Android Unikernel: Gearing Mobile Code Offloading Towards Edge Computing

Song Wu, Chao Mei, Hai Jin, Duoqiang Wang

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
E-mails: {wusong, meicorl, hjin, dqwang}@hust.edu.cn

Abstract

Recently, *Mobile Cloud Computing* (MCC) which utilizes cloud to enhance mobile device's performance, is becoming more and more popular. A typical approach of MCC is to offload some computation-intensive tasks onto cloud servers to execute and fetch results back. However, this schema suffers greatly from the long-distance network transmission latency and server boot-up latency, leading to high-delay response, which is unacceptable for most real-time applications. *Mobile Edge Computing* (MEC) or *Mobile Fog Computing* (MFC) can drastically reduce transmission latency by offloading the tasks onto the edge servers without transferring to remote data centers. However, traditional virtual machines (VM) or containers used in MCC are too heavyweight for resource-constrained environment of edge or fog servers. In this paper, we argue that enhanced unikernel can be used as task runtime in MEC or MFC to efficiently support mobile code offloading. To achieve this goal, we put forward the concept of Rich-Unikernel which aims to support various applications in one unikernel while avoiding their time-consuming recompilation. Following the design of Rich-Unikernel, we implement a not only lightweight but also flexible runtime for offloaded codes, called Android Unikernel, by integrating basic Android system libraries into OSv unikernel. Our experiment shows, compared with VM and container, Android Unikernel introduces much less boot-up delay, memory footprint, image size and energy consumption.

Keywords: Unikernel, Code Offloading, Computation Offloading, Edge Computing, Fog Computing, Library Operating System

1. Introduction

Over the last decades, mobile device processors are becoming more and more powerful, but the increasing speed of performance requirements of mobile applications (such as augmented reality, image processing, 3D games, AI applications) is much faster than the increasing speed of processors' performance, so that many computation-intensive applications may not be performed efficiently on mobile devices themselves. To solve this problem, *Mobile Cloud Computing* [1] comes into being, which uses cloud infrastructure to enhance the performance of mobile devices. In this field, a well-known approach is *computation offloading*, also called *code offloading* [2], which splits mobile application into small tasks, transfers the computation-intensive tasks to cloud to execute, and then fetches results back, thus improving execution speed of applications and reducing energy consumption of mobile devices. Nevertheless, most of the existing related works [3, 4, 5, 6, 7] mainly focus on the implementation of offloading frameworks. It is difficult to response to mobile requests in time, due to the long-distance transmission delay of network, which may results in even worse user experience than local execution, especially when the network environment is poor.

MEC [8][9] and MFC [10], which suggest processing data at the edge of network, are potential to address the issues of network transmission delay, bandwidth cost, as well as data safety and privacy. By offloading tasks onto nearby edge servers instead of remote data centers, network latency can be reduced sharply.

However, unlike traditional cloud infrastructure which is generally made up of resource-abundant data centers, MEC and MFC, emphasizing on real-time computation or in-sute data processing, are mainly based on resource-constrained edge devices such as WiFi access points, home gateways, femtocells, and cellular base stations. Therefore, it places higher requirements on designing the runtime in such devices or edge servers. The most important requirements are less resource overhead (e.g. memory footprint, disk usage and energy consumption), faster boot speed, and easier on-demand deployment. Given the long boot-up delay and heavy resource cost of virtual machine, it is not advisable any more to use VM in MFC and MEC scenarios. As for container [11], although it costs much little resource than virtual machine, it is still not fast enough to startup to provide just-in-time service. Besides, the isolation and security of container are often criticized by industry and academia [12].

Unikernel [13], which fits all the demands above, seems to be a perfect choice. Unfortunately, to use unikernel as runtime to handle mobile offloading requests in MEC or MFC, we still need to overcome several challenges, such as time-consuming recompilation, slow customization, and mobile code supporting (elaborated in Section 2). In this paper, we introduce a new enhanced unikernel runtime to support efficient mobile code offloading in MEC or MFC. We firstly put forward the concept of Rich-Unikernel, aiming to support various mobile applications in one unikernel while avoiding their time-consuming recompilation. Following the design of Rich-Unikernel, we implement Android Unikernel based on OSv unikernel [14]. Our Android Unikernel is a lightweight and flexible runtime that enables to run offloaded mobile codes in the form of unikernel on resource-constrained devices in MEC or MFC. There are two important components in Android Unikernel, DynamicLinker and Libdroid. DynamicLinker is mainly in charge of linking offloaded application codes into unikernel dynamically, so we do not have to generate an unikernel for each application at compile-time. Libdroid is a set of Android libraries used to support offloaded mobile codes in the form of LibOS (Library Operating System)[13][15][16] in Android Unikernel. With the advantages of Android Unikernel, our experimental results show that runtime boot-up delay can be significantly reduced from more than 1 minute in the case of VM and seconds in the case of container (Rattrap [17]) to less than a second, while saving memory footprint, disk usage and energy consumption. Moreover, when deployed to edge servers, the average offloading response latency can be sharply reduced. In summary, this paper makes the following contributions:

- We propose the idea of Rich-Unikernel which is an enhanced unikernel for the scenarios with changing applications. Unlike conventional unikernel customized and compiled for each application, Rich-Unikernel provides a more general runtime for similar applications by integrating basic and common system libraries.

- We design a new runtime, Android Unikernel, following the idea of Rich-Unikernel, for efficient mobile offloading in MEC and MFC. Compared with traditional runtime like Android VM or Android container, our proposed runtime is more lightweight in bootup time, memory footprint, image size and energy consumption, so that it can be used in resource-constrained environment and for just-in-time services.

- We refactor a part of Android system libraries that are necessary for offloaded mobile codes and integrate them with OSv to generate Android Unikernel so as to support Android applications' code offloading. As far as we know, it is the first effort to try unikernel in offloading field.

- We propose a method to rapidly *unikernelize* an application by dynamically linking application code to Android Unikernel instead of recompiling and repackaging application code with system libraries as traditional unikernel does. It significantly decreases the building time of unikernels and makes just-in-time mobile offloading services possible.

The rest of this paper is organized as follows. Section 2 presents the background of mobile code offloading and unikernel along with some related work, what follows is our motivations. Section 3 describes the concept of Rich-Unikernel and the design and architecture of our Android Unikernel based edge runtime. The implementation of Android Unikernel is described in section 4. In section 5, we evaluate our Android Unikernel based runtime and compare its performance and resource overhead to traditional VM-based runtime and container-based runtime. Finally, we conclude this paper, talk about some limitations of our approach and describe our future work in section 6.

2. Background and Motivation

2.1. Mobile Code Offloading

Code offloading is a commonly used approach in the field of MCC, which derives from the initial notion of cyber foraging [18]. The basic idea of code offloading is to offload the computation-intensive tasks (such as a method or a process) inside a mobile application to the cloud to execute, as shown in Figure 1. This architecture contains a client and a server. The client represents for mobile devices which offload computational codes to cloud on demand, and the cloud server is in charge of emulating mobile runtime environment and handling offloading requests within it.

Many researchers have made great contributions to this field and proposed many excellent offloading frameworks. MAUI [3] is the first pioneer in this field, which offloads manually annotated mobile computation-intensive methods to cloud so as to improve applications' performance and save energy

of mobile devices. CloneCloud [4] and COMET [6], which can be regarded as the improved version of MAUI, enable thread-level offloading without manual annotation. Advancing on previous work, ThinkAir [5] focuses on the elasticity and scalability, and it enables parallelizing method-level offloading. And to coordinate task offloading among mobile devices and get offloading results quickly at the same time, [19] proposes an agent-based MCC framework to enable the device to receive offloading results faster by making offloading decision on the agent. Besides these offloading frameworks, many other research works focus more on offloading decisions, such as [20], in which offloading decision is formulated as an NP-hard 0 − 1 nonlinear integer programming problem with time deadline and transmission error rate constraints, making offloading more efficiency and energy-saving. All of these works have contributed greatly to the research of computation offloading.

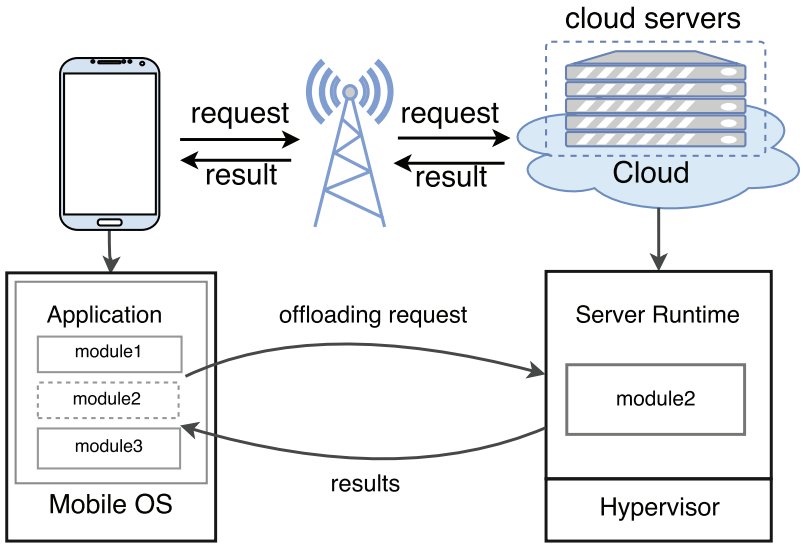


Figure 1: Basic Offloading Architecture

Unfortunately, in traditional Mobile Cloud Computing, all of the offloading frameworks face a performance bottleneck. Since they mainly focus on implementation details of offloading, both network transmission latency and cloud-side overhead are rarely considered. In order to decrease the network latency, Mobile Edge Computing [8][9] and Mobile Fog Computing [10] are proposed. By offloading mobile codes to adjacent edge servers instead of remote data centers, network transmission latency can be significantly reduced

[7], thus greatly improving the QoS (Quality of Service). Based on MFC and MEC, many studies in offloading have been conducted to guarantee the performance of computation offloading, such as [21, 22, 23, 24, 25]. These works, though sharply reduce network latency, place huge burden to edge servers. Since the server runtime is mainly based on traditional VM with a full Android-x86 system running in it, which is too heavyweight for resource-constrained edge servers. Besides, the boot time of VM is too long to provide just-in-time service on demand, so we have to startup servers in advance if we want to guarantee the quality of service (QoS), which, in turn, further increases the resource overhead and energy consumption.

To further reduce the runtime overhead, many lightweight virtualization technologies (such as Container [11]) have been introduced in MFC and MEC. Rattrap [17] uses LXC (Linux Container) to serve as offloading cloud server runtime, greatly reduces overhead at cloud side, compared to VM. ParaDrop [26] introduces a specific edge computing framework, which uses LXC to do resource isolation work and enables third-party softwares to be installed directly on the gateway, thus introducing less overhead at edge.

2.2. Library Operating System: Unikernel

Unikernel is a highly-specialized single-address-space immutable disk image, constructed by linking an application only with its necessary libraries at compile-time through library operating systems, of which all the services, from device drivers to schedulers to network stack, are implemented in the form of libraries that can be linked directly with applications [13]. Compared to VM and container, unikernel has the advantages of small, fast and secure. As illustrated in [27], Unikernel is suitable for IoT (Internet of Things) edge, due to its small footprint and flexibility. [28] implements unikernel-based elastic CDNs for video streaming in distribution networks, significantly improving the performance of video delivery. [29] implements a NFV platform with MirageOS unikernels, which does not rely on current cloud orchestration or SDN. In this paper, we try to use unikernel as server runtime to handle mobile code offloading requests in MFC and MEC scenarios, while saving resources at edge servers.

Figure 2 shows the difference among container, VM and unikernel. Container performs much better than VM in resource overhead and bootup speed, since it shares kernel with host OS at the cost of less isolation and less security. On the contrary, VM is good at isolation and security but too heavyweight. Imagining that you just want to run a very simple application, but

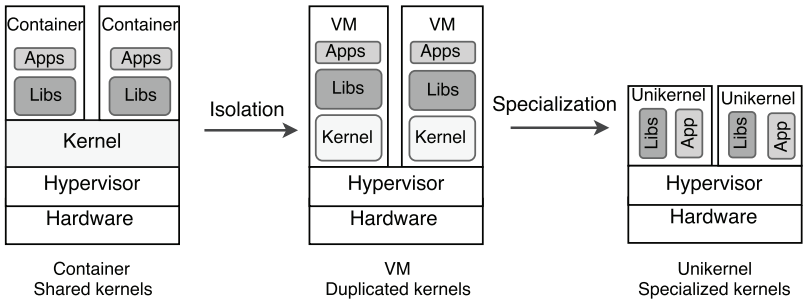


Figure 2: Differences among Container, Virtual Machine and Unikernel

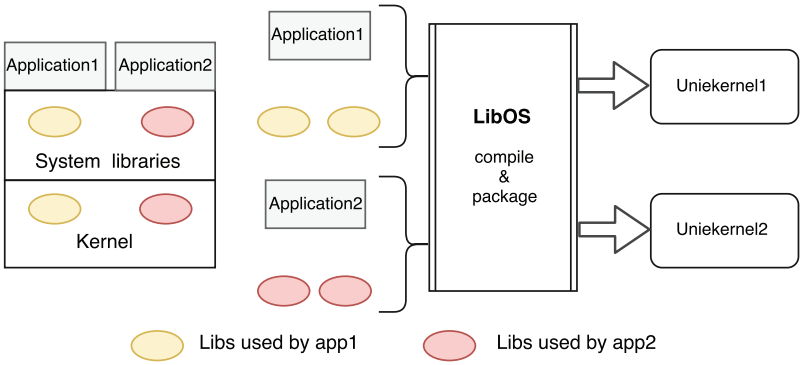


Figure 3: Building Process of Unikernel

you have to run a whole operating system and pay for its huge cost, even though most of system functions are not used. Unikernel eliminates the redundancy and complexity of virtual machine and container by specializing an application into a standalone image along with its necessary system libraries, combining the advantages of virtual machine and container while eliminating their disadvantages.

As figure 3 shows, when building an application into an unikernel, only the necessary system libraries and kernel functions used by the application are compiled into the unikernel. Therefore, unikernel image is very small. It usually takes only about KB to MB disk space. Besides, both application codes and kernel codes run in the same address space without any system call and context switching. Therefore, codes in unikernel can run in the most efficient way.

To summarize, unikernel has a lot of advantages over virtual machine and container. Firstly, small image size. Its image is one to two orders of mag-

nitude smaller than virtual machine image and container image. Secondly, fast boot. Unikernel can boot in milliseconds due to it small image. Thirdly, small footprints. Unikernels are often orders of magnitude smaller than traditional OS deployments. Fourthly, improved security. Unikernel reduces the amount of code deployed, which reduces the attack surface. Finally, easy to deploy. Since everything an application needs have been packaged into the image, it can be boot up directly on any kind of hypervisor or even bare metal without any additional configuration. All of the above advantages make us believe that unikernel is a potential candidate to be used as server runtime in MFC and MEC.

2.3. Motivation

Despite the advantages of unikernel, there still exists some challenges when using unikernel in MEC or MFC as server runtime for mobile code offloading. Firstly, unikernel is immutable disk image specialized for a particular application (as shown in Figure3), which means that we have to generate an unikernel for each application. However, offloading scenarios often meet various applications, so immutable unikernel is improper in code offloading scenarios, and we need a new kind of unikernel each of which can serve a series of applications. That is, we need to strengthen unikernel's versatility properly. Secondly, since the building process costs seconds of time, it is inadvisable to compile offloaded application codes into unikernel after offloading requests arrive. We must find a method to *unikernelize* an application as soon as possible. Finally, we need an Android LibOS to convert Android application codes into unikernel, but existing LibOS used to create unikernel are mainly based on traditional Linux, which does not provide support for Android features. So our last challenge is to make unikernel enable offloaded Android application codes.

To use unikernel as edge runtime for mobile code offloading, we must address the above three challenges. In next section, we will explain in detail how we address these issues.

3. System Design

3.1. Overview

To use unikernel as our edge server runtime, we take a series of measures to overcome the three challenges mentioned in motivation. For the first challenge, we devise Rich-Unikernel, a kind of more general purpose unikernel

than conventional unikernel, so that it does not need to be customized for each application. And for the second challenge, we propose a method to immediately *unikernelize* an application by pre-building an Android Unikernel and dynamically linking the application into Android Unikernel at runtime, where Android Unikernel is the instantiation of Rich-Unikernel in the mobile offloading scenario. By this way, we can eliminate the time-consuming compiling process after an offloading request arrives, thus greatly reducing the response latency. Finally, for the last challenge, we port a part of essential Android libraries to pre-built Android Unikernel to support offloaded Android application codes.

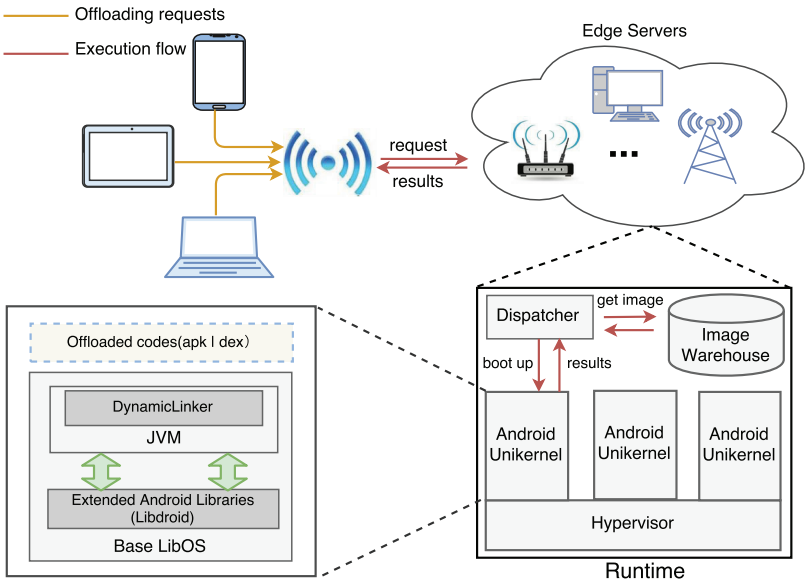


Figure 4: Unikernel-Based Edge Computing Architecture

Through above solutions, we finally achieve the goal of using unikernel as runtime to handle various mobile offloading requests on edge servers. We design our Android Unikernel based edge computing platform for Android code offloading in MFC or MEC. Figure 4 shows the overview of system architecture. Since there are already many brilliant offloading frameworks (mentioned in Section 2), we do not intend to talk too much about it. Our work mainly focus on server runtime in edge devices. The runtime contains three parts: Image Warehouse, Dispatcher and virtual servers (running Android Unikernels). The Image Warehouse is where pre-built Android

Unikernels are stored. The Dispatcher is in charge of handling mobile offloading requests, and starting up a virtual server for each request. And the servers are virtual machines running Android Unikernels fetched from Image Warehouse. Besides basic system libraries and a JVM runtime provided by LibOS, Android Unikernel contains two important parts, DynamicLinker and Libdroid. DynamicLinker is used to quickly unikernelize offloaded codes without on-line compilation. Libdroid is responsible for providing Android system call to JVM so that JVM can interpret those bytecodes which invoke Android APIs.

In what follows, we will further explain the methods or components used in Android Unikernel. We firstly explain the notion of Rich-Unikernel, then we will explain how to *unikernelize* applications based on our pre-built Android Unikernels, and how to ensure Android application codes to run correctly in Android Unikernel.

3.2. Rich-Unikernel

We define Rich-Unikernel as a kind of unikernel that is more general than conventional unikernel. Conventional unikernel is immutable and customized for each application. That is, once the application changes, we have to recompile and reconstruct the application into a new unikernel, even if the change is very small. However, Rich-Unikernel is not specialized for a particular application and it can be regarded as a base unikernel for a series of applications. All of the system libraries needed by these applications have already been packaged into the base unikernel, so it is able to run different applications (one application at a time).

Figure 5 shows the difference between conventional unikernel build-process and our Rich-Unikernel based build-process. In our method, Rich-Unikernel is compiled offline and constructed by LibOS. This process is similar to conventional unikernel build-process (shown in Figure 3), what different are the libraries. For Rich-Unikernel, it contains the common libraries used by a series of applications, while conventional unikernel only contains the libraries used by a particular application. Besides, Rich-Unikernel does not contain application codes at first, and it just has a tool called Application Loader which is able to load application codes into unikernel at runtime. When a Rich-Unikernel boots up, Application Loader starts automatically and waits for offloading request. Once an offloading request arrives, Application Loader links its corresponding offloaded codes into base unikernel online, thus the

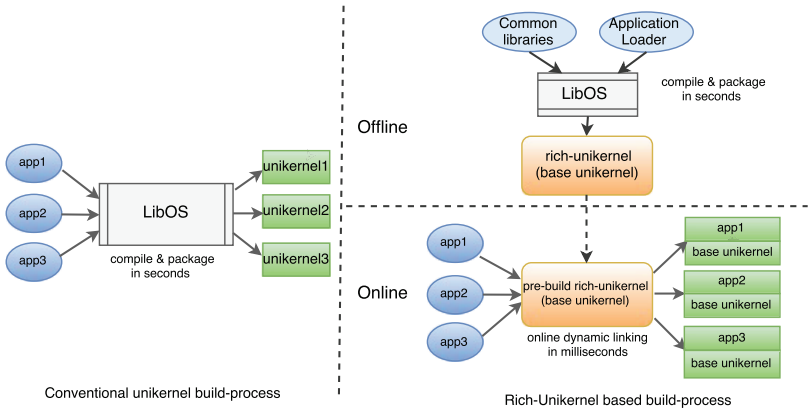


Figure 5: Difference Between Conventional Unikernel and Rich-Unikernel

application is unikernelized without any compilation. Finally, it uninstalls the application automatically after its running. By this way, we do not have to recompile application codes and its system libraries. Instead, we can pre-build some Rich-Unikernels and dynamically change a Rich-Unikernel into the unikernel corresponding to a particular application when a request arrives, which significantly reduces response latency of our system.

Although Rich-Unikernel breaks the original principle of unikernel such as immutable runtime and specialization for each application, and introduces a little bigger size of image, we think the design of Rich-Unikernel is valuable for the scenarios with changing applications like mobile code offloading where a general lightweight server runtime is needed and its response latency is of importance.

3.3. Unikernelization

In this section, we describe how to *unikernelize* an application by dynamically linking application to a pre-built Rich-Unikernel. There exists many LibOS projects (such as OSv [14]), in which application module is independent from other kernel modules, so the fastest way to convert an application into its corresponding unikernel is to replace the application module without changing the other kernel modules, thus saving the time of recompiling these modules.

So the problem becomes how to change the application module while eliminating unnecessary recompiling works. Since our basic LibOS contains

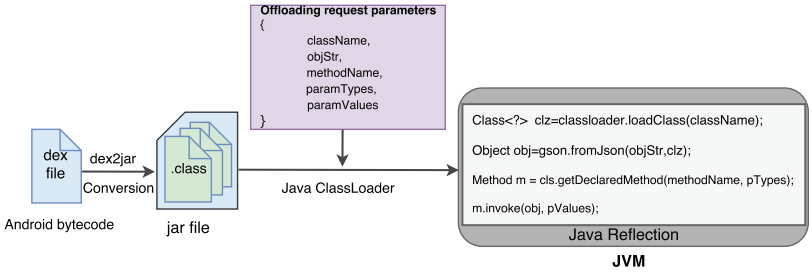


Figure 6: Workflow in DynamicLinker

a JVM runtime, we come up with *Java Reflection*[1], through which we can dynamically load applications at runtime. However, before this operation, we must do a bytecode convertion work in advance. Because the offloaded codes are transferred in the forms of apk file or dex file, where dex is the format of Android bytecode, this kind of bytecode always runs in Dalvik or ART (both Dalvik and ART are Android version of the JVM but structurally different from JVM), and JVM can not load classes from these bytecodes directly.

Therefore, we provide a tool named DynamicLinker, which is an implementation of Application Loader. Figure 6 shows the sketch map of workflow in DynamicLinker. Firstly, it converts Android bytecode (.dex) to Java bytecode (.jar) by using *dex2jar*[2], before loading Java classes into JVM. After the byecode conversion process, we can dynamically load the Java classes specified by the offloading requests, and reconstruct the object instance according to the Json string in request parameters. Finally, we get the specified method that needs to be executed, and then invoke it through Java Reflection. Thus we achieve the goal of dynamically loading and running offloaded Android codes in pre-built Android Unikernels.

Although the byecode conversion process is an extra step compared to VM-based runtime and container-based runtime (where a full Android operating system is prepared), it is still significant. On one hand, this process is very fast, almost done in a few milliseconds. On the other hand, if we do not convert Android bytecodes into Java bytecodes, we have to provide a Dalvik or ART VM in Android Unikernel to run these codes. However, as we mentioned before, both Dalvik and ART are actually JVM for Android, and

[1]https://docs.oracle.com/javase/tutorial/reflect/
[2]https://github.com/pxb1988/dex2jar

they have a lot of functions in the same way as JVM. So it is too redundant to add Dalvik or ART to Android Unikernel where JVM already exists.

3.4. Libdroid: Extended Android Libraries

Android Unikernel can load and verify Java classes from Android bytecodes through DynamicLinker. However, the offloaded codes may involve some classes and APIs of Android OS, such as Bitmap and Logger, and JVM in Android Unikernel can not correctly interpret this part of bytecodes. That is, there should be an Android library to support such offloaded codes in Android Unikernel.

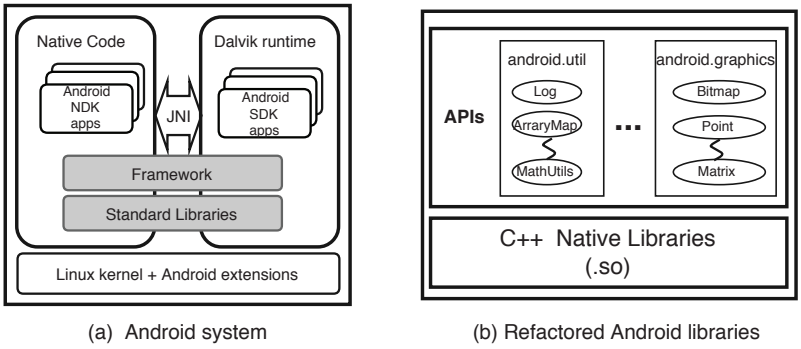


Figure 7: System Library Refactoring

To address the above issue, we port a series of necessary Android system libraries to Android Unikernel as an individual module called Libdroid. As shown in Figure 7(a), Android system contains four layers: Application, Framework, Libraries and Linux kernel. Application layer mainly includes some Android applications, framework layer mainly contains Android APIs for developing Android applications, library layer is made up of all kinds of system libraries and native C++ libraries, and kernel layer is based on Linux kernel, including all kinds of Android hardware drivers. As shown in Figure 7(b), our refactoring work mainly focuses on the framework layer and system library layer. The API layer of Libdroid corresponds to the framework layer in Android system, including those components which may be used by offloaded codes. These APIs are mainly data structure relevant APIs, such as Bitmap, Bundle, Matrix. And the library layer mainly contains low-level C++ native libraries that are needed by upper APIs. Other system libraries, such as *libc*, are already included by LibOS. We can reuse these libraries by

properly modifying them, so we do not have to port this part of codes. As for the kernel layer, most Android drivers for hardware devices, like Camera, USB, WIFI, Bluetooth and all kinds of sensors, are useless in offloading scenarios. Since there is no such device on the server side and codes that can be offloaded must not depend on these devices. Particularly, Binder, which is the Android interprocess communication mechanism, is also unnecessary for Libdroid, because current unikernels do not support multiprocess and all the offloaded codes run in a single process without any interprocess communication. Therefore, we do not care the drivers in kernel layer. As a result, Libdroid only contains needed API frameworks and their low-level C++ implementations, leading to its small size. Therefore, we can implement our Android Unikernel without vastly increasing its image size.

After the library extension, what we need to do is to integrate these libraries with LibOS so that we can package them into unikernel through LibOS's compiling system. During the execution of offloaded codes, when JVM meets a function call to Android system API, we change it to call the corresponding implementation in the extended libraries inside the unikernel. So these codes can run correctly in Android Unikernel.

3.5. Image Warehouse

Since building an unikernel image needs seconds of time, in order to ensure real-time response to mobile offloading requests, we prepare some unikernel images in advance by the Image Warehouse which is a repository storing pre-built Android Unikernel images. Once an offloading request arrives, the Dispatcher can immediately fetch an Android Unikernel from Image Warehouse and start it up, thus saving the time of building the unikernel. And when a server finishes handling its offloading tasks, the image used by it will also be restored in Image Warehouse.

In case of a sudden spike in traffic occurs, Image Warehouse is configured with the ability of auto-scaling. It can configure and deploy copies of existing images to serve the demands. This auto-scaling happens so quickly that an incoming connection can trigger the creation of new server and the new server can then handle that request immediately. When the demand dies down again, Image Warehouse removes redundant images again so as to save limited storage space. With this ability, we can be more elastic, raising and lowering capacity to precisely meet demand and therefore only spending what we actually need when we really need it.

4. Implementation

We implement the prototype of our Android Unikernel based offloading system on a personal desktop which acts as edge server. It is equipped with one two-core Intel(R) Core(TM) i5-4590 3.30Ghz processor, 4GB DRAM and 300GB HDD, running Ubuntu16.04 LTS.

In our implementation, we choose OSv[3] [14] as basic LibOS. There are several open source LibOS projects[4], such as MirageOS, IncludOS, LING and so on. We finally choose OSv mainly because of two reasons. Firstly, language compatibility issues. Since Android is compatible with most Java APIs and our goal is to run offloaded Android application codes in unikernel, by using OSv which provides a JVM runtime, we can save a lot of work on handling language compatibility problems that many other unikernels may face [13][30]. Secondly, in OSv unikernel, application module is completely detached from kernel modules, and this gives us a chance to add or delete the application part without affecting kernel parts, which provides the basis of dynamically linking applications and easily adding extended Android libraries to unikernel. Based on the above two reasons, we think that OSv is the most suitable LibOS to implement our Android Unikernel.

For the offloading framework, we choose the framework of ThinkAir [5] and modify it according to our runtime. And the extended Android libraries (Libdroid) are ported from Android-x86 5.1-rc1. We consolidate OSv with these Android libraries and modify its compiling system. When building an Android Unikernel, Libdroid is also packaged into the disk image along with Dynamiclinker. When an Android Unikernel boots up, the Dynamiclinker in it runs automatically and links the application received from dispatcher, and then runs the application codes in JVM.

Our project source code is publicly available online at `https://github.com/cgcl-codes/Libdroid`. Since our project is based on OSv unikernel, you should firstly install OSv (see *osv.io* for detailed steps), and then follow the steps to build Android Unikernel.

[3]http://osv.io/

[4]http://unikernel.org/projects/

5. Evaluation

5.1. Experiment Setup

In this section, we compare Android Unikernel based server runtime with two other server runtimes:

- **VM-based Server Runtime**: This platform is the one that ThinkAir [5] uses, which runs a full Android-x86 system in VirtualBox. Each VM is equipped with 1 vCPU and 600MB of memory.
- **Container-based Server Runtime**: Rattrap [17], which runs a virtual Android runtime in LXC (called Cloud Android Container) by extending OS kernel with Android drivers on demand.

And for the offloading workloads, we choose four typical computation-intensive Android applications of different categories, which have been widely used in previous researches [6, 22], as our benchmark workloads.

- **ChessGame**: An interactive Android chess game based on CuckooChess Engine, which ranks top 200 in Computer Chess Rating Lists 40/40[5]. It represents those computational offloading workloads with intensive network communications.
- **FaceDetect**: A face recognition application, based on Google FaceDetector which is actually implemented by Java Native Interface (JNI) written in C++. We choose this application to represent the computation-intensive image processing workloads along with file transfer.
- **VirusScan**: A virus scanning application which scans the filesystem and checks each file with a virus database. It represents those computation tasks with intensive I/O operations.
- **Linpack**: The most popular benchmark for testing the floating-point performance of computer systems, by using gaussian elimination method to solve linear algebraic equations. Here we choose it as a delegate of purely computational workloads.

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

Type	Memory Footprint	Image Size	Bootup Energy Consumption
Android-x86 VM	516MB	1.4GB	334.2J
Cloud Android Container	128MB	1.02GB	49.4J
Android Unikernel	52MB	70MB	15.6J

Table 1: System Overhead Comparison

All of these four applications are installed on a HuaWei Honor V9 smartphone, which acts as the offloading client. And the server machine is what we mentioned in Section 4. Both client and server are under the LAN WiFi network situation. Next, we compare Android Unikernel with the other two server runtimes in four aspects: system overhead, boot time, performance and energy consumption.

5.2. System Overhead

For system overhead, we mainly focus on memory footprint, disk usage and bootup energy consumption, since these are the most limited resources at edge servers compared to data centers. Table 1 shows the overhead of three kinds of instance (Android-x86 VM, LXC-based Android Container and our Android Unikernel).

Memory footprint is an important indicator to evaluate a system, especially in MFC and MEC scenarios where memory resource is not as abundant as data centers. Less memory footprint means that we can startup more instances at the same cost of physical resources. In Table 1, the memory footprint is actually the minimal memory size we need to specify before starting a VM or a container. During our test, Android-x86 VM requires at least 516MB memory to boot up, and LXC-based Android Container requires at least 128MB. However, our Android Unikernel needs only 52MB to boot up, which is 10% of VM's footprint and less than half of container's footprint. Therefore, our Android Unikernel based runtime has great advantage over conventional VM-based runtime and container-based runtime in terms of memory overhead.

As for disk usage, it is measured by image size. VM contains a whole kernel and runs a full operating system. As a result, it costs too much disk space. For Rattrap, since it still runs a full Android environment in Linux Container, its disk usage is still too huge when booting up a container

instance, while Rattrap does a lot of work to share a plenty of common system libraries between containers. However, instead of running a full operating system, our Android Unikernel only contains a basic Java runtime and the ported Android libraries needed by offloaded applications, leaving out most needless kernel codes. As a result, it cost only about 70MB disk space, saving more than 90% disk space compared to Android VM and Android container. Besides, this advantage gives us chance to quickly transmit and deploy unikernel images between edge severs through network.

Bootup energy consumption is the average quantity of energy used by each instance to bootup. We use the method in [31] to evaluate the energy consumption of each instance. Here we only consider the energy consumption of bootup, and the energy consumption during applications' execution is highly related to application workloads, which we will describe later. From the table we can see our Android Unikernel based runtime is competitive in energy saving. VM costs a lot of energy mainly because of its huge image and long bootup-duration with I/O operations to load disk image. Container performs much better since it shares many resources with host OS and bootup faster. Our Android Unikernel, due to its small image and fast boot, costs the least energy.

5.3. Boot Time

We have already know from Table 1 that our Android Unikernel is much smaller than Android-x86 VM image and Android Container, which means it can be loaded from disk in less time. As a result, it can be booted up very quickly. Figure 8 compares average boot time of Android Unikernel against Android Container and Android-x86 VM, where both unikernel image and Android-x86 VM image are booted in KVM. In this figure, the y-coordinate represents the boot time of each instance, where the boot time is measured from startup to the point when runtime environments finish startup and be connected to the Dispatcher. And the x-coordinate represents the maximal memory size that can be used by each instance, which is specified manually through hypervisor or Cgroup (for Android Container) before startup.

As shown in Figure 8, in terms of boot time, Android-x86 VM is not competitive at all, because it takes tens of seconds to startup. As for Android container, it reduces boot time to about 2s by introducing OS-level virtualization, but it is still not fast enough to provide just-in-time services. By contrast, an Android Unikernel instance takes only less than 1s to boot up. Besides, its startup speed is the least affected as memory size increases from

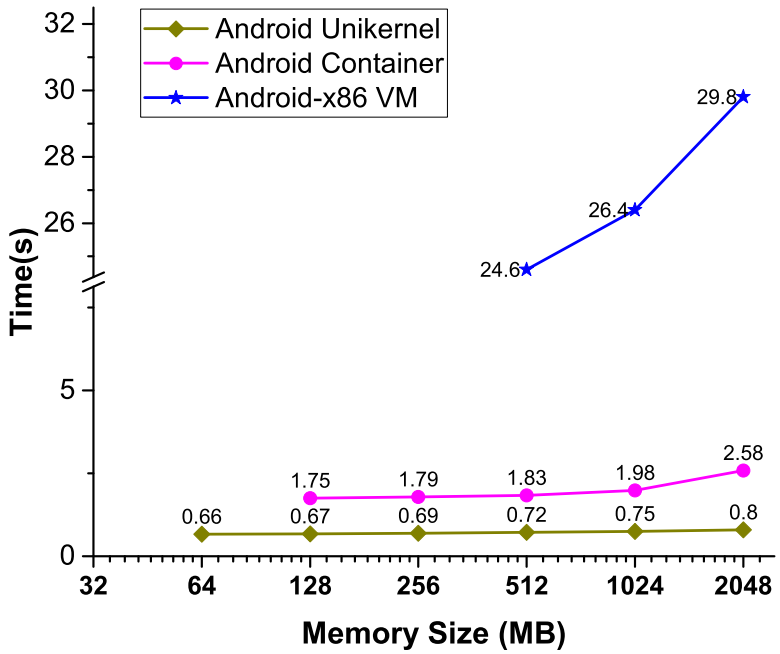
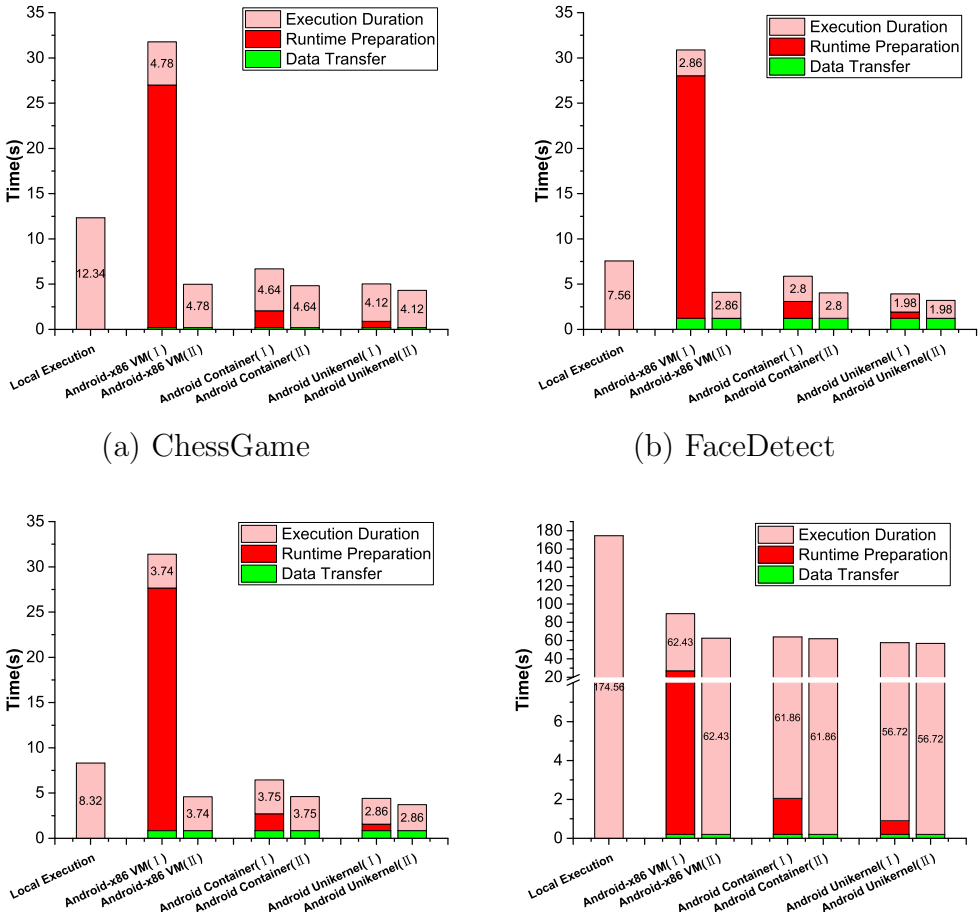


Figure 8: Boot Time Comparison

64MB to 2048MB. Such fast boot time is sufficient to ensure the real-time response to most offloading requests in MFC and MEC scenarios.

5.4. Performance

In this section, we compare the performance of the benchmark applications in the cases of Android Unikernel and other two traditional runtimes. Figure 9 shows the average performance of three different runtimes when addressing different workloads. We compare applications' execution time of running locally at smartphone (Local Execution) to the execution times of offloading execution with three different server runtimes. When using offloading execution, the application's execution time is divided into three parts: *Data Transfer*, *Runtime Preparation* and *Execution Duration*, where *Data Transfer* represents the time of transferring codes, offloading parameters and necessary files, *Runtime Preparation* represents the average boot time of VM or container, and *Execution Duration* is the actual code-execution time. For



(a) ChessGame (b) FaceDetect

(c) VirusScan (d) Linpack

Figure 9: Performance Comparison of Four Different Types of Applications

ChessGame, *Execution Duration* is the average time of searching each step; for FaceDetect, it is the average time of identifying the faces in a picture; for VirusScan, it represents the time of scanning files and checking if any virus exists; and for Linpack, it is the average time of solving a set of linear equations.

It is worth mentioning that only the first offloading request of each TCP connection has to face a cold start of VM or container. The VM or container will keep alive until the corresponding connection be released. Therefore, we draw two series of histogram for each server runtime. The first histogram (I) represents the average performance of the first request, and the rest is represented by the second histogram (II). From Figure 9, we can easily gain three observations:

- In terms of response performance, our Android Unikernel based runtime is outstanding under both cold start and non-cold start situations, with average 60% speed-up over local execution.
- When addressing short-time computation tasks, such as ChessGame and FaceDetect, VM may performs even worse than local execution due to it long boot-up delay.
- Besides the boot time, Android Unikernel is also better at Execution Duration (as marked in the pink area in the figure). As codes in unikernel run in the same address space without overhead of context switching and system call, code execution is more efficient.

All in all, Android Unikernel based edge runtime is more efficient and suitable than traditional VM-based runtime and container-based runtime to be used under MFC and MEC scenarios to handle offloading requests.

5.5. Energy Consumption of Different Workloads

Figure 10 compares the energy consumption of the four benchmark application workloads in different runtimes. For Chessgame, y-coordinate represents the average energy consumed to search each step; for FaceDetect, y-coordinate represents the average energy consumption of identifying the faces in a picture; for VirusScan, y-coordinate is the average energy consumption to scan files and check if any virus exists; and for Linpack, it is the average energy consumption of solving a set linear equations.

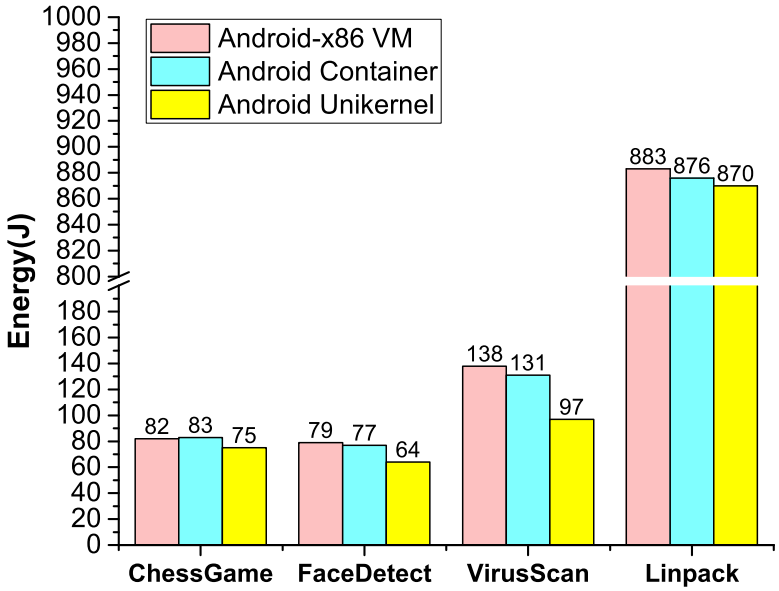


Figure 10: Energy Consumption of Different Workloads

From Figure 10, we can easily draw a conclusion that our Android Unikerne is more energy-efficient than the other two runtimes, especially when addressing computational tasks accompanied with intensive I/O operations, such as VirusScan, which has more I/O system calls. In VM and container, system calls are all accompanied with system overhead. However, codes in unikernel run in the same address space without system call and overhead of context switching, so it costs much less energy.

6. Conclusion and Future Work

In this paper, we present Android Unikernel, a lightweight runtime designed for mobile computation offloading under MFC and MEC scenarios. By pre-building base Android Unikernel and running offloaded mobile codes in them, the proposed runtime can not only effectively reduce response latency, but also reduce resource overhead. Our evaluation shows that our Android Unikernel performs much better than traditional VM based run-

time and container-based runtime in boot time, system overhead (memory footprint, disk usage, bootup energy consumption) and energy consumption, which makes Android Unikernel suitable for Mobile Fog Computing and Mobile Edge Computing in handling real-time offloading tasks.

Although our Android Unikernel has many advantages compared to virtual machine and container, limitations still exist. One of the most obvious limitation is multi-process applications are not allowed, which is the inherent limitation of unikernels. Therefore, those codes that need to fork new process can not be offloaded and run in Android Unikernel. In the future, We plan to support multi-process applications according to the method mentioned in [32]. Besides, our Android Unikernel is just a specific instance of Rich-Unikernel in the scenario of mobile code oofloading. But in our opinion, the idea of Rich-Unikernel is also applicable in other scenarios (like Internet of Things (IoT), mobile app testing) where a more general unikernel is needed. So our next future work is to broaden the idea of Rich-Unikernel in more scenarios, instead of limiting its usage to mobile code offloading.

7. Acknowledgements

This research is supported by National Key Research and Development Program under grant 2016YFB1000501, National Science Foundation of China under grants No. 61732010 and 61472151, and the Fundamental Research Funds for the Central Universities under grants 2016YXZD016 and 2015TS067.

References

[1] N. Fernando, S. W. Loke, W. Rahayu, Mobile cloud computing: A survey, Future generation computer systems 29 (1) (2013) 84–106.

[2] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, R. Buyya, Mobile code offloading: from concept to practice and beyond, IEEE Communications Magazine 53 (3) (2015) 80–88.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: Making smartphones last longer with code offload, in: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10), ACM, 2010, pp. 49–62.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, Clonecloud: Elastic execution between mobile device and cloud, in: Proceedings of the Sixth Conference on Computer Systems (EuroSys'11), ACM, 2011, pp. 301–314.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, X. Zhang, Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading, in: Proceedings of the 31st International Conference on Computer Communications (INFOCOM'12), IEEE, 2012, pp. 945–953.

[6] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, X. Chen, Comet: Code offload by migrating execution transparently, in: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), USENIX, 2012, pp. 93–106.

[7] M. Satyanarayanan, P. Bahl, N. Davies, The case for vm-based cloudlets in mobile computing, IEEE Pervasive Computing 8 (4) (2009) 14–23.

[8] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, V. Young, Mobile edge computing-a key technology towards 5g, ETSI White Paper 11 (11) (2015) 1–16.

[9] M. Satyanarayanan, The emergence of edge computing, Computer 50 (1) (2017) 30–39.

[10] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe, Mobile fog: A programming model for large-scale applications on the internet of things, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing (SIGCOMM'13), ACM, 2013, pp. 15–20.

[11] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, in: Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07), ACM, 2007, pp. 275–287.

[12] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, C. A. De Rose, A performance isolation analysis of disk-intensive workloads on container-based clouds, in: Proceedings of the

23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'15), IEEE, 2015, pp. 253–260.

[13] D. S. Charalampos Rotsos, Unikernels: Library operating systems for the cloud, ACM Sigplan Notices 48 (4) (2013) 461–472.

[14] A. Kivity, D. L. G. Costa, P. Enberg, Osv: Optimizing the operating system for virtual machines, in: Proceedings of 2014 USENIX Annual Technical Conference (ATC'14), USENIX, 2014, p. 61.

[15] D. R. Engler, M. F. Kaashoek, J. O'Toole, Exokernel: An operating system architecture for application-level resource management, ACM Sigops Operating Systems Review 29 (5) (1995) 251–266.

[16] I. M. Leslie, D. Mcauley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. Hyden, The design and implementation of an operating system to support distributed multimedia applications, Selected Areas in Communications IEEE Journal on 14 (7) (1996) 1280–1297.

[17] S. Wu, C. Niu, J. Rao, H. Jin, X. Dai, Container-based cloud platform for mobile computation offloading, in: Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS'17), IEEE, 2017, pp. 123–132.

[18] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, H. Yang, The case for cyber foraging, in: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, ACM, 2002, pp. 87–92.

[19] Z. Kuang, S. Guo, J. Liu, Y. Yang, A quick-response framework for multi-user computation offloading in mobile cloud computing, Future Generation Computer Systems 81 (2018) 166–176.

[20] K. Liu, J. Peng, H. Li, X. Zhang, W. Liu, Multi-device task offloading with time-constraints for energy efficiency in mobile cloud computing, Future Generation Computer Systems 64 (2016) 1–14.

[21] X. Jin, Y. Liu, W. Fan, F. Wu, B. Tang, Multisite computation offloading in dynamic mobile cloud environments, Science China Information Sciences 60 (8) (2017) 089301.

[22] Y. Wang, M. Sheng, X. Wang, L. Wang, J. Li, Mobile-edge computing: Partial computation offloading using dynamic voltage scaling, IEEE Transactions on Communications 64 (10) (2016) 4268–4282.

[23] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, Y. Zhang, Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks, IEEE Access 4 (99) (2017) 5896–5907.

[24] Y. Mao, J. Zhang, K. B. Letaief, Dynamic computation offloading for mobile-edge computing with energy harvesting devices, IEEE Journal on Selected Areas in Communications 34 (12) (2016) 3590–3605.

[25] X. Tao, K. Ota, M. Dong, H. Qi, K. Li, Performance guaranteed computation offloading for mobile-edge cloud computing, IEEE Wireless Communications Letters PP (99) (2017) 1–1.

[26] D. Willis, A. Dasgupta, S. Banerjee, Paradrop: A multi-tenant platform to dynamically install third party services on wireless gateways, in: Proceedings of the 9th ACM Workshop on Mobility in the Evolving Internet Architecture (MobiArch'14), ACM, 2014, pp. 43–48.

[27] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, J. Ott, Consolidate iot edge computing with lightweight virtualization, IEEE Network.

[28] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, F. Huici, Unikernels everywhere: The case for elastic cdns, in: Proceedings of the 12th ACM Sigplan/Sigops International Conference on Virtual Execution Environments (VEE'17), ACM, 2017, pp. 15–29.

[29] Nfv platforms with mirageos unikernels, `http://unikernel.org/blog/2016/unikernel-nfv-platform`, 2017.

[30] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad, K. Begnum, Includeos: A minimal, resource efficient unikernel for cloud services, in: Proceedings of the 7th International Conference on Cloud Computing Technology and Science (CloudCom'15), IEEE, 2015, pp. 250–257.

[31] W. Jiang, F. Liu, G. Tang, K. Wu, H. Jin, Virtual machine power accounting with shapley value, in: Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS'17), IEEE, 2017, pp. 1683–1693.

[32] M. Kanatsu, H. Yamada, Running multi-process applications on unikernel-based vms, SOSP.