2017 IEEE International Parallel and Distributed Processing Symposium

Container-Based Cloud Platform for Mobile Computation Offloading

Song Wu*, Chao Niu*, Jia Rao†, Hai Jin* and Xiaohai Dai*
* Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology Wuhan, 430074, China
Email: {wusong, chaoniu, hjin, seafooler}@hust.edu.cn
† The University of Texas at Arlington, USA Email: jia.rao@uta.edu

***Abstract*—With the explosive growth of smartphones and cloud computing, mobile cloud, which leverages cloud resource to boost the performance of mobile applications, becomes attractive. Many efforts have been made to improve the performance and reduce energy consumption of mobile devices by offloading computational codes to the cloud. However, the offloading cost caused by the cloud platform has been ignored for many years. In this paper, we propose *Rattrap*, a lightweight cloud platform which improves the offloading performance from cloud side. To achieve such goals, we analyze the characteristics of typical offloading workloads and design our platform solution accordingly. Rattrap develops a new runtime environment, Cloud Android Container, for mobile computation offloading, replacing heavyweight virtual machines (VMs). Our design exploits the idea of running operating systems with differential kernel features inside containers with driver extensions, which partially breaks the limitation of OS-level virtualization. With proposed resource sharing and code cache mechanism, Rattrap fundamentally improves the offloading performance. Our evaluation shows that Rattrap not only reduces the startup time of runtime environments and shows an average speedup of 16x, but also saves a large amount of system resources such as 75% memory footprint and at least 79% disk capacity. Moreover, Rattrap improves offloading response by as high as 63% over the cloud platform based on VM, and thus saving the battery life.**

I. Introduction

Mobile cloud computing, which offloads computation on mobile devices to a cloud platform, can enable execution of computationally intensive applications on mobile devices with enhanced user experience.The challenges lie in how to seamlessly integrate mobile runtime environments into cloud platforms and how to perform mobile computation offloading in a cost-effective manner. Existing work mainly focused on the design of code offloading frameworks and addressed the issues of code partitioning, offloading decision and methodology. The design of a cloud platform to support mobile computation offloading has largely been ignored. Previous studies assumed that computation from the cloud is ubiquitous and always ready to use.

Many practical issues arise when building a real mobile cloud. The cloud should provide on-demand execution environments for mobile computation codes, e.g., the Android mobile OS. Many offloading studies [1–4] used virtual machines (VMs) to host mobile OS in the cloud. Hardware virtualization allows different guest OSes to co-exist in the cloud platform. The main drawbacks of using VMs in mobile cloud are the long VM startup time and high virtualization overhead. Interactivity and mobility are the keys to the success of mobile cloud computing, but heavyweight VM solutions cannot meet these requirements [5, 6]. Though pre-starting VMs can reduce the VM startup time, it would inevitably incur high resource cost because the number of offloading requests is probably large [4].

OS-level virtualization, a.k.a, container-based virtualization [7–10] has recently attracted much attention due to its near-native performance and low virtualization overhead. Building a mobile cloud using container-based virtualization is a promising idea, but presents several challenges. First, the container must run the same OS as the host. Much effort is needed to customize a container to support mobile apps. Second, OS containers offer less isolation between mobile OSes, making it possible to exploit optimizations between mobile apps to further improve performance. However, there lacks a comprehensive study of offloading performance to guide the optimizations.

In this paper, we address the above two challenges and design *Rattrap*, a mobile offloading cloud platform based on OS containers. To support the execution of mobile apps in the cloud, we develop *Cloud Android Container*, a mobile OS environment built directly on general purpose server OS. We perform a comprehensive study of offloading performance in representative mobile workloads, which motivates the development of two optimizations for mobile offloading: *shared resource layer between multiple apps* and *mobile code cache*. Experimental results using LXC container and Android apps show significant performance improvement on code offloading compared to VM-based cloud platforms. The startup time of the mobile OS is reduced from more than 28s using VMs to less than 2s with Cloud Android Container. Our container-based cloud saves as much as 75% memory footprint and at lease 79% disk usage by hosting the mobile OS in the cloud. Moreover, the performance of code execution has been improved by as much as 40%. In summary, this paper makes the following contributions:

- We discuss the behaviors of mobile cloud applications, characterize representative offloading workloads, and summarize the problems current cloud platform faces. These provide the guidelines of how to make the cloud side better serve offloading requests.

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.47

- We devise the idea of dynamically extending the host OS kernel with mobile OS drivers and make the mobile cloud platform based on OS-level virtualization come true. By implementing drivers for various kernel features, containers can run certain different OSes, which partially breaks the kernel limitation of OS-level virtualization.
- With containers running mobile environments, we propose a set of techniques to improve its efficiency, including customization of mobile OS, sharing common resources, and code cache mechanisms, which makes our work no longer a simple code runtime environment.

II. Background

A. Basic Offloading Mechanism

With the rise of mobile cloud computing, the idea of offloading computation has been a research hotspot for a long time. Unlike existing mobile applications which run locally and directly request data from content providers, mobile devices can offload parts of the workloads to the cloud, making cloud a powerful worker to process computation tasks. Generally, a basic offloading system [2, 3, 11–14] is composed of two parts. The client side runs on mobile devices, which controls offloading computational code to the cloud when needed. The cloud side is responsible for handling offloading requests and provides runtime environments.

Much research effort has been duplicating mobile runtime environments in the cloud to support offloading. To achieve this, many frameworks [2, 3] leverage the virtualization technology, namely mobile OS virtual machine. The cloud platforms in these frameworks have nearly the same runtime environments as mobile devices, which makes it straightforward to run mobile code on the cloud side. However, to use a VM instance, the cloud has to install and boot a guest OS in the VM, which incurs substantial delay. For example, a *Standard Medium* instance (m1.medium) on Amazon EC2 has an average setup time of 28.4 seconds, which is undesirable for mobile requests. Therefore, on-demand deploying VMs is an expensive operation and not suitable for real-time mobile offloading [6].

B. OS-level Virtualization

While the traditional system virtualization provides strong fault isolation, OS-level virtualization is much more efficient, especially for server consolidation. Containers, as the center of OS-level virtualization, have recently emerged as a lightweight alternative to hypervisor-based virtualization [15]. Unlike virtual machines, containers usually imposes much less overhead, since they share the host OS kernel and do not suffer the overhead of resource virtualization.

Unfortunately, the requirement of sharing host OS kernel makes OS-level virtualization unsuitable for mobile cloud, as general purpose OSes are incompatible with the mobile OS and does not support running mobile code. This limitation is a vital challenge of designing a container-based platform for mobile cloud computing.

III. Workload Analysis

In this section, we perform a characterization of benchmark applications in used in previous mobile offloading research, and discuss the problems of existing cloud platforms based on VM. As will be discussed later in Section IV, these findings about offloading workloads motivated our system design.

A. Experiment Setup

In this series of experiments, four representative Android applications of four different categories, which have been widely adopted by previous researches [1, 12–14], are used as our benchmark workloads.

- *Image tools* are the most common benchmarks used in previous researches and represent the computation-intensive workloads with file transfer. **OCR** (Optical Character Recognition) is based on the Google Tesseract library whose real computation is implemented with Java Native Interface (JNI) code written in C++.
- *Games* interact with user continually, representing workloads with intensive network communications. **ChessGame** is an Android port of the CuckooChess Engine which ranks top 200 in Computer Chess Rating Lists 40/40[1].
- *Anti-Virus* has been adopted increasingly since the rapid malware emergence. **VirusScan** checks the target with virus database search and spawns more I/O requests than other benchmarks.
- *Mathematical tools*, like **Linpack**, are implemented in ordinary Android Java. They are often used to represent pure computation.

The client applications run on 5 Android devices and use the Java reflection techniques to enable the offloading of computation codes. The mobile code runtime environments on the cloud is based on Android-x86 [16] VM. More details about experimental settings, like the configuration of cloud servers, can be found in Section VI.

B. Performance Penalty of Runtime Preparation

As described in [6], the key reason that existing cloud approaches are not suitable for mobile cloud applications is the unacceptable long startup time of VMs. In this section, we adopt all 4 workloads and evaluate how runtime preparation affects the performance. The experiment is performed with stable LAN WiFi, in which we suppose the best network environment is provided and take no account of the instability of network.

We divide the process of offloading into 4 phases. *Computation Execution* is the pure execution stage of offloaded tasks on the cloud. *Runtime Preparation* is the setup phase for mobile code runtime after the offloading requests arrive. *Network Connection* is the process of establishing a connection between mobile devices and cloud resources. *Data Transfer* is the time spent to transfer necessary data for offloading tasks. Figure 1 shows phase details and offloading speedups of

[1] http://www.computerchess.org.uk/ccrl/4040/

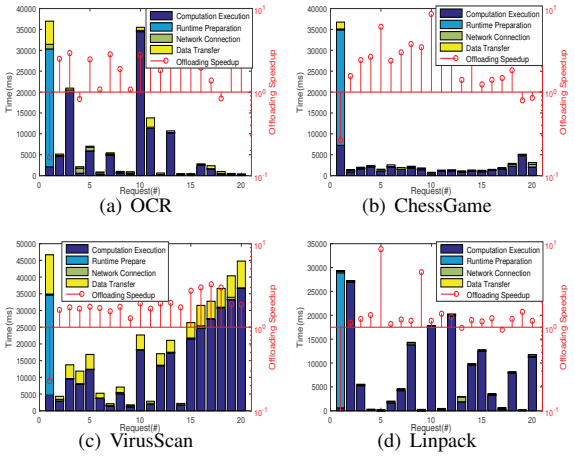

Fig. 1. Phase details and offloading speedups when running different workloads with the existing cloud platform. The first 20 offloading requests are investigated.

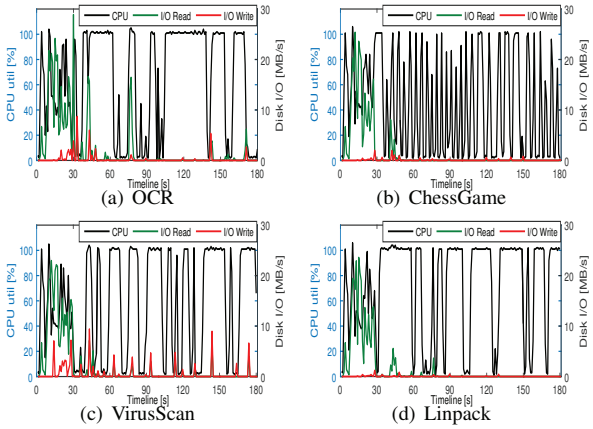

Fig. 2. System load in offloading process of different applications.

different workloads. *Offloading speedup* refers to the ratio of local execution time and offloading response time. When offloading speedup is larger than 1, code offloading outperforms local execution; otherwise, we call it an *offloading failure*.

Observation 1: Through Figure 1, it is clearly observed that each VM encounters a serious offloading failure for the first request, which is caused by the long runtime preparation. Each time a new code execution environment is needed, offloading requests have to face the cold start of cloud runtime. Since the cloud platform is based on Android-x86 VM, which is a relatively heavyweight resource model, the startup time usually lasts even longer than the pure computation time. This leads to poor user experience of mobile apps.

Implication 1: In light of the above, we believe that the long startup time of VMs with mobile OS causes the performance degradation at the beginning stage of offloading requests. Pre-loading VMs is an intuitive way to mitigate such offloading failures, but it will inevitably reduce the server resource utilization and increase the complexity of the system. Leveraging a lightweight and fast-boot cloud resource model may change the game.

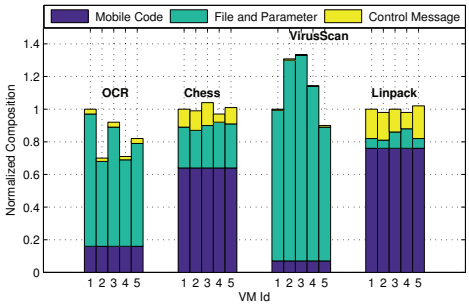

Fig. 3. Composition of migrated data with different workloads.

C. Cloud Server Load

In this subsection, we analyze the impact of offloading on cloud server load in previous experiments, and discuss the properties of offloading tasks. Figure 2 describes the timeline graphs (with one second granularity) of CPU and I/O utilization on the cloud server.

Observation 2: During the boot of VMs (0-30s), the server loads show similarities between different workloads. After the stage of runtime preparation, offloading tasks are handled and the CPU load increases to 100% once new request arrives. In ChessGame, the computation are relatively small, leading to the high fluctuation in the CPU load.

In terms of system I/O load, after the startup, it appears an I/O increase for a short period of time because of receiving mobile codes and loading them into runtime by ClassLoader[2]. After this, workloads show different I/O loads. OCR and VirusScan, which demand more migrated data for execution, incur short-term I/O boosts when requests arrive. In contrast, other workloads call for few data transfer and bring about small I/O loads.

Implication 2: The experiment results suggest that most of offloading tasks are computation-intensive jobs initially and followed by a short-term I/O-bound phase. Considering the performance loss caused by VMs, especially the overhead of I/O virtualization, the cloud platform should take both computing performance and I/O performance as the priorities of performance improvement.

D. Duplicate Code Transfer

We now look into data transfer in the process of offloading. From Figure 1, we notice that the first computational task in VM comes with significantly more data transfer time, because the new runtime lacks the mobile code to be executed. To understand whether all migrated data is necessary, we evaluate the composition of migrated data of each Android VM on cloud, as shown in Figure 3. The migrated data includes mobile codes to be executed, files and parameters that specify offloading tasks and control messages managing offloading procedures. Since our offloading framework is based on Java reflection, the mobile codes in the experiment are app files.

Observation 3: Through Figure 3, it is clearly observed that in the process of offloading, the transmission of the same

[2]http://developer.android.com/reference/java/lang/ClassLoader.html

mobile codes occurs in every VM. This is because VMs are completely isolated and clients have to push mobile codes into each one of them. For workloads which require no additional file transfer, like ChessGame and Linpack, the mobile code accounts for more than 50% of migrated data. The duplicate code transfer causes the long data transfer time every time a new VM is started. We also find that the migrated data size of single request is relatively small due to the limit on network bandwidth of mobile devices.

Implication 3: The duplicate data transfer in existing frameworks is inefficient and ideally the same mobile code should be transferred only once. One way to solve this problem would be to build the mobile code cache mechanism.

E. Redundancy of Mobile Environments

In this experiment, we carefully profile which part of the mobile OS is essential, and which part is less important or even unnecessary to offloading. After the experiments above are finished, we check the last access time of each part of Android OS and try to find which files are never used during the offloading process.

Observation 4: We find out that 771MB out of 1.1GB files (68.4% of the entire OS) are never accessed by offloaded codes, which are composed of unnecessary modules and libraries. Most of them are for the hardware of mobile devices, like Camera support and sensor driver. We also notice that the same */system* folder occupies 985MB space (87.4% of the entire OS) in each Android VM. This means offloading codes of different applications probably only require a uniform runtime environment on the cloud, while it exists serious redundancy with current cloud platforms.

Implication 4: According to the observation above, it seems to be unwise to run an entire mobile OS in VMs separately. By removing the unnecessary parts and sharing the system libraries, we can improve the disk utilization of code runtime environment and make it more lightweight.

IV. System Design

Through the above observations and implications, we conclude that existing cloud platforms cannot serve mobile computation offloading well. In this section, we describe Rattrap, a new lightweight cloud platform for mobile computation offloading.

A. Overview

Based on previous analysis of typical mobile workloads, we design the cloud platform with four primary goals: (1) keep the deployment of code runtime environments fast, (2) eliminate redundancy and overlap of mobile environments, (3) minimize performance overhead in the cloud, and (4) reduce duplicate code transfer.

Figure 4 provides an overview of Rattrap's system architecture. We use Android as the mobile environment since our prototype implementation is based on it. On the cloud side, Rattrap mainly consists of 4 components: *Cloud Android Container* provides the execution environments for mobile

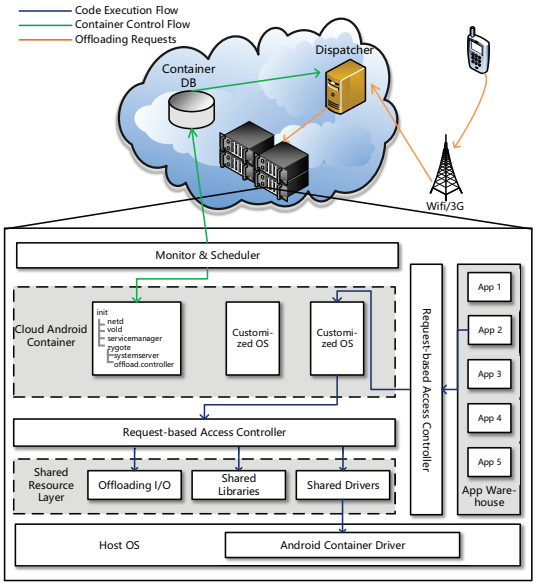

Fig. 4. Overview of Rattrap architecture.

code; *Shared Resource Layer* packs the shared resources of containers to avoid redundancy and improve performance; *App Warehouse* implements the code cache mechanism to get rid of duplicate data transfer; and *Request-based Access Controller* secures the cloud platform.

In what follows, we will present, in details, the design of above core components in Rattrap and handle the following challenges:

- *How to deal with the gap between the host OS and Android?* This is the key step to ensure that Cloud Android Container actually works on the cloud (§IV-B1).
- *How to adapt Android boot process for container boot and make it as fast as possible?* This work means changing Android startup procedures from device environments to container environments, while guaranteeing the efficiency (§IV-B2).
- *How to construct the appropriate OS for mobile computation offloading?* Rattrap employs a subset of entire Android OS as the runtime environments, just enough to support mobile computation offloading (§IV-B3).
- *How to eliminate redundant storage and manage the common resources among containers?* Shared Resource Layer considerably reduces the size of single container by sharing common files, and boosts I/O performance with sharing offloading I/O (§IV-C).
- *How to implement mobile code cache mechanism?* With App Warehouse, code cache takes charge of indexing and fetching executed codes for arrived offloading requests (§IV-D).

Note that some other components are designed to ensure Rattrap function normally: *Dispatcher* handles the new arrived offloading requests and allocates execution environments for them; *Container DB* stores information of Cloud Android Containers as basis of resource management;

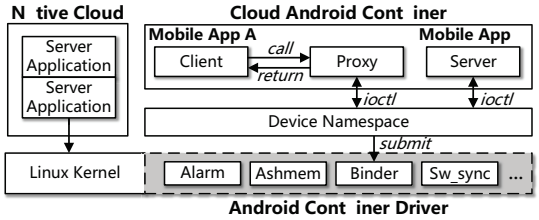

Fig. 5. To solve the kernel incompatibility problem, Android Container Driver dynamically extends the initial kernel with Android drivers.

Request-based Access Controller remedies the deficiency of container's lightweight isolation mechanism; and *Monitor & Scheduler* conducts resource scheduling at process-level, rather than at VM-level in existing platforms.

B. Cloud Android Container

The code runtime environment is one of the most important part in Rattrap. With OS-level virtualization, Rattrap introduces a novel environment, *Cloud Android Container*, to replace Android VM to support mobile code execution. Each Cloud Android Container holds individual Android OS and has its own process space, root file system and network resources. To conduct Cloud Android Container and guarantee the efficiency, we carry out 3 primary designs as follows.

1) Android Container Driver: Containers share OS kernel interfaces with little overhead, but are unable to support multiple kernels (in our case, Android kernel and Linux kernel). The kernel restriction has significantly weaken the generality of containers and we propose Android Container Driver to solve this problem in Rattrap.

Android kernel is considered as a special version of the mainstream Linux kernel, with additional drivers including Alarm (Real Time Clock based alarm for timer messages), Binder (Android interprocess communication mechanism), Logger (lightweight RAM log driver) and so on. In official Android, these drivers are built directly into the kernel since they are essential as soon as devices boot. To enable the features, we have to integrate this part of drivers with host OS kernel on the cloud and recompile it, which leads to poor expandability. Fortunately, in our case, Android OS is running inside containers, which means Android drivers are unnecessary until Cloud Android Containers are started. That is to say, Android kernel features will not have to be built-in drivers.

Therefore, we propose *Android Container Driver*, the kernel module package which contains specific Android drivers to dynamically extend the running kernel for container environments. In stead of linking drivers statically to the kernel, Android Container Driver implements Android features in the form of loadable kernel modules. The cloud is able to support Cloud Android Container by simply loading modules whenever necessary, which ensures easy transformation between common cloud platforms and Rattrap.

Figure 5 describes the Binder driver model in Android Container Driver as an example. Like other Android kernel drivers, Binder is a pseudo driver which has no corresponding physical device. Pseudo drivers are not hardware-related and thus our implementation of Android Container Driver will work for all hardware platforms. The pseudo devices (*e.g.*, `/dev/binder`) are initiated only when Android Container Driver is loaded, resulting in kernel extension without rebuilding or rebooting cloud servers. With the extended kernel, mobile applications inside containers are able to make Android-specific system calls. As will be mentioned in Section IV-C, Android drivers are shared among containers. We leverage device namespace [17] to implement device isolation and multiplexing for Alarm, Binder and Logger. The workflow of device namespace framework is modified to adapt to Rattrap since it is originally designed to operate with mobile devices instead of cloud.

More importantly, Android Container Driver proposes the idea of running operating systems (*e.g.*, Chrome OS [18], embedded Linux) with differential kernel features in cloud containers. By implementing the kernel differences as driver modules, the cloud platform will be able to dynamically support containers running multiple Linux-based operating systems and the reconstruction of cloud platforms is quite simple without interfering native server applications. Meanwhile, it also delivers flexibility and efficiency. In particular, the extended drivers are only included when certain containers are started, and unloaded when they are no longer needed to avoid wasting memory. This usage model enhances the flexibility of containers and partially breaks the kernel limitation of OS-level virtualization.

2) Android Boot in Cloud Android Container: Android boot process is another concern in Rattrap, since we focus on its adaptation to cloud container environments and try to reduce the latency caused by container startup. Android in mobile devices works on flash memory and its boot process relies on the RAMDisk-based images. Thus, Android OS boots quite differently from existing operating systems which can directly start in containers. To enable Android in Cloud Android Containers, Rattrap modifies its startup sequence so that instead of loading kernel and ramdisk in boot process, the preprocessing is accomplished before we start containers.

Figure 6 compares the boot process in mobile devices and in containers. Android device takes 4 steps to finish booting, while Cloud Android Container boot jumps directly to the "terminus". In particular, there are three important differences, as in the *gray box*. First, bootloader loads Android OS after its power-on self-tests, but does no exist in Cloud Android Containers. Instead of loading kernel, containers directly acquire kernel functions by sharing the host OS kernel. Second, mounting root file system is normally handled by the kernel initialization, while we construct the file system according to `initrd.img` before starting a Cloud Android Container. Containers are populated with Android rootfs and start directly by executing `/init`. Third, `/init` is the beginning of user-space processes and starts *Zygote* which then initializes core system services. In order to make the init process work in Rattrap and optimize the boot time, we

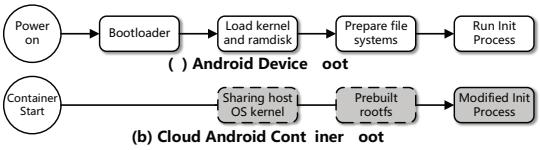

Fig. 6. Android device boot vs. Cloud Android Container boot.

modify the original init process.

3) Customized OS for Offloading: Our analysis (§III-E) shows that only 31.6% of the entire Android OS is actually needed for processing offloading requests. To keep Cloud Android Containers lightweight and avoid wasting cloud resources, Rattrap customizes the composition of OS to replace the original Android as the mobile cloud environment. Unlike the official version of Android, the customized OS here is designed to support offloaded codes only and has much smaller size. Employing the customized OS makes Cloud Android Container more lightweight, without affecting the computational performance.

Specifically, mobile OS in Rattrap behaves as a computing environment which is quite different from the one on a smartphone. First, plenty of system components are taken away since they are worthless here. Apart from the kernel drivers we mentioned in Section IV-B1, most of Android drivers are for hardware devices, like Camera and Bluetooth, which are unnecessary for offloading requests and occupy big space. In our profiling with Android 4.4 (Kitkat), the redundancies mainly include 20 built-in Android apps, 197 shared library files (.so), 4372 kernel modules (.ko) and 396 firmware libraries (.bin). Besides, the customized OS works in a totally new mode, without system UI, telephony, user interact and many other features. Some of the features are central services of Android's architecture, especially the rendering and display part, and simply removing them will most certainly cause a crash. The customized OS solves the problem by restraining calls for these services. When the invocation is inevitable, we fake the key interfaces with direct returns so that the system will not find the absences.

C. Shared Resource Layer

Shared Resource Layer is designed to handle two problems. First, our analysis (§III-E) describes that overlapping exists between different runtime environments, and sharing this part of OS can avoid waste of disk space. Second, as discussed in Section III-C, disk I/O is also important for offloading requests besides computation. So, Rattrap proposes *Sharing Offloading I/O* and markedly improves I/O performance.

Containers often use layered file system to support system images and COW (copy-on-write) at the file system level, like Docker [19] combined with AUFS [20] (Another Union File System). Based on this, Shared Resource Layer is built to take care of common data and share them between Cloud Android Containers. In Rattrap, the system libraries stripped by our OS customization are the main components of shared data. By eliminating duplication of these files, the size of a single Cloud Android Container becomes about 50 times smaller.

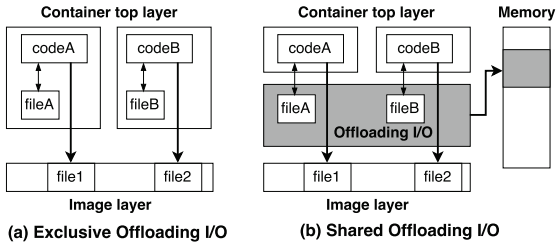

Fig. 7. Rattrap implements Sharing Offloading I/O layer with in-memory file system.

Android drivers in kernel extension are also shared resources, because even with their necessity, only a few offloading codes interact with them, making it meaningless to provide exclusive drivers for individual Cloud Android Container.

Moreover, Rattrap implements the offloading I/O layer with in-memory file system to boost its performance. We define offloading I/O as the I/O operations executed by offloaded codes, mainly related to transferred files. The introduction of in-memory file system is based on two reasons. First, migrated data in offloading is one-time deal, making the volatility of in-memory file system no longer a concern. "Burn after reading" keeps the size of in-memory file system small and ensures privacy protection. Second, the design presents an interesting tradeoff between I/O performance and memory footprint. Fortunately, offloading requests usually have no large file transmission (§III-D), meaning that the data size in offloading I/O layer is relatively small. With minor extra footprints, Rattrap can conduct offloading I/O in memory.

Figure 7a describes that existing containers handle I/O operation inside their own layer, namely *Exclusive Offloading I/O*. However, this design goes against the integration of in-memory file system, since separating offloaded data in multiple Cloud Android Containers makes it difficult to build the in-memory layer. To solve this problem, Rattrap puts offloading file objects in one shared layer instead of inside multiple container top layers. Figure 7b illustrates this design, as we called Sharing Offloading I/O. The combination of in-memory file system and Sharing Offloading I/O makes Rattrap very efficient, especially for workloads with more I/O operations.

D. App Warehouse and Mobile Code Cache

To solve the problem of duplicate code transfer (§III-D), we implement the App Warehouse in Rattrap to manage offloaded codes from different mobile apps. Figure 8 describes the mobile code cache mechanism based on App Warehouse. In Rattrap, the code transfer happens when the application sends its first offloading request, once and for all. Then, App Warehouse will preserve the code and maintain a cache table with the code information. After that, offloading tasks of same operations will have same `Reference` and look for codes from App Warehouse by `AID`, rather than deliver them again. Moreover, the cache table contains additional mapping relationships between mobile codes and Cloud Android Containers (`CID`). With these information, the Dispatcher tends

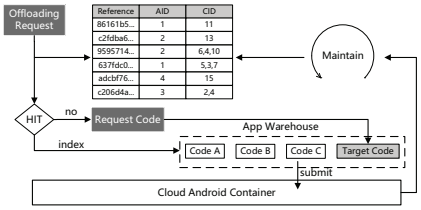

Fig. 8. Mobile Code Cache Mechanism

to allocate offloading tasks to the Cloud Android Container where requests from the same application have been executed before, which saves the time for loading codes. By caching mobile codes, Rattrap significantly reduces the migrated data size, and thus reduces the energy consumption of mobile devices and users communication costs.

E. Security Discussion

In mobile cloud environment, the cloud server running offloaded codes may serve different apps (like Cloudlet [21]). If one of the offloaded apps carries virus or malware, it would cause serious security problems to all of the users. Besides, although container-based runtime is more lightweight, it is less robust since OS-level virtualization is a lightweight isolation mechanism [7]. What's more, our shared-based architecture (Shared Resource Layer and App Storehouse) is likely to be used by malicious codes.

All of the above risks make us to provide an additional security guard, Request-based Access Controller, for Rattrap. It automatically analyzes the offloading requests with information received and generates the permission table for them. Offloading requests from the same application share one permission table, which means the analysis happens only once for each mobile app. This controller works by filtering every workflow that comes out of the Cloud Android Container and records the violation of permission checks. When the number of violations reaches the threshold, offloading requests from this app will be blocked.

V. Implementation

We implement the prototype of Rattrap on our server machines. Each server contains 2 six-core Intel Xeon X5650 2.66Ghz CPUs with 16GB of DRAM and 300GB HDD, running Ubuntu 15.04. The original cloud server can be extended to Rattrap by simply loading Android Container Driver, without kernel recompiling or any operating system modification. Compared with traditional Android VMs, we implement Cloud Android Container based on Linux Container (LXC) [10] as the mobile code runtime environments. We choose LXC as our virtualization solution because it has been widely used in production environments and its derivative, Docker, is the most popular container technology at present.

The modifications of Android drivers are based on Linux Kernel 3.18.0. As we mentioned above, the driver isolation and multiplexing is based on device namespace [17]. Since the device namespace patch is based on Android kernel, efforts are made to integrate device namespace into Linux kernel and make it work with cloud servers. The construction of Cloud Android Container and customization of Android OS are based on a self-built Android-x86_4.4_r2 image which we consider most stable version by then. In our prototype implementation, tmpfs is used as the in-memory file system for sharing offloading I/O.

The clients run on Android devices equipped with both WiFi and cellular network (3G/4G) connections. Rattrap leaves the offloading details in clients to existing offloading frameworks and only cares about the cloud side. The power consumption measurement is based on PowerTutor [22].

The source code of Rattrap is publicly available online at `https://github.com/CGCL-codes/Rattrap`.

VI. Evaluation

A. Experiment Setup

All experiments are run on server machines mentioned in Section V. In our experiments, we compare Rattrap with two other cloud platforms as baseline systems:

- **VM-based Cloud Platform**: The current cloud platform whose code runtime environment is usually based on Android-x86 [16] running in VirtualBox. Each Android-x86 VM is configured to run with 1 vCPU and 512MB of memory.
- **Rattrap(W/O)**: Rattrap without optimizing the Cloud Android Container, meaning we only replace VM with Container and employ NO OS optimization, shared resource design and code cache mechanism.

We still use the typical offloading workloads introduced in Section III as our benchmarks and evaluate Rattrap in 4 mobile scenarios:

- **LAN WiFi**: Mobile devices and the cloud server are in the same LAN, stable and fast.
- **WAN WiFi**: WAN WiFi has about 60ms latency connecting the cloud server through public IP, but stable.
- **3G**: 3G is used for Internet access, unstable, with high latency and limited bandwidth, whose upstream bandwidth is 0.38Mbps and downstream bandwidth is 0.09Mbps.
- **4G**: 4G gets better network conditions than 3G, but less stable than WiFi since the change of context, whose upstream bandwidth is 48.97Mbps and downstream bandwidth is 7.64Mbps.

B. Runtime System Comparison

Before introducing mobile cloud applications, we compare Cloud Android Container in Rattrap with Android VM in the traditional cloud platform. We look at several primary factors to evaluate the code runtime environments. Table I shows a portion of our results, where CAC represents Cloud Android Container.

We test the setup time by capturing the time it takes for runtime environments to finish startup and be connected to the Dispatcher. By introducing OS-level virtualization, Cloud Android Container without optimization can still achieve

TABLE I
OVERHEADS OF CODE RUNTIME ENVIRONMENTS

Code Runtime	Setup Time	Memory Footprint	CPU Allocation	Disk Usage
Android VM	28.72s	512MB	1vCPU	1.1GB
CAC (non-optimized)	6.80s	128MB	1vCPU	1.02GB
CAC	1.75s	96MB	1vCPU	7.1MB

TABLE II
TOTAL NUMBER OF DATA TRANSMITTED WITH DIFFERENT BENCHMARKS

Workload	Download (KB)			Upload (KB)		
	Rattrap	W/O	VM	Rattrap	W/O	VM
OCR	154	152	152	29440	34233	35047
Chess	34	34	34	4788	14011	13301
VirusScan	1738	1582	1572	91973	99375	98895
Linpack	11	11	11	169	776	705

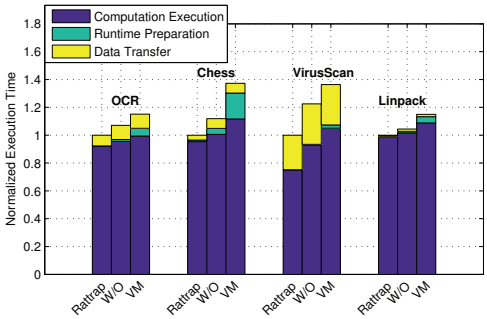

Fig. 9. Average performance of offloading requests.

4.22x speedup of preparation time. With our optimization of Android boot, the speedup of setup time becomes much more significant (16.41x).

The memory footprint is actually from the configuration we specified before the runtime starts. Android VM requires at least 256MB of memory and is recommended to run with 512MB of memory. We allocate 128MB of memory to each Cloud Android Container without optimization, because we observe that the maximum memory usage in the offloading process is 110.56MB and it happens when the container boots. Similarly, the initial memory for the optimized Cloud Android Container is 96MB since its maximum memory usage is 96.35MB.

As for the disk usage, since we provide common system libraries through Shared Resource Layer, a single Cloud Android Container occupies less than 7.1MB, while the size of entire Android OS in containers or VMs is around 1GB.

C. Performance for Different Applications

In this section, we evaluate the performance difference between Rattrap and traditional cloud platforms with different workloads. To eliminate the impact of network connection, we run mobile apps in LAN WiFi. In order to model the user behavior, for each set of experiment, we use 5 Android devices running offloading workloads, and the same inflow of requests is used for both Rattrap and VM-based cloud.

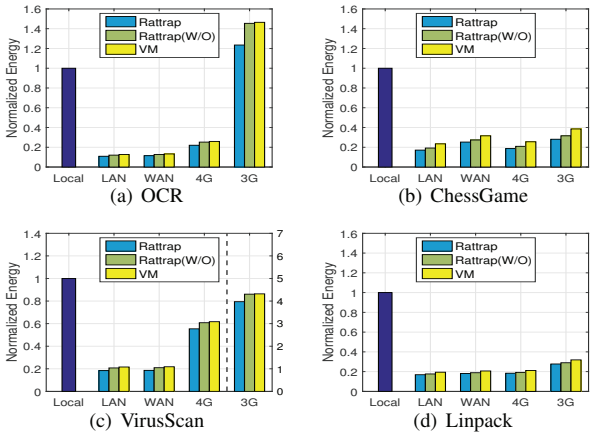

Fig. 10. Average power consumption of offloading requests in various network scenarios.

Figure 9 shows the average performance of offloading requests in different workloads. We observe that the average runtime preparation time improves 4.14-4.71x with Rattrap(W/O) and 16.29-16.98x with Rattrap, which significantly influences the total execution time of offloading requests.

The data transfer time also achieves speedups from 1.17x to 2.04x with Rattrap, while Rattrap(W/O) gets no improvement since it has no code cache mechanism. Table II shows the total migrated data size in different workloads with Rattrap, Rattrap(W/O) and VM-based cloud. We can see the upload data size is obviously decreased by leveraging Rattrap, which indicates the importance of our code cache mechanism. To be noted, we notice that OCR and VirusScan have small app size compared to their parameter data size, which makes the improvements inconspicuous.

As for the pure computation time, we observe that Rattrap(W/O) already achieves 1.02-1.13x speedups since Cloud Android Container gets rid of the hardware virtualization overhead. Meanwhile, Rattrap gets even better performance (1.05x-1.40x speedups) and most of these advantages benefit from the sharing offloading I/O. The performance speedups for pure computation codes, like Linpack, is relatively small, since VM introduces not much overhead and leaves Rattrap little room for improvements. The behavior of VirusScan is quite different from other applications. According to our analysis in Section III, VirusScan introduces more I/O operations and thus achieves a higher speedup since containers get better I/O performance than VM-based platforms, while Rattrap improves further by introducing in-memory file system.

D. Power Consumption in Different Network Scenarios

In this section, we evaluate how Rattrap saves more battery power in various network scenarios. To achieve fair comparison, at runtime we record the network conditions and application states of offloading tasks with Rattrap, and then we replay these requests for the other two baseline systems.

The power consumption results are shown in Figure 10. The consumed energy is normalized to the one when running the workload apps entirely on the mobile devices. It is clearly

observed that, in most cases, mobile computation offloading can extend battery life markedly, especially for workloads without additional file transmissions, like ChessGame and Linpack. Besides, we have three observations from the results.

First, Figure 10 shows that both Rattrap and Rattrap(W/O) can save more energy for mobile devices than existing cloud platforms. This indicates that Cloud Android Container shortens the cloud response and thus extends the battery life. However, Rattrap(W/O) has limited improvements since it can not benefit from the code cache and the mobile OS optimization.

Second, we notice that compared with VM-based cloud platforms, Rattrap outperforms VM by 1.37x with ChessGame, while it is less superior with VirusScan (1.13x) and Linpack (1.15x). The differences are caused by the properties of workloads. Offloading requests from ChessGame have less computation and thus the profit from runtime preparation accounts for higher share of energy savings than Linpack. On the other hand, the power consumed by file transmissions in VirusScan is considerable and unavoidable, and thus depresses the energy efficiency of Rattrap.

Third, from Figure 10(a) we find that Rattrap outperforms VM by 1.22x while the average improvement drops when the network gets worse. This means that for OCR, as network conditions become poorer, the gap of power consumption between VM-based solutions and Rattrap becomes smaller. We observe similar phenomena in VirusScan, but not in other workloads. The reason is that with the rise of latencies and the decrease of network bandwidth, the offloading bottleneck for workloads which require additional files is no longer the computation performance but the long-time file transfer, which is not improved in Rattrap. This means for offloading requests with a lot of file transmissions, Rattrap's energy-saving advantage is affected by the network conditions.

E. Performance with Trace-based Simulation

In this section, we use trace-based simulation to evaluate the performance of Rattrap further. The trace data is from the Livelab dataset [23]. The dataset consists of real-world app access traces, and we simulate offloading requests with these timestamps of access records as the start time. For fair comparison, we use a separate experiment to obtain the local execution time for calculating speedup.

With various workloads, the speedup distributions of Rattrap, Rattrap(W/O) and VM-based cloud platforms present a good similarity. Without loss of generality, we just present the CDF of speedups for ChessGame with our trace simulation, as shown in Figure 11. Our evaluation shows that Rattrap always achieves better speedups. For example, 54.0% of offloading requests with Rattrap get higher speedup than 3.0x, while the result is 50.8% for Rattrap(W/O) and only 11.5% for the VM-based cloud platform.

In addition, we also find an interesting detail in the experiments. Compared with Rattrap(W/O), Rattrap is slightly dominant for offloading requests with speedup higher than 2.0x, since offloaded code of ChessGame is more like pure

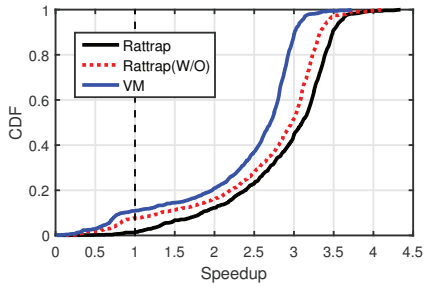

Fig. 11. Rattrap improvements with real-world access traces.

computation and the improvement of Rattrap is not so obvious. On the other hand, we observe that Rattrap handles offloading failures much better than other two baseline systems. Specifically, Rattrap(W/O) fails to make 7.7% of requests benefit from offloading, which is close to the VM-based cloud platform (9.7%), while Rattrap's offloading failure is only 1.3%. This is because the start time is less than 2s for Rattrap, which is pretty close to just-in-time deployment of runtime environments. Most of offloading requests do not have to suffer from the long runtime preparation, which is the major cause of offloading failure.

VII. Related Work

To implement mobile computation offloading, many studies have been conducted. MAUI [11] takes advantage of managed code and provides method level code offloading for Microsoft .NET applications. CloneCloud [2] and COMET [12] modify Dalvik Virtual Machine to implement thread level offloading without access to the program source code. ThinkAir [3] customizes communications between cloud and devices in application layer and thus requires additional code annotations. These frameworks focus on implementation details of offloading and employ plain VM to support offloaded codes.

Instead, some other related offloading systems run the cloud side in the form of service or component and no longer needs the mobile OS simulation. Zhang et al. [14] give Android applications computation offloading capability by automatically app refactoring. Sapphire [13] builds its own offloading object *(Sapphire Object)* and runtime kernel *(Deployment Kernel)* to separate application logic from deployment code. However, these frameworks require extra deployment efforts and their runtime environments differs from that on mobile devices and thus expose strict limitations on offloaded codes.

Closer to our concern, some researches enable improvement on cloud side for offloading. CMCloud [1] is a novel cost-effective cloud platform which detects potential QoS failures by performance estimation and guarantees QoS requirements by VM migration. COSMOS [4] introduces strategies for cloud resource management and offloading decision, attempting to sustain offloading at low cost. These works pay more attention to resource allocation strategy, or concern more about the cost-effective feature. Unlike efforts as discussed above, Rattrap focuses not on management strategies but on problems of the cloud platform itself.

Container technology provides an alternative solution for building cloud platforms. A great benefits brought by container have been widely confirmed, including better system efficiency than VMs [8], low-overhead process migration [7] and so on. To make container technology more efficient and secure, a lot of researches have been conducted recently. For example, Slacker [15] optimizes the storage model to accelerate container startup and Scone [24] secures containers with Intel SGX. These works advance container technology and our system may also benefit from them.

Some other efforts have been attempting to combine OS-level virtualization with mobile usage. Cells [17] leverages OS-level virtualization to run one foreground virtual phone and multiple background phones in the same mobile device with modest overhead. ParaDrop [25] proposes a multi-tenant edge computing framework by dynamically installing third party services with LXC on wireless gateways. Unlike them, Rattrap attempts to introduce container for mobile OS and aims to improve offloading performance from the cloud side.

VIII. Conclusion and Future Work

In this paper, we present Rattrap, a lightweight container-based cloud platform designed for mobile computation offloading. We observe that traditional virtual machine-based cloud platform would incur significant overhead when dealing with mobile cloud applications. Rattrap mitigates such drawbacks by introducing OS-level virtualization to provide execution environments for mobile codes and bringing in a set of designs to conduct an efficient cloud platform. Our evaluation of a Rattrap prototype shows that Rattrap can significantly improve both the system performance and energy efficiency for mobile computation offloading.

To broaden the applicability of our work, we plan on making Rattrap available on public clouds, like Amazon EC2 [26], by building an open Amazon Machine Image (AMI). We will also explore the possibility of Rattrap implemented on Docker, which may bring about the real just-in-time provision of Cloud Android Container. Moreover, we would like to find more use cases for Cloud Android Container, like Internet of Things (IoT), mobile app testing, etc.

IX. Acknowledgements

This research was supported by National Key Research and Development Program under grant 2016YFB1000501, 863 Hi-Tech Research and Development Program under grant No. 2015AA015303, and National Science Foundation of China under grant No. 61232008.

References

[1] D. Chae, J. Kim, J. Kim, J. Kim, S. Yang, Y. Cho, Y. Kwon, and Y. Paek, "Cmcloud: Cloud platform for cost-effective offloading of mobile applications," in *CCGrid'14*, pp. 434–444.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *EuroSys'11*, pp. 301–314.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *INFOCOM'12*, pp. 945–953.

[4] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: Computation offloading as a service for mobile devices," in *MobiHoc'14*, pp. 287–296.

[5] Y. Duan, M. Zhang, H. Yin, and Y. Tang, "Privacy-preserving offloading of mobile app to the public cloud," in *HotCloud'15*, pp. 18–18.

[6] M. Shiraz, S. Abolfazli, Z. Sanaei, and A. Gani, "A study on virtual machine deployment for application outsourcing in mobile cloud computing," *J. Supercomput.*, vol. 63, no. 3, pp. 946–964, 2013.

[7] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *OSDI'12*, pp. 361–376.

[8] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *EuroSys'07*, pp. 275–287.

[9] "Openvz," http://en.wikipedia.org/wiki/OpenVZ.

[10] "Linux container," http://en.wikipedia.org/wiki/LXC.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *MobiSys'10*, pp. 49–62.

[12] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *OSDI'12*, pp. 93–106.

[13] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *OSDI'14*, pp. 97–112.

[14] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *OOPSLA'12*, pp. 233–248.

[15] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *FAST'16*, pp. 181–195.

[16] "Android-x86," http://www.android-x86.org/.

[17] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A virtual mobile smartphone architecture," in *SOSP'11*, pp. 173–187.

[18] "Chrome os," http://www.chromium.org/chromium-os.

[19] "Docker," http://www.docker.com/.

[20] "Advanced multi layered unification filesystem," http://aufs.sourceforge.net/.

[21] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct 2009.

[22] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *CODES/ISSS'10*. ACM, 2010, pp. 105–114.

[23] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: Measuring wireless networks and smartphone users in the field," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 3, pp. 15–20, Jan. 2011.

[24] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in *OSDI'16*, pp. 689–703.

[25] D. Willis, A. Dasgupta, and S. Banerjee, "Paradrop: A multi-tenant platform to dynamically install third party services on wireless gateways," in *MobiArch'14*, pp. 43–48.

[26] "Amazon ec2," http://aws.amazon.com/ec2/.