Aciom: Application Characteristics-aware Disk and Network I/O Management on Android Platform

Hyosu Kim, Minsub Lee, Wookhyun Han, Kilho Lee, Insik Shin*
Dept. of Computer Science
KAIST, South Korea
{hskim, minsub, hanwook, khlee}@cps.kaist.ac.kr, insik.shin@cs.kaist.ac.kr

ABSTRACT

The last several years have seen a rapid increase in smart phone use. Android offers an open-source software platform on smart phones, that includes a Linux-based kernel, Java applications, and middleware. The Android middleware provides system libraries and services to facilitate the development of performance-sensitive or device-specific functionalities, such as screen display, multimedia, and web browsing. Android keeps track of which applications make use of which system services for some pre-defined functionalities, and which application is running in the foreground attracting the user's attention. Such information is valuable in capturing application characteristics and can be useful for resource management tailored to application requirements. However, the Linux-based Android kernel does not utilize such information for I/O resource management. This paper is the first work, to the best of our knowledge, to attempt to understand application characteristics through Android architecture and to incorporate those characteristics into disk and network I/O management. Our proposed approach, Aciom (Application Characteristics-aware I/O Management), requires no modification to applications and characterizes application I/O requests as time-sensitive, bursty, or plain, depending on which system services are involved and which application receives the user's focus. Aciom then provides differentiated I/O management services for different types of I/O requests, supporting minimum bandwidth reservations for time-sensitive requests and placing maximum bandwidth limits on bursty requests. We present the design of Aciom and a prototype implementation on Android. Our experimental results show that Aciom is quite effective in handling disk and network I/O requests in support of time-sensitive applications in the presence of bursty I/O requests.

*A corresponding author

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.4.3 [**Operating Systems**]: File Systems Management—*Access Methods*; D.4.4 [**Operating Systems**]: Communications Management—*Network Communication*

General Terms

Algorithms, Design, Performance

Keywords

Android Platform, Application Characteristics Awareness, I/O Management

1. INTRODUCTION

The last two decades have seen an explosive spread of mobile phones all over the world. Recently, a rapidly growing portion of mobile subscribers uses smart phones for more advanced computing ability. Smart phone users are projected to overtake feature phone users in the US at the end of this year [5]. Android [2] is introduced to deliver an open-source software platform tailored to mobile devices. Android is reported to be the world's best-selling smart phone platform [3] and projected to take 49% of the smart phone market in 2012 [4].

The Android platform provides a whole open-source software stack, from operating systems through middleware to applications. Android applications, written in Java, run on their own separate virtual machines over a Linux-based kernel. The Android middleware provides access to a set of native libraries for performance optimization and third-party libraries such as OpenGL and Webkit. In many mobile devices, including smart phones, the applications running in the foreground usually perform user-interactive tasks. The Android middleware keeps track of which application is running in the foreground and which ones are in the background, and the kernel makes use of this information in CPU scheduling and memory management to better support the foreground application's responsiveness. However, the Android platform does not yet incorporate such information into I/O management, although I/O management is quite critical to the performance of interactive embedded applications because embedded devices are generally subject to insufficient disk and network capacity.

For example, interactive applications, such as multimedia applications, demand time-predictable handling of their I/O requests. I/O-intensive applications, such as download managers, often make

bursty I/O requests, striving to consume all the spare bandwidth in the system. I/O requests can be of different priorities depending on the applications running in the system. The I/O managers of the existing Linux-based Android kernel are designed to maximize throughput by handling I/O requests in a fair manner, without paying attention to the characteristics of the I/O requests. This could easily lead to cases in which, for example, the I/O requests of a multimedia application experience longer latencies than their requirements and the application fails to deliver its intended quality-of-service (i.e., yielding screen delays and pauses).

Many studies [7, 8, 11, 18] have shown that operating systems can make better resource management decisions when the characteristics of resource requests are available to the operating systems. For instance, the Redline operating system [18] expects users or system administrators to come up with the resource requirement specifications of interactive applications. For example, the specification of an interactive multimedia application, *mplayer*, can indicate its CPU requirement of 5ms out of every 30ms period. Redline then makes use of this specification in determining the scheduling parameters of the application, including its disk request priority. However, such a specification requires manual efforts of expertise upon every different system environment. The mClock hypervisor I/O manager [7] introduces a scheduling algorithm that can accommodate the different characteristics of disk requests, assuming that each virtual machine is able to determine the specification parameters of its disk request requirements. However, this study [7] does not address how to determine such specification parameters appropriately.

The goal of our paper is to provide differentiated I/O management services for Android applications according to their characteristics. For example, prioritized or guaranteed I/O management services are provided for user-interactive applications such that those applications can produce prompt or delay-bounded responses to users. In view of this goal, we developed Aciom (Application Characteristics-aware Disk and Network I/O Management) on the Android platform. Aciom takes advantage of the Android architecture to understand the characteristics of applications dynamically. The Android middleware offers various system services to facilitate application development. For example, multimedia applications can make use of the Android media server system service to play a movie. Such information can be useful to figure out application characteristics. That is, when the media server service is subject to time-sensitive operations, any application that wants to use this media server service can be considered as time-sensitive as well. As such, Aciom characterizes Android applications as time-sensitive, bursty, or plain depending on which system services they use and on whether or not they run in the foreground. Aciom then provides different disk and network I/O management services for different types of applications. For a time-sensitive application, Aciom aims to guarantee a certain amount of I/O bandwidth reservation to these applications in order to provide low-latency I/O request handling. For a bursty application, it places a maximum limit on the I/O bandwidth allocation for the application, in order not to impose a negative effect on time-sensitive I/O requests. We developed an Aciom prototype on Android, and our experimental results show that Aciom is quite effective in supporting the performance requirement of a time-sensitive application in the presence of bursty I/O requests.

The rest of this paper is organized as follows. Section 2 introduces the overall architecture of the Android platform, and describes system services that the Android platform provides. Section 3 presents an overview of our proposed platform (Aciom). In Section 4 and Section 5, we describe the details of Aciom's differentiated I/O service mechanism for disks (Section 4) and networks (Section 5). Section 6 shows the evaluation results, and Section 7 presents related work. Finally, we conclude our paper in Section 8 with ideas for future work.

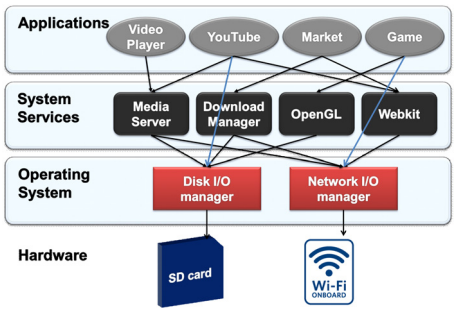

Figure 1: The overall architecture of Android platform

2. ANDROID PLATFORM

Android is an open-source, software platform for mobile devices that provides a Linux-based operating system, a set of Java applications for basic mobile device features, and middleware (cf. Figure 1). Each Java application runs within its own instance of the *Dalvik* virtual machine, and each instance runs in its own process with a unique Linux user identification in order to isolate applications within the platform. The middleware offers a set of Java and native libraries to facilitate application development. The Java libraries, called *application framework*, offer many kinds of services related to application semantics, such as management of the lifecycle of application components. They also enable the use and replacement of application components upon request and provide access to the native libraries through JNI (Java Native Interface). The native libraries are written in C/C++ and expose device-specific and/or performance-sensitive functionalities. We will now discuss the topics necessary to understand our I/O management scheme.

2.1 System Service

The Android middleware offers a set of Java and native libraries, called *system services*, to support a range of core system functionalities. As an example, the system services include *ActivityManager* to manage the lifecycle of application components and *LocationManager* to take care of location (e.g., GPS) updates.

A special process, called *service manager*, maintains a list of system services available in the platform. Applications make use of system services through the Android IPC mechanism, called *binder*. For example, when an application wants to playback a movie, it first requests permission from the service manager to use the media player service. Once the permission is granted by the service manager, the application is able to inform the media player service of the media type and the source location of the movie through the binder. The media player service, instead of the application,

Classification	Android System Service
Always bursty	Download service Storage service Package manager service
Always Time-sensitive	Media service Audio service Telephony service
Always plain	Layout inflater service Power management service
F/B dependent	Sqlite service Search service

Figure 2: Android system service characterization with I/O requests

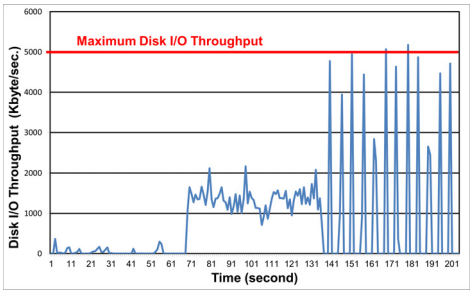

Figure 3: Examples of I/O request pattern

then loads the media file from the source location, decodes it according to its media type, and displays the file.

2.2 Foreground/Background Applications

In many mobile devices such as smart phones, the applications running in the foreground generally perform user-interactive tasks and the applications that do not need interaction with users run in the background. The Android middleware keeps track of which application is running in the foreground and which applications run in the background, and the kernel incorporates this information into the CPU scheduling and memory management. The kernel employs a CFS (completely fair scheduler) algorithm for CPU scheduling, and it gives a higher priority to the foreground application. The Android memory management performs application-based memory reclamation. When this system finds no free space in the memory, it kills one or more background applications to reclaim the memory from the application(s). As such, Android favors the foreground application in the CPU and memory management, however, it does not yet support the foreground application in disk and network I/O management.

3. ACIOM OVERVIEW

This section presents an overview of the Aciom (Application Characteristics-aware I/O Manager) system. The goal of Aciom is to provide differentiated I/O management services for Android applications according to their characteristics. For example, Aciom delivers prioritized or guaranteed I/O management services to user-interactive applications such that those applications can produce prompt or delay-bounded responses to users. Toward this goal, this section first classifies applications according to the characteristics of their I/O requests and discusses the limitations of the current Linux I/O management in support of application characteristics. It then outlines how Aciom can capture application characteristics in Android and how they can be incorporated into differentiated disk and network management services.

Application characterization with I/O requests. Applications can be classified as *time-sensitive* if they impose timing constraints on their I/O requests, or as *time-insensitive* otherwise. As an example, let us consider a multimedia application that plays a live streaming video at 30 FPS (frame-per-second). The kernel should be able to handle the incoming packets of this application with a low latency, say, a latency smaller than 33ms. Time-insensitive applications can be further characterized as *bursty* if they often make a burst of I/O requests, or *plain* if they do not. Bursty applications tend to consume all the bandwidth available in the system and often delay time-sensitive I/O requests. Hence, an I/O management scheme is required that can satisfy the timing constraints imposed by time-sensitive requests, even in the presence of bursty requests. Figure 2 shows how we categorize Android services.

Figure 3 shows three different patterns of application I/O requests. The first phase is measured from the web-kit service (plain), the second phase is from the media service (time-sensitive), and the third phase is from the download server (bursty).

There is another types of application characteristic, named F/B dependent (Foreground/Background dependent). From the user's viewpoint, applications running on mobile devices, such as smart phones, can have different characteristics according to the user attention. In other words, some applications can have two different types of characteristics depending on whether the application is user-interactive or not. For example, let us consider an internal database service, called SQLite service. Such a service can be considered as time-sensitive, if it provides database handling, which usually includes intensive I/O requests, for the foreground application. Or, it can be classified as bursty if no foreground application makes use of its database I/O requests. In the latter case, it could be considered as harmful to time-sensitive applications.

Limitations of the current Linux platform. The Linux I/O managers do not incorporate the characteristics of applications into policies that determine when and in what order I/O requests are serviced. Their default policies are designed to allocate I/O bandwidth in such a way that the system throughput is maximized, oblivious to the applications' requirements associated with disk and network requests.

This can be detrimental to especially time-sensitive applications. For example, consider a time-sensitive multimedia application making I/O requests at the same time as a download application is making a burst of I/O requests. In this case, the Linux I/O managers can handle a batch of requests coming from the download application ahead of the multimedia application's requests, imposing longer delays on the latter requests. This could lead to a significant degradation of quality of the multimedia service. The multimedia application can play a movie at a much lower FPS, producing unacceptable random screen delays and pauses.

As such, Linux does not provide applications with differentiated I/O management services. Therefore, it is inappropriate for embedded systems with insufficient resource capacity, and this requires a new application characteristics-aware I/O manager for embedded systems.

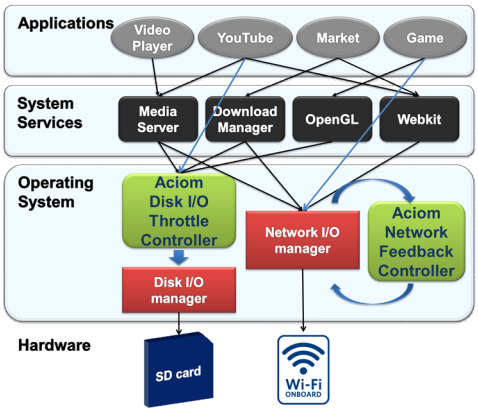

Figure 4: The overall architecture of Aciom platform

Application characteristics available in Android. Commodity applications do not provide their own characteristics, and this makes it difficult for operating systems to figure out their performance requirements. Profiling the behavior of applications can reveal their resource usage pattern but does not help much to identify whether applications are demanding delay-sensitive I/O requests. However, it is relatively easier to estimate the characteristics of commodity applications in Android.

The Android platform provides a set of system services to applications for performance-sensitive or device-specific functionalities. For example, there is a system service, called *download manager*, that handles long-running HTTP downloads. A client application can pass a URI to the download manager, asking to download the URI to a particular destination file. The download manager will conduct the download in the background, taking care of the HTTP interactions and retrying downloads over various failures such as disconnection. As such, the client application is not involved in any of the network operations, and the download manager takes care of all the networking processes for the client application. Each system service has its own characteristics on I/O operations. Since each system service is supposed to provide its own pre-defined functionalities, its characteristics can be known in advance. As an example, the download manager can make a burst of network and disk requests. When an application makes use of a certain system service, we can then infer the characteristics of the application from the system service's characteristics, even though the application does not announce its characteristics explicitly. For example, when a media player application makes use of the media server system service, the media player can be considered as a time-sensitive application.

Application characteristics-aware I/O manager. Aciom makes use of two schemes to characterize Android applications. The applications that make use of system services will be characterized as time-sensitive, bursty, or plain depending on which system services they use. For applications that do not use system services, Aciom uses the characteristics of consumer electronics in which the foreground application usually performs user-interactive tasks. That is, Aciom characterizes such applications as time-sensitive if they run in the foreground, or bursty if they are running in the background.

Aciom employs different I/O management services for different types of applications. For a time-sensitive application, Aciom reserves a certain amount of I/O bandwidth such that the application's I/O requests can be handled in time. For a bursty application, Aciom limits the bandwidth allocation in order not to interfere with the reserved bandwidth allocations of the time-sensitive applications. If no time-sensitive application is running, then Aciom does not place any bandwidth limit on the bursty applications. For a plain application, Aciom does not perform any special treatment but takes care of its I/O request in the same way as does the Linux I/O manager.

Aciom aims to allocate as much bandwidth as necessary to time-sensitive applications so that they can handle their requests properly. However, information is not available on exactly how much bandwidth is required to handle requests that are subject to application requirements. Aciom estimates the current required bandwidth of a time-sensitive application based on how much bandwidth was requested in previous situations. In fact, such an estimate can be inaccurate when predicting the current or near-future bandwidth requirements, in particular when applications start showing different request behaviors. Hence, Aciom maintains reserved extra bandwidth, called *headroom*, to give a margin to time-sensitive applications. When time-sensitive applications require more bandwidth than was estimated, they can utilize the headroom to resolve the extra requirement.

Even though the Aciom disk and network managers share the same design principles, they employ different techniques to estimate the bandwidth requirements of time-sensitive applications, to provide reserved bandwidths to time-sensitive applications, and to limit the bandwidth allocations to bursty applications. This is mainly because the disk and network devices exhibit different characteristics. For example, applications generally make disk requests for a large chunk of data in a relatively long interval, while network requests are made for a small chunk of data within a shorter request interval. Furthermore, Android applications generally receive packets rather than send them out. This means that network requests will generally be initiated outside of the system, while disk requests will be generated inside the system. Such differences lead Aciom to employ different management mechanisms. Details of this are explained in Section 4 for disks and in Section 5 for networks.

4. DISK I/O MANAGEMENT

The Linux I/O manager does not incorporate the characteristics of applications into its disk I/O scheduling policy. Its default policy is designed to allocate disk I/O bandwidth in such a way that the system throughput is maximized, sometimes to the detriment of application demands. This section describes the proposed disk I/O manager, which dynamically captures an application's requirements for disk requests and allows the requests to meet their requirements based on the information provided by the Android architecture.

4.1 Linux Disk I/O Management

The Linux-based Android block device manager uses the *Complete Fairness Queuing (CFQ)* elevator algorithm for disk I/O management. The main purpose of CFQ is to ensure a fair allocation of the disk bandwidth among all the applications that initiate disk I/O requests as well as to maximize the system-wide throughput.

To achieve a fair allocation of the disk bandwidth, CFQ uses a two-level queuing approach with 64 internal I/O queues and a

single I/O dispatch queue. When an application issues a disk I/O request, the request is inserted into one of the internal I/O queues. CFQ makes use of a hash function that converts the process identifier of the application into the index of an I/O queue. Therefore, requests coming from the same application are always inserted into the same I/O queue. Requests in an I/O queue are sorted according to their initial sector numbers. The CFQ elevator then scans the I/O queues in a round-robin fashion, finds the first non-empty queue, and moves a batch of requests from the selected I/O queue to the tail of the dispatch queue. The elevator repeats a sequence of those actions in a work-conserving mode, and this way, the CFQ elevator algorithm guarantees the maximization of the system-wide throughput.

In order to avoid *disk request starvation*, which occurs when the elevator policy ignores a request for a very long time, each request is assigned a *deadline*. Every read request is given a default deadline of 125 ms, while the default deadline for write requests is 250 ms. The elevator keeps scanning the sorted I/O queues sequentially, unless a request deadline expires. If so, the elevator moves that expired request to the tail of the dispatch queue.

However, this CFQ policy is not aware of the applications' requirements, which are imposed to disk requests. This can cause time-sensitive requests to wait longer in the I/O queues and/or in the dispatch queue, resulting in a violation of their timing requirements.

4.2 Aciom Disk I/O Management

The main purpose of the Aciom disk I/O manager is to provide differentiated services to different types of applications.

Time-sensitive requests. A time-sensitive application aims to guarantee a certain amount of disk bandwidth such that its disk I/O requests can be handled in time. Even though the Android architecture helps to successfully classify disk requests as time-sensitive, it does not reveal how much bandwidth should be allocated to those time-sensitive requests to avoid any performance loss for their application. Hence, Aciom tries to figure out directly how much bandwidth each time-sensitive application will require. In order to estimate this requirement, Aciom keeps track of how often the application makes time-sensitive disk requests and how large the data size of each request is. Let R_i^k denote the k-th disk request from a time-sensitive application i, A_i^k denote its arrival time, and S_i^k denote its data size, respectively. At this point, Aciom first computes the projected period (denoted as $\bar{T_i^k}$) as a weighted combination of the previous value and a newly available data, which is the arrival time difference between the two consecutive requests, $R_i^{(k-1)}$ and R_i^k, i.e.,

$$\bar{T_i^k} = \alpha \cdot \bar{T_i^{k-1}} + (1-\alpha) \cdot (A_i^k - A_i^{k-1}),$$

where $\bar{T_i^0}$ is defined as a default period for the time-sensitive application i, $A_i^0 = A_i^1 - \bar{T_i^0}$, and the parameter α controls how much we rely on the history in determining the project period. The value can be chosen in a range $[0, 1]$. Let $\bar{S_i^k}$ denote the average data size of a request R_i^k, which is calculated as follows:

$$\bar{S_i^k} = \beta \cdot \bar{S_i^{k-1}} + (1-\beta) \cdot (S_i^k),$$

where $\bar{S_i^0} = S_i^1$ and S_i^1 is defined as the first request's size of an application i. The parameter β works in a way similar to that of the parameters α, and $\beta \in [0, 1]$. Finally, let B_i denote how much bandwidth an application i wants to consume on average for its disk operations, and this value can be computed as follows:

$$B_i = \bar{T_i^k} \times \bar{S_i^k},$$

when the application i makes the k-th disk request.

Now, we have an estimation of the bandwidth requirement (B_i) for a time-sensitive application i. We then consider how to allocate the bandwidth of B_i to the application i. Aciom aims to allocate such bandwidth by taking care of a request of size $\bar{S_i^k}$ within every interval of length $\bar{T_i^k}$. To achieve this, Aciom takes advantage of a request deadline. It is assigned such that a single request should be served within an interval of length $\bar{T_i^k}$. Let D_i^k denote the deadline of a request R_i^k, and it is determined as

$$D_i^k = D_i^{k-1} + \bar{T_i^k},$$

where $D_i^0 = A_i^1$. Since the k-th request of size $\bar{S_i^k}$ will be handled before its deadline D_i^k, Aciom can provide disk I/O bandwidth of at least B_i to the time-sensitive application i.

Bursty requests. Bursty applications often seek to consume all the available disk bandwidth in the system and can have a negative influence on time-sensitive disk requests. Hence, Aciom places maximum limits on the disk bandwidth allocations to bursty applications such that the reserved bandwidth for time-sensitive applications should be provided independent of bursty requests.

Aciom maintains an additional single internal queue, called a *pre-release* queue, for bursty disk requests, in addition to the existing I/O queues and the dispatch queue of the Linux disk manager. Every bursty disk request is initially placed into the pre-release queue and is assigned an absolute release time. Each bursty request will remain in the pre-release queue until its release time expires. Upon expiry, the request is moved to one of the I/O queues. Aciom adaptively determines the release times of the bursty requests such that they cannot consume bandwidth greater than their maximum limit. In order to find the maximum possible limit of a bursty application, Aciom first calculates the total remaining bandwidth that does not violate the total reserved bandwidth for time-sensitive applications. The total remainder bandwidth will be then equally distributed to individual bursty applications. Let $RB_i(t)$ denote the remainder bandwidth of a bursty application i at time t, and it is calculated as

$$RB_i(t) = \frac{C_{max} - \left(\sum_{j \in TA} B_j(t) + H(t)\right)}{|BA|},$$

where C_{max} is the maximum disk capacity, TA and BA are the sets of time-sensitive applications and bursty applications, respectively, $B_i(t)$ is a projected average bandwidth of a time-sensitive application i at time t, and $H(t)$ is the headroom size at time t. For the k-th request of a busty application i, let V_i^k denote the actual size and $\bar{V_i^k}$ denote the projected average size. Then, $\bar{V_i^k}$ is computed as

$$\bar{V_i^k} = \gamma \cdot \bar{V_i^{k-1}} + (1-\gamma) \cdot (V_i^k),$$

Then, we can compute how many bursty requests can be handled per second. When a bursty application i makes its k-th request, let N_i^k denote the number of bursty requests to be handled per second at that time. Then, N_i^k is computed as

$$N_i^k = RB(t)/\bar{V_i^k}.$$

Then, let L_i^k denote the release time of the k-th request of the bursty

application i, which is defined as

$$L_i^k = L_i^{k-1} + 1/N_i^k,$$

where $L_i^0 = A_i^1 - T_i^1$.

5. NETWORK I/O MANAGEMENT

The current Linux network management subsystem is separate from applications. Its management scheme is designed to maximize the system throughput, which is sometimes inconsistent with application requirements. In order to bridge from network management to application requirements, Aciom extends the Linux network subsystem to make corresponding connections between sockets and applications. Even though the existing Linux socket is able to find the corresponding application, its method takes time since it goes through a number of the data structure. We maintain the value in the socket so as to represent which socket corresponds to which process, in order to reduce overhead.

5.1 Linux Network Management

The Linux network subsystem makes use of two queues, the *input* and *output* queues, to hold incoming and outgoing packets, respectively. The subsystem takes care of those packets by default in a First-Come-First-Server manner. Linux does not only support FCFS, but also supports some other policies to determine the order in which packets are served, for example, to improve their QoS. However, such policies make use of networking-related parameters, such as IP addresses and port number, which do not directly disclose application characteristics. Therefore, the Linux network subsystem may allocate network resources to applications in a way that does not correspond to the user's anticipation. For example, when a user makes a video call while downloading files in the background, any networking management scheme that provides the video call application with a lower bandwidth than its requirement, will lead to a degraded QoS for the video call.

5.2 ACIOM Network Management

The Aciom network manager primarily aims to handle network I/O requests according to application characteristics. Similar to the role of the Aciom disk manager does, the network manager also tries to secure the required network bandwidth for time-sensitive applications at the expense of limiting the bandwidth allocation for bursty applications. However, the Aciom network manager faces several different challenges in achieving this goal, compared to the case of disk management, because there are prominent differences between network and disk I/O requests.

For example, disk I/O requests are synchronous, while network I/O requests are asynchronous, in a sense that disk I/O requests are initiated by the applications inside the system while incoming packets are initiated by applications outside the system. This introduces a big difference into the process of I/O management. Disk managers are able to directly throttle disk I/O requests through local queue management in order to limit disk bandwidth allocations, however, network managers are not capable of simply limiting the network bandwidth allocations for incoming packets by local queue management. In addition, disk I/O requests have a large range of data size, from a single sector size of 512 bytes to any arbitrary large size of up to 2048 KB, while network I/O requests are of a relatively smaller size up to 1024B (i.e., up to the maximum single network frame size). Thus, network I/O management is subject to having to handle a much larger number of requests in a shorter interval, compared to disk I/O management. These problems make it infeasible to directly apply the Aciom disk management techniques to network management.

Unlike the Aciom disk manager, which allocates bandwidth via local queue management, the Aciom network manager employs flow control for bandwidth allocation. Aciom makes use of flow control of the TCP protocol; each TCP connection is called a *flow*. TCP flow control service is designed to resolve speed mismatches between senders and receivers [9]. A fast sender can overflow the network device buffer of a slow receiver, and flow control can be performed to slow down the sender. A TCP packet includes a parameter, called *window size*, that enforces flow control. The receiver determines the value of window size and sends it to the sender through its ACK packet. The window size gives the sender an idea of how much free buffer space is available at the receiver. A proper control of window size allows the sender to properly control the transmission speed. Note that each individual TCP flow has its own window size parameter, and a change to the window size of a flow does not interfere directly with another flow. Employing the existing TCP flow control as it is, we also note that Aciom does not impose any extra data transmission over the network.

However, changing window size to manipulate bandwidth is very difficult to implement in the controller. This is because the controller should contain an analytical network model to derive the appropriate window size. There have been considerable studies of the analytical network model [12, 10]. However, most of these are improper because this model is complex when targeting device, because devices consist of complex floating point calculations, and so a controller needs to be implemented as a user-level application.

To resolve this problem, we use a simple proportional controller, called a P controller.

$$W_{out}(t+1) = W_{out}(t) + K_p(B_d(t) - B_c(t)$$

where $W_{out}(t)$ is the window size at time t in order to control bandwidth, $B_d(t)$ is the desired bandwidth at time t, $B_c(t)$ is the current bandwidth at time t, and K_p is the proportional gain.

The P controller periodically calculates the window size. Since we make the framework at the kernel level, where floating point calculations are limited, a simple P controller can be easily implemented with very low cost of overhead.

In Aciom network I/O management, the window size of the time-sensitive application's flow is never modified in our framework. It always reserves as much bandwidth as possible within the given network bandwidth. Aciom tries to change the window size of bursty application's. With the P controller, the window size of a bursty application can be calculated as:

$$W_{out}(t+1) = W_{out}(t) +$$

$$K_p(C_{max} - (\sum_{j \in TA} B_j(t) + H(t)) - \sum_{j \in BA} B_j(t))$$

where C_{max} is the maximum network bandwidth.

As mentioned in section 3, network devices usually receive packets within a short time interval. Hence, bandwidth measurement using methods in the disk I/O could incur large overhead in the network I/O. In the network I/O, we use two step phases to measure the bandwidth of each network flow. First, we accumulate the size of the network packet whenever the Linux network stack finishes decoding the packets. Since each Linux socket corresponds

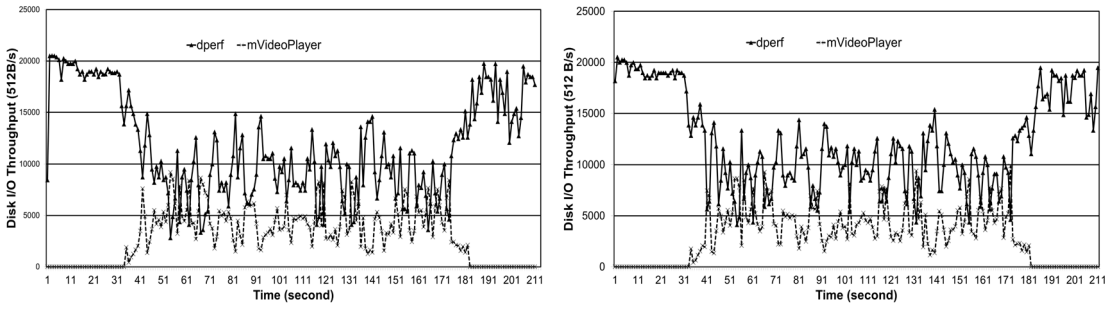

Figure 5: The disk I/O bandwidth comparison

to each network flow, the accumulated value is held by the socket. Second, a kernel thread that is activated periodically(In our experiment every 500ms) figures out the sum of the time-sensitive applications' cumulated receive packet size and also that of the bursty applications'. Finally, the kernel thread calculates the window size of bursty applications with the controller.

Compared to the disk I/O management, the network I/O management does not provide straight bandwidth partitioning. Since it is based on the P controller, network management tries to divide bandwidth through given rules.

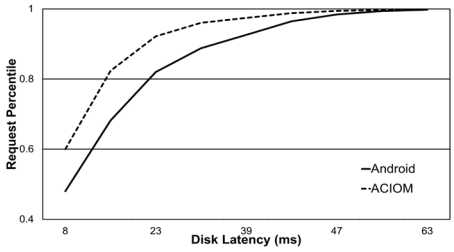

Figure 6: The disk I/O latency comparison

6. EXPERIMENTAL EVALUATION

We have built a prototype of Aciom based on the Android 2.2 platform and carried out experiments on the disk and network I/O managements both on Android and on Aciom. This section describes our prototype implementation and experiment environments and discusses experimental results. The main purpose of our experiments is to examine the following important questions: (1) Why is Aciom needed on Android? (2) How effective is Aciom in supporting the performance requirements of time-sensitive applications, in the presence of bursty applications.

6.1 Prototype Implementation

We implemented Aciom by modifying the block device scheduling layer and network packet handling layer of a Linux kernel (version 2.6.35) in Android 2.2. The changes required were small: the overall implementation took roughly 450 lines of C code to modify an existing I/O management layer and implementing a new kernel thread to control network flows. Our resulting I/O manager is lightweight, which is important because an embedded device has low computing power and the I/O path has a critical impact on the overall system performance. We constructed our prototype on a $Blaze^{TM}$ mobile development platform. $Blaze^{TM}$ is equipped with a Texas Instruments' OMAP4430 dual-core processor, 1GB of RAM, 8GB flash memory storage, and 802.11n Wi-Fi wireless connectivity.

6.2 Disk Management

This subsection presents the experimental environment and results for disk I/O management. We performed disk experiment to first show the need to incorporate application characteristics into the disk I/O management by demonstrating that the Linux-based Android kernel does not distinguish disk I/O requests coming from different applications, which results in the poor performance of time-sensitive applications in the presence of bursty disk requests. Our disk experiment was carried out to examine whether Aciom can successfully distinguish time-sensitive and bursty disk requests and to determine how well it manages the effect of bursty requests on time-sensitive applications.

6.2.1 Experimental Environment

Our disk experiment was performed with two Android applications: *mVideoPlayer* and *dperf*. The *mVideoPlayer* application is a free video player that can be downloaded from the Android market [1], and the *dperf* application is a disk I/O-intensive, custom application that we developed for our experiment. During our experiment, *mVideoPlayer* ran in the foreground and played a movie by using the *media server*, which is one of the typical time-sensitive Android system services. Specifically, it played the movie, "Toy Story 3" from an MPEG-4 format media file of 316 MB size. The *mVideoPlayer* application played the movie at a target of 30 FPS for 141 seconds with a resolution of 1920 x 1080. On the other hand, *dperf* ran in the background, making random I/O requests (75% reads) of 128 Kbytes every 75 ms. Aciom automatically classified the *mVideoPlayer* as time-sensitive because it made use of the time-sensitive media server, and *dperf* as bursty because it did not use any system service but ran in the background.

Five parameters were configured for the Aciom disk management: the maximum disk I/O bandwidth (C_{max}), the default period of the time-intensive application ($\bar{T^1}$), the period weight parameter (α), the size weight parameter (β), and the size of headroom ($H(t)$). C_{max} was obtained from the hardware specification

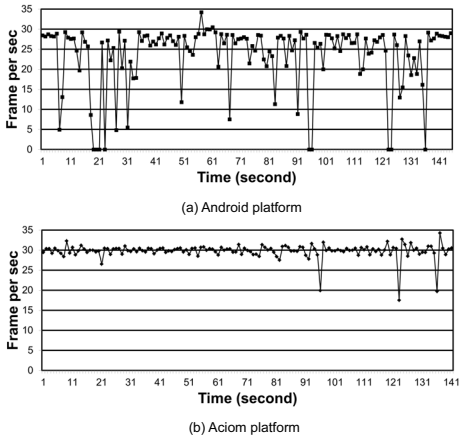

Figure 7: Frames per second of a media server

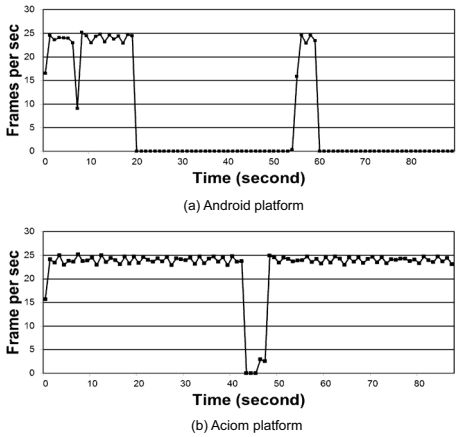

Figure 8: Frames per second of a media server

(10MB/s). $\bar{T^1}$ was set at 125ms as a default read request deadline for Linux. Both of the weight parameters α and β were assigned to 0.95 in order to better account for previous history more. $H(t)$ was also set at $0.1 \times C_{max}$. We would like to note that optimally configuring the parameters of α, $beta$, and $H(t)$ is important for system performance. However, a detailed discussion of how to find these optimal values dynamically is out of the scope of the paper.

6.2.2 Experimental Results

We ran the two applications both on Android 2.2 and on our Aciom prototype. Figure 5 compares the performance of Android and Aciom disk managements in terms of the disk throughput of the two applications. While *dperf* ran all the time during the 220 seconds of the experiment, *mVideoPlayer* ran for only around 150 seconds, from 32 seconds after the beginning of the experiment. The figure shows that *dperf* consumes the bandwidth of roughly 10 MB within the intervals of [0,32] and [181, 211]. This shows that *dperf* can consume a bandwidth of 9.6 MB as desired within the two intervals of [0, 32] and [181, 211] both on Android and on Aciom, while *mVideoPlayer* makes no disk request within those intervals. This indicates that when no time-sensitive requests are made, Aciom does not place any maximum limit on the disk bandwidth allocation for bursty disk requests. However, *dperf* has to compete with *mVideoPlayer* for disk bandwidth when the latter application starts making disk requests from 32 second to 181 second. During the interval [32, 181], Android and Aciom show different behavior in allocating disk bandwidth to the two applications. Figure 5 shows that both Android and Aciom allocate bandwidth to the two applications in a similar fashion. This shows that even though Aciom makes use of a headroom reservation to maintain the bandwidth margin for time-sensitive applications, it does not decrease the system-wide throughput. In fact, time-sensitive applications require low-latency disk I/O management. That is, even if *mVideoPlayer* receives a comparable disk bandwidth on Android and on Aciom, we need to look at the latency of its disk requests to see whether it is receiving proper disk I/O management service.

Figure 6 shows the distribution of the latency of disk I/O requests coming from *mVideoPlayer* both on Android and on Aciom. The figure indicates that a majority of time-sensitive disk requests experienced a much smaller latency on Aciom than on Android. For example, 80% of *mVideoPlayer*'s disk requests were handled in 12 ms on Aciom, but required 21 ms on Android. Also, Aciom took care of 87% of the requests in 20 ms, while Android performed only 75% in 20 ms.

Disk latency is critical to the performance of many time-sensitive applications. Figure 7 shows the performance of *mVideoPlayer* in terms of how many frames per second (FPS) it displayed. The video clip in the experiment was designed to be displayed at 30 FPS. However, *mVideoPlayer* could not meet the required FPS many times on Android, experiencing high fluctuations. On the other hand, *mVideoPlayer* was able to maintain the required 30 FPS much more steadily on Aciom.

6.3 Network Management

This subsection describes our experimental environment and results for network I/O management.

6.3.1 Experimental Environment

This experiment includes two applications: *YouTube*and *iPerf*. *YouTube* and *iPerf* are both third-party Android applications that can be downloaded from the Android market [1]. *YouTube* plays a video clip via network streaming with RTSP (Real-Time Streaming Protocol, and *iPerf* is a network-intensive application that is designed for performance measurement of network flows. In our network experiment, *YouTube* played a streaming video at 25 FPS from the YouTube website, making use of the Android media server for downloading, decoding, and displaying of the streaming data. *iPerf* received a bunch of network packet from a remote challenger, who maintained the network packet handling results for performance measurement. During the experiment, *YouTube* ran in the foreground all the time, and *iPerf* ran in the background and started receiving packets from 10 seconds after the beginning of the experiment. This method allowed *YouTube* to receive some streaming data and to keep them into a buffer.

Four parameters were configured for the Aciom network I/O management: the maximum network I/O bandwidth (C_{max}) was set at 2.4Mbps, the size of headroom ($H(t)$) was set at 200Kbps, the proportional gain (K_p) was assigned to 0.5, and the period of the network kernel thread was set at 500ms.

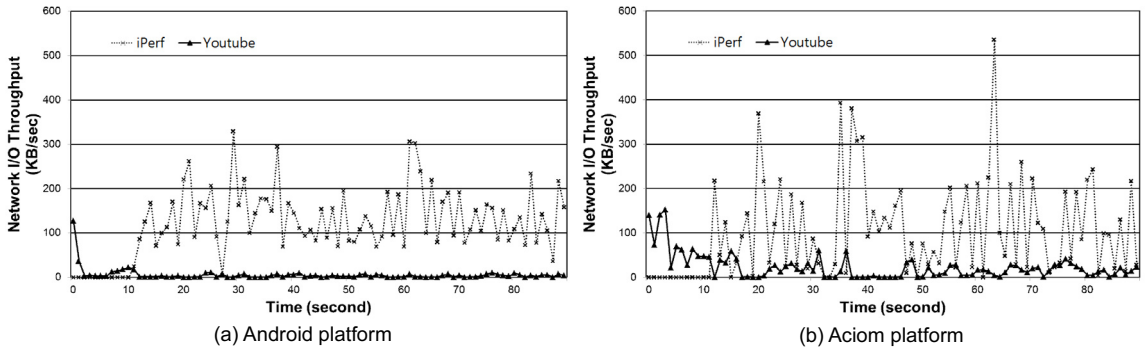

Figure 9: The network I/O bandwidth comparison

6.3.2 Experimental Result

Our network experiment was designed to show how effectively the Aciom network manager can take care of network flows for the purpose of supporting time-sensitive applications, while bursty network flows strive to consume all the available network bandwidth.

Figure 9 shows the network throughput of each flow on Android and on Aciom. When *iPerf* starts to consume network bandwidth from the 10th second, *YouTube*, as can be seen in Figure 9(a), received hardly any network bandwidth allocation on Android, on the other hand, it can be seen in Figure 9(b) that *YouTube* is given some network bandwidth allocation in Aciom. This is because the Aciom network manager favors time-sensitive applications by limiting the bandwidth allocations for bursty applications. It reduces the window size of *iPerf*'s TCP flow in order to slow down its sender. Therefore, the total bandwidth consumed by *iPerf* is 140 Kbps on Android, while it is 121 Kbps on Aciom.

We next examined how such bandwidth allocations affect the video quality of *YouTube*. Figure 8 shows how many frames per second *YouTube* displayed when running on Android and on Aciom. Figure 8(a) shows that *YouTube* experienced serious problems during a considerable amount of time in the experiment. It did not proceed at all (i.e., screen pause) for 64 seconds out of the 90 seconds of the experiment. On the other hand, it can be seen in Figure 8(b) that *YouTube* was able to display the movie in a much more stable manner on Aciom. The application mostly displayed the movie at $23 \sim 25$ FPS and experienced a degradation for only $2 \sim 3$ seconds. This indicates that Aciom is quite effective in supporting the performance requirement of time-sensitive applications through network I/O management.

7. RELATED WORK

Operating systems often take specifications from time-sensitive applications to determine the quantity of resources those applications will demand. In Rialto [8], applications shown to make use of system calls to inform the resource manager of their own estimate of the quantity of resource required over any period of time. In Redline [18], the resource requirement specifications of applications are assumed to be written by users or system administrators and passed to the resource manager through files. Rialto and Redline then make use of those specifications to reach scheduling decisions. SMART [11] also takes resource requirements from multimedia applications through system calls, and it notifies those applications as to whether their requests can be satisfied or not. This allows multimedia applications to take proper actions upon notification; they may degrade their quality-of-service accordingly. Those approaches basically assume that the specification of resource requirements will be provided explicitly, while our approach does not employ such an assumption and requires no modification to applications at all.

Several real-time scheduling algorithms [15, 6, 17] are proposed to schedule disk I/O requests taking their timing constraints into account. The SCAN-EDF algorithm [15] schedules disk I/O requests in an increasing order of their application deadlines, breaking ties according to the SCAN algorithm. The WRR-SCAN algorithm [17] aims to maximize disk throughput subject to the timing constraints of the disk requests. The algorithm provides disk bandwidth reservation to each time-sensitive applications according to its timing constraints, and it allocates the remainder of bandwidth to non-real-time applications. The above approaches were found to work well when the system uses a disk as its secondary storage. However, the above approaches do not fit in embedded systems, since most embedded devices use flash memory as its secondary storage.

A few studies [14, 13] have been performed on flash storage-aware I/O buffer scheduling. The CFLRU algorithm [14] takes into consideration a flash memory property that has different costs between read and write: write operations are more expensive. CFLRU uses this property to reduce the total cost. To reduce write operations, the algorithm holds dirty requests in the buffer cache as long as possible until the CFLRU starts to suffer system performance. [13] extends the CFLRU algorithm to improve efficiency by partitioning the buffer cache into several regions according to different operation costs. Even though this method considers flash memory properties well, it only aims to improve overall I/O efficiency, and it does not consider the quality of time-sensitive applications.

The mClock I/O manager [7] is introduced for hypervisor disk I/O scheduling. Assuming that each virtual machine (VM) specifies its own disk bandwidth requirement as well as its minimum and maximum requirements, the mClock hypervisor I/O manager aims to allocate disk bandwidth to VMs according to their requirement specifications. Even though the mClock I/O manager is most closely related to the Aciom disk I/O manager, there are some difference between them. While it is valid for mClock to assume that a time-sensitive VM would not require more bandwidth than its max-

imum requirement, such an assumption does not hold for Aciom, and this entails techniques to estimate the required bandwidth of time-sensitive requests and to recover when the required bandwidth is underestimated.

PSM-throttling [16] is proposed in order to reduce network module energy consumption in mobile devices. These devices generate a network module on/off duty-cycle, which periodically changes the network module's state between on and off, so that mobile devices are able to save energy when the devices are in the off state. These devices employ a zero-fixed window size packet to make senders transmit their packets only when devices are activated. [11] also makes use of zero window size to notify senders that the buffer of the receiver side is full. However, instead of immediately advertising a non-zero window when application starts to read data out of the receive buffer, this system waits to advertise the non-zero window size until its window has considerably increased, in order to avoid a silly window syndrome. This method configures the buffer occupancy threshold when a non-zero window is advertised (immediately following a zero window) as a tunable parameter to adapt to various network conditions.

There have been several studies of the analytical network model [12, 10]. These have aimed to develop an analytical characterization of the throughput as a function. The parameters of the model are loss rate, RTT(round trip time), and maximum window size. This model considers TCP congestion control in order to establish an accurate model. However, Aciom does not employ an analytical network model to calculate proper window size, since the model still has difficulty implementing in the kernel.

8. CONCLUSION

To the best of our knowledge, this paper presents the first attempt to understand application characteristics through Android architecture and to incorporate those characteristics into disk and network I/O management. Our prototype implementation shows that our proposed approach can support time-sensitive applications, allowing them to meet their performance requirements even when running with bursty I/O applications.

Our future work will include extending the proposed approach to other aspects of resource management, such as power management. In addition, in this paper, we consider only application characteristics. However, context (i.e., location, time) can also affect an application's behavior, and this can be important information in resource management. We plan to develop a context-aware system of resource management in order to better support application performance requirements.

9. ACKNOWLEDGEMENT

This work was supported in part by the IT R&D Program of MKE/KEIT [2011-KI002090, Development of Technology Base for Trustworthy Computing], Basic Research Laboratory (BRL) Program (2009-0086964), Basic Science Research Program (2011-0005541), and the Personal Plug&Play DigiCar Research Center (NCRC, 2011-0018245) through the National Research Foundation of Korea (NRF) funded by the Korea Government (MEST), and KAIST-Microsoft Research Collaboration Center.

10. REFERENCES

[1] https://market.android.com.

[2] http://www.android.com.

[3] http://www.canalys.com/pr/2011/r2011013.html.

[4] http://www.engadget.com/2011/04/07/gartner-android-grabbing-over-38-percent-of-smartphone-market-i/.

[5] http://www.mobileburn.com/news.jsp?id=9125.

[6] R.-I. Chang, W.-K. Shih, and R.-C. Chang. Deadline-modification-scan with maximum-scannable-groups for multimedia real-time disk scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium(RTSS)*, 1998.

[7] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] M. B. Jones, D. L. McCulley, A. Forin, P. J. Leach, D. Rosu, and D. L. Roberts. An overview of the rialto real-time architecture. In *Proceedings of ACM SIGOPS European Workshop*, 1996.

[9] J. F. Kurose and K. W. Ross. *Computer Networking third edition*. Addison Wesley, 2005.

[10] V. Misra, W.-B. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of tcp-windowsize behavior. In *Technical Report ECE-TR-CCS-99-10-01, Performance*, 1999.

[11] J. Nieh and M. S. Lam. The design, implementation and evaluation of smart: a scheduler for multimedia applications. In *Proceedings of ACM Symposium on Operating Systems Principles*, 1997.

[12] J. Padhye, V. Firoiu, and D. T. J. Kurose. Modeling tcp throughput: a simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM*, 1998.

[13] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn. A cost-aware page replacement algorithm for nand flash based mobile embedded systems. In *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT)*, 2009.

[14] S.-Y. Park, J.-U. K. D. Jung, J.-S. Kim, and J. Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[15] A. Reddy, J. Wyllie, and K.B.R.Wijayaratne. Disk scheduling in a multimedia i/o system. In *Proceedings of the first ACM international conference on Multimedia*, 1993.

[16] E. Tan, S. C. Lei Guo, and X. Zhang. Psm-throttling: Minimizing energy consumption for bulk data communications in wlans. In *Proceedings of IEEE International Conference on Network Protocols*, 2007.

[17] C.-H. Tsai, E. T.-H. Chu, and T.-Y. Huang. Wrr-scan: a rate-based real-time disk-scheduling algorithm. In *Proceedings of the 4th ACM international conference on Embedded software(EMSOFT)*, 2004.

[18] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2008.