Operating System Convergence: An Example via the Maru OS Project

William White
Core Kernel / RTS Team
Red Hat, US
Auburn University, AL
chwhite@redhat.com

Xiao Qin
Department of Computer Science and Software Engineering
Auburn University, AL
xqin@auburn.edu

2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00163

***Abstract*—Converging multiple form-factors and devices into one cohesive system is not necessarily a new idea; however, there has been a growing interest in achieving the design goal of convergence in recent years. This growing interest is due partly to a desire to maintain only one operating system. Some implementations also include the ability to run both in desktop mode and mobile mode simultaneously. These mobile systems attach to a lapdock or monitor via one or more cables to activate the desktop mode. These converged systems have mobile hardware constraints; however, current system on chip (SOC) improvements may make running these systems much more realistic. Several techniques are available to achieve convergence. This paper attempts to outline these design principles while also presenting an example architecture for using virtualization techniques to achieve operating system convergence. Additionally, this paper proposes a solution to attach the mobile device to the lapdock wirelessly using Bluetooth and Miracast. The virtualization techniques used to achieve this convergence can apply to various form-factors, application ecosystems, and computational platforms.**

***Index Terms*—Operating System Convergence, Linux, Android, LXC, OS-level Virtualization, IoT, Containers, Application Ecosystem, Pixel 2, Application Compatibility Layers, Virtualization, Convergence, Lapdock, Miracast**

I. Introduction

Maru OS [1] is a project that runs a Linux container on top of an Android kernel. Isolation between the Android and Linux userspace works by running the Linux-specific userspace packages inside an LXC container. LXC provides os-level virtualization (Also known as virtual environments) in the form of isolated containers. A single host operating system (The Linux Kernel) shares resources between an Android and Linux userspace. Consider that Android is just a Linux distribution with a custom patchset applied to an upstream Linux Kernel. This fact means that Android can include any of the features that a modern Linux operating system contains, including the ability to isolate user space environments on top of a shared kernel. Linux has multiple container implementations that have evolved over the years. This project focuses on LXC because LXC virtualizes sets of userspace processes. It does so in a similar manner as a virtual machine, without the duplication of additional operating system resources. LXC is already in the Linux Kernel; however, Android disables many parts of the upstream kernel, including LXC. To make LXC work, the team made modifications to the Android kernel configs to include the LXC configuration. From here, an isolated LXC container launches that contain a complete workstation or desktop-like distribution that runs on top of the Android kernel. Such an implementation is called operating system convergence.

There have been a handful of projects that have appropriately implemented convergence. Such projects include Windows Continuum, Ubuntu Touch, and Samsung Dex. Each project has had a fatal flaw: A lack of application support or application ecosystem. It does not matter how exemplary the implementation is; no one will use it if the system does not have a full-fledged app ecosystem. The Windows Mobile OS project and Windows Continuum never hit mainstream success because Microsoft could not provide an equivalent application ecosystem as the competition (Android and iOS). Samsung Dex has a fantastic mobile application ecosystem; however, it lacks desktop application support. Ubuntu Touch has come the closest regarding app support; however, it does not provide a traditional UI for end-users to use. A proper system should also include a conventional mobile UI that is (architecturally speaking) decoupled from the underlying system. This paper attempts to outline the architecture of a system that provides both a robust application ecosystem and a UI that is familiar enough for the end-user to use. A system can achieve convergence through a bottom-up rewrite of the diverged operating systems, or virtualization techniques can be used (operating system convergence).

It is important to note that Maru OS is an open-source project initially created by Preetam D'Souza, and as such, this paper is based on existing open-source solutions. The primary focus of this paper is to give a holistic view of the Maru OS architecture. Additionally, this paper highlights different techniques used in converging systems and building compatibility layers. This paper makes the following four direct contributions to the Maru OS project:

1) *Pixel 2/XL:* We ported the Pixel 2/XL to Maru OS.
2) *Wireless Lapdock Proposal:* We provide an architectural diagram and solution to use devices completely wirelessly with a lapdock device.
3) *Modernizing the Maru OS Framework:* We investigate rebasing the project on Halium and Lineage 17.1.
4) *Architectural Outline:* We documented and outlined the

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00163

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

Maru OS project's architecture, which is currently not thoroughly outlined.

II. Background Information

A. What are Virtual Environments?

Understanding how a virtual environment behaves is crucial to understanding this project. Simply put, a virtual environment is a type of os-level virtualization where a set of userspace processes group together to form a sort of container. Each resulting container shares splices of the underlying kernel, hence the name os-level virtualization. A container implementation does not have to include strong isolation; however, strong isolation is a desirable trait in all modern container implementations. We define *strong isolation* as a mechanism that restricts a defined set of userspace processes from accessing other undefined processes and from accessing undefined hardware resources (resources not assigned to the virtual environment), as well as restricting other userspace processes from accessing the original set. While ungraceful exits can theoretically occur (And have been exploited in the past), such exits are considered bugs (exploits) and are not allowed by a properly isolated container. A classic example of an unisolated virtual environment would be Linux's chroot command, where the root context changes from the host's root to a new directory that acts as the virtual environment's new root directory [2]. Linux provides strong isolation functionality through two main features: namespaces and cgroups. Namespaces allow for process-level isolation while cgroups allow for hardware-level isolation (Resource allocation done on a container level instead of an operating system level). Two examples of container runtimes that take advantage of strong isolation are LXC and Docker [2]. Many people may be familiar with Docker, where the intent is to run just enough dependent processes to get applications running. This design is excellent for the current cloud computing model (cloud-native), where you want to deploy just what you need for your web application. Docker represents a type of container called application containers. However, this type of container is not holistic enough to virtualize an entire desktop environment. Here, LXC shines as it provides a container runtime designed to run a complete system, instead of a single process, inside a virtual environment. This type of container is called a system container. In this sense, we can achieve the same functionality with LXC as traditional virtual machines as long as our processes are compatible with the underlying operating system. Moreover, we can accomplish this without duplicating additional kernel-level resources making for a more performant system [2].

It is important to note that a key architectural feature that makes all of this possible is Linux's monolithic kernel architecture. While it is entirely outside of the scope of this paper to break into the microkernel vs. monolithic kernel argument, we should explain two properties of a monolithic kernel. First, a monolithic kernel has less context switching between userspace and kernelspace due to more functionality being inside the kernel. This reduction in context switching makes for a more performant system when compared to a traditional microkernel [3]. Secondly, the monolithic kernel contains most of the system functionality, which means that the system can share more functionality among containers than possible with a microkernel. The tradeoff between the performance and security of such systems has an inverse relationship; however, an adequately configured container can be considered secure [4].

B. Virtual Environments vs. Virtual Machines vs. Emulation

Before continuing, we will quickly differentiate between different types of virtualization. To start with, let us look at emulation. Emulation is the mimicking of non-existing physical hardware or system software. That is, to use software to mimic the functionality of a physical piece of hardware or a software component that does not exist in the host system. Notable examples include emulation of the PowerPC instruction set on an x86_64 CPU and the emulation of x86_64 on arm64 [5]. A minor example of this could include using SSD to create a memory device to expand RAM in the system. Another example is emulating a TPM chip [6] for an operating system that requires this to boot.

Emulation is different and frequently confused with virtual machines. A virtual machine is a type of isolation that isolates an entire operating system (or kernel) by splitting up the physical hardware and sharing them among these isolated guest operating systems [7]. In such a stack, the isolation occurs around both the kernelspace and userspace. With virtual machines, the hardware gets spliced up using either existing physical hardware or emulated hardware for other systems to use. Virtual machines do not mimic non-existing hardware without an emulation layer being present. A virtual machine simply splices up hardware for others to use. Lastly, we have virtual environments, which we will not cover in-depth because the previous section goes over this. When compared to virtual machines, virtual environments are almost always faster [8]. This performance difference is because each virtual machine adds another operating system to the stack. In fact, in some cases, the performance of a virtual environment approaches that of bare-metal (native) [9]. The only overhead added is the overhead of the container runtime itself. That is not to say that virtual machine level isolation is not needed. If the workflow needs this isolation level, taking a performance hit is well worth it. It is more to say that using a virtual environment is preferred when possible. Again, this debate goes outside of the scope of this paper. It is also outside the context of this paper to go into the security between containers and traditional virtual machines. However, keep in mind that LXC based virtual environments provide protection via strong isolation. Also, keep in mind that there is an inverse relationship between system performance and system security (isolation). Moreover, isolating the entire OS could have profound performance implications.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

C. Why not use KVM?

As discussed, virtual environments provide strong isolation with better performance than VMs. This performance boost alone is a reason not to implement the project using a hypervisor (KVM). One could argue that a virtual machine would be better from a maintenance perspective. In such an argument, the design of a system would choose to lose performance to make the system easier to maintain over multiple devices. KVM would also provide uniform device driver management for the Linux virtual machine. While this initial sense is understandable, it is not wise to lose performance unless a hard requirement requires engineers to do so. For example, running the initial system inside KVM would require running any other virtual machine in a nested environment, resulting in catastrophic performance loss. In this example, if we want to add Windows or Mac OS application support, we may need an additional virtual machine. Any additional virtual machines will be nested if the core system is already in a virtual machine. Nested virtualization is not preferred.

Nevertheless, there is another consideration outside of the design: Qualcomm. When Qualcomm first began writing their SOC firmware, they did not in-vision their SOC use for anything other than mobile phones. Because of this, they never pictured their customers wanting to run virtual machines. The ARM architecture permission outline dictates security level EL0 to EL3, where EL3 is the highest permission level reserved for firmware drivers. The purpose of EL2 is to run hypervisors [10]. Qualcomm decided to use EL2 to load their Qualcomm Trusted Execution Environment (QTEE) instead of reserving it for hypervisors/virtualization. Qualcomm could have done this without destroying EL2's definition by loading DRM management into a trusted VM at EL1 or EL0. This implementation is how upstream Linux kernel developers [11] have proposed Qualcomm fix their issue. However, Qualcomm has not yet resolved this issue. Moreover, if Qualcomm does fix this, it would be unlikely for older chipsets to inherit these changes. At the time of writing, Qualcomm has effectively killed any ability to run KVM on some of their SOC. Qualcomm currently dominates the mobile SOC market, which hinders the ability to run virtual machines inside the Maru OS project (Even inside the LXC container).

D. What is Operating System Convergence?

In the most simple terms, convergence creates a system that seamlessly scales between different device form factors. Think of a convergent system as a system that seamlessly operates on mobile, tablet, and desktop interfaces [12]. Two of the best commercial examples of a converged OS are the now-discontinued Windows Continuum and the currently developed Samsung Dex. Here, one device operates as the user's cell phone, tablet, and laptop/workstation. A mobile device (i.e., a cell phone with video-out support) can be plugged into a monitor or laptop shell to transform into a desktop or tablet interface.

Ideally, the OS is written from the bottom up so that the developers only maintain one operating system. Examples of this approach include Ubuntu Touch, Samsung Dex, and Windows Continuum. Here, the entire system is written from the bottom (kernel) up to the graphical stack (compositor and desktop environment) in a way that scales among multiple device form-factors as one maintained operating system. *Convergence* is the term that refers to this kind of system design. This approach is not always possible, and one of the biggest challenges to this form of convergence is fragmentation between operating system designs and their different implementations.

The convergence of Android and Linux userspace does not strictly follow this definition. While the kernel [13] is mostly converged, the respective userspace implementations have diverged, and there are several key differences. Two significant differences to note are Android's use of Bionic over the traditional glibc [14] and Android's use of the SurfaceFlinger compositor over Wayland. Propositions to fix these already exist, which we will outline later. Because the kernel is already converged, userspace implementations can use Linux containers to join most of the system together. Containers make a great choice because they have low overhead, allow for seamless integration with the host OS, and provide an abstraction layer around the perspective userspace layers. We define this type of convergence as *operating (OS) convergence*. While the long-term goal should be a bottom-up design, this approach is useful for backward compatibility and application compatibility among various operating systems. This paper aims to give an overview and architectural example of this approach to converging operating systems into one cohesive system.

Other projects and examples use virtualization techniques to converge operating systems and add application compatibility. WINE is a well-known compatibility framework that allows Windows native binary code to execute on top of a Linux kernel. WINE runs as a compatibility layer by implementing the Windows API in a virtual environment and uses a "virtual kernel" loaded as a kernel module to translate system calls between the two incompatible kernels [15]. The original implementation of WSL (Windows Subsystem Linux) had a similar implementation on top of the Windows kernel to provide native Linux binary support in Windows systems. These implementations can have considerable overhead due to the translation required for system calls (emulation). These implementations also have an issue with compatibility and stability. In these cases, it is sometimes better to run virtual machines to allow the entire operating system to be isolated. A project named Noah implemented a Linux compatibility layer by running a lightweight Linux guest in a virtual machine that allowed for the execution of Linux binary applications on both Mac OSX and Windows [16]. WSL2 took a similar approach by moving its compatibility layer into a lightweight Linux virtual machine. The convergence of Android and Linux does not fall under this category because the translation between Bionic to glibc provided by libhybris has minimal overhead. Regardless of the virtualization technique chosen, the long-term goal when converging operating systems is to run the virtualized system as a backend service that provides

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

compatibility to a cohesive system.

E. Importance of the Application Ecosystem

Additionally, the system may need to ensure backward compatibility and application support from one operating system ecosystem. We believe that one of the core reasons Windows Continuum failed was a lack of a diverse and robust application ecosystem. Samsung Dex also falls short by only offering mobile applications in desktop mode. The best solution would be to take Android's massive application ecosystem and combine it with Linux's already existing ecosystem, which would provide a system that contains an application ecosystem that meets almost every workload and use case. It is important to note that Ubuntu Touch has achieved this with Waydroid (formally Anbox) [17] by running a barebones android container and piping it forward to their custom UI. Maru OS wants to take a slightly different approach than Ubuntu Touch. Maru OS provides the application ecosystem of both Android and Linux while also keeping a traditional Android and desktop interface.

III. Related Work

A. Motorola Webtop and Ubuntu for Android

Running a workstation environment on the end user's phone is not new. In 2011, Motorolla released the Atrix 4G, which ran the Webtop Platform. This platform ran an Ubuntu rootfs inside of a rooted chroot [18]. The developers enforced security by using the TOMOYO Linux Mandatory Access Control framework. When plugged into a lapdock accessory, the phone would power the lapdock to behave like a desktop workstation. The MobileView feature would mirror the Android display to the Ubuntu chroot to allow for android applications to be used [19]. The Motorolla Webtop platform is one of the first examples of a converged system and is the only commercial example of Android and Linux being converged into one cohesive system.

Another example is the Ubuntu for Android project, which was announced in 2012 [20]. This project achieved the same goal as webtop, but instead of using a lapdock shell, it provided a small dock for a monitor. Android for Ubuntu was deprecated by 2014, and the project was superseded by the Ubuntu for Phones project, which eventually became the Ubuntu Touch project. Both Webtop and Ubuntu for Android are spiritual predecessors to Maru OS. The main difference is that Maru OS uses LXC containers providing stronger isolation and security.

B. Ubuntu Touch and PostmarketOS

Ubuntu Touch and PostmarketOS both represent a bottom-up approach to convergence. The ubports documentation describes convergence as a system with the capability to "use the same code base" on multiple hardware and form factors, with the goal being to "provide a consistent experience, whether you are using a big screen with a mouse and keyboard or on a small touchscreen" [12]. PostmarketOS also achieves the same style of convergence. Both PostmarketOS and Ubuntu Touch can plug the phone into a lapdock or monitor to launch desktop mode. This type of convergence is the design that most systems should strive to achieve. These distros include a new interface (the mobile UI) that most end users will not be familiar with and a limited application ecosystem. Additionally, this kind of rewrite can be extremely costly from a development perspective. So while this approach is preferred, sometimes the level of complexity and requirements may need a layered approach. Even with these cases, the long-term goal should be to achieve a bottom-up approach to convergence.

C. C-Android, Anbox, Halium, and Waydroid

One of the most significant issues that arise from convergence is a lack of compatible applications in the new system. This compatibility problem can be solved using operating system convergence. In this solution, the system is running in a virtualized environment that allows for application compatibility. When it comes to converging Linux and Android into one system, the compatibility problem occurs because Android drivers are written in Bionic while Linux traditionally uses glibc [21]. Various projects use Android containers to try and solve this problem.

C-Android is a research project to bypass detection and prevention of analysis techniques done inside Android. C-Android loaded Android inside an LXC container and loaded the analysis tools on the Linux Host, which allowed the system to get around the detection and prevention of analysis techniques performed in Android. This architecture allowed Android to run unmodified while analysis tools ran undetected in the Linux host. The researchers architected a system that used Linux init to boot. This sequence was done by loading the Android kernel and running an init sequence inside a Linux host. The init sequence loaded the analysis tools and the LXC container run time. The Android environment starts its init sequence in a container on top of this. This solution aims to provide a complete Android environment and experience as opposed to providing support for anyone Android Application [22]. Since C-Android provides an Android kernel, Bionic drivers are already provided for the Android container to consume. While the Android container can talk directly to the drivers, any Linux application running would not be able to access them. C-Android was not trying to solve this particular compatibility problem. An ideal solution would let both converged systems interact with the hardware.

Other solutions aim to run Android without affecting the native Linux application's ability to talk with the hardware. The Ubuntu Touch project developed Anbox, which ran Android inside of an LXC container and used QEMU for GLES emulation [22]. While this provided decent performance, the reliance on emulation was a performance bottleneck, and Anbox-Halium replaced this. Halium is a project that is architecturally very similar to C-Android. Halium aims to provide a unified HAL for projects needing to run GNU C on devices with pre-installed Android. Halium runs a Linux kernel and uses Systemd to init the system and load Android into an LXC container [23]. The main difference is in Halium's

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

use of the libhybris library. Libhybris provides a solution to the incompatibility problem by translating Bionic system calls to glibc calls which allow Bionic-based hardware to run on GNU C-based systems [21]. Halium and libhybris allow a device to run "proprietary Android userspace drivers" on a "downstream kernel" by running the needed Android parts in a container [24]. In the case of Anbox-Halium, the Halium project replaces the QEMU GLES emulation to allow the container to have direct access to the hardware (hardware-acceleration). Anbox-Halium eventually morphed into Waydroid, a version of Anbox compatible with the new Wayland compositor and provided Halium support for devices that need it [25]. Both Anbox and Waydroid attempted to provide just enough of an Android ecosystem to allow for individual Android application compatibility.

IV. System Architecture

Fig. 1 represents an architectural diagram of the Maru OS project. This diagram shows each project component. The system's three main components are the Android Kernel, the Linux Container (LXC), and the Android userspace.

A. Android Kernel

The kernel is already mostly converged due to Linux and Android using the same underlying kernel, and the most significant addition for Maru OS was patching LXC back into the kernel. The main difference is Android's use of Bionic for device drivers. These drivers differ from traditional Linux, where device drivers are written and compiled in glibc, which causes a compatibility problem where Linux applications cannot access hardware. Maru OS has a solution to re-direct audio so that direct access is not needed; however, emulation of the GPU occurs through software acceleration. This emulation can cause a considerable performance overhead, especially for graphically intensive applications. A project called libhybris exists to solve this compatibility problem that translates Bionic and glibc calls. Fig. 1 shows how a libhybris implementation would look inside the LXC container with Maru OS.

B. Android Userspace

The Android userspace does not run in an isolated container. Maru OS adds a few features that manage and converge the Linux container inside the userspace. For the most part, the Android host has been left alone with only minor additions. Let us go over only the services vital to controlling the container:

- *LXC and Busybox Packages Added:* Busybox brings in a set of POSIX compliant utilities. LXC is the userspace tool used to interface with the LXC runtime.
- *Maru Settings and iPerspectiveService:* The Maru Settings application allows for the container to be started or stopped graphically. iPerspectiveService provides the functionality to maintain the state (start, stop, I/O-direct enable/disable, status check).
- *mflinger (Android Server) and mclient (Linux Client):* These are the services that manage the HDMI framebuffer.

Mflinger sits on the Android side and handles the manipulation and creation of the Linux display buffer. Android has an internal display (labeled iDisplay Framebuffer in Fig. 1), and Maru OS assigns this to the internal display (marked by Display Controller in Fig. 1). Every Android phone will have at least one framebuffer device for the internal display. Things get interesting when considering additional framebuffers. Some devices will have a physical framebuffer that the system assigns to an HDMI device (represented by HDMI Framebuffer), and these devices have physical video-out support. Typically, when Android connects to an HDMI cable, the screen on the iDisplay Framebuffer is duplicated to the HDMI Framebuffer, and the system mirrors the Android screen to the HDMI device. Maru OS assigns the HDMI Framebuffer to the LXC container instead. This change means that the Linux composite manager writes to it instead of Android mirroring the HDMI Framebuffer. The handling of the assignments of these buffers is where mflinger comes into play. The mflinger service is responsible for detecting which framebuffer is designated for the HDMI output and then assigning this buffer as the Linux framebuffer. From here, mflinger acts as the server-side Android service to interact and manipulate that buffer.

There may not be a physical framebuffer already assigned for HDMI use. In this case, the phone does not have physical video-out support. When there is no physical framebuffer, mflinger detects no HDMI framebuffer and creates a virtual framebuffer that emulates the HDMI display. Third-party devices can connect the phone to a monitor for devices lacking a physical buffer that rely on the virtual framebuffer to power the graphics:

- *DisplayLink:* An adapter that provides a physical framebuffer over USB, allowing for good performance. The drivers can install in Android; however, it is proprietary software and could cause licensing problems. Additionally, the use of two cords is a requirement.
- *Chromecast:* The virtual framebuffer (HDMI) can be screen-casted to a Chromecast. Chromecast requires a wireless access point (No WiFi-Direct). It also requires Google Play Services, which could also have licensing concerns.
- *Miracast:* Miracast is an open-source project that allows for the virtual framebuffer to be screen-casted directly to a receiver via WiFi-Direct. Miracast is the preferred method for the Pixel 2/XL. Miracast support can be enabled on Android and via Gnome on Linux.

C. Linux Container

The LXC container runs on top of the Android Kernel. The Maru OS distro itself uses Debian Buster as its base. XFCE is used as the desktop environment to ensure that the Linux environment does not use unneeded resources. The mclient daemon manipulates the HDMI framebuffer, which acts as a complement to the mflinger service. A simple explanation is that mclient translates the composite manager's request to manipulate a buffer over to mflinger, which handles the

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

actual manipulation of the buffer. The mclient daemon handles creating the shared memory realm and initializing the display on the Linux side. From here, mclient translates composite calls into this shared memory realm, where mflinger creates a surface buffer out of it. A special device file descriptor references the buffer to avoid the consumption of duplicate memory.

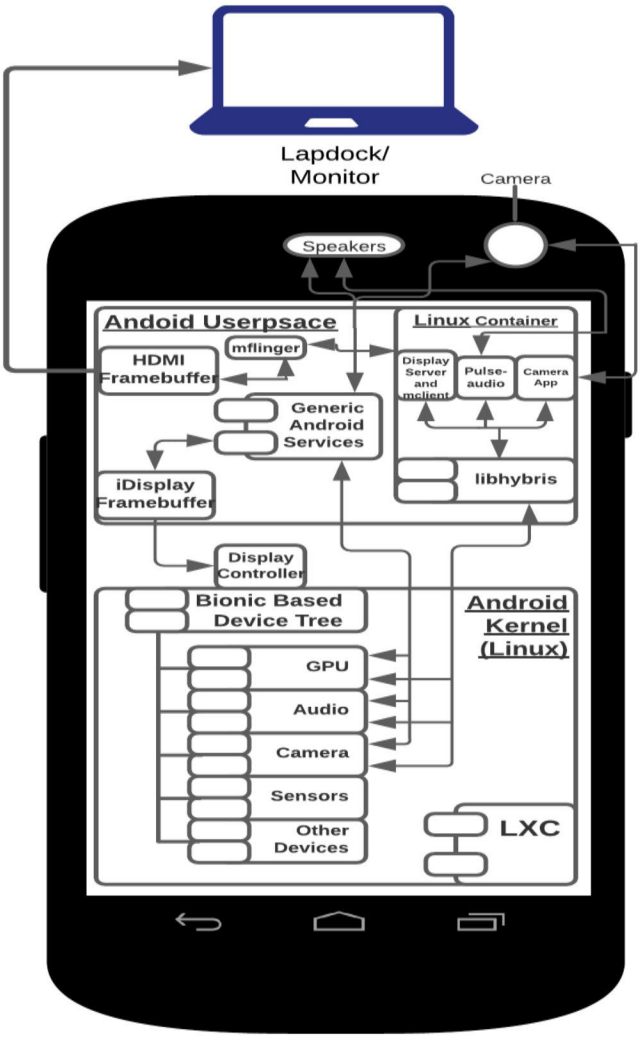

Fig. 1. Maru OS Project Architecture Diagram. Libhybris and Halium are planned as future work and are not currently implemented into the project.

V. Proof of Concept and Device Port

The paper's main scope was to complete the Maru OS Pixel 2/XL port. The expectation was to perform research on additional tasks; however, any additional implementation work was considered a bonus. We took this mentality because only a single developer worked part-time on the port. In each subsection, we will go over the completed tasks.

A. Port Maru OS to Pixel 2/XL

The main focus was to provide an official port of Maru OS for the Pixel 2 and Pixel 2 XL devices. In order to do this, we completed the following tasks:

- *LXC Patchset:* The LXC patchset was applied to the device-specific kernel config files.
- *Add Maru into Device Tree:* The Maru OS build-specific pieces were added so that the build could pick up the Maru-specific vendor configs.
- *SEPolicy Fixes:*
  1) *dac_override:* Android 9 created a neverallow rule for dac_override, which uncovered an issue. The dac_override type allowed the container to have the ability to override root permissions. This issue was fixed by changing Maru OS's SEPolicy to no longer include a rule to allow dac_override.
  2) *maru_file:* This issue was specific to Treble devices. The policy type maru_file did not have permission to interact with the attribute core_data_file_type. The core_data_file_type attribute was added to the maru_file permission type to fix this.
  3) *Accepted Upstream:* Both patches were accepted upstream.
- *Pixel 2/XL Device Trees Created Upstream:* We created four projects upstream to handle the Pixel 2/XL device and kernel-specific changes. We merged the device-specific code upstream. Currently, beta builds have been generated for testing.

We performed usability and performance tests on the Pixel 2 XL to test the system's capability. Fig. 2 shows a screen capture of the LXC container running multiple productivity applications. The productivity applications are listed below:

- *Application Install:* A large (500MB) application install is performed in the top terminal
- *HD YouTube Video:* On the right-hand side of the screen, a 1080P YouTube video is played in Firefox.
- *Eclipse IDE:* On the left side, the Eclipse IDE is loaded with a basic bubble sort C example program [26] and the terminal below this shows the bubble sort program running.
- *Android Environment:* A YouTube video is being played simultaneously in the Android environment.
- *Resource Usage:* Fig. 2 also includes a second screenshot taken immediately after the first and shows the resource usage of the productivity test.

Fig. 3 and 4 show a test designed to show the system under load. The figures and results are listed below:

- *HD YouTube Video:* The same 1080P video from Fig. 2 was left open playing on the system. This video can be seen on the upper right side of the screen in Fig. 3.
- *GIMP:* GIMP was launched, and a high-quality screenshot of the Android Home screen was opened and shown on the upper left side of the screen in Fig 3.
- *Blender:* In Blender, the provided demo "Fishy Cat" was loaded, and the scene was actively rendered. This rendering can be seen on the bottom left side of the screen in Fig 3.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

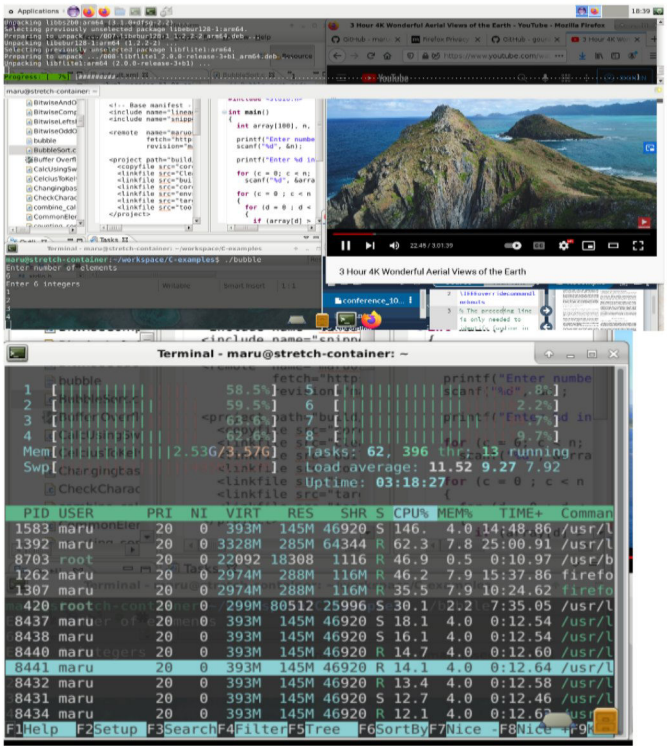

Fig. 2. Maru OS: Productivity Applications
A test of multiple productivity applications showing the system has spare resources. The second image shows the load of the system while we performed this test.

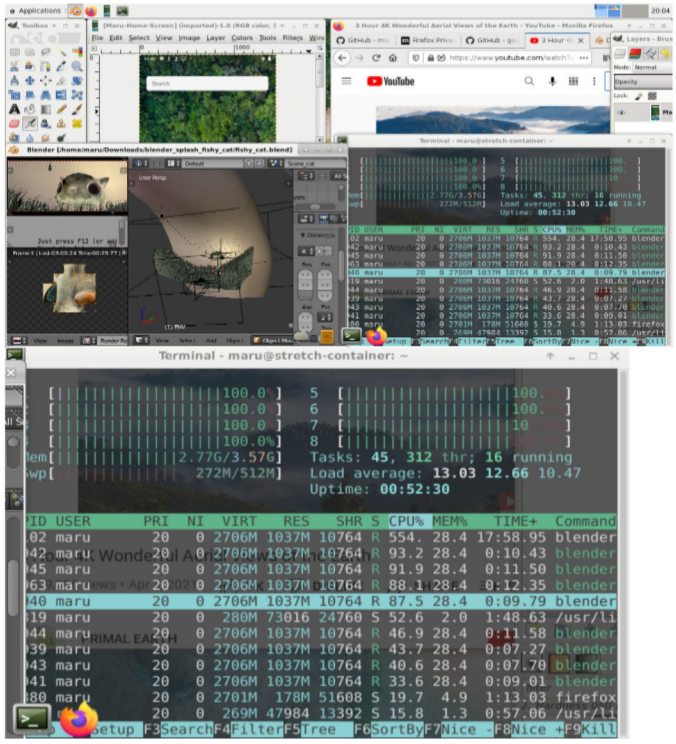

Fig. 3. Maru OS: Resource Usage
A test showing the system operating at full load as well as it's resource usage.

- *Android Environment Load:* Fig. 4 shows a screenshot of the Android environment running OpenArena, a relatively heavy 3D game, at the same time as the load test in Fig. 3
- *Resource Usage:* Fig 3 shows a screenshot of the resource usage, which shows CPU usages maxed out and RAM almost maxed out. The game in Fig. 4 runs with no lag despite the load generated in the LXC container.

It is important to note that the game is offloading the processing primarily to the GPU and not the CPU; however, this shows the system can handle a load from multiple heavy-weight applications simultaneously. The Pixel 2 XL hardware is dated, and a more modern phone with increased RAM and processing power would handle these tests even better.

B. Partially Ported Maru OS to Lineage 17.1

We spent a week attempting an initial port of Maru OS to the Android 10 based Lineage 17.1. We completed the following items:

- *Maru Patchset:* A patchset for the Maru specific changes were generated against the projects git repositories and applied against Lineage 17.1.
- *Disabled Deprecated Features:* Some of the core features were disabled due to libraries being deprecated. These deprecations caused build problems, and disabling them allowed us to test that the rest of the system booted correctly.



Fig. 4. Maru OS: Resource Usage Android Environment
OpenArena is running in the Android environment at the same time as the test in Fig. 3 without lag.

- *Test LXC:* The system was booted, and the LXC container started. The logic needed to assign the HDMI framebuffer was disabled, so it could not attach its display. This partial-port shows that with minor work, the project can be ported to Android 10.

C. Nexdock 360 Lapdock and Miracast Support

The NexDock 360 [27], a device referred to as a lapdock, acts as a laptop shell with no internal processing power of its own. It includes a massive battery, video-in device (Micro-HDMI), display, mouse, keyboard, and a single thunderbolt port to attach the mobile device. The Thunderbolt port supports

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

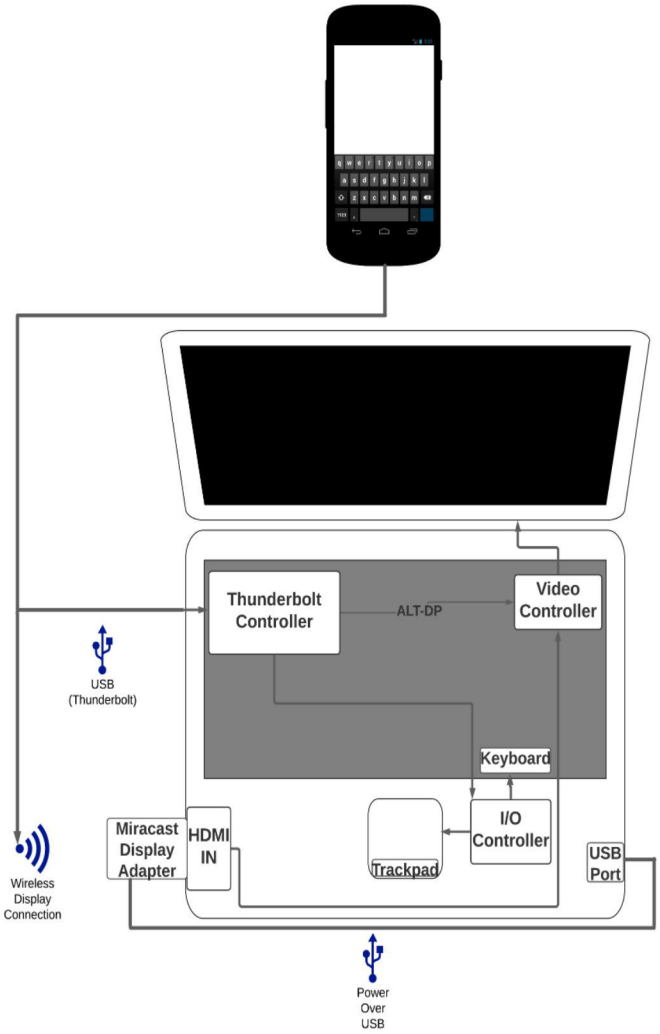

Fig. 5. Nexdock 360 powered by Maru OS device amd Miracast

This diagram is a representation of the POC used to demo Fig. 2, 3, and 4.

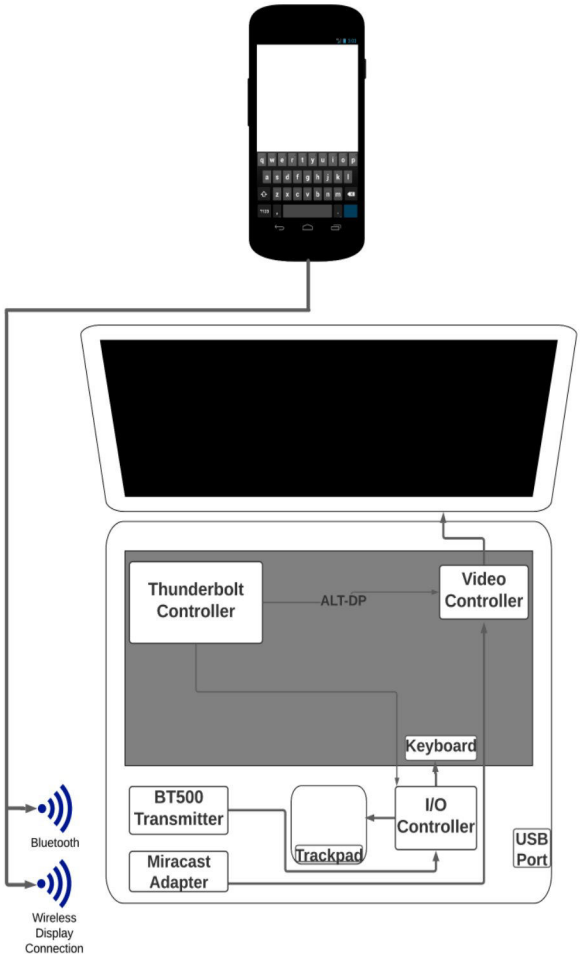

Fig. 6. Lapdock Wireless POC for Maru OS

The lapdock could be modified to have both Miracast receiver and a BT-500 Bluetooth transmitter embedded in the device. This solution would allow for Maru devices to attach the lapdock completely wirelessly.

Alt-DP, meaning that supported devices can use this port for USB, video-out, and I/O. Devices that support video-out (Not through Alt-DP) can still send video to the Micro-HDMI port; however, this requires an additional cord. Phones that do not support video-out do not have a native way to send video to the lapdock but can still attach the keyboard and trackpad via the Thunderbolt port. The Pixel 2/XL falls into the latter display problem.

A great solution to this display problem is using Miracast. Miracast allows casting device buffers directly to receivers with no active internet connection via Wi-Fi Direct. Another significant benefit was that Miracast uses existing framebuffer devices. This usage indicates that a device that supports physical video-out could beam a display without the performance loss of an emulated framebuffer. If a device has no physical video-out support, then the emulated framebuffer could still be beamed to the display to gain video-out support. Miracast has implementations for both Android and Linux, which was a huge bonus. We built a POC around this by adding a Microsoft Wireless Display Adapter to the Micro-HDMI port. Fig. 5 shows a diagram of this POC.

The POC shown in Fig. 5 is not truly wireless, but it requires only one cord to connect the device to the lapdock. We designed a solution to allow the phone to be attached to the lapdock entirely wirelessly. This feature could be achieved by attaching the mouse (trackpad) and keyboard to a Bluetooth transmitter, allowing Maru devices to connect to the lapdock wirelessly. The phone would only need to be attached to the Thunderbolt port to act as a USB controller with this setup. We attempted a POC around this design by attaching a BT-500 USB transmitter [28] to the lapdock's thunderbolt port. Since the lapdock lacked an internal USB controller, the USB transmitter could not attach to the I/O. Additionally, since there was only one Thunderbolt input, there was no

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

way to attach both a USB controller device and the BT-500 transmitter simultaneously. While the external POC failed, this design could be achieved internally by embedding the BT-500 transmitter into the lapdock and attaching the transmitter directly to the trackpad and keyboard I/O. While the POC in Fig. 5 works, the Miracast receiver is bulky and requires a USB power cable attached to a USB port on the opposite side of the device. A cleaner solution would be to embed the Miracast display receiver inside the lapdock. Fig. 6 shows a diagram of a truly wireless solution that embeds the BT-500 transmitter and Miracast receiver inside the lapdock. A more hardware savvy individual could make a POC with both of these embedded into the lapdock allowing phones to power the lapdock wirelessly.

VI. Future Work

One of the main goals of this project and paper was to perform research to provide a baseline for future work. Maru OS is an excellent example of achieving operating system convergence; however, the software base is aging. Because of this, we need to add a handful of enhancements. Maru OS would also immensely benefit from moving the Android userspace into an LXC container and incorporating the Halium framework. Other projects have already achieved parts of the framework needed to do this, and future work should investigate these projects further.

A. Halium and Waydroid

Maru OS would benefit from implementing libhybris support so that Linux applications would have direct hardware access (hardware-acceleration). We originally intended to implement libhybris inside the Linux container. The Halium project aims to build a framework around libhybris; however, this framework runs the Android userspace inside an LXC container and boots out of a Linux host. Halium runs Android inside of a container in combination with libhybris in order to allow a standard Linux distribution to run on Bionic-based devices [23]. In this sense, Android is run as a backend service to provide compatibility to a Linux userspace; however, we can leverage Halium to do more. Waydroid's project intention is to provide Android application compatibility. Waydroid can be combined with Halium to provide this compatibility on ARM-based Bionic devices [25]. Additionally, Waydroid can have Android extend its display to run a full Android home launcher [29]. We could have Waydroid assign the Android container to the phone's internal display controller. This designation would allow Android to function as the primary mobile experience and Linux to act as the primary workstation experience. This solution would provide similar functionality as Maru OS while also inheriting all of the features of Halium and Wayland.

1) Decoupling Android UI: The approach described above provides end-users a familiar mobile UI converged into a traditional Linux distribution. While this solves the problem, it should not be the long-term solution. The Android UI could be decoupled from the Android container and moved into the Linux userspace. This decoupling would shift the role of the Android container back to the role of a backend service without sacrificing the familiarity of Android. This approach would allow the option to eventually deprecate the Android container without removing the UI from the converged system.

B. Fedora Mobility

Android is one of the most popular mobile operating systems with a sizeable decentralized community. This decentralized community has caused a fragmentation problem among the Android kernel ecosystem, which can delay (and in some cases completely stop) updates and CVE fixes from propagating to a device [13]. One possible solution would be to unify Android under a single kernel ecosystem. While there are plenty of distribution kernels (such as Debian), Fedora offers a great opportunity as it is the upstream for the entire Enterprise Linux family of distributions. Fedora Mobility is still in its infancy but could provide a great baseline to run Android on a unified enterprise kernel. Currently, the Fedora kernel is missing paramount functionality due to security concerns, namely the ashmem module. These requirements mean that additional work is needed to port Waydroid to Fedora securely.

C. Functional Safety of Environment

Another vital area to investigate is the functional safety of the environment. While LXC containers provide strong isolation, a couple of additional areas need investigation. We could further improve this design by running the Linux and Android userspace inside their own containers. This architecture would allow the host to gain a certain amount of immutability, predictability, and determinism, which would increase both the performance and security of the converged system.

Another area needing investigation is how Android handles IPC, particularly binder. Android launches each application as single isolated DVM instances that resemble distributed systems, and binder acts as a remote call IPC to coordinate these instances. There is also a kernel-level module that orchestrates the IPC stack [30]. Android's application isolation is interesting because it resembles how application containers (Ie: Docker) isolate web applications. Another interesting topic is the effect of running two separate IPC mechanisms at the kernel level on strong isolation.

1) Android Compatibility on the Edge and Far Edge (Fog): The importance of functional safety is stressed even more by the possibility of adding this kind of Android compatibility in the Edge and Far Edge. IoT has exploded in the last decade, and Android, in particular, is widespread. Adding Android compatibility on top of a safety-certified kernel could prove beneficial. C-Android has already shown the possibility of running Android in a cluster with a fast-restore mechanism [22]. We would also like to investigate how real-time Android environments (such as RTDroid [31]) would profit from running on a Linux kernel specifically tuned for real-time workloads.

VII. Conclusion

Maru OS shows a proof of concept system that uses operating system convergence to combine operating systems

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.

with different device form-factors into one cohesive system. We show with this POC that these virtualization techniques can also add application compatibility to a system's application ecosystem. These results indicate that modern mobile devices with larger RAM banks and faster processors could handle these tasks and possibly even replace lightweight notebook computers. In this paper, we introduced a system design that would allow the mobile device to attach completely wirelessly to a lapdock or monitor. This type of system architecture could represent the perfect remote client as one single device serving multiple form-factors. This design could be incredibly versatile with the industries move to the cloud, edge, and far edge services that allow for resource-heavy processing to be offloaded remotely. Additionally, we could apply this type of system convergence to other form-factors and computer platforms. One particular area of interest is real-time Android workloads in the edge and far edge, as well as the safety and functional certification of such a system.

ACKNOWLEDGMENT

This paper is dedicated in memory of Anthony B. Massaro. We want to acknowledge the work done by Preetam D'Souza, the original creator of Maru OS. Preetam put together the basic framework to make porting and refactoring of this project possible. Maru OS is an open-source project, and we would like to thank each contributor to the project. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants IIS-1618669, OAC-1642133, CCF-0845257. William White's work is supported by Red Hat.

REFERENCES

[1] P. D'Souza. Maru. Accessed on Jan. 28, 2022. [Online]. Available: https://maruos.com/

[2] M. Eder, "Hypervisor-vs. container-based virtualization," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 1, 2016.

[3] H. Miao, "Analysis of practicality and performance evaluation for monolithic kernel and micro-kernel operating systems," *International Journal of Engineering (IJE)*, vol. 5, no. 4, p. 277, 2011.

[4] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of os-level virtualization technologies," in *Secure IT Systems*, K. Bernsmed and S. Fischer-Hübner, Eds. Cham: Springer International Publishing, 2014, pp. 77–93.

[5] W. Chen, Z. Wang, and D. Chen, "An emulator for executing ia-32 applications on ARM-based systems." *J. Comput.*, vol. 5, no. 7, pp. 1133–1141, 2010.

[6] Y. Shi, B. Zhao, Z. Yu, and H. Zhang, "A security-improved scheme for virtual tpm based on KVM," *Wuhan University Journal of Natural Sciences*, vol. 20, no. 6, pp. 505–511, 2015.

[7] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, pp. 267–272, 2012.

[8] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016, pp. 1–6.

[9] D. Beserra, M. K. Pinheiro, C. Souveyet, L. A. Steffenel, and E. D. Moreno, "Performance evaluation of os-level virtualization solutions for hpc purposes on soc-based systems," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, pp. 363–370.

[10] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A comprehensive survey," *ACM Computing Surveys*, vol. 51, pp. 1–36, 01 2019.

[11] J. Edge, "KVM for Android," Nov. 2020, Accessed on Jan. 28, 2022. [Online]. Available: https://lwn.net/Articles/836693/

[12] "Convergence," The UBports project, Tech. Rep., Accessed on Jan. 28, 2022. [Online]. Available: https://docs.ubports.com/en/latest/humanguide/other-design-considerations/convergence.html

[13] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the Android kernel patch ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3649–3666. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng

[14] "Bionic," *The Android Open Source Project*, 2009, Accessed on Jan. 28, 2022. [Online]. Available: https://android.googlesource.com/platform/bionic/

[15] N. Giri, V. Nandgaonkar, and R. Gosavi, "Virtual operating system for windows to linux migration," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 2017, pp. 2125–2127.

[16] T. Saeki, Y. Nishiwaki, T. Shinagawa, and S. Honiden, "A robust and flexible operating system compatibility architecture," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 129–142. [Online]. Available: https://doi.org/10.1145/3381052.3381327

[17] E. Abdi, J. West, and R. Błedowski, "Waydroid," accessed on Jan. 28, 2022. [Online]. Available: https://waydro.id/

[18] "Webtop," *Motorola Webtop*, 2012, Accessed on Jan. 28, 2022. [Online]. Available: https://sourceforge.net/projects/motorola-webtop.motorola/files/WT-1.2.0-X/WT-1.2.0-110%2BWT-1.2.0-135/

[19] R. Paul, "Hands-on: Motorola Atrix's Ubuntu-powered WebTop experience," Mar. 2011, Accessed on Jan. 28, 2022. [Online]. Available: https://arstechnica.com/gadgets/2011/03/motorola-atrix-the-ubuntu-powered-webtop-experience/

[20] "Ubuntu for Android," Apr. 2013, Accessed on Jan. 28, 2022. [Online]. Available: https://web.archive.org/web/20140930201948/ubuntu.com/phone/ubuntu-for-android

[21] J. Poutiainen, "Mer cellular telephony architecture: Software analysis and description," 2015.

[22] N.-T. Chau and S. Jung, "Dynamic analysis with android container: Challenges and opportunities," *Digital Investigation*, vol. 27, pp. 38–46, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287618301257

[23] "Halium," 2017, Accessed on Jan. 28, 2022. [Online]. Available: https://halium.org/

[24] "Hybris," postmarketOS, Tech. Rep., Accessed on Jan. 28, 2022. [Online]. Available: https://wiki.postmarketos.org/wiki/Hybris

[25] E. Abdi, "android_device_halium_anbox," *GitHub repository*, 2021, Accessed on Jan. 28, 2022. [Online]. Available: https://github.com/waydroid/android_device_halium_anbox

[26] randerson112358, "C-programs," *GitHub repository*, Accessed on Jan. 31, 2022. [Online]. Available: https://github.com/randerson112358/C-Programs

[27] "Nexdock," accessed on Jan. 31, 2022. [Online]. Available: https://nexdock.com/explore-nexdock/

[28] "Bluetooth adapter for keyboard mouse," accessed on Jan. 31, 2022. [Online]. Available: http://handheldsci.com/kb/

[29] "Setting up waydroid only sessions," Waydroid, Tech. Rep., Accessed on Jan. 31, 2022. [Online]. Available: https://docs.waydro.id/faq/setting-up-waydroid-only-sessions

[30] T. Schreiber, "Android binder," *A shorter, more general work, but good for an overview of Binder. http://www. nds. rub. de/media/attachments/files/2012/03/binder. pdf*, 2011.

[31] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, and L. Ziarek, "Rtdroid: A design for real-time android," *IEEE Transactions on Mobile Computing*, vol. 15, no. 10, pp. 2564–2584, 2016.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:16:49 UTC from IEEE Xplore. Restrictions apply.