PAPER

Research on Building an ARM-Based Container Cloud Platform

Lin CHEN[†a)], **Xueyuan YIN**[†b)], **Dandan ZHAO**[††], **Hongwei LU**[††], **Lu LI**[††], ***and*** **Yixiang CHEN**[†††], ***Nonmembers***

SUMMARY ARM chips with low energy consumption and low-cost investment have been rapidly applied to smart office and smart entertainment including cloud mobile phones and cloud games. This paper first summarizes key technologies and development status of the above scenarios including CPU, memory, IO hardware virtualization characteristics, ARM hypervisor and container, GPU virtualization, network virtualization, resource management and remote transmission technologies. Then, in view of the current lack of publicly referenced ARM cloud constructing solutions, this paper proposes and constructs an implementation framework for building an ARM cloud, and successively focuses on the formal definition of virtualization framework, Android container system and resource quota management methods, GPU virtualization based on API remoting and GPU pass-through, and the remote transmission technology. Finally, the experimental results show that the proposed model and corresponding component implementation methods are effective, especially, the pass-through mode for virtualizing GPU resources has higher performance and higher parallelism.

key words: *cloud computing, ARM, hypervisor, containers, GPU*

1. Introduction

According to latest statistics of ARM, cumulative shipments of ARM chip partners have reached about 240 billion chips by 2022. Due to ARM chips have rich capabilities, such as lower energy consumption, low-cost and supporting extensible design, which have been employed into diverse fields (e.g., set-top boxes, smart phones, PCs and servers). For the server field, by employing virtualization technologies, ARM chips have been applied to build various cloud services including cloud mobile phone, cloud gaming, and cloud computer. Associated with it, ARM cloud involves virtualization technologies of CPU, memory, GPU, network and storage.

Primarily, CPU and memory and IO could be virtualized by employing a hypervisor (also named virtual machine monitor, VMM) [1] or a container [2]–[4] middleware: (1) For hypervisor model, the vital appliances including CPU, memory, IO, interrupt, clock, etc. must have essential hardware virtualization features [5], which could contribute to the realization of hypervisor functions. A typical realization is based on an ARM/KVM model, taking Phytium server chips (e.g., FT-2000+/64) as examples to illustrate [6], whose key virtualization features are realized with the help of corresponding hardware virtualization capabilities, such as CPU providing a separate execution environment EL2 for hypervisor, and memory managed by a MMU supporting two-level page table transformations, and IO integrating with an IOMMU to support device allocation and access directly. Besides, on FT-2000+/64 server platform configured with dual-cores, the Benchmark statistics show that the average loss ratios of CPU computing and memory access are both about 3%. Moreover, some scholars have carried out ARM/KVM virtualization researches on ARM-v8 platform (Firefly-RK3399). Huang et al. [7] are focus on deploying a guest OS through the mechanism of cross compiling UEFI, and realizing dual-screen display by modifying codes of QEMU display device. The results show that user performance is not good, due to the constraint of low performance hardware. Ma [8] points that ARM/KVM solution is better than docker in terms of the functional integrity and file copy. (2) For container model, various products for containerizing Linux or Android guest are derived. For providing Linux containers, there are LXC, Docker, LXD, Kata containers, Linux VServer, Singularity, Rkt, gVisor. Thereinto, LXC, Docker and LXD have good storage feature supporting, Kata-containers has good features of isolation [9]. The comparisons between LXC, Docker and LXD as shown in Table 1.

For offering Android containers, which can provide varieties of SaaS services (e.g., cloud mobile phone, cloud gaming, cloud testing). Typical solutions are: Ubuntu Touch [11] is a mobile OS released by Canonical, and designed for mobile Touch devices such as smartphones and tablets, and its Android containers (instantiated by employing LXC technology) run on Ubuntu. Anbox [12], [13] abstracts hardware access and integrates core system services into a GNU/Linux system, which uses Linux technologies (e.g., USER, PID, UTS, NET) to separate the Android OS from host; moreover, a derivate solution named xDroid of Anbox is born. Huawei has launched a cloud-based mobile phone solution named Monbox [14], which provides simulation test, trial promotion, cloud gaming and mobile office. Tencent pushes out a cloud-based Android mobile named VMI (Virtual Mobile Infrastructure), which is composed of a server (including Android containers and corresponding dockdroid daemons) and a client. The Condroid project [15]–[17] is launched by Arclab, Zhejiang University, which employs LXC tools to containerize Android guests on an Android host OS, and peripheral supports are implemented.

Manuscript received February 10, 2023.
Manuscript revised July 3, 2023.
Manuscript publicized August 7, 2023.
[†]The authors are with Xingzhe AI, Chengdu, 610065 China.
[††]The authors are with Lingyue Cloud, Chengdu, 610065 China.
[†††]The author is with Arm China, Shanghai, 201101 China.
a) E-mail: linchn@yeah.net
b) E-mail: yinxueyuan@foxmail.com (Corresponding author)
DOI: 10.1587/transfun.2023EAP1016

Copyright © 2024 The Institute of Electronics, Information and Communication Engineers

Table 1 Feature comparisons between LXC, Docker and LXD.

LXC (Born in 2008)	Docker (Born in 2013)	LXD (Born in 2014)
A command line tool without daemon process and rest API; LXC container cannot be migrated across hosts.	Applications packaged by Docker can be migrated across machines or platforms.	Compared with KVM, the deployment density is increased at least 10 times, startup speed is accelerated more than 94%. [10]

LXC VS Docker	• The early docker implementation is based on LXC; • Since version 0.9, Docker has launched Libcontainer as a container driver to replace LXC;
Docker VS LXD	• Docker is mainly used to host applications, while LXD container provides a complete OS; • They can complement each other: Docker applications can be run inside a LXD container.
LXC VS LXD	• LXD daemon makes up for deficiency of LXC; • LXD focuses on running an OS in a container.
Additional Remarks	• The various low-level Linux container runtimes (e.g., LXC) have no essential differences, because all of them rely on Cgroups, NameSpaces, etc.

Table 2 Model feature comparisons between hypervisor and container.

Indicator	Hypervisor Model	Container Model
Isolation	• Excellent; • An isolate virtual machine (VM) OS [19, 20].	• Good; • An isolate user-space.
Performance	• VM performance is lower than that deployed in a physical host.	• The container performance is close to that running in a physical host. [21,22]
Disk Use Size	• About several or tens of GB.	• In the range of several MB to GB.
Instantiation Time	• Average; • In the range of seconds to minutes.	• Excellent; • About several milliseconds or seconds.
Concurrency	• Poor; • Dozens of VM instances can run in a server.	• Excellent; • Hundreds of containers can run in a server.
Portability	• Average; • Features of hardware, VMM etc. need to be considered.	• Excellent; • Cross-platform running is supported inherently.
Operating Expense	• Different OSs need to be managed and maintained.	• Unified host OSs needs to be maintained, which is light work.
Capital Expenditure	• Each instance needs to pay for license.	• The model reduces the number of licenses.

Inspired by Condroid, a solution named Clondroid [18] is developed by ITRI, Taiwan. The key feature comparisons between hypervisor and container as shown in Table 2.

Secondly, with the dramatic increase of graphical applications, GPU resources are also indispensable. For GPU virtualization, virtual GPU (vGPU) or GPU pass-through or API remoting technologies are leveraged usually. Huawei Monbox provides GPU resource by employing the pass-through technology [14], and corresponding GPU driver must be transplanted into Android OS, thus, Android rendering framework needs to be adapted. The ReDroid (RemoteanDroid) [23] provides an Android container with a GPU acceleration function, which employs mesa3D graphics library to realize GPU acceleration. Some game manufacturers use Ampere Servers and Nvidia's vGPU technology to build cloud game solutions.

Moreover, NV (Network Virtualization), NFV (Network Function Virtualization) and SDN (Software Defined Networking) could be leveraged to construct virtual network environment; LVM (Logical Volume Manager) and distributed storage system (e.g., Ceph, GlusterFS) cloud be employed to provide storage resource pools; Based on NFV and SDN, the security service property of "Encapsulate multiple security appliances into one" can be realized.

Furthermore, quota management for virtualized resources is an indispensable part, after a virtual appliance (e.g., a VM or a container) has been initialized, if its resource quota is not set or too many resources are configured, the server will be overloaded. Contrarily, if too few resources are allocated, resources will be wasted and the service quality of the container will be affected. Simple resource quota upper limit setting makes it difficult to make full use of server resources (e.g., OpenStack quota system, which sets limits on resources such as amount of CPU that a project can use).

Finally, remote transmission is an essential part of service delivery in cloud environments. Different from the traditional simple remote-control scenario as shown in Table 3, with the rapid development of network technologies (e.g., 5G, fiber-optic network) and various types of cloud services, nowadays, users are concerned about two indicators: image quality [24] and time delay. According to statistics, the end-to-end access delay of the Huawei Monbox solution is about 100ms. Tencent VMI transfers the display images to the user client at a specified frame rate. (Typical representatives of commercial transmission protocols on x86 platform are Citrix ICA, VMWare PCoIP, Microsoft RDP, RemoteFX, HP RGS, NoMachine NX, Huawei HDP etc.)

Overall, existing research efforts focus more on a specific segment of building ARM cloud (e.g., ARM hypervisor model or employing Android as a host OS), and there is less public detailed information and experience about building a complete solution. To provide references for constructing an ARM-based cloud, the corresponding techniques and solutions are investigated, compared, and realized in Sect. 1, which contains indicators of technology and economy aspects. In Sect. 2, the ARM cloud framework is formal defined, and typical solutions or products are summarized. In Sect. 3, a specific ARM-based cloud platform is implemented by employing technologies of the container, GPU

Table 3 Differences of 3 main remote application scenarios.

Scenarios	Differences
Simple Remote Control	• The controlled object does not have to be in a cloud (e.g., any computer at home). • Basic interactive control is supported (e.g., command characters). • Users can tolerate delays of hundreds of milliseconds. • Only a few hundred Kbps or a few Mbps of bandwidth is required.
Virtual Desktop Infrastructure (VDI)	• The computing and storage resources are in cloud, and GPU resources are optional. • The delivery objects are Windows and Linux OS mainly. • Audio and video playback need to be supported. • Common computer peripherals need to be supported (e.g., keyboard, U disk, etc.) • User can tolerate delays of tens of milliseconds, typical value is less than 60ms. • About less than 10 Mbps of bandwidth is required.
Cloud Gaming	• The computing and storage resources are in cloud, and GPUs or vGPUs are needed. • High image quality requirements need to be satisfied (e.g., 2K/4K@90FPS/144FPS). • Game peripherals need to be supported (e.g., controller). • User only tolerate delays of tens of milliseconds, and typical value is less than 50ms. • For an esports player, the time delay needs to be less than 20ms. • More than 50Mbps of bandwidth is required in 1080P@144FPS scenario.

pass-through and API remoting, and remote transmission. The experiments and discussions are given in Sect. 4. The summary and a prospect to the research of this paper are discussed in Sect. 5.

2. Virtualization Framework Formal Definition

2.1 Definition

With the goal of providing a conceptual and concise description of existing virtualization systems or solutions, and proposing essential components of the system (at this moment, there is no discussion on the detailed states and transitions of various elements in the system), a virtualization framework was formally designed and defined.

The framework labeled F is defined as a four-tuple formally: $F = (Hardware, HostOS, Middleware, GuestOS)$,

- *Hardware*: a specific type of a hardware platform (including x86 and ARM), $Hardware = \{hw_{x86}, hw_{ARM}\}$.
- *HostOS*: an OS type of the host OS (including Linux, Android, Windows, MacOS), $HostOS = \{hos_{Linux}, hos_{Android}, hos_{Windows}, hos_{MacOS}\}$.
- *Middleware*: a virtualization model (including hypervisor model such as Xen, KVM, VirtualBox etc., and container model such as LXC, LXD, Docker etc.), $Middleware = \{Hypervisor, Container\}$, $Hypervisor = \{h_{Xen}, h_{KVM}, h_{VirtualBox}, h_{Commercial}\}$, $Container = \{c_{LXC}, c_{LXD}, c_{Docker}, c_{commercial}\}$, $h_{commercial}$ and $c_{commercial}$ represent proprietary products of *Hypervisor* and *Container* respectively.
- *GuestOS*: an OS type of the guest OS (including Linux, Android and Windows), $GuestOS = \{gos_{Linux}, gos_{Android}, gos_{Windows}\}$.

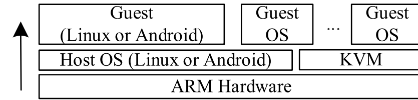


Fig. 1 A hierarchical framework of an ARM/KVM model.

- Any element in F can be assigned as *NULL*, which represents that item is not assembled (e.g., partial products without employing virtualization technologies, which employs physical ARM clusters).
- For a specified physical server i, which satisfied the following specific constraints:
  (a) $hw_i \in Hardware$, $hos_i \in HostOS$, and $(hos_{x86})_i \neq (hos_{ARM})_i$.
  (b) $Middleware_i = \{h_i, c_i\}$, $h_i \in Hypervisor$, $c_i \in Container$. If $h_i \neq NULL$ and $c_i \neq NULL$, then $\exists(h_i \wedge c_i) \neq NULL$, which represents that there are compatible hypervisor virtualization technologies and container technologies. For example, in a host environment with KVM virtualization technology, the LXC or LXD or Docker container technology can continue to be deployed.

According to the above definitions and specific constraints, the following features of F could be obtained.

(1) By employing constraint (a), Servers with different CPU architectures can reasonably install host operating systems suitable for their own hardware (x86 or ARM), so as to realize the basic management and resource scheduling of the underlying hardware.

(2) By employing constraint (b), the compatibility of different virtualization technologies can be guaranteed, enhancing the diversity and flexibility of server resource pooling.

(3) By providing different guest systems, various business needs can be met. Furthermore, when virtual machines or containers are instantiated, the isolation requirements between them could be satisfied by employing the mechanism of communication access control [19].

2.2 ARM VMM and Container Framework

According to above F definition and the typical ARM hypervisor realization is *ARM/KVM*, which framework $F_{ARM/KVM} = (hw_{ARM}, \{hos_{Linux}, hos_{Android}\}, h_{KVM}, \{gos_{Linux}, gos_{Android}\})$. The typical ARM/KVM realization as shown in Fig. 1.

Moreover, for offering container services (e.g., Linux container, Android container), the framework is $F = (hw_{ARM}, \{hos_{Linux}, hos_{Android}\}, Container, \{gos_{Linux}, gos_{Android}\})$. When $GuestOS = gos_{Android}$, F can offer cloud mobile phone, cloud gaming service, etc.

Currently, partial representative products or (open source) solutions of Android hypervisor, container, and farm as shown in Table 4, which shows the essential elements of every realization framework and important features.

Table 4 Keys of Android hypervisor and Android container and Android farm solutions.

Model	Solutions	Framework and Key Points
Hypervisor	ARM/KVM	$F_{ARM/KVM} = (hw_{ARM}, \{hos_{Linux}, hos_{Android}\}, h_{KVM}, \{gos_{Linux}, gos_{Android}\})$
		The model for general description.
	Genymotion	$F_{Genymotion} = (hw_{x86}, \{hos_{Linux}, hos_{Windows}, hos_{MacOS}\}, h_{VirtualBox}, gos_{Android})$
		For offering $gos_{Android}$ services, due to primarily constrained by the factors of performance and high price, $gos_{Android}$ is often realized based on a x86 server. Based on AOSP (Android Open-Source Project), various versions of Android x86 OS could be produced, which could be deployed on a x86 platform (e.g., a physical x86 computer, or a x86 VM instance launched by h_{KVM} or $h_{VirtualBox}$).
Container	Ubuntu Touch	$F_{Ubuntu\ Touch} = (\{hw_{ARM}, hw_{x86}\}, hos_{Linux(ubuntu)}, c_{LXC}, gos_{Android})$
		It focuses on ARM devices.
	Anbox, xDroid	$F_{Anbox\&xDroid} = (\{hw_{ARM}, hw_{x86}\}, hos_{Linux(ubuntu)}, \{c_{LXC}, c_{LXD}, c_{Docker}\}, gos_{Android})$
		OpenGL ES acceleration is realized by simulation, but compatibility is poor, due to it only supports ARM64-V8A applications.
	Huawei Monbox	$F_{Huawei\ Monbox} = (\{hw_{ARM}, hw_{x86}\}, hos_{Linux(ubuntu)}, c_{commerical}, gos_{Android})$
		Drawing experience from Anbox, GPU pass-through is adopted.
	Tencent VMI	$F_{Tencent\ VMI} = (\{hw_{ARM}, hw_{x86}\}, hos_{Linux(ubuntu)}, c_{LXC}, gos_{Android})$
		Building on LXC and OpenGL ES API remoting technologies.
	Condroid, Clondroid	$F_{Condroid\&Clondroid} = (hw_{ARM}, hos_{Android}, c_{LXC}, gos_{Android})$
		LXC tools are transplanted to Android host to realize containerization.
	ReDroid	$F_{Redroid} = (\{hw_{ARM}, hw_{x86}\}, hos_{Linux}, c_{Docker}, gos_{Android})$
		Mesa3D is introduced to realize GPU acceleration.
Farm	ARM Farms	$F_{ARM\ Farms} = (hw_{ARM}, hos_{Android}, Null, Null)$
		Evaluation based on cost performance and compatibility, partial products without employing virtualization technologies, which employs physical ARM Farms (clusters). They are consisted of many independent ARM processors and communicate with each other or external objects through high-speed networks, every Android guest OS is deployed inside the independent hardware unit. A typical solution is Baidu ARM Cloud Mobile.

3. A Containerized Android Framework

To discuss the construction process and key technologies of ARM cloud, according to Table 1 and compatibility with leading virtualization standards, an ARM cloud framework for cloud Android gaming and testing is built as shown in Fig. 2. The ARM-based container model is the best choice with the greatest potential and Docker is chosen; and Ubuntu Linux OS can be well compatible with mainstream virtualization standards as the host OS, which has an advantage of integrating unified management tools easily (e.g., K8S); and challenges such as Binder (that is an Inter-Process Communication mechanism provided by the Android system) virtualization, Binder isolation, and display virtualization are eliminated inherently.

The proposed framework consists of an ARM cloud and terminals, which communicate with each other through remote transmission subsystem. The cloud employs four hierarchical layers: a hardware layer, a kernel layer, a middleware layer and a container layer. The physical layer provides resources of physical CPU, memory, storage, network and GPUs. Then, a Linux kernel and middleware are built on top of it to containerize Android instances. Besides, an agent and a client are deployed inside an Android container and a terminal respectively, which are responsible for transmitting screen image, audio data and input events (e.g., keyboard press, screen touch) to each other.

As a whole, the ARM cloud framework can be divided into three key parts to implement: (1) Android container subsystem: On the ARM server, Ubuntu and Docker are used to build a basic container environment. Then, an algorithm suitable for dynamic adjustment of container partial resources is implemented and integrated into virtualization middleware. (2) GPU sharing subsystem: The principles, advantages and disadvantages of the two methods (API remoting and GPU pass-through) are discussed and implemented in the test environment. (3) Remote transmission subsystem: Taking an Android terminal as an example, a remote transmission subsystem composed of client and server is implemented. At the same time, the reasons and benefits of implementing the server and client in different levels of Android system are analyzed.

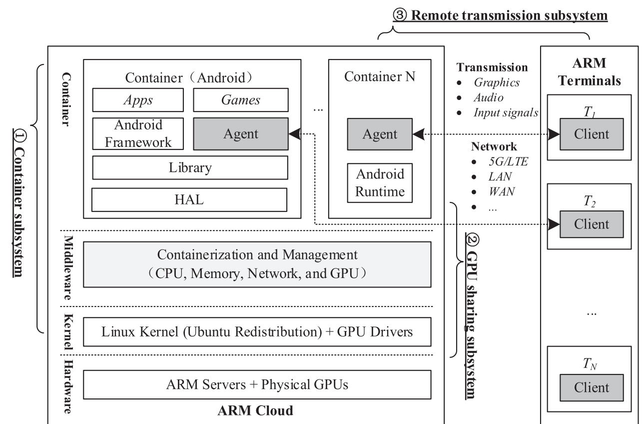


Fig. 2 The ARM cloud framework for offering Android container services.

The realization framework is formally described as: $F = (hw_{ARM}, hos_{Linux(Ubuntu)}, c_{Docker}, gos_{Android})$.

3.1 Android Container Subsystem

In addition to the usual container lifecycle management (e.g., create, running, pause, delete) and image management by employing Docker, for more efficient use of server resources, a mechanism of resource dynamic allocation was designed.

The mechanism contains three modules: (1) Resource monitoring module: a statistical tool based on Telegraf, InluxDB and Grafana, which is responsible for collecting the runtime load of host system and containers regularly; (2) Resource quota module: it is responsible for calculating container resource quota requirements and generating resource quota policies; (3) Quota setting module: it is responsible for delivering and executing policies.

The above three modules are realized and deployed into the middleware layer in Fig. 2, which has benefits of high reliability and security of the external monitoring model. When obtaining the specific resource load information, Telegraf can obtain the corresponding load information of the host server by using various plug-ins (e.g., CPU, network, disk); With the Docker plug-in (e.g., call '*docker stats*' API to retrieve the usage of container resources, including CPU, Memory, Network I/O etc.), we can get the load information of the Docker container. Afterwards, the data is stored in InluxDB and visualized using Grafana.

The resource adjustment algorithm is described as Algorithm 1, and the notations used as follows:

- C_{Total}: all containers (ci) running on an ARM server, $C_{Total} = \{c1, \ldots, ci, \ldots, cn\}$.
- p_{ci}: a resource quota policy of container c_i, which is a two-tuples: $p_{ci} = (cr_{MiniReq}, cr_{Capping})_{ci}$. The two items represent indexes of minimum resource requirement and resource quota capping for c_i respectively.
- P_{Total}: a resource quota policy set of C_{Total}, $P_{Total} = \{p_{c1}, \ldots, p_{ci}, \ldots, p_{cn}\} = \{\ldots, (cr_{MiniReq}, cr_{Capping})_{ci}, \ldots\}$.
- THR_{Idle}: a determination coefficient, which is used to determine whether the remaining resource of a server is sufficient. According to industry experience, THR_{Idle} is set to 0.5.
- $THR_{Efficient}$: an effective resource coefficient. According to experience, $THR_{Efficient}$ is set to 0.9.
- $PR_{Capping}$: a resource capping of a server, $PR_{Capping} = (PR_{Physical} \times THR_{Efficient})$, and $PR_{Physical}$ represents resources of a server (including CPU, memory, network and disks).
- $T_{Interval}$: an interval collection cycle of the resource monitoring module, $T_{Interval}$ is set to 10 seconds.
- $CR_{ConUsage}$: according to $T_{Interval}$, a set of the resource usage of containers in C_{Total} are collected and noted as $CR_{ConUsage} = \{res_{c1}, \ldots, res_{ci}, \ldots, res_{cn}\}_{T_{Inverval}}$.
- $CRS_{T_{Interval}}$: according to $T_{Interval}$, C_{Total} resource load could be noted as $CRS_{T_{Interval}} = (\sum_{i=1}^{n} res_{ci})_{T_{Interval}}$.
- The container ck without setting a value of resource capping,in other words, $(p_{ck} \otimes cr_{Capping}) = NULL$ is

Algorithm 1: Resource adjustment algorithm

Input: res_{ci} , $PR_{Capping}$ (*Resource usage of ci , Resource capping of a server*)
Output: p_{ci} (*Resource quota policy*)

1. while True:
2. if $CRS_{T_{Interval}} \le (PR_{Capping} \times THR_{Idle})$ then
3. sleep ($T_{Interval}$); continue; /*The server has idle resources*/
4. end if
5. if $CRS_{T_{Interval}} > PR_{Capping}$ then /*The server has been overloaded*/
6. $CR_{Exceed} \leftarrow CRS_{T_{Interval}} - PR_{Capping}$ /*The exceed resource*/
7. Find max res_{cj} in $CR_{ConUsage}$, and migrate cj to an idle server;
8. $PR_{Available} \leftarrow res_{cj} - CR_{Exceed}$ /*The free physical server resource*/
9. $C_{Total} \leftarrow C_{Total} - \{cj\}$
10. $CRS_{T_{Interval}} \leftarrow CRS_{T_{Interval}} - res_{cj}$
11. Array_I ← []
12. if $PR_{Available} < 0$ then
13. $CR_{Exceed} \leftarrow CR_{Exceed} - res_{cj}$
14. else then
15. for ck in P_{Total}
16. if $(p_{ck} \otimes cr_{Capping}) = NULL$ then /*No resource capping*/
17. $Wgt_{ck} \leftarrow \frac{res_{ck}}{CRS_{T_{Interval}}}$ /*The weight of res_{ck} */
18. $p_{ck} \leftarrow (cr_{MiniReq}, Max(cr_{MiniReq}, PR_{Capping} \times Wgt_{ck}))_{ck}$
19. else then
20. Add ck to Array_I
21. end if
22. end for
23. $CRS_{Capping} \leftarrow \sum_{i=1}^{m} (p_{ci} \otimes cr_{Capping})$ /* $CRS_{Capping}$ is the sum of the resource capping of all containers in C_{Total} */
24. $CRS_{Available} \leftarrow 0$; $\triangle Sum \leftarrow 0$;
25. if $CRS_{Capping} > PR_{Capping}$ then
26. for cq in Array_I
27. $Wgt_{cq} \leftarrow \frac{res_{cq}}{CRS_{T_{Interval}}}$
28. $p_{cq} \leftarrow (cr_{MiniReq}, Max(cr_{MiniReq}, PR_{Capping} \times Wgt_{cq}))_{cq}$
29. end for
30. end if
31. Array_II ← []
32. for ci in C_{Total} /*Calculate the available resources $(cr_{Available})_{ci}$ for every container*/
33. $(cr_{Available})_{ci} \leftarrow (p_{ci} \otimes cr_{Capping}) \times THR_{Efficient} - res_{ci}$
34. if $(cr_{Available})_{ci} > 0$ then
35. $CRS_{Available} \leftarrow \sum_{i=1}^{k} (cr_{Available})_{ci}$
36. $p_{ci} \leftarrow (cr_{MiniReq}, Max(cr_{MiniReq}, cr_{Capping} - (cr_{Available})_{ci} \times Wgt_{Available}))_{ci}$
37. Perform p_{ci} to ci
38. else if $(cr_{Available})_{ci} < 0$ then

39. $\vartriangle cr_{ci} \leftarrow \frac{res_{ci}}{THR_{Efficient}} - (p_{ci} \otimes cr_{Capping})$

40. $\vartriangle Sum \leftarrow \sum_{i=1}^{z} \vartriangle cr_{ci}$

41. Add ci to Array_II

42. end if

43. end for

44. $PR_{HostFree} \leftarrow PR_{Capping} - CRS_{Capping}$ /*The free resources in the Host system are not allocated to containers*/

45. $RS_{Available} \leftarrow CRS_{Available} + PR_{HostFree}$ /*All idle resources available for containers on this server*/

46. for ci in Array_II

47. if $PR_{HostFree} \geq \vartriangle Sum$ then /*Allocate from $PR_{HostFree}$ */

48. $p_{ci} \leftarrow (cr_{MiniReq}, cr_{Capping} + \vartriangle cr_{ci})_{ci}$ /

49. else if $RS_{Available} \geq \vartriangle Sum$ then/*Allocate from $RS_{Available}$ */

50. $p_{ci} \leftarrow (cr_{MiniReq}, cr_{Capping} + \vartriangle cr_{ci})_{ci}$

51. else if $RS_{Available} < \vartriangle Sum$ then/*Allocate from $RS_{Available}$ */

52. $p_{ci} \leftarrow (cr_{MiniReq}, cr_{Capping} + \frac{\vartriangle cr_{ci}}{\sum_{i=1}^{z} \vartriangle cr_{ci}} \times RS_{Available})_{ci}$

53. Perform p_{ci} to ci

54. end if

55. end for

56. end if

57. end if

58. sleep ($T_{Interval}$)

59. end while

satisfied.

Taking into account user experience, in an ARM server environment with massive resources, such as hundreds of CPU cores (e.g., Dual Ampere Altra Max processors), the idle resource determination coefficient THR_{Idle} can be further reduced and the effective resource coefficient $THR_{Efficient}$ can be further improved. When creating a container, optional configurable resource restriction policies for the container are available. If the container is not configured with a resource usage limit (it is assigned the value to *NULL*), when malicious programs within the container continuously consume resources, the corresponding resources on the host will be exhausted, which will also affect the normal running of other containers on the same host.

3.2 GPU Sharing Subsystem

The GPU Sharing Subsystem frameworks and comparison of the two methods (API remoting and GPU pass-through) are shown in Fig. 3.

The GPU resources are the essential factor to ensure 3D applications could run correctly, and video could be played more smoothly. To provide GPU resources (using a Nvidia or AMD GPU graphics card and corresponding drivers) to Android container, two methods are explored: (1) API remoting method (labelled Method I) and GPU pass-through method (Method II).

3.2.1 Method I: API Remoting

Based on the OpenGL ES libraries, OpenGL libraries, and GPU Drivers, to achieve GPU resource sharing in API forwarding mode, the following transmission process of the instruction stream is realized: When 3D applications (e.g., game) are launched in an Android container, a series of in-

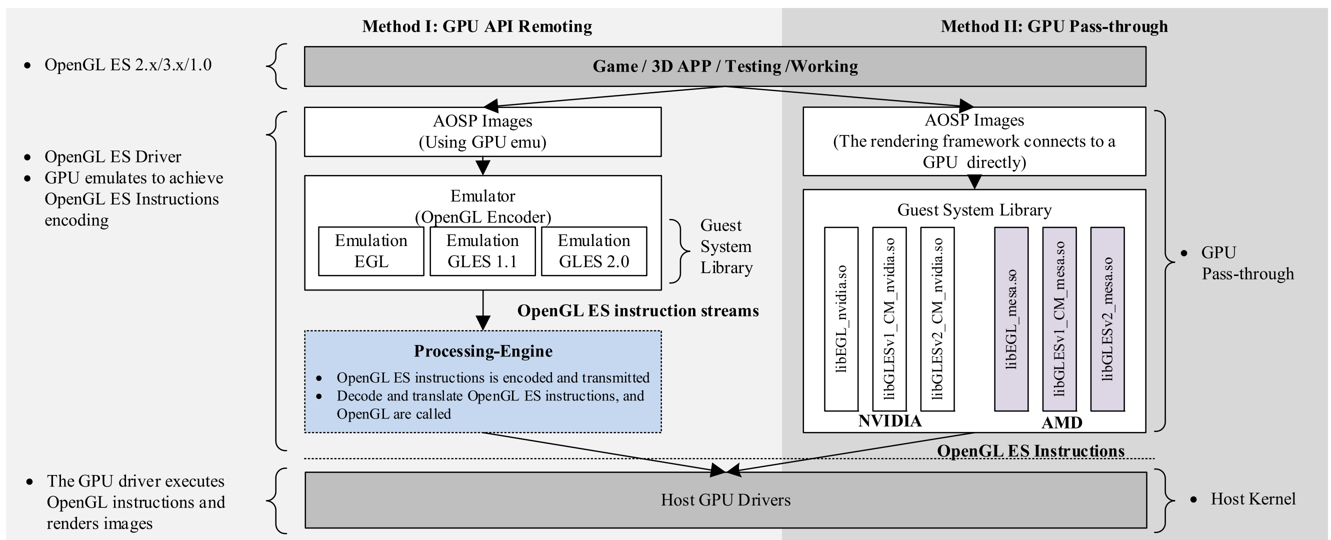


Fig. 3 GPU resource sharing subsystem.

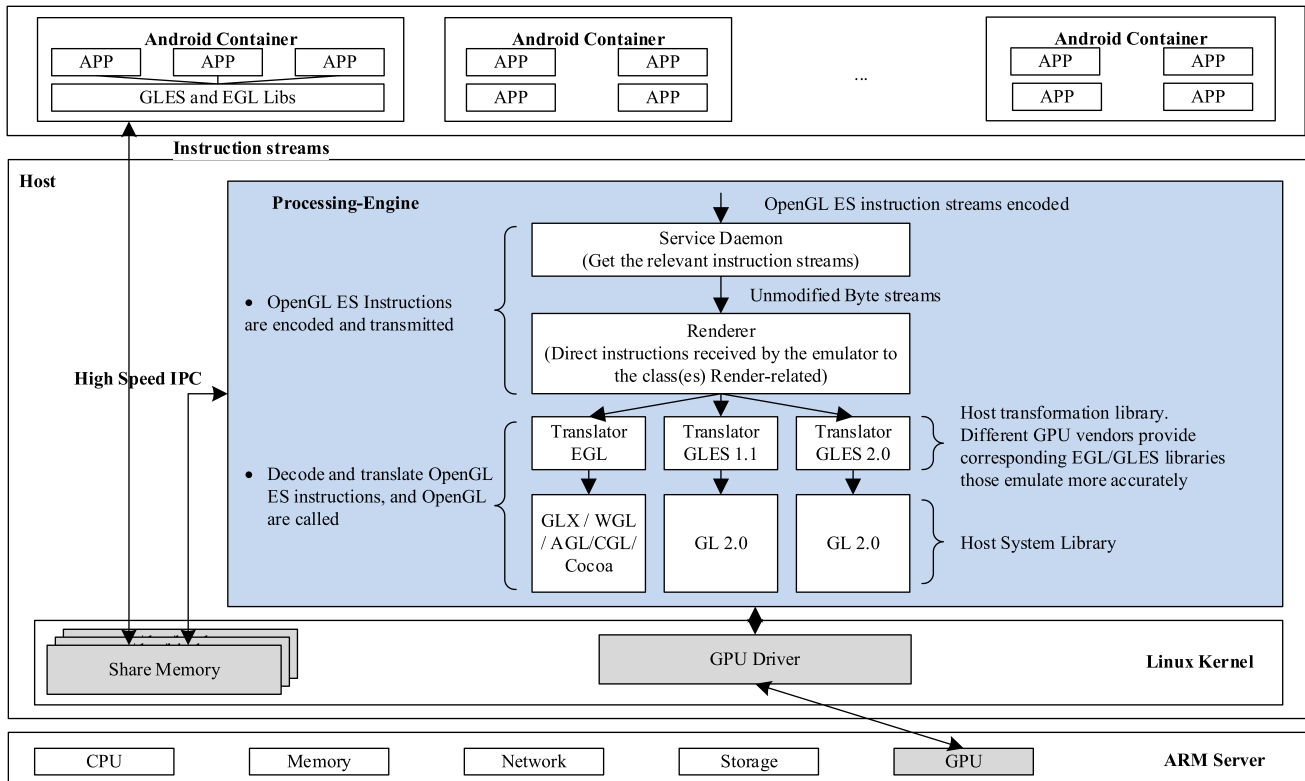


Fig. 4 The API remoting processing engine.

struction streams of OpenGL ES will be encoded firstly; then, they will be transmitted to Processing-Engine runs inside the host OS through a high-speed channel. Finally, the encoded instructions will be decoded, translated into OpenGL instructions, executed in the original sequence, and returned execution results. Due to the requirements of delivering OpenGL ES instruction stream to the middleware at a high speed must be satisfied, we employed the mechanism of share memory to realize the communication channel as shown in Fig. 4.

Specifically, when system service processes (surfaceflinger process, system_server process, gralloc process, etc.) inside Android and Applications are started, the graphics-related API will be called through libEGL.so. Then, according to Android properties settings, the corresponding functions in libEGL_emulation.so, libGLESv1_CM_emulation.so and libGLESv2_emulation.so will be called. After that, the function call will be encoded through libOpenglSystemCommon.so, and sent to the Processing-Engine on the Host through high-speed IPC channel. Finally, the Processing-Engine module decodes the function call and calls GPU driver to execute.

The advantage of method I is that a wide range of GPUs can be supported and without risk of GPU vender lock-in, which only depending on OpenGL features supported by GPUs and corresponding drivers. But the intermediate layer brings some performance loss.

3.2.2 Method II: GPU Pass-Through

The traditional x86 pass-through mainly distributes the designated GPU directly to the designated VM for independent use; and the PCI device is directly passed to the VM, which is responsible for the initialization, read-write and other operations of the device, and the entire device is monopolized by the VM. However, in a containerized environment, there is no process of device initialization; every Android system uses GPU card resources in the form of an application, which is equivalent to multiple applications using a GPU card at the same time to achieve resource sharing. Moreover, by the GPU pass-through method, a GPU device is needed to assign to an Android container by pass-through technology, and its driver is ported into the container for interacting with the Android's rendering framework.

Specifically, it is similar to the API-remoting process. Take AMD graphics card as an example, when system service processes and Applications inside Android are started, the corresponding functions in libEGL_mesa.so, libGLESv1_CM_mesa.so and libGLESv2_mesa.so from Mesa will be called through libEGL.so according to Android properties settings, and then those functions will be performed directly on AMD GPU through the matching driver.

To achieve GPU resource sharing in pass-through mode, the following works has been done: The focus is on compiling Mesa and installing it into the Android container, which is from AOSP (Android Open Source Project)

Codebase, and the stability of the program is improved (e.g., eliminated the issue of Android processes crashing when running some Apps).

The approach has higher efficiency and performance due to there is no intermediate layer for instruction redirection, while it requires a corresponding driver in every specific container. Pessimistically, the current adaptation progress is discouraging: it needs long time for vendors to develop and modify various GPU drivers (non-opensource). Moreover, the method charges according to the number of container instances, which is also an unavoidable economic cost issue, for example, according to the investment of NVIDIA's vGPU (virtual GPU) in VDI scenario, the overall investment of an instance (including hardware and software) is very expensive, which costs equivalent to or more than a PC, but brings poor performance and worse user experience. It is also an important inducement for the birth of farm solution.

3.3 Remote Transmission Subsystem

We took an Android terminal as an example to construct the transmission subsystem, as shown in Fig. 5. The framework has two modules: (1) An *Agent:Service* module running in the container, which is responsible to capture and encode screen images, audios, and inject control signals (e.g., keyboard, mouse, game pad); (2) An *Agent:Client* module running in the terminal, which is designed to decode video and sound, and intercept above control signals.

The two modules were deployed at different layers based on the following considerations:

(1) The *Agent:Service* module was deployed into Android application framework layer.

- Screen image capture: the image data of desktop or application interface could be captured in this layer by accessing "*GraphicBuffer*" object and "*Image_getLockedImage()*" function, which has advantages of minimal delay and is transparent to the application layer.
- Audio capture: audio could be captured by accessing "*AudioRecorder*" object directly.

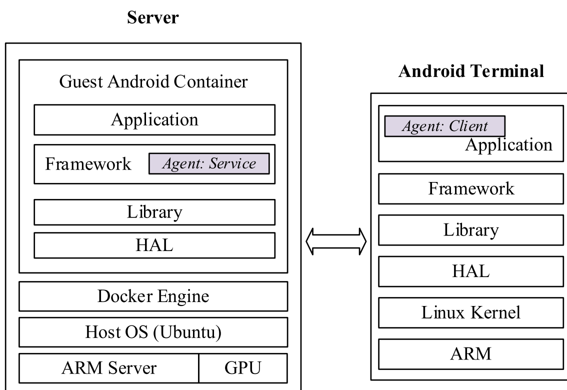


Fig. 5 Remote transport system framework.

- Control signals injection: after receiving the control signals, format conservation will be performed according to the data format specifications among different layers.

(2) The *Agent:Client* module was placed in the application layer of terminal.

- Image display: obtain the encoded video streams for decoding, rendering and display; there is an advantage of optimal compatibility in this layer.
- Audio playback: get the audio streams sent by *Agent:Server* and decode it.
- Control signals interception: call corresponding classes from the system library to intercept input events (e.g., keyboard, mouse, touch) directly, which has the advantage of the lowest complexity.

4. Experiments

According to Fig. 2, we employ two ARM server, Gigabit Ethernet switches, Gigabit NICs and GPUs to build an ARM cloud platform. The key hardware and software parameters as shown in Table 5 and Table 6. Other essential parameters will be given in the corresponding subsections.

4.1 Container Functions

A container was initialized on Server I with Nvidia GTX 1650 card, and key parameters of the container were: 2 CPUs, 8 GB memory capacity, 256 GB disk space; the container image quality was set to 720P@60 fps; container OS is Android 9 (Pie). The AIDA64 was used to verify GPU characteristics of container without/with GPU resource, as shown in Fig. 6 and Fig. 7 respectively, the container with GPU resources can support a higher version of OpenGL ES

Table 5 Hardware parameters of experiment platform.

Indicator	Server I	Server II
Model	Ampere Mt.Jade	Phytium R2215
CPU	2x Ampere Altra CPU (80 ARMv8 64-bit CPU cores, Max 3.0GHz, Current 2.8GHz)	1x FT-S2500 CPU (64 ARMv8 64-bit CPU cores, 2.1GHz)
Memory	256GB (DDR4 8x32GB)	Memory 32GB (DDR4 1x32GB)
Disk	2x Samsung PM983 960GB NVME SSD	1x Toshiba AL15SEB120N 1.2 TB SAS
GPU	NVIDIA: GeForce GTX 1650;	NVIDIA: GeForce GTX 1650; AMD: Radeon RX 590.

Table 6 Software parameters of experiment platform.

OS	Ubuntu 18.04.5 ARM64	
Kernel	5.3.0-26-generic	
GPU Drivers	NVIDIA	NVIDIA-Linux-aarch64-455.04.20-grid.run
	AMD	xserver-xorg-video-amdgpu_22.0.0-1_arm64.deb
Docker	20.10.2-0ubuntu1~18.04.2 arm64	
LXC	3.0.3-0ubuntu1~18.04.1 arm64	
Container OS	Android 9 (Pie), Android 12 (S)	
Benchmark Tools	AIDA64 for Android, version: 1.79	
	AnTuTu for Android, version: 9.3.1	
Game sample	Anonymous Me (a fighting game and published by Chengdu Dragonest Co., Ltd)	

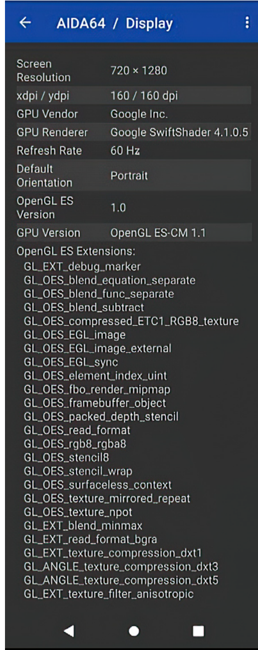


Fig. 6 The AIDA64 running in the container without GPU.

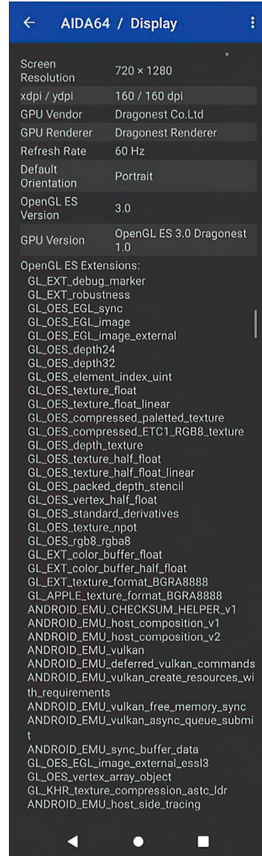


Fig. 7 The AIDA64 running in the container with GPU (API remoting).

(version: 3.0) and more features.

Then, AnTuTu benchmark tool was used to test container performance, due to the version of OpenGL ES supported by the container without GPU is only 1.0, AnTuTu was not functioning properly, while it could run in the container with GPU properly as shown in Fig. 8 (due to commercial copyright and license reasons, this section does not discuss the detailed practices of container with NVIDIA pass-through GPU at this stage, and taking AMD GPUs as alternatives to discuss in Sect. 4.3.).

Moreover, the mobile game Anonymous Me for Android was selected as a game sample, the terminals were Google Pixel and Huawei mobile phones. The key encoded parameters of video and audio are:

- Video: "*Preset: ultrafast Tune: zerolatency Profile:main Rate Control: ABR, bitrate:4000, vbv_buffer_size:4000*";
- Audio: "*opus 44.1khz stereo VBR*".

According to statistics, the cloud gaming consumed bit rate was about 4 Mbps, and SSIM (Structural Similarity) and PSNR (Peak Signal to Noise Ratio) between source images in container and decoded images in terminal are shown in Fig. 9 and Fig. 10. The average values of SSIM and PSNR are about 96% and 38.3 dB respectively; the average time delay is about 50 ms in our LAN test environment.

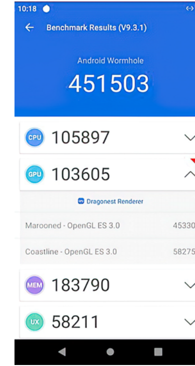


Fig. 8 The AnTuTu running in the container with GPU (API remoting).

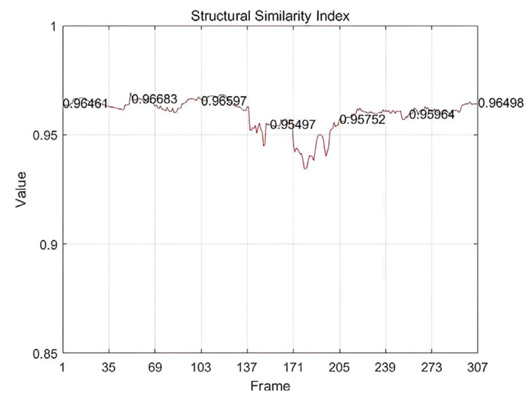


Fig. 9 The SSIM statistics.

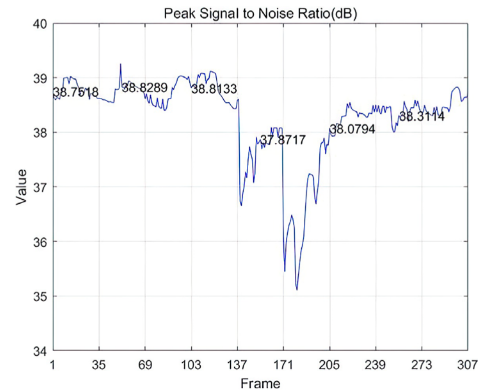


Fig. 10 The PSNR statistics.

4.2 Container Resource Adjustment

We applied the algorithm proposed in Sect. 3.1 to the following scenario: Initialize 9 Android containers inside Server I, and CPU quotas were configured too large and would cause the server overload. The monitor cycle $T_{Interval}$ = 10 *seconds*; the container OS is Android 9. The container specific parameters as shown in Table 7.

As shown in Fig. 11, all containers and the server have run in full load by running '*AndroidStressTest*' tool inside

Table 7 The CPU resource parameters of different containers.

Container ID	CPU (Cores)	More Information
c1	16	Total Cores (Current): 16 There would be a sudden demand for resources: Maximum CPU requirement was 24.
c2, c3	12	Total Cores: 24
c4, c5	16	Total Cores: 32
c6, c7	20	Total Cores: 40
c8, c9	24	Total Cores: 48
Total demand	Total Cores (Current): $160 > \left(80 \times 2 \times THR_{Efficient}\right) = 160 \times 0.9$	

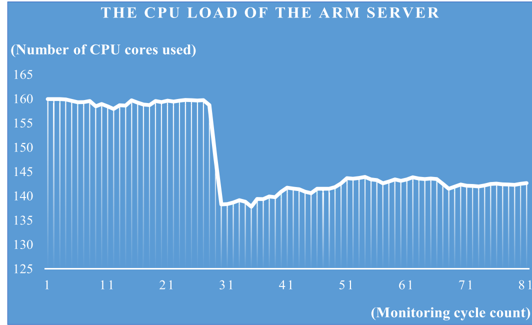


Fig. 11 CPU load curve of Server I.

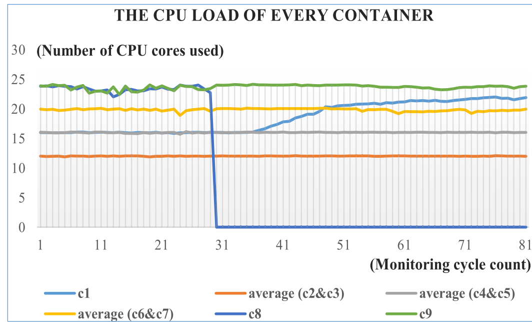


Fig. 12 CPU load curve of every container.

every container. After enabling the resource configuration adjustment algorithm in the 30th monitoring cycle, *c8* with the largest load was chose to be migrated, and the server load was reduced to less than 90%, and there were CPU resources remaining as shown in Fig. 12. Furthermore, a resource burst request event is simulated of container *c1* between the 31st and 38th monitoring cycles, which needs additional 8 CPU; at this juncture, according to the proposed algorithm, for *c1*, a new policy $p_{c1} = \left(cr_{MiniReq}, cr_{Capping} + \Delta cr_{c1}\right)_{c1}$ will be generated and executed, which would allocate resources to *c1* from free resources. As shown in Fig. 12, p_{c1} will be updated iteratively in a limited number of monitoring cycles. The final result is: Within the 90% load range of the server $THR_{Efficient}$, the resource request of *c1* was satisfied maximally.

Moreover, the setting of resource monitoring time intervals needs to be comprehensively judged based on specific business applications and the user's tolerable waiting time, such as the different tolerate delays of three remote

Fig. 13 Four of eight instances are running concurrently on Server II (from left to right, and from top to bottom, the terminals are Huawei Mate 20, Google Pixel 3, Huawei P40 pro and Huawei Mate 40 pro).

application scenarios in Table 3. If a very short-time interval is set, it will result in more computational overhead and worse user experience.

The current technological research focuses on single resource bottlenecks, such as CPU resource scheduling for compute-intensive tasks in the test case. The method can be extended to resource scheduling for network-intensive tasks and disk IO intensive tasks, as dynamic resource adjustment (Scale up or scale down) does not cause container process exceptions. However, for memory resource management, only online memory expansion operation is generally carried out. If online memory reduction is carried out, which will trigger '*out of memory*' exception.

According to the industrial practices, in order to reduce the difficulty of system management, easier to plan hardware, and reduce the difficulty of system fault handling, the vast majority of cloud platform tend to deploy the same type of computing tasks on the same server. Therefore, scheduling of single type resources is the first stage of our research. With the improvement of server performance, the comprehensive dynamic scheduling of multi-dimensional resources will also be the focus of the next research.

4.3 Parallel Performance

Several Android containers were initialized on Server II and key parameters were: 4 CPUs, 8 GB memory capacity, 256 GB disk space; container OS is Android 12 (S); the mobile game '*Anonymous Me*' for Android was selected. The detailed data of one test case (720*1280@60 fps) as shown in Fig. 13 and Table 8.

In the game scenario concurrency performance test, due to higher graphical requirements, Android 12 was selected as the container OS. However, for the following factors: (1) For Pass-through mode, NVIDIA GPU lacks drivers adapted for Android 12 that can only be provided by NVIDIA; (2) For API remoting mode, under the host OS environment of Ubuntu aarch64, AMD does not provide official driver; The currently available drivers are Mesa drivers compiled by the community, which cannot meet the realization requirements. Therefore, we choose AMD RX 590 and NVIDIA GTX 1650 with similar performance as

Table 8 The detailed test data of Anonymous Me cloud gaming in an Android container.

Key Parameters: Model: Pass-through; Image: 720*1280@60fps; GPU: AMD RX590.				
Number of parallel containers	Server II Load Average			Frames Per Second (FPS)
	GPU	CPU	Memory	
1	8~15	4%	10%	60FPS
2	17~24	7%	15%	60FPS
3	36~43	9%	20%	60FPS
4	57~65	15%	24%	60FPS
5	83~92	23%	28%	59~60FPS
6	97~98	30%	32%	50~55FPS
7	97~100	32%	36%	42~46FPS
8	97~100	34%	41%	36~39FPS

Table 9 The summarized statistics of Anonymous Me cloud gaming in Android containers.

Model	GPU	Parallel Tests	
		30FPS@720P	60FPS@720P
Pass-through	AMD RX 590	10	5
API Remoting	GeForce GTX 1650	2 (27~30FPS)	1 (46 ~ 48FPS)

the GPUs for the comparative test of the two modes: Pass-through mode uses AMD RX 590, and API remoting mode employs NVIDIA GTX 1650. The statistics summarized as shown in Table 9.

The test statistics show that the pass-through mode has higher performance: due to the API remoting mode will lead to the loss of many GPU features; and limited by the openness of the graphics card manufacturer to the driver. No matter what mode is employed, to obtain higher parallelism, it is necessary to conduct research and optimization in combination with the hardware, drivers, and 3D applications.

5. Conclusions and Future Work

Servers and terminals based on ARM have been deployed into cloud computing industry (e.g., cloud gaming, cloud mobile, cloud computer/cloud desktop). In this paper, to provide a complete reference for building ARM-based cloud, the mainstream solutions and key technologies were investigated and compared: for general computing resource virtualization, technologies and mainstream solutions of hypervisor model (e.g., Phytium solution, ARM/KVM) and container model (e.g., LXC, Docker, LXD, Ubuntu Touch, Anbox, Monbox, ReDroid) were discussed; an algorithm suitable for dynamic adjustment of partial container resources was implemented, which focused on single resource bottlenecks; for GPU virtualization, GPU pass-through and API remoting technologies were researched. Then, the remote transmission as an essential part of cloud service delivery was detailed analyzed (including scenarios of simple remote control, virtual desktop infrastructure and cloud gaming). Furthermore, a specific ARM-based cloud platform was implemented by employing technologies of the container, GPU pass-through and API remoting. Finally, the relevant practices had been done and the statistics proved the effectiveness of the framework and techniques.

Given that the IaaS platform is composed of independent physical servers, using the framework and realization methods proposed in this paper, a platform meets the requirements of massive resources could be built by horizontally expanding physical servers. Moreover, the division of security domains with the capacity of defense-in-depth for different business networks must be considered.

The formal description in the paper currently mainly focuses on the components of the system, with the goal of providing a conceptual and concise description of existing systems or solutions, and proposing essential components of the system. At present, there is no discussion on the detailed states of various elements in the system. Next, we will conduct in-depth research on the life cycle state transition of each element in the model (such as building a finite-state machine and state transition diagram for each element), security constraints, formal verification or reduction, etc.

Similar to the ecological development of X86 architecture chips, the success of ARM-based virtualization (e.g., the high rate of adoption, wide range of applications) mainly depends on the factor of cost performance (price-performance ratio), which is determined by factors of the device price, virtualization efficiency and number of parallel instances, especially for GPU resources. With the support of container technology, the parallel performance of ARM cloud servers (the cost-effectiveness of single-channel cloud service) is mainly limited by the management and allocation of GPU resources. For different scenarios, implementing fine-grained virtualization technology from GPU cores (CUDA cores) and GPU memory are feasible optimization methods, so as to improve the energy efficiency and user experience for general-purpose computing and graphics rendering scenes respectively, while they require the support of graphics card manufacturers. In addition, in graphical interactive scenarios such as cloud gaming and cloud design, by introducing a dedicated encoding card, the GPU only needs to do the rendering work, then the density of ARM server cloud services cloud be further increased. In the future, we will carry out related research work in the above fields.

Acknowledgments

This work was supported in part by the Science and Technology Service Industry Demonstration Project of Sichuan Province of China under grant No. 2021GFW023 and in part by the Arm Technology (China) Joint Research Project under grant No. CMA-CLA-01337.

References

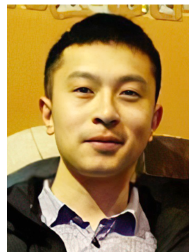
[1] J.P. Huai, Q. Li, and C.M. Hu, "Research and design on hypervisor based virtual computing environment," J. Software, vol.18, no.8, pp.2016–2026, Aug. 2007. DOI: 10.1360/jos182016
[2] B.H. Yang, W.J. Dai, and Y.L. Cao, Docker Primer, pp.4–9, China Machine Press, Beijing, China, 2018.
[3] Y.H. Chen, "Development of container-based virtualization technology," ICT Journal, vol.1, no.167, pp.11–16, Sept. 2016.
[4] Wikipedia, "OS-level virtualization," https://en.wikipedia.org/wiki/OS-level_virtualization, accessed Feb. 2. 2023.
[5] H. Yin, "Research on virtualization technology based on the server

made in China," Electron. World, vol.23, no.1, pp.103–104, Feb. 2020.
[6] Phytium, OpenGCC, From Terminal to Cloud: Full Stack Solution White Paper Based on Phytium Platform, 2nd ed., pp.1–81, Phytium, Tianjin, China, 2020.
[7] Y.F. Huang, "Virtualization on ARM machine platform," M.S. thesis, Dept. Computer Science & Technology, Huazhong Sci.& Tec., Hubei, China, 2019. DOI: 10.27157/d.cnki.ghzku.2019.003846
[8] H.Y. Ma, "Experiments about the performance of full virtualization and containers on ARM-v8 platform," Proc. 2019 ITEEE, Sanya, China, pp.394–403, 2019. DOI: 10.12783/DTCSE/ITEEE2019/28777
[9] F. Olivier, M. Fabien, and N. Florent, "A review of native container security for running applications," Proc. 17th MobiSPC, pp.157–164, Leuven, Belgium, 2020. DOI: 10.1016/j.procs.2020.07.025
[10] E. Alexia, "Infographic: LXD Machine containers from Ubuntu," https://ubuntu.com/blog/infographic-lxd-machine-containers-from-ubuntu, accessed Feb. 2. 2023.
[11] Faun., "Container Architecture," https://wiki.ubuntu.com/Touch/ContainerArchitecture, accessed Feb. 2. 2023.
[12] AMPERE, NETINT, Android in the cloud on ARM native servers, 1st ed., London, United Kingdom, Canonical, 2020.
[13] F. Simon, "Anbox," https://github.com/anbox/anbox, accessed Feb. 2. 2023.
[14] J.D. Chen, "Huawei cloud: Kunpeng cloud mobile phone architecture decryption," Proc. HDC Together, pp.1–16. Guangdong, China, 2020.
[15] W.Z. Chen, L. Xu, G.X. Li, and Y. Xiang, "A light-weight virtualization solution for Android devices," IEEE Trans. Comput., vol.64, no.10, pp.2741–2751, Oct. 2015.
[16] K.M. Shi, "The design and implementation of Android desktop cloud system based on LXC," M.S. thesis, Dept. Computer Science & Technology, Zhejiang University, Hangzhou, China, 2018.
[17] Arclab, "Condroid," http://condroid.github.io/, accessed Feb. 2. 2023.
[18] T. Ian, iFan, H. Victor, "Android Containerization," https://github.com/clondroid, accessed Feb. 2. 2023.
[19] X.Y. Yin, X.S. Chen, L. Chen, G. Shao, H. Li, and S. Tao, "Research of security as a service for VMs in IaaS platform," IEEE Access, vol.6, pp.29158–29172, 2018. DOI: 10.1109/ACCESS.2018.2837039
[20] X.Y. Yin, X.S. Chen, L. Chen, and H. Li, "Extension of research on security as a service for VMs in IaaS platform," Security and Communication Networks, vol.2020, pp.1–16, 2020. DOI: 10.1155/2020/8538519
[21] B. Peng, P. Yang, Z.C. Ma, and J.G. Yao, "Performance measurement and analysis of ARM embedded platform using Docker container," Journal of Computer Applications, vol.37, no.S1, pp.325–330, June 2017.
[22] L. Xu, Z.H. Wang, and W.Z. Chen, "The study and evaluation of ARM-based mobile virtualization," International Journal of Distributed Sensor Networks, vol.2015, no.1, pp.1–11, Oct. 2014.
[23] Z.Y. Zhou, "Remote-android," https://github.com/remote-android, accessed Feb. 2. 2023.
[24] K. Su, "Research on virtualization techniques for cloud desktop services," Ph.D. dissertation, Dept. Computer Science & Technology, Zhejiang University, Hangzhou, China, 2017.

Lin Chen was born in Yibin, China, in 1983. He received his PhD. degree in computer science and technology from Sichuan University, Chengdu, China. He is currently VP of Chengdu Xingzhe AI Co., Ltd., China. His research interests include cloud computing, network security and SDN.

Xueyuan Yin was born in Yunnan, China, in 1988. He received his PhD. degree in computer science and technology from Sichuan University, Chengdu, China. He is currently CEO of Xingzhe AI and CTO of Dragonest Co., Ltd., China. His main research interests include cloud computing and network security.

Dandan Zhao was born in Xinxiang, China, in 1993. She received her M.S. degree in computer technology from Sichuan University, Chengdu, China. She is currently working for Lingyue Cloud Co., Ltd., China. Her research interests include Linux OS, cloud computing and security.



Hongwei Lu was born in Jianyang, China. He received his M.S. degree in Microelectronic & Solid State Electronic from Zhejiang University, Zhejiang, China. He is currently working for Lingyue Cloud Co., Ltd., China. His research interests include cloud computing and security.

Lu Li was born in Chengdu, China. He is currently working for Chengdu Lingyue Cloud Co., Ltd., China. His research interests include cloud computing, audio and video transmission.

Yixiang Chen was born in Shanghai, China. He is currently working for Arm Technology (China) Co. LTD, Shanghai, China. His research interests include cloud computing, ARM virtualization technology and cloud gaming.