Container-Based Customization Approach for Mobile Environments on Clouds

Jiahuan Hu, Song Wu(✉), Hai Jin, and Hanhua Chen

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan 430074, China
{zjsyhjh,wusong,hjin,chen}@hust.edu.cn

Abstract. Recently, mobile cloud which utilizes the elastic resources of clouds to provide services for mobile applications, is becoming more and more popular. When building a *mobile cloud platform* (MCP), one of the most important things is to provide an execution environment for mobile applications, e.g., the Android mobile *operating system* (OS). Many efforts have been made to build Android environments on clouds, such as Android *virtual machines* (VMs) and Android containers. However, the need of customizable Android execution environments for MCP has been ignored for many years, since the existing OS customization solutions are only designed for hardware-specific platforms or driver-specific applications, and taking little account of frequently-changing scenarios on clouds. Moreover, they lack a unified method of customization, as well as an effective upgrade and maintenance mechanism. As a result, they are not suitable for varied and large-scale scenarios on clouds. Therefore, in this paper, we propose a unified and effective approach for customizing Android environments on clouds. The approach provides a container-based solution to custom-tailor Android OS components, as well as a way to run Android applications for different scenarios. Under the guidance of this approach, we develop an automatic customization toolkit named AndroidKit for generating specific Android OS components. Through this toolkit, we are able to boot new Android VM instances called AndroidXs. These AndroidXs are composed of OS images generated by AndroidKit, which can be easily customized and combined for varied demands on clouds.

Keywords: Mobile cloud · Execution environment · Android · Container-based customization approach · AndroidKit · AndroidX

1 Introduction

Currently, mobile cloud, which leverages elastic resources of cloud platform to provide services for mobile applications, is becoming more and more attractive. There are many scenarios depend on MCP for different requirements. Mobile computation offloading [1–3], which is able to offload parts of workloads to cloud, exploits rich computing resources of cloud platform to enhance performance of

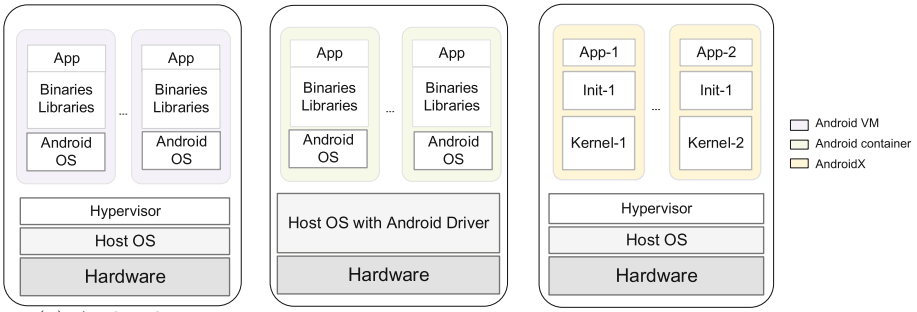
S. Li (Ed.): GPC 2018, LNCS 11204, pp. 155–169, 2019.
https://doi.org/10.1007/978-3-030-15093-8_11

mobile applications and reduce power consumption of mobile devices. And cloud-based mobile testing [4], which uses elastic cloud infrastructure to test mobile applications for different application requirements, has been widely used.

Many challenges need to be faced when building a practical MCP. One of the typical challenges is how to build a mobile execution environment, e.g., the Android mobile OS. Technically speaking, Android OS is not an ordinary Linux distribution although it is built on top of the standard Linux kernel. There are many differences between Linux and Android. For example, Linux uses X11 or Wayland to run a GUI, but Android uses SurfaceFlinger. What's more, Android relies on the special kernel features (e.g., Binder IPC subsystem [5]), which simply do not exist on Linux. Therefore, it is a challenging job to launch Android applications on MCP, since the vast majority of MCPs are currently based on ordinary GNU/Linux distributions.

Using VMs as the MCP execution environments is a practical solution. Projects like shashlik [6] or genymobile [7] use an emulator, which is essentially a VM, to run the Android environment. The emulator creates an entire emulated system with independent kernel to provide rich functionality. By targeting an emulator, they avoid the hardware compatibility problems, but it causes a lot of resource costs since each VM runs a full copy of an OS and suffers high virtualization overhead. In contrast to this, containers (e.g., Docker [8], LXC [9], and rkt [10]), as the center of lightweight virtualization technologies, have significantly lower overhead when compared to VMs. However, because of the shared kernel mechanism, many efforts need to be made when using containers as MCP runtime environments for running mobile applications. Anbox [11] puts a full Android-x86 [12] OS into a LXC container and runs the Android-x86 OS under the same kernel as the host OS does. It communicates with the host system by using different pipes and sends all hardware access commands by reusing what Android implements within the QEMU-based emulator. Rattrap [13] is a container-based cloud platform. It provides an on-demand execution environment for running Android codes through *Cloud Android Container* (CAC), which runs an Android-x86 OS inside a LXC container with dynamic driver extensions. However, the common drawback of Anbox and Rattrap is their hardware compatibility problem, which results in only a handful of Android applications can be run normally.

Another challenge is the unified customization model for frequently-changing usage scenarios on mobile clouds. The need for customizable OS arises when the existing OS components can not match the specific use cases (such as resource-constrained hardware platforms and driver-specific applications). Many research works have been made on exploring the customization of OS. Exokernel [14] and Nemesis [15] customize OS by restructuring the OS with the target application into a set of libraries. Think [16] provides a highly flexible programming model for building flexible OS kernels from components, and [17] designs a system that has the ability to guide Linux images customization for scientific applications. The common problem of these OS customization works [14–17] is the lack of a unified customization model and an effective upgrade and maintenance mechanism, since



(a) Android VM (b) Android container (c) AndroidX

Fig. 1. The difference between Android VM, Android container, and AndroidX

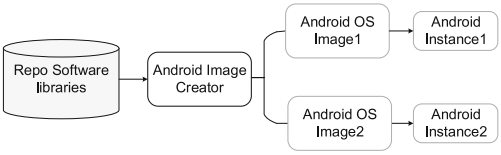
they are only designed for a particular type of platform or device and do not focus on the various frequently-changing scenarios on clouds. In other words, they are inappropriate for varied demands on clouds.

According to the above analysis, in this paper, we mainly address two challenges mentioned above. Inspired by the previous OS customization works, we present a customizable Android execution environment called AndroidX. As Figs. 1(c) and 2(b) show, AndroidXs are built from system images and run with them. From Fig. 1(c), we can find that one of the AndroidX instances is launched with the Kernel-1 image, and another is launched with the Kernel-2 image, but they share the Init-1 image. This kind of combination effectively improves the utilization of images and simplifies the maintenance and update process of the whole system. By leveraging the customizability and portability of Docker images [8], AndroidXs can be easily customized for varied scenarios. In addition, from Fig. 1(c), it can be also observed that AndroidXs combine the strong hardware-enforced isolation and compatibility of VMs and the flexibility of containers.

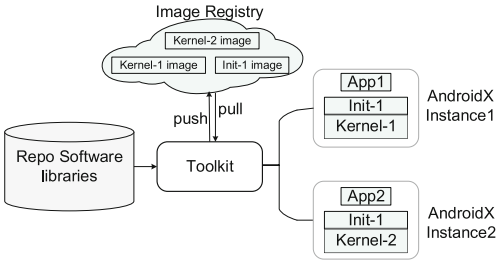
In particular, the main contributions of this paper are summarized as follows:

1. We present a unified approach for customizing Android OS. The approach provides a container-based solution for building Android OS system component images. Following the guidance of this approach, we develop an automatic customization toolkit called AndroidKit, which is much suitable for varied demands on clouds by leveraging the customizability and portability of Docker images.
2. We give a prototypical implementation called AndroidX with the help of the toolkit. The AndroidXs combine the great hardware-enforced isolation and compatibility of VMs and the flexibility and portability of containers. To demonstrate the usability and practicability of the toolkit described here, we use it to implement a type of lightweight AndroidX specifically customized for mobile computation offloading.

The rest of this paper is organized as follows: Sect. 2 explains the research motivation, which gives us an important guideline on designing AndroidX.



(a) The customization process of Android OS



(b) The customization process of AndroidX

Fig. 2. The comparison of customization process for Android and AndroidX

Then in Sect. 3, we describe the architecture and design of AndroidX in detail, as well as AndroidKit that we developed. Section 4 shows an implementation of AndroidX. After that, in Sect. 5, we evaluate the usability of the toolkit by using a case study. In Sect. 6, we describe and review some related works. Finally, a conclusion is summarized in Sect. 7.

2 Research Motivation

First of all, when building a MCP, the cloud platform should provide a runtime environment that has the ability to run mobile applications, e.g., the Android mobile OS. The need for customizing Android OS exists because varied cloud-based scenarios have their own demands for runtime environments. For example, since mobility and interactivity are the keys to the success of computation offloading, the cloud execution environment, e.g., the tailor-made, trimmed-down Android OS, should be as lightweight as possible and only contain the functionality that computation offloading depends on. By contrast, cloud-based mobile testing, which needs MCP to provide the full integrated testing environments, requires the cloud platform has multiple versions of mobile OS for testing different application requirements, e.g., Android OS with a particular architecture or kernel. Therefore, there is a demand for custom-tailored Android OS for varied scenarios on clouds.

Second, many efforts [18,19] have been made to implement the customization of Android OS, but they only focus on the hardware-specific devices or driver-specific applications. From Fig. 2(a), we can clearly observe that traditional customization approaches of Android OS have the following problems. On one hand, the image creator needs to customize complete system images

for different hardware devices and platforms, which results in low reuse of OS image components. On the other hand, traditional approaches require developers have a deep understanding of Android OS and be familiar with each customization step, which is a challenging job for developers. In addition, due to the lack of a unified authentication and update mechanism, it is difficult to ensure the maintainability and security of OS components.

Finally, as known to all, a lot of scenarios use Android VMs and Android containers as on-demand execution environments for running Android applications on clouds. The Android VMs, as shown in Fig. 1(a), have a full copy of an Android OS and run on top of the specific hypervisor. The main defects of Android VMs are their heavy overhead of virtualization and slow startup speed, which are unable to meet the requirements for some scenarios on clouds that require low time-delay, high interactivity, and mobility [20,21]. The Android containers (shown in Fig. 1(b)), by contrast, are much more lightweight than Android VMs because they share kernel with host OS and suffer small overhead [13]. However, as an alternative solution to Android VMs, Android containers are not appropriate for multi-tenant deployments on clouds because of their poor compatibility and weak isolation. Therefore, it is necessary to customize such an execution runtime environment with great isolation and compatibility like VMs, as well as the flexibility like containers.

3 System Design

Through the above analysis and discussion, we conclude that the current execution environments can not meet the needs of varied scenarios on mobile clouds. In this section, we introduce AndroidX, a customizable Android runtime environment for running Android applications on clouds, as well as AndroidKit, a toolkit that we have developed to create Android VM images around specified applications.

3.1 Overview

Based on the previous analysis of the deployment of Android on GNU/Linux platform and the traditional OS customization approaches, and taking into account the current demands for varied scenarios on clouds, we design AndroidX with the following primary targets:

1. **Customizable components:** The customizability of OS components is important when faced with the varied demands on clouds. Our AndroidX is designed for the purpose of customizable OS.
2. **Easy tooling and easy iteration:** The previous research works do not consider the iterability and maintainability of system components, since they are only designed for a certain hardware-specific platform or driver-application, and without taking into account frequently-changing usage scenarios on clouds.

3. **Immutable infrastructure:** The immutable infrastructure, as the term suggests, is comprised of immutable components. It makes service maintenance as simple as installing fresh copy of applications and removing the old versions. With the advantages of repeatable deployments and scalability, it is widely used on cloud environments.

We use Linux platform as the running and testing environments since our prototype implementation is based on it. Figure 3 provides an overview of AndroidX's architecture. It can be clearly observed that AndroidX has two core components. One is a set of tools with command line interfaces called AndroidKit that is used to parse input *yaml* configuration file and generate Docker images. The other is a hypervisor-agnostic runtime, which is able to launch Docker images with a new VM instance.

According to the above targets that we presented, the following four primary challenges have to be overcome: (1) keep the customization process of Android OS simple enough and make it iterate as quickly as possible on the development of system components, (2) be compatible with the *Open Containers Initiative* (OCI) specification for Docker containers, (3) minimize performance overhead as far as possible, and (4) have the speed of containers and excellent isolation of VMs.

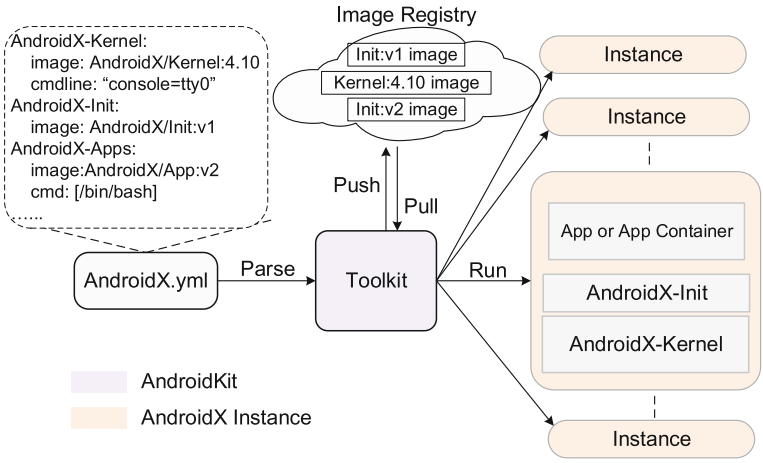


Fig. 3. Overview of AndroidX architecture

3.2 AndroidKit

One of our targets is to achieve easy tooling and easy iteration. To achieve this target, we provide a set of tools called AndroidKit for images building. The toolkit contains an automatic build system that has the ability to create minimalistic Docker images for specific applications. In general, it builds the

images according to the corresponding package source that consists of a directory containing a Dockerfile, which contains the steps to build the package. From Fig. 3, we can find that the toolkit uses a *yaml* configuration template as the input file, which contains the specific system modules that need to be customized.

As we heavily exploit Docker containers, which allow users to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package, for building tailor-made Android OS, we are able to ensure the freshness and integrity of generated images by Docker content trust. Moreover, benefit from the immutable and self-contained features of the images, we can test them for continuous development and delivery. Because of the rich tools provided by Docker, it is very easy for users to add or update the existing system components for different application requirements, which is much suitable for varied demands on clouds.

After building or pulling the images mentioned in the *yaml* configuration file, AndroidKit is able to run the generated images with a new VM instance called AndroidX, which is designed to be architecture agnostic and compatible with the OCI specification that allows users to run Docker images on any hypervisor. Internal to the AndroidX instance in Fig. 3, a minimalist Android Kernel called AndroidX-Kernel is booted directly by hypervisor, the AndroidX-Kernel employs a tiny Android initialization service called AndroidX-Init to load the Docker images from host and then launch them. Through containing applications within separate VM instances and kernel spaces, AndroidXs are able to provide more excellent workload isolation than containers, and security advantages like VMs, which are much suitable for multi-tenant cloud environments.

3.3 AndroidX

Another of our targets is able to launch Docker images with a new Android VM instance, which is called AndroidX. After pulling user-defined Docker images, AndroidKit assembles them into bootable images, then they are launched directly by the toolkit on plain hypervisor, e.g., *qemu*. AndroidX promises immutable infrastructure by eliminating the middle layer of Guest OS. Since the image is run as an $initramfs$, upgrades are done by updating the system components externally. This makes AndroidX immutable, but persistent storage can be attached by adding `-drive` parameter when booting up.

The customization process of AndroidX is shown in Fig. 3. It can be clearly observed that the customization is processed by the order of AndroidX-Kernel, AndroidX-Init, and AndroidX-Apps, which are defined in the input *yaml* configuration file.

(1) AndroidX-Kernel: The AndroidX-Kernel section defines the kernel configuration. One of the features of AndroidX is the customizable kernel. As the previous works shown [11,13], Anbox and Rattrap use LXC to run an entire Android-x86 OS in a container. However, since containers share kernel with host, if users want to add extra new kernel features, they have to modify the host kernel configuration and recompile the kernel, which is a big challenge for

non-kernel developers. What's more, it is not suitable for multi-tenant environments on clouds because of the weak isolation of containers. In contrast to this, AndroidX has its own independent kernel, and we provide a set of useful tools to simplify the customization process of kernel.

To build the AndroidX-Kernel, we divide the kernel configuration file into two parts according to the content of configuration options. One is the general-purpose configuration of Android that acts as a baseline, and the other is the user-optional kernel configuration, which can be replaced or added according to different hardware platforms or application requirements. When creating tailor-made AndroidX-Kernel images, AndroidKit can take a set of user-provided kernel options, and then uses the merge_config script that we provided to generate a minimalist configuration file, which can be used to build the device-specific kernel with user-provided features enabled. This helps AndroidKit create more streamlined kernel images.

(2) AndroidX-Init: The AndroidX-Init section consists of basic init process image called *initrd*, which is unpacked directly into the root filesystem and contains an Android init program. This and the AndroidX-Kernel can be booted directly on plain hypervisor. Unlike other Linux based systems, which use combinations of `/etc/inittab` and `init` programs included in busybox, Android uses its own initialization program, which parses an *init.rc* script that including actions to mount the basic filesystem, set system properties, and start the specified Android system services.

To bring up containerd [22], an industry-standard container runtime, and use runC [23] to run application containers, we migrate containerd and runC to Android-x86 runtime environment. The original runC program has no ability to launch application containers since the `pivot_root` system call, which is used to change root filesystem by runC, does not work on a $ramfs$ or $tmpfs$ root filesystem. Instead, AndroidX first creates a new $tmpfs$ as root filesystem and then uses `switch_root` system call to change root filesystem in an *initrd* shell script provided by Android-x86. Since this is done before starting Android init program, runC is able to support `pivot_root` system call without errors. In addition, as runC provides a native Go implementation for creating containers with a few Linux kernel features (e.g., *namespaces* and *cgroups*), the options of *namespaces* and *cgroups* must be selected in AndroidX-Kernel configuration file except the IPC *namespace* since Android uses binder for interprocess communication. Through our efforts, AndroidX-Init can finally bring up containerd and use runC to run application containers (rootless containers can be launched successfully we tested).

(3) AndroidX-Apps: The AndroidX-Apps section shows a list of images for running applications and services. It contains the specific apps and services that need to be launched when AndroidX starts. The final goal of AndroidX-Apps is that the specific apps and services can be emitted directly both by AndroidX-Init and runC. Since AndroidX-Init has the native ability to run Android applications and services, there is no need to do extra efforts when exploiting AndroidX-Init to run Android applications and services. By contrast, as we use runC to launch

application containers, some kernel features (e.g., *cgroups* and *namespaces*) and the Go runtime environment must be supported. In our efforts, rootless containers currently can be instantiated successfully by runC.

As we hope that more than one AndroidX instance could share the system components as much as possible, we exploit the read-only feature of the Android system partition, which can be attached by adding `-drive` parameter and shared with other instances when booting AndroidX instances. Benefit from this, the average disk usage size of each AndroidX instance decreases and gets close to Android containers when more and more AndroidX instance are started. Since we design AndroidX for immutable infrastructure, AndroidX-Apps can be attached by using SD card and data partitions for persistent storage, which will be identified by tailor-made AndroidX initialization program when booting up.

4 Implementation

We have implemented the prototype of AndroidX on our machine. The machine contains an Intel Core i5-7200 2.50 GHz CPU (2 cores) with 16 GB of DDR4 RAM and 256 GB HDD, running Ubuntu 16.04. In our current version, the implementation of AndroidX consists of two parts. One of them provides a rich set of tools named AndroidKit for images building. We develop the toolkit with the guidance of the container-based customization approach. Another is the execution runtime environment, which can be easily customized for varied scenarios on clouds.

As we mentioned above, the build process of AndroidX heavily leverages Docker images for packaging, and all intermediate images are referenced by digest to ensure reproducibility across its build process. To guarantee the freshness and integrity of the images, all of the generated images will be signed using Docker content trust. After building the Docker images, AndroidX instance can be booted directly on plain hypervisor by targeting with AndroidX-Kernel+AndroidX-Init. Through combining with the portability of app container images, AndroidX is able to allow users to build, ship, and run apps anywhere, without considering the underlying technology stack.

The construction of AndroidX is based on a series of Android-x86_6.0_r3 components, which can be customized for varied demands. In our prototype implementation, we provide a lot of templates, e.g., Dockerfiles, for users customize OS components. The source code of AndroidX is publicly available online at https://github.com/CGCL-codes/AndroidX.

5 A Case Study

To demonstrate the usability and practicability of AndroidKit, we use it to implement a type of lightweight AndroidX, which is based on Android-x86 and specifically customized for mobile computation offloading. These AndroidXs provide the excellent workloads isolation like Android VMs, as well as the extremely fast

Table 1. Performance comparison for Android container, Android VM, and AndroidX. For the experiment we allocate a single core and 1024 MB of memory to each test instance.

Runtime	Boot time	Memory footprint	Disk usage	CPU allocation	Memory allocation
Android container	1.8 s	96 MB	1044 MB	1vCPU	1024 MB
Android VM	31.8 s	493 MB	1728 MB	1vCPU	1024 MB
AndroidX	3.9 s	270 MB	1101 MB	1vCPU	1024 MB

instantiation time like Android containers. In this section, we present an evaluation of these customized AndroidXs, including the comparisons of startup time, memory footprint, and disk usage with standard Android VMs and Android containers. All experiments are run on a machine mentioned in Sect. 4.

5.1 Boot Time

Boot time is a critical performance evaluation point in many cloud computing scenarios. Since we evaluate the toolkit by using it to implement a type of lightweight AndroidX specifically customized for mobile computation offloading (which offloads computational codes to clouds and requires low time-delay), we want to measure how long AndroidKit takes to create and boot such an AndroidX instance.

The main limiting factors of instantiation time are the image size of VMs and the number of processes that need to be started. LightVM [24] has demonstrated that startup times grow linearly with VM image size by booting the same unikernel VM from images of different sizes in the experiment. The reason why large VMs slow down instantiation time can be summarized as follows: launching a large VM instance needs time to read the image from disk, parse it and finally run it in memory. Inspired by this, we have made a great effort on optimizing AndroidX for mobile computation offloading, and in our efforts, AndroidKit can generate compact AndroidXs which have the speed of containers. Table 1 compares boot times for a noop AndroidX instance against a noop Android VM, as well as a noop Android container. Time is measured form booting to the point where instantiation is finished.

In order to find the most time-consuming process during the whole startup of standard Android VM instances, we try to take a look at the CPU usage when starting a noop Android VM. For the measurement we use bootchart [25] and adb [26] tools to get a noop Android VM's CPU utilizations. The bootchart is a tool for performance analysis and virtualization of the GNU/Linux boot process, and the adb tool which is specially designed for Android, has the ability to analyze the startup log of each process. As shown in Fig. 4, the Android VM reaches a maximum CPU utilization of about 100% after 7 s of startup and lasts for a period of time. To find out which processes occupy a large amount of CPU resources at that time, we try to get the time consumption of each process in

the whole startup process of the VM by leveraging bootchart and adb tools, and find that dex2oat, package scanning, and class preloading take up about 72.7% of the total system boot time and consume a lot of CPU resources at that time. To shorten the time taken by these processes, we reduce the number of preloaded packages, as well as the preloaded classes and resources when customizing AndroidX for mobile computation offloading. The optimized result is shown in Fig. 5. This figure shows the time consumption comparison of class preloading, package scanning and dex2oat during the whole startup for Android VM, AndroidX, and Android container. From this figure, we can conclude that the time consumption of optimized processes is only 15.7% of the non-optimized. In addition, after using a pre-prepared data partition, we almost eliminate the time taken up by the dex2oat process. In order to minimize the size of VM image, we reduce the functionality of AndroidX, such as Camera and Bluetooth that will not be used on cloud environments. In our efforts, the size of the final image generated by AndroidKit is reduced from the original 1728 MB to 1101 MB. As a result, the startup time of an optimized AndroidX instance can be shortened to 3.9 s (shown in Table 1), which is much lightweight than Android VM, and as fast as Android container.

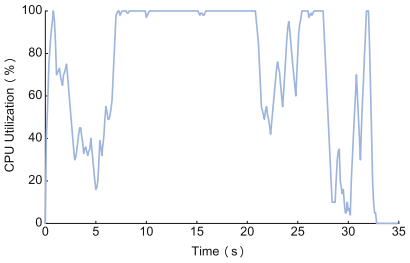


Fig. 4. CPU usage during the whole startup of a noop Android VM

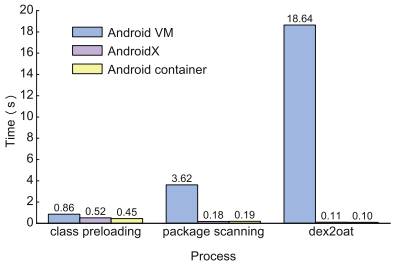


Fig. 5. Time consumption comparison of class preloading, package scanning, and dex2oat during the whole startup for Android VM, AndroidX, and Android container

5.2 Memory Footprint

As known to all, in order to concurrently launch multiple instances on a single host, the most effective solution is to reduce per-instance memory footprint. One of the advantages of Android containers is that they typically need less memory than Android VMs because they use a common kernel with host OS and have smaller root filesystems. By contrast, Android VMs, where each instance has their own independent kernel and runs an entire Android OS, suffer more resource overhead than Android containers when concurrently running multiple instances on a single machine. In our next analysis, we try to find if the memory footprint of each compact AndroidX is close to Android container.

We observe, as others [27], that most VMs and containers run a single application on clouds. By reducing the functionality of AndroidX to include only what is necessary for that specified application, we are able to reduce the memory footprint of each AndroidX instance. With the help of `adb shell procrank` command, we get exact memory footprint of a noop AndroidX instance. Table 1 shows the memory usage of a noop AndroidX instance against a noop Android VM and a noop Android container (which is essentially a Rattrap instance). From the table, we can conclude that the memory usage of a noop AndroidX instance is only 54.7% of a noop Android VM. The reason why we use a noop AndroidX instance is that the cloud execution environment for mobile computation offloading has little association with applications, and AndroidKit has the ability to generate AndroidX for different application requirements since we provide a lot of templates for customization.

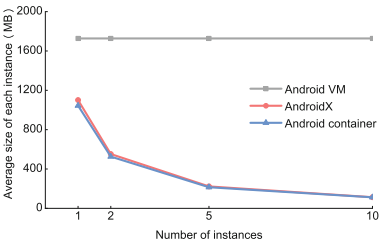


Fig. 6. The average disk usage size of each instance for Android VM, AndroidX, and Android container

5.3 Disk Usage

As we mentioned above, one of the main limiting factors of instantiation time is the image size of instance. In order to shorten the startup time and improve the disk utilization of the offloading code execution environment, Rattrap analyzes the entire Android-x86 OS files and finds out that 68.4% of them are never accessed by offloaded codes, which are composed of unnecessary libraries and modules. By removing the unnecessary parts and sharing the system libraries, an optimized Rattrap instance only takes up 7.1 MB space and has faster startup speed.

Inspired by this, we put forward the idea of sharing partitions. AndroidX instance has the ability to share the data and system partitions by the predefined `-drive` parameter, which can be identified by AndroidKit. In this way, the disk usage of each AndroidX instance is close to a Rattrap instance, which is less than 10 MB. From Table 1 and Fig. 6, we can find that an AndroidX instance takes up about 1101 MB of disk. However, the average disk usage size of each AndroidX instance decreases and gets close to Android containers when more and more Android instances are launched. By contrast, as Fig. 6 shows, the average disk usage size of each Android VM instance always remains the same

when the number of running instances increases, since Android VMs lack a sharing mechanism. As a result, AndroidX has a great disk utilization when running multiple instances, which is more suitable for large-scale scenarios on clouds.

6 Related Work

A lot of container-based virtualization technologies (e.g., Docker, LXC, and rkt) have been widely deployed on cloud platforms because of their low resource overhead and great scalability. However, as known to all, the weak isolation of containers has caused some security problems on multi-tenant cloud environments [28]. Unlike containers, VMs which are based on hypervisor technologies, have excellent hardware-enforced isolation, but they cause high resource overhead since each VM runs a full copy of an OS. Intel sets out to build hypervisor-based container named *Intel Clear Containers* (ICC) [29] by combining the best benefits of VMs and Linux containers. Kata Containers [30], which combines technology from ICC and Hyper [31], tries to run Docker containers on agnostic hypervisors to provide the workload isolation like VMs, as well as the portability like containers. A part of our design idea comes from ICC and Hyper, but as we have already shown, AndroidXs are specifically designed for the various frequently-changing scenarios on mobile clouds.

Traditional operating systems, e.g., Linux, focus on the versatility and integrality of system functions and contain the entire software stack with the tradeoff of overhead and efficiency. By contrast, unikernels [27], which are designed for supporting cloud services rather than desktop applications, are tiny VMs that pack the minimalistic OS with the target application into a single bootable VM image. Many research works have been made on constructing unikernels (e.g., OSv [32], MirageOS [33], and ClickOS [34]), and the common goal of these unikernels is to run single application on a single machine to eliminate the redundancy and provide great performance. Through booting directly on plain hypervisor, they are able to avoid the hardware compatibility problems suffered by traditional library operating systems (e.g., Exokernel [14] and Nemesis [15]).

7 Conclusion

This paper presents AndroidXs as customizable execution environments for Android applications on clouds, as well as AndroidKit as a toolkit for building customizable Android OS images. Our idea comes from the experience of running Android applications on Linux platform, as well as *Intel Clear Containers* and Hyper open source projects. After investigating the relevant research works of OS customization, and inspired by the design of unikernels, we propose a container-based approach for customizing mobile cloud execution environment. Under the guidance of this approach, we have developed a set of tools named AndroidKit, which is able to generate images for customizable AndroidXs. To demonstrate the practicability of the AndroidKit, we use it to implement a type

of lightweight AndroidX specifically customized for mobile computation offloading. The AndroidXs are composed of OS images generated by AndroidKit, and in our efforts, rootless containers are now able to be brought up by runC in AndroidX instances.

Acknowledgements. This research is supported by National Key Research and Development Program under grant 2016YFB1000501, and National Science Foundation of China under grants No. 61732010 and 61872155.

References

1. Cuervo, E., et al.: MAUI: making smartphones last longer with code offload. In: Proceedings of MobiSys, pp. 49–62. ACM (2010)
2. Chun, B., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of EuroSys, pp. 301–314. ACM (2011)
3. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: Proceedings of INFOCOM, pp. 945–953. IEEE (2012)
4. Mobile testing. https://en.wikipedia.org/wiki/Mobile_application_testing
5. Android binder. https://elinux.org/Android_Binder
6. Shashlik. http://www.shashlik.io/
7. Genymobile. https://www.genymobile.com/
8. Docker. https://www.docker.com/
9. Lxc. https://en.wikipedia.org/wiki/LXC
10. Rkt. https://coreos.com/rkt/
11. Anbox. https://anbox.io/
12. Android-x86. http://www.android-x86.org/
13. Wu, S., Niu, C., Rao, J., Jin, H., Dai, X.: Container-based cloud platform for mobile computation offloading. In: Proceedings of IPDPS, pp. 123–132. IEEE (2017)
14. Engler, D.R., Kaashoek, M.F., O'Toole, J.: Exokernel: An operating system architecture for application-level resource management. In: Proceedings of SOSP, pp. 251–266. ACM (1995)
15. Leslie, I.M., et al.: The design and implementation of an operating system to support distributed multimedia applications. IEEE J. Sel. Areas Commun. **14**(7), 1280–1297 (1996)
16. Fassino, J., Stefani, J., Lawall, J.L., Muller, G.: Think: a software framework for component-based operating system kernels. In: Proceedings of ATC, pp. 73–86. ACM (2002)
17. Krintz, C., Wolski, R.: Using phase behavior in scientific application to guide linux operating system customization. In: Proceedings of IPDPS. IEEE (2005)
18. Shanker, A., Lai, S.: Android porting concepts. In: Proceedings of ICECT, vol. 5, pp. 129–133. IEEE (2011)
19. Yaghmour, K.: Embedded Android: Porting, Extending, and Customizing. O'Reilly Media Inc., Sebastopol (2013)
20. Duan, Y., Zhang, M., Yin, H., Tang, Y.: Privacy-preserving offloading of mobile app to the public cloud. In: Proceedings of HotCloud, pp. 18–18. ACM (2015)
21. Shiraz, M., Abolfazli, S., Sanaei, Z., Gani, A.: A study on virtual machine deployment for application outsourcing in mobile cloud computing. J. Supercomput. **63**(3), 946–964 (2013)

22. Containerd. https://containerd.io/
23. Runc. https://blog.docker.com/2015/06/runc/
24. Manco, F., et al.: My VM is lighter (and safer) than your container. In: Proceedings of SOSP, pp. 218–233. ACM (2017)
25. Bootchart. http://www.bootchart.org/
26. Android debug bridge. https://en.droidwiki.org/wiki/Android_Debug_Bridge
27. Madhavapeddy, A., Scott, D.J.: Unikernels: the rise of the virtual library operating system. Commun. ACM **57**(1), 61–69 (2014)
28. Container security. https://arxiv.org/abs/1507.07816
29. Intel clear container. https://clearlinux.org/containers
30. Kata container. https://katacontainers.io/
31. Hyper. https://hypercontainer.io/
32. Kivity, A., et al.: Osv - optimizing the operating system for virtual machines. In: Proceedings of ATC, pp. 61–72 (2014)
33. Madhavapeddy, A., et al.: Unikernels: library operating systems for the cloud. In: Proceedings of ASPLOS, pp. 461–472. ACM (2013)
34. Martins, J., et al.: Clickos and the art of network function virtualization. In: Proceedings of NSDI, pp. 459–473. ACM (2014)