2021 Zooming Innovation in Consumer Technologies Conference (ZINC) | 978-1-6654-0417-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ZINC52049.2021.9499302

Linux container solution for running Android applications on an automotive platform

1st Srdjan Usorac
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
srdjan.usorac@rt-rk.uns.ac.rs

2nd Bogdan Pavkovic
Faculty of Technical Sciences
University of Novi Sad
Novi Sad, Serbia
bogdan.pavkovic@rt-rk.uns.ac.rs

***Abstract*—An existing problem in the automotive industry is how to use Android alongside Linux operating system on a single SoC. The challenge is especially exacerbated between the driver assist applications (developed on a variant of a real-time Linux) and infotainment applications (developed on the Android). Due to the rapid surge of the automotive industry in the past years, there is already an existing solution on the market that uses system hypervisors to run two or more operating systems side by side on a single SoC. Hypervisors have resources shared between the two systems, which is a major bottleneck point for further system optimization. On the other hand, several existing open-source solutions enable Android to run on top of the Linux operating system inside a separate container, with the benefit of dynamic resource allocation. We give an overview of the current development state of container-based solutions and propose one solution for a new approach: running Android inside Linux container technology on an automotive-grade platform. Our proposed approach integrates different classes of automotive applications and enables secure inter-system communication on top of container-based technology. Our future work will expand the implementation of our current solution with additional Android features, to run more demanding Android applications in a secure Linux environment.**

***Index Terms*—Automotive, Android, Hypervisor, Linux Container**

I. Introduction

The software for autonomous cars can be broadly separated into two major regions: automotive machine vision (AMV) and in-vehicle infotainment (IVI). AMV region usually has a dedicated system on chip (SoC) in charge of managing advanced driver assistance systems (ADAS) i.e. essential systems for autonomous driving. AMV region usually has a Linux-based real-time operating system (OS) support [1]. On the other hand, the IVI region provides driving information and consumer entertainment inside a vehicle. The infotainment region usually has an Android operating system ported on a separate SoC, with support for consumer quality applications.

Although two distinct in-vehicle regions are designed for completely different purposes, the interaction between regions is necessary in several cases. For example, the information about camera objects, provided by the machine-vision algorithm, needs to be displayed on the main driver display subsystem (provided by the Android automotive application). Traditionally, the only way for the two different regions to interact is through an external bus connector. We consider that such an approach is insufficiently performative when considering high-range ADAS algorithms.

A solution for interaction between two regions problem already exists, through the usage of hypervisors (embedded system virtualization mechanism) [5]. Hypervisors enable both Android and Linux operating systems to coexists on a single SoC [7]. Furthermore, QNX hypervisors have applications in the automotive domain, supporting parallel usage of both Android and Real-Time Operating System (RTOS) [3]. Inter-OS communication is established using inter-process communication (IPC) mechanisms. Information about device driver surroundings is also provided. QNX hypervisors have been evaluated on the Qualcomm Snapdragon SoC, a commonly used platform in today's automotive industry [4].

With hypervisors, both operating systems can share graphical information between different applications [2]. When using hypervisor technology for inter-operating system communication, secure rendering of multimedia is possible [6].

Even though the hypervisor solution is currently widely used as a solution for porting the Android operating system to the IVI domain of an automotive environment, it has its limitations. The concept of using hypervisors is that system resources are statically distributed across two or more selected guest operating systems [7]. Each of the two operating systems uses CPU and RAM given by the system hypervisor. To overcome the shortcoming of static resource allocation, further research has shifted towards dynamic allocation of system resources, managed by a single host operating system [9].

In this paper, we propose an integrated system approach for placing Android inside a containerized environment. We propose a container-based solution to be implemented as a part of the Linux host system, which runs on top of an automotive-based SoC. By using container-based technology [9], we will be able to provide functionalities such as:

- Android and Linux operating systems will both share the same kernel of the host system.
- Different classes of algorithms will be able to communicate with an already running Linux host system. These different classes include ADAS algorithms (on Linux) and cluster display applications (on Android).
- Data sharing will be enabled between the two integrated systems.

- Only a single host operating system runs directly on SoC, while the other system runs as a process.

II. Related Work

Hypervisors are a widely accepted solution in the automotive industry, used for developing Android and Linux on the same SoC [8]. Hypervisors manage the integration of IVI and ADAS functionalities [7]. Furthermore, the hypervisor virtualization mechanism provides support for secured rendering graphics and multimedia between guest operating systems [6] [2]. Consequently, research results demonstrate that it is possible to shield mission-critical software from uncritical software applications, which co-exist on the same SoC [8]. However, hypervisors introduce restrictions in static memory distribution across multiple guest operating systems [8].

Running Android inside the container has been a focus for certain research groups for the past several years. Linux Container (LXC) [10] is an operating system virtualization technology, managing the run of a guest operating system on top of an already present host operating system. LXC container doesn't provide full operating system virtualization, since it uses the host system kernel. LXC technology establishes this partial virtualization scheme by using two characteristics, provided by the kernel:

- **Control groups (Cgroups)** - used to limit, account, isolate, and control system resource usage, such as CPU and RAM, for a certain process group. LXC containers utilize Cgroups to manage custom system resource allocation. Also, this technology provides dynamic resource allocation management.
- **Namespaces** - technology that controls the visibility of system resources to the system process. LXC utilizes the rich toolset of Namespaces to run its operating system as a custom process with its own resource sets, isolated from the host operating system environment.

LXC has access to host system resources through a special container configuration mechanism. The host kernel needs to support Cgroups and Namespaces for LXC to be functional in the user-space. LXC user-space tools provide the guest operating system its root filesystem [10]. By default, the container provides a minimal set of kernel device drivers needed to run a Guest operating system. Finally, the option is given to run the Android operating system inside the LXC container. We just need to modify the required kernel and user space utilities and deploy the required Android images to the container.

The foundation of LXC has been covered by many research groups. Heavy argumentation is given on why containers have the edge over full virtualization solutions, such as system hypervisors [10]. The basic argument is that through LXC, host operating systems could dictate the number of shared system resources for guest operating systems (CPU, RAM). A good example of deployment of Android inside the container comes from the mobile device domain, where the SD card has been used for booting of Android [10].

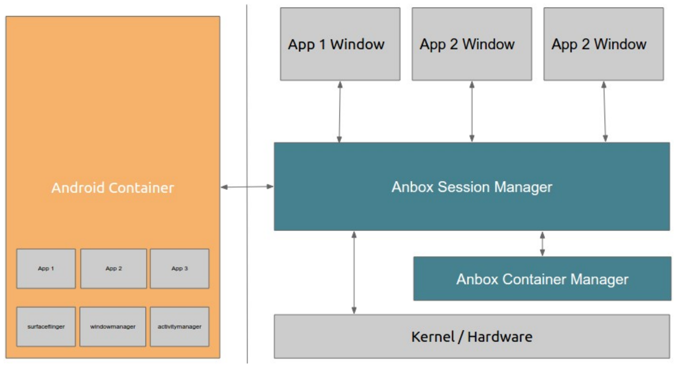

Fig. 1. Anbox open-source solution architecture [11]

Anbox (Android in a box) is an open-source solution that puts the Android system inside LXC container [11]. By abstracting hardware access and integrating core system services, Anbox establishes communication with a GNU/Linux host operating system. Integration of Android services is done through the usage of SDL (Simple DirectMedia Layer) [12]. SDL is a Linux host system library that provides access to system graphics, audio, and peripherals.

Anbox is implemented as a custom Android device, brought up from the Android Open Source Project (AOSP). One of the main key points is that the Anbox inherits from the Goldfish device. Goldfish device is mainly used to run the Android system inside the AOSP emulator. Because of Goldfish inheritance, Anbox is using the same graphics acceleration stack as the Android emulator, the android-emugl. Android-emugl uses special QEMU pipes technology for the optimized rendering of the Android graphical stack.

On the Linux host system side, the SwiftShader library is used to facilitate software-based graphical rendering to the host system [13]. This hardware-independent 3D graphical rendering makes the Anbox portable to a variety of system architectures, regardless of proprietary graphics drivers used.

In Figure 1, we displayed the architecture of the Anbox solution. Anbox solution architecture consists of two major parts:

- **Container Manager** - core component for monitoring the run of LXC container. It runs on the host system and manages mount points and network bridge communication with Android inside LXC.
- **Session Manager** - host system application in charge of communication between Android guest and Linux host operating system. This component enables Android applications to appear on the host system graphical shell environment as separate application windows.

Although we consider Anbox as a great example of Android container technology, it has a setback regarding system performance. As said in the previous segment, Anbox inherits the Goldfish device as a starting point, with an emulator background. Although Anbox has successfully implemented graphics acceleration, it is still based around software rendering which utilizes CPU core component.

In the automotive industry, the real-time performance of critical applications is of utmost importance. Automotive applications should have indirect access to hardware resources (such as GPU for graphics), which will result in better overall performance. Furthermore, the Anbox device is currently compatible only with Android 7.1.1. while Android 10 is widely being used in the automotive industry for SoC SDK support.

Building on the Anbox solution, Chau et al. [14] have focused on defining an approach for running Android applications inside LXC, with direct access to system peripherals. In other words, a guest operating system, Android, will have direct access to hardware peripherals such as GPU, audio/video cards, etc. A mechanism of applying control group rules is defined in the LXC configuration. Authors provide a full performance description of Android inside LXC, compared with Android emulators, and even real mobile phone devices [14].

In our proposal, we consider porting Android inside LXC because of better performance and extension of real physical hardware limitations. We build up our solution on the shoulders of the presented related work. We managed to create a concept for running Android applications inside LXC on the host operating system, with direct hardware access, and for specific deployment in an automotive environment.

III. Proposed Solution

We propose a solution for running Android inside LXC on top of the Linux operating system, with direct access to selected host system resources. We present our solution architecture in Figure 2. We could see from the solution architecture that both host and guest container operating systems use a common kernel. Our solution architecture consists of two parts: Linux host environment, Android container device.

Starting from the bottom of the architecture, the first component is the kernel. For our solution, the kernel needs to support the following features:

- **Cgroups** and **Namespaces** - crucial features for LXC support.
- **Ashmem** - (Android Shared Memory) is a shared memory allocation used inside the Android operating system.
- **Binder** - inter-process communication (IPC) mechanism used for accessing different layers of the Android operating system.
- **Binderfs** - filesystem for Android binder IPC, which enables dynamic addition/removal of binder devices inside different IPC namespaces.

Our solution uses Wayland as the default display server. This means we are targeting automotive platforms which have Wayland support enabled. SPURV project also chose Wayland as the default host system display server [15]. We inherited this approach, considering that there is already a working example, and also implemented support for Wayland [15].

The IVI-Shell is a key component of our system since it gives support for touch input events from the Wayland display server and forwards events to Android. We also have LXC library support on the Linux host system, with options for mounting device drivers to the guest system, bridging network connection, and preparing the Android filesystem for the boot process. Container Manager is a user-space application in charge of configuring the LXC container, executing, and maintaining a stable run of the Android operating system inside the container. Also, Container Manager is in charge of forwarding graphics information from Android to the LXC.

The Android container device is an Android operating system, suitable for running inside the LXC container. This device is configured from the Android Open Source Project (AOSP) official release version 10. The two key configurations that we added to the Android container device are Linux host internet support and an adaptation of the Android graphical subsystem.

Our solution architecture aims to make Android applications act like any other applications running on the Linux host system. Thus, it is essential to provide a portable bridge communication for the Android graphical subsystem. We configured components from the Android Native and HAL layers. We implemented a custom Android HWComposer, that acts as a Wayland client, displaying data on the Wayland display server [15]. We configured the Graphics Engine to communicate through the LXC directly with the host system GPU driver.

The bridge communication for both HWComposer and Graphics Engine components is achieved through Container Manager, which forwards information to the LXC. After that, the LXC forwards graphical information to the host GPU driver through mount points. LXC also notifies the host system Wayland server about new display information.

With our proposed solution, Android and Linux applications co-exist on GNU/Linux host system. By using our proposed solution, we can demonstrate the feasibility of a typical automotive use case: the Linux parking assistance application providing information to the Android display map application. Finally, we can share data between different classes of automotive applications without the need for a different SoC development environment.

IV. Implementation

In the previous chapter, we described the architecture of our solution. By using Linux containers, we can mount device partitions from Android directly onto the Linux host system. In this chapter, we present the implementation of our solution on Qualcomm Snapdragon 8155 automotive reference board [4]. With the platform, we got access to Qualcomm Technologies (QTI) yocto project, which enabled us to make our own Linux system distribution. We applied the required modifications to support LXC technology, using cgroups and namespaces.

In parallel, we worked on configuring our Android device for the container. One of the major challenges we had was to adapt the Android system to support the graphics environment of our Qualcomm board. We managed to overcome this issue by configuring Wayland as our primary Linux display environment because Qualcomm Linux has support for Wayland. As described in the previous chapter, we implemented a

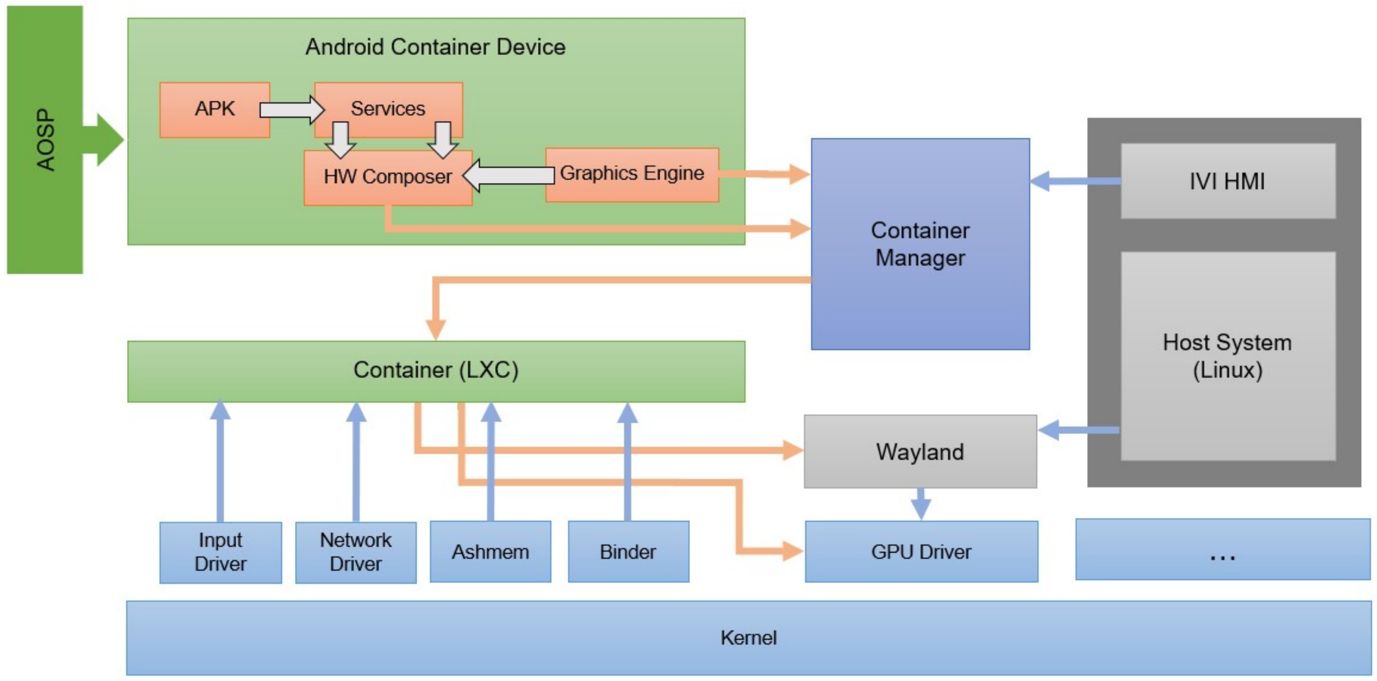

Fig. 2. Proposed solution for Android inside container

custom HWComposer that is in charge of displaying Android graphical information on the Linux host system.

After bringing up both the Linux and the Android side of the solution, we have all the required data to run Android inside LXC on the Qualcomm board. In Figure 3, our solution test case is presented. We currently have support for simple Android graphics and news applications. We implemented a custom launcher subsystem, which has both Android and Linux application icons presented. This shows that both Linux and Android applications could be executed in the same way, when pressing desired application icon.

V. Future Work

We currently have implemented an Android container solution on the Qualcomm reference board. This platform has support for the QNX hypervisor system. We are currently in the process of configuring hypervisor with Android and Linux guest operating systems. After successful implementation, we plan on making available the performance comparison between our proposed solution, basic Android in LXC, and Android hypervisor. Furthermore, we plan to highlight further pros and cons observed in a real-world experimental environment.

Next, we are planning to optimize our custom Android system. Our goal is to boot Android in LXC with a minimal set of services, and packages needed for the system to function. For a set of user-attractive applications, we will disable as many system services as possible.

After successful optimization, we will continue to expand the Android container to utilize new features, such as GPS, Audio, Video, and Bluetooth. Finally, we plan to focus our future research work on inter-OS security, regarding communication between a QM system (Android) and an ASIL-B system (Automotive Grade Linux).

VI. Acknowledgement

This research(paper) has been supported by the Ministry of Education, Science and Technological Development through the project no. 451-03-68/2020-14/200156: “Innovative scientific and artistic research from the FTS (activity) domain”.

References

[1] M. Milosevic, M. Z. Bjelica, T. Maruna and N. Teslic, ”Software Platform for Heterogeneous In-Vehicle Environments,” in IEEE Transactions on Consumer Electronics, vol. 64, no. 2, pp. 213-221, May 2018.
[2] M. Z. Manic, M. Z. Ponos, M. Z. Bjelica and D. Samardzija, ”Proposal for graphics sharing in a mixed criticality automotive digital cockpit,” 2020 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 2020, pp. 1-4.
[3] N. Pajic and M. Bjelica, ”Integrating Android to Next Generation Vehicles,” 2018 Zooming Innovation in Consumer Technologies Conference (ZINC), Novi Sad, Serbia, 2018, pp. 152-155.
[4] Lantronix. ”SA8155P Automotive Development Platform.” https://www.lantronix.com/products/sa8155p-automotive-development-platform (accessed January 29, 2021).
[5] P. Yu et al., ”Real-time Enhancement for Xen Hypervisor,” 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Hong Kong, China, 2010, pp. 23-30.
[6] M. Ilic, T. Andjelic, N. Zmukic and M. Z. Bjelica, ”Support for rendering multimedia at digital vehicle instrument cluster,” 2017 25th Telecommunication Forum (TELFOR), Belgrade, Serbia, 2017, pp. 1-4.
[7] S. Karthik et al., ”Hypervisor based approach for integrated cockpit solutions,” 2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), Berlin, Germany, 2018, pp. 1-6.
[8] K. Lampka and A. Lackorzynski, ”Using Hypervisor Technology for Safe and Secure Deployment of High-Performance Multicore Platforms in Future Vehicles,” 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genoa, Italy, 2019, pp. 783-786.
[9] Á. Kovács, ”Comparison of different Linux containers,” 2017 40th International Conference on Telecommunications and Signal Processing (TSP), Barcelona, Spain, 2017, pp. 47-51.

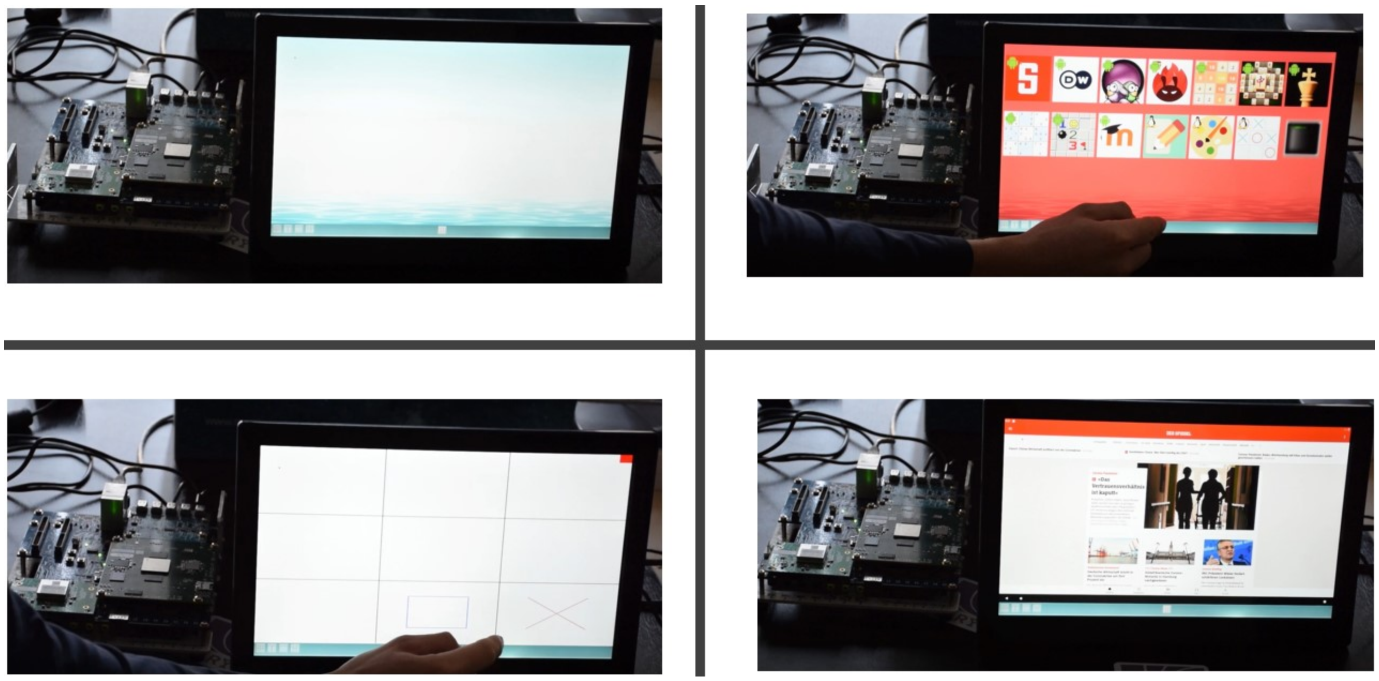

Fig. 3. Android container on Qualcomm SA8155: default Linux session (upper-left), list of Android/Linux applications (upper-right), running Linux application (lower-left), running Android application (lower-right)

[10] Xin Li, Hee-Kyung Moon and Sung-Kook Han, ”A Design and Implementation of Universal Container,” Information Technology and Computer Science (ITCS), Amman, Jordan, 2016, pp. 137-143.

[11] GitHub. ”Anbox.” https://github.com/anbox/anbox (accessed January 29, 2021).

[12] libSDL. ”Simple DirectMedia Layer.” https://www.libsdl.org (accessed January 29, 2021).

[13] SwiftShader. ”SwiftShader.” https://swiftshader.googlesource.com/SwiftShader (accessed January 29, 2021).

[14] Ngoc-Tu Chau and Souhwan Jung, ”Dynamic analysis with Android container: Challenges and opportunities,” Digital Investigation, vol. 27, 2018, pp. 38-46.

[15] Robert Foss. ”Running Android next to Wayland.” https://www.collabora.com/news-and-blog/blog/2019/04/01/running-android-next-to-wayland/ (accessed January 30, 2021).