quality of service. Although the cumulative buffer writing delay is not a direct measure of the music playback quality either, we found that it reflects the quality of music playback more accurately than the writing delay of a single block.

Normally, the decoding and buffer-writing activity takes 15ms to 20ms under the CFS when there are no other active tasks. Thus, we defined the buffer-writing delay as the difference between the elapsed decoding and buffer-writing time and 20ms. For example, when decoding and buffer-writing a block takes 100ms, the buffer-writing delay of this block is 80ms.

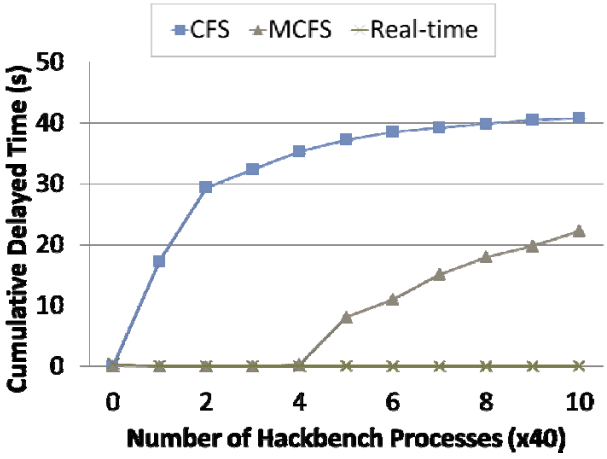

Figure 8. Cumulative buffer-writing delay of the music player given a varying number of Hackbench process groups.

Figure 8 shows the cumulative buffer writing delay during the music playback under the CFS, MCFS, and with the real-time support framework.

Under the CFS, buffer-writing delay occurs even with only one group of Hackbench processes running. This tendency is similar to that shown in the experiments with the alarm clock task. The buffer-writing delay increased steadily with increasing numbers of Hackbench process groups until reaching 40,000ms.

The MCFS significantly suppressed buffer-writing delay. The delay did not exceed 28,000ms in the worst case, approximately 30% better than the CFS. Also, noticeable delay occurred only beyond four concurrent Hackbench groups. With the real-time support framework, as expected, no significant buffer-writing delay occurred.

When there are a small number of Hackbench tasks, the scheduling delays of a prioritized task are usually short and frequent. On the contrary, when there are a large number of Hackbench tasks, the scheduling delays of a prioritized task are generally long and infrequent.

This tendency is illustrated by the differences between the slope patterns of Figure 7 and Figure 8. Even when buffer-writing delays occur, the music may play without noticeable QoS degradation if the delays are sufficiently short that they do not empty the buffer. Although there are a lot of buffer-writing delays with a small number of Hackbench process groups under the CFS, as Figure 8, these delays did not lead to jitters or suspension of music playback, as shown in Figure 7.

In order to analyze the performance overhead of the

proposed schemes, we measured the average completion time of the Hackbench tasks during the music player experiments. Figure 9 shows the average execution time normalized to the results under the CFS.

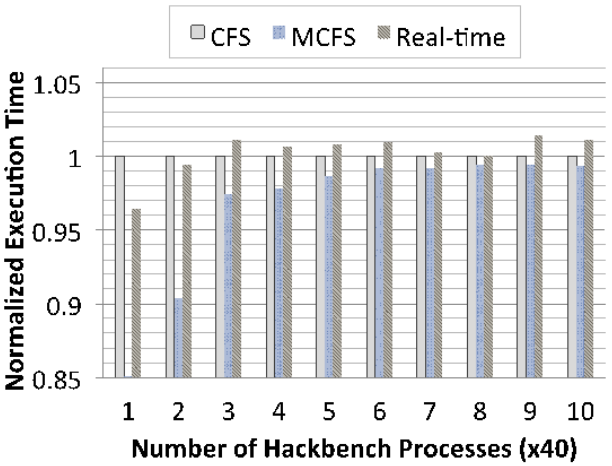

Figure 9. Normalized execution time of Hackbench benchmark with a concurrently running music player under different scheduling schemes.

The performance under the MCFS was better than that under the CFS because the MCFS suppress context switches. The real-time support framework also reduces the number of context switches given small numbers of Hackbench process groups by scheduling the music player preferentially to other tasks. With the large numbers of Hackbench process groups, Hackbench performance degraded slightly under the real-time framework because a relatively large portion of processor time was being devoted to the music player in comparison to the other two schemes.

However, in all cases, the proposed schemes reduced performance by less than 2% in comparison to the CFS.

V. Conclusion

Despite many benefits, the JVM architecture has a significant weakness: applications inside JVMs cannot fully utilize the kernel-level real-time support because the JVM layer hinders applications from informing the kernel of their timeliness requirements.

Our research improves the responsiveness of JVM-based smart phone applications by modifying the conventional scheduler so that it places a higher value on task priorities than on fairness. In addition to this, we proposed a cross-layer real-time support Java API library that delivers the timeliness requirements of applications to the kernel scheduler.

The evaluation showed that the modified kernel scheduler significantly reduced the response latency without requiring modification or rebuilding of existing applications. In addition, when applications were modified to use the proposed cross-layer real-time support framework, the scheduling latency of those applications was limited to few milliseconds even under heavy load.

However, the proposed schemes still cannot always provide real-time responsiveness because it does not automatically detect scheduling dependency relationships among applications and services, and, in addition, some system-related threads cannot be prioritized to the real-time class. We are conducting research to address these issues.

References

[1] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," presented at the Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012.

[2] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, "Exhausting battery statistics: understanding the energy demands on mobile handsets," presented at the Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds, 1851327, 2010.

[3] Y.-H. Lee, P. Chandrian, and B. Li, "Efficient Java Native Interface for Android Based Mobile Devices," presented at the Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications, 2011.

[4] J. J. Labrosse, MicroC/OS-II : the real-time kernel, 2 ed., CMP Books, 2002.

[5] J. A. Stankovic and R. Rajkumar, "Real-Time Operating Systems," Real-Time Systems, vol. 28, pp. 237-253, 2004.

[6] S. Oikawa and R. Rajkumar, "Portable RK : a portable resource kernel for guaranteed and enforced timing behavior," presented at the Proceedings of the IEEE Real-Time Technology and Applications Symposium, 1999.

[7] J. Ready, "VRTX : a real-time operating sytems for embedded microprocessor applications," IEEE Micro, vol. 6, pp. 8-17, 1986.

[8] C. Maia, L. Nogueria, and L. M. Pinbo, "Evaluating Android OS for Embedded Real-Time Systems," presented at the Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp. 63-70, 2010.

[9] Y. Woo, J. Cho, D. Lim , and E. Seo, "Cross-layer real-time support for JVM-based smartphone systems," presented at the Proceedings of the 2012 IEEE International Conference on Consumer Electronics, pp. 592-593, 2012.

[10] J. W. Muchow, Core J2ME Technology and MIDP, first ed., Prentice Hall PTR, 2001.

[11] G. Bollella and J. Gosling, "The real-time specification for Java," Computer, vol. 33, pp. 47-54, 2000.

[12] J. C. Pang, G. C. Shoja, and E. G. Manning, "Providing soft real-time quality of service guarantees for Java threads," Concurrency and Computation: Practice and Experience, vol. 15, pp. 521--538, 2003.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, vol. 39, pp. 1175-1185, 1990.

[14] W. v. Hagen, "Real-time and performance improvements in the 2.6 Linux kernel," Linux Journal, vol. 2005, 2005.

[15] S. Kleiman and J. Eykholt, "Interrupts as threads," ACM SIGOPS Operating Systems Review, vol. 29, pp. 21-26, 1995.

[16] E. Seo, J. Jeong, S. Park, J. Kim, and J. Lee, "Catching two rabbits: adaptive real-time support for embedded Linux," Software: Practice and Experience, vol. 39, pp. 531-550, 2009.

[17] J. Kay and P. Lauder, "A fair share scheduler," Communnications of the ACM, vol. 31, pp. 44--55, January 1988.

[18] C. S. Pabla, "Completely fair scheduler," Linux Journal, vol. 2009, 2009.

[19] D. Bovet and M. Cesati, Understanding the Linux Kernel, third ed., O'Reilly Media, 2000.

[20] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," presented at the Proceedings of the International Conference on Embedded Software, pp.39-48, 2011.

[21] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the Xen hypervisor," presented at the Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment, pp. 97-108, 2010.

[22] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," presented at the Proceedings of IEEE Annual Symposium on Foundations of Computer Science, Los Alamitos, CA, USA, 1978.

[23] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, "Design and implementation of a comprehensive real-time Java virtual machine," presented at the Proceedings of the 7th ACM and IEEE international conference on Embedded software, 1289967, 2007.

[24] C.-t. Man, P. Li, and Y. Li, "Study of Priority Inversion in Embedded Linux," presented at the Proceedings of the 1st International Conference on Innovative Computing, Information and Control, 2006.

[25] S. K. Card, T. P. Moran, and A. Newell, The Psychology of Human-Computer Interaction, Lawrence Erlbaum Associates, 1983.