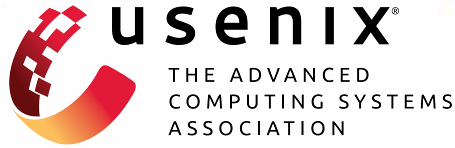


Towards Intelligent Automobile Cockpit via A New Container Architecture

Lin Jiang and Feiyu Zhang, *Xi'an Yunzhiji Technology;* Jiang Ming, *Tulane University*

https://www.usenix.org/conference/nsdi24/presentation/jiang-lin

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Towards Intelligent Automobile Cockpit via A New Container Architecture*

Lin Jiang†
Xi'an Yunzhiji Technology, China
pppaass@163.com

Feiyu Zhang
Xi'an Yunzhiji Technology, China
1123022283@qq.com

Jiang Ming†
Tulane University, USA
jming@tulane.edu

Abstract

An intelligent cockpit is now crucial in automobiles, not just to provide digital instrumentation and in-vehicle controls but also to offer a wide range of entertainment functionalities. To cater to the demands of these intelligent vehicles, the automotive industry starts employing virtualization technology to offer a unified hardware and software architecture that can simplify system management and enhance resource utilization. Particularly in the domain of intelligent cockpits, virtualization can tightly integrate systems with different criticality levels (e.g., safety and real-time) on a single hardware platform, improving inter-system communication quality and the timely response to user-initiated requests. Currently, microhypervisor virtualization has been used in production to achieve intelligent automobile cockpit. However, in addition to the performance concern and high production costs, this solution is suffering from the global shortage of chips capable of running microhypervisor systems.

Our key insight is that, most functions within intelligent cockpit systems are non-safety-critical and non-real-time multimedia tasks. Based on this characteristic, in this paper we present *AutoVP*, a new cockpit virtualization architecture. The hardware foundation of AutoVP consists of two low-cost chips: 1) a consumer-grade System-on-Chip (SoC) multi-core processor as the main chip; 2) a typical automotive-grade Microcontroller Unit (MCU) as the auxiliary chip. The MCU auxiliary chip is responsible for hosting real-time and safety-critical tasks, while the SoC main chip primarily handles multimedia tasks, such as entertainment systems and digital instrumentation. Further more, we construct an Android container virtual environment on the SoC main chip. This environment integrates multiple media functions onto a single chip, resulting in efficient utilization of chip computational resources and high system scalability. Our comparative performance evaluation demonstrates that AutoVP is a cost-effective and efficient solution to build intelligent cockpits.

*Operational Systems Track
†Corresponding authors.

1 Introduction

The automotive electrical and electronic systems comprise hundreds of sensors, actuators, and Electronic Control Units (ECUs). These units are responsible for running various subsystems, such as instrumentation, entertainment, and advanced driver assistance systems. They collaborate while remaining relatively independent, ensuring the highest level of performance, safety, and functionality. Today's automobiles are approaching the limits of their complexity. In the future, the automotive industry will provide always-connected vehicles, operating advanced autonomous driving functions and an increasing array of cutting-edge applications. In this environment, both the hardware and software components within vehicles are experiencing exponential growth in line with the increasing number of applications, leading to an explosion in the complexity of vehicle architectures [1]. Furthermore, the proliferation of connectivity and applications also results in a larger attack surface [2]. This distributed computing architecture is making automotive electrical and electronic systems increasingly bulky, with challenges emerging in wiring, thermal management, and power distribution [3].

A promising approach to address the aforementioned problem involves the adoption of a virtualization solution. Virtualization primarily serves the purpose of integrating multiple business subsystems that originally operated on different ECUs [4, 5], thus reducing software and hardware costs, and shortening product time-to-market. At the beginning of the 21st century, this technology was first adopted as a centralized software technique for avionics systems [6]. Nowadays, several commercial virtualization solutions in the market have achieved significant success in safety-critical applications, including aerospace, national defense, and healthcare. There is currently a drive to promote virtualization as the preferred integrated solution within the automotive electrical and electronic architecture [1].

In the automotive landscape, vehicles can be categorized into various functional domains, including the Powertrain domain, Chassis domain, Body/Comfort domain, Cockpit/In-

fotainment domain, and Autonomous Driving domain [7]. The Cockpit/Infotainment domain encompasses both safety-critical tasks such as instrumentation and driver assistance, as well as non-critical multimedia entertainment functions like central control and heads-up displays. These subsystems operate relatively independently while collaborating, interacting directly with users to provide intelligent experiences [8, 9]. By employing a centralized virtualization solution to integrate multiple subsystems within the Cockpit/Infotainment domain, it can reduce costs, enhance collaborative processing efficiency, and optimize user experience.

Currently, commercial intelligent cockpit virtualization products are dominated by the hypervisor-based solution [10, 11], such as QNX Hypervisor [12], INTEGRITY Multivisor [13], and PikeOS Hypervisor [14]. They enable the operation of multiple virtual systems on a single set of hardware (e.g., ECU), with each virtual system running in a relatively independent virtual environment, hosting different functions.

These vehicle hypervisor solutions typically adopt a microkernel architecture [15, 16]. However, compared with user-attractive Android applications, microkernel architecture has a notable drawback, namely the absence of a rich software ecosystem [17–21]. For instance, if the cockpit needs to run advanced driver assistance systems (ADAS) functions (e.g., automated parking), they require support for deep learning, computer vision, video encoding/decoding, and 3D graphics, necessitating substantial resource investment for development. Furthermore, existing hypervisor solutions must handle both safety-critical tasks and complex multimedia operations, leading to specific hardware requirements. Consumer-grade SoC chips or even standard automotive-grade MCU chips fail to meet these demands. Automotive-grade SoC chips are capable of accommodating such intricate software systems [22], but their development entails significant challenges and high production costs. Even worse, the auto industry is facing a global shortage of automotive-grade SoC chips [23–25].

We have three observations motivating a new cockpit virtualization design: 1) only a minority of functions within an intelligent cockpit are safety-critical (e.g., instrumentation and driver assistance), while the majority of modules classified as non-critical operations (e.g., all multimedia entertainment tasks); 2) a typical automotive-grade MCU chip can host these real-time and safety-critical functions, while a cheap consumer-grade SoC chip (e.g., smartphone chips) can handle the remaining non-critical functions; 3) these two kinds of chips are affordable, and they have not been significantly impacted by recent supply chain shortages. As a result, this paper presents a cost-effective virtualization architecture for constructing an intelligent cockpit, called *AutoVP*.

As shown in Figure 1, AutoVP employs Android container technology [26, 27] to integrate non-safety-critical subsystems, such as instrument display, central control system, and passenger entertainment system, into a single entity, which is deployed on a cheap consumer-grade SoC chip. Container

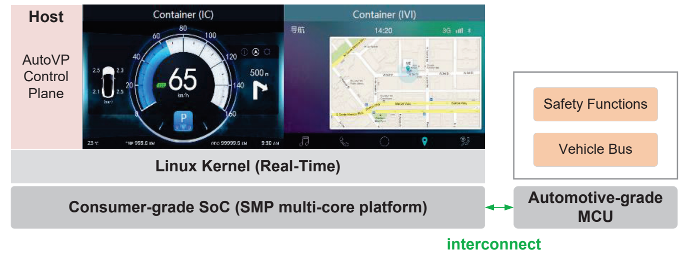


Figure 1: AutoVP's mixed-criticality decoupled design.

is a lightweight virtualization technology [28–30] that can efficiently leverages the computational power of the SoC chip while ensuring the independent operation of each business subsystem, thereby enhancing service quality. Safety-critical tasks such as safety monitoring and vehicle bus, are handled by an automotive-grade MCU chip. Interconnection between the SoC chip and MCU chip is facilitated through inter-chip communication technologies, forming a complete intelligent cockpit system. Please note that AutoVP's mixed-criticality decoupled design also complies with automotive functional safety requirements, as outlined in ISO 26262 [31].

We developed a new Android container framework on top of Cells [26, 27]. This container imposes no specific hardware requirements and incurs very small virtualization performance overhead, making it suitable for widespread deployment across various low-power SoC chips. Besides, in the automotive industry, the deployment of safety-critical tasks using automotive-grade MCU chips is a well-established and mature method [32]. In addition to container-based virtualization, AutoVP's differences in design also involve monitoring mechanisms, implementing corrective measures for abnormal behaviors, and ultimately isolating the safety-critical tasks from the business functions. Hence, deploying the AutoVP-powered cockpit system within intelligent vehicles is relatively straightforward.

We compared AutoVP with a commercial hypervisor product on the same automotive-grade platform. Our extensive performance experiments demonstrate that AutoVP incurs significantly lower performance overhead. When examining various aspects such as CPU overhead, memory usage, power consumption, startup time, peripheral performance, and frame rate, AutoVP's container outperforms the hypervisor solution.

In a nutshell, we make the following key contributions:

- We propose a mixed-criticality decoupled architecture to address the need for centralized deployment of various business subsystems within the intelligent cockpit domain, all while offering ease of construction, low production costs, and small performance overhead.
- We present a new Android container framework tailed for intelligent cockpits. Our work represents the latest progress in mobile container-based virtualization, and it is an ideal solution to host the non-safety-critical subsystems that require display screens for user interaction.

- Our evaluation and real-world deployment demonstrate that AutoVP is a viable in-vehicle virtualization alternative to mitigate the ongoing automotive chip crisis.

Real-world Deployment AutoVP has been deployed in two flagship electric vehicle models under a leading automotive manufacturer.* The installation volume in the past year has exceeded one million units.

Open Source We have released a prototype of AutoVP's container to facilitate reproduction and reuse, as all found at https://github.com/jianglin-code/AutoVP.

2 Background and Related Work

In this section, we first provide background information on the evolution of in-car virtualization. We also review existing approaches for intelligent cockpit virtualization and identify their limitations, which have prompted our work. Next, we underline the need of using the Linux container technology for in-car virtualization. Finally, we introduce the specific Android container framework that we leverage to implement AutoVP's container.

2.1 The Development of In-Car Virtualization

With the emergence of multi-core embedded System-on-Chip (SoC) devices, the integration of multiple applications with varying levels of criticality on a single platform has become increasingly popular. This platform is referred to as a mixed-criticality system, which needs to meet various requirements, including real-time constraints, operating system (OS) scheduling, and memory/OS isolation [33].

The cockpit system of intelligent vehicles represents a typical mixed-criticality system. It comprises non-critical subsystems such as entertainment, networking, and voice systems, as well as safety-critical subsystems like the vehicle bus and advanced driver assistance systems (ADAS). The non-critical subsystems often involve multimedia subsystems that rely on hardware modules for tasks such as display, encoding/decoding, networking, AI, and voice processing. Therefore, these subsystems are typically deployed on SoC chips to provide users with a seamless intelligent experience. On the other hand, safety-critical subsystems must adhere to functional safety requirements to ensure error-free operation [31]. They are usually deployed on automotive-grade MCU chips, offering users a stable safety and automated experience.

A key challenge in the design of mixed-criticality systems is the isolation of software applications with varying degrees of criticality on a common hardware platform. In the automotive domain, a common practice for isolating safety-critical applications is through the proliferation of multiple hardware Electronic Control Units (ECUs). These control units are dedicated to various tasks, including basic operations like intelligent infotainment, as well as critical functions such as automated parking and adaptive cruise control. However, this approach is highly inefficient, as many of the resources within these ECUs often remain underutilized. However, with the evolution of multi-core architectures and the introduction of new hardware extensions such as virtualization, securely executing multiple applications on the same platform while reducing costs and vehicle weight to enhance resource utilization has become feasible [33].

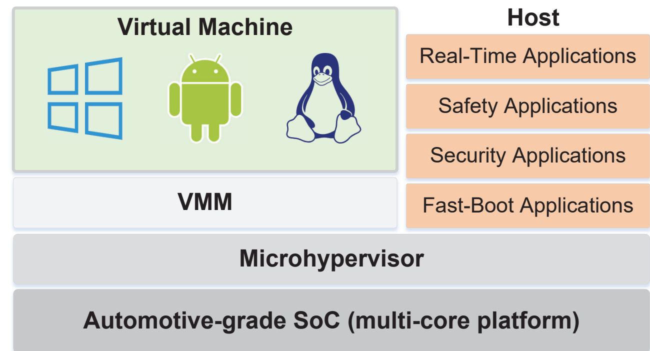


Figure 2: In-vehicle microhypervisor architecture.

Virtualization is considered a solution for isolating operating systems within virtual machines, and its advantages include cost reduction through the abstraction of the host platform [10]. Additionally, features provided by a hypervisor, such as the abstraction of memory, CPU, and interrupts, aid in the isolation of operating systems. This approach, as exemplified by commercial products like QNX Hypervisor [12], INTEGRITY Multivisor [13], and PikeOS Hypervisor [14], can be applied to create mixed-criticality systems. A common industry practice is to employ a hypervisor system with a microkernel architecture to ensure real-time capabilities and functional safety [15, 19]. We will further discuss the pros and cons of this approach in §2.2 and §2.3.

Android Automotive [34] is a new vehicle OS based on Android, designed to run in-vehicle infotainment systems and pre-installed Android applications. However, Android Automotive is not a virtualization solution and thus lacks a mechanism for system isolation. If several different types of services are running on the same Android Automotive system at the same time, sharing system resources, such as network, storage, multimedia, etc., these services will interfere with each other, affecting the user experience and in-vehicle safety.

2.2 Microkernel + Hypervisor

Embedded systems in vehicles often adopt microkernel-based operating systems due to their inherent real-time and security features. Such microkernel OSs are responsible for hosting safety-critical tasks; besides, they employ hypervisor techniques to run virtual OSs that have a richer software ecosystem, addressing the limitation of their own software ecosys-

*Until the publication of this paper, we have not been granted authorization to disclose the name of the automotive manufacturer.

Table 1: Comparison of three in-vehicle chips. The unit prices displayed represent their median ranges.

Chips	Production Cost	Unit Price ($)	Supply Chain Shortages?
Automotive-grade SoC	High	300∼500	Yes
Consumer-grade SoC	Low	70∼150	No
Automotive-grade MCU	Minimum	5∼10	No

tem [12–14]. Running this type of software-configured kernel is referred to as a "microhypervisor," which combines both microkernel and hypervisor functionalities [35].

The microhypervisor is responsible for providing resource containers, execution contexts, scheduling, inter-process communication, and synchronization mechanisms for its user mode. As shown in Figure 2, the user mode of the microhypervisor can directly run applications, making it suitable for executing real-time tasks, fast boot tasks, and functional safety tasks, among others. Furthermore, the user mode of the microhypervisor can also run virtual machines through a virtual machine manager, making it suitable for running OSs with a rich software ecosystem, such as virtual Linux systems.

From a functional safety perspective, because virtual OSs have a vast software ecosystem that can support complex hardware for business tasks, while the microhypervisor system can directly host safety-critical tasks, this approach is a feasible design for in-vehicle virtualization. The hypervisor host domain serves as the guardian of the virtual machine domain, similar to the role of an independent MCU.

2.3 Limitations of In-Vehicle Microhypervisor

However, in-vehicle microhypervisor solutions present two significant challenges. One is the requirement for a specific high-end hardware base, and the other is the performance of inter-process communication (IPC), which can become a bottleneck for the entire software-hardware system.

The microkernel can provide verified real-time capabilities and a minimal trusted computing base for safety-sensitive applications, thus requiring a hardware platform with safety and reliability. Furthermore, the microkernel can run complex multimedia and intelligent AI tasks by executing virtual machines. In this scenario, it demands a hardware foundation with substantial computational power, abundant peripheral modules such as display units, GPU modules, AI components, and network modules. Consumer-grade SoC chips lack safety and reliability assurances, while automotive-grade MCU chips often cannot meet the computational and peripheral requirements of complex multimedia tasks. Typically, automotive-grade SoC chips are needed to simultaneously run safety-critical and non-critical tasks. However, as shown in Table 1, the production costs of automotive-grade SoC chips are exceptionally high, and they are currently being adversely affected by global supply chain shortages [23–25].

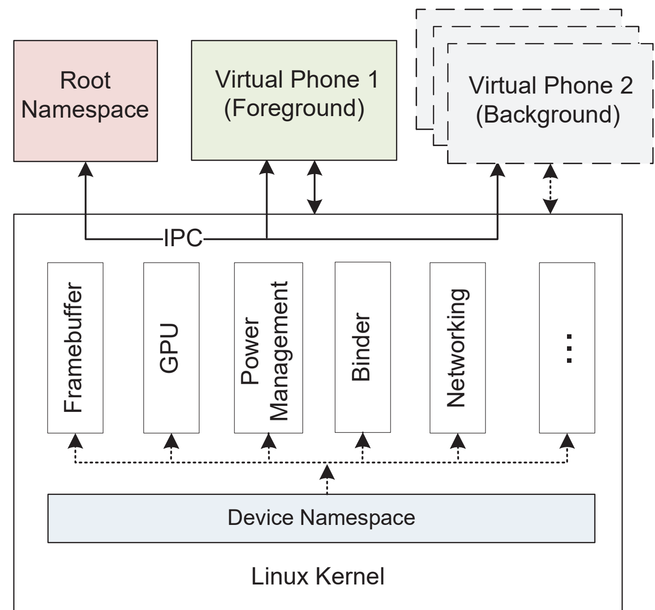


Figure 3: The overview of Cells [26, 27]. Only the virtual phone running in the foreground is displayed at any time.

Furthermore, within microkernels, any communication between different user processes is based on IPC, and this is an operation-intensive process. For instance, if a client process writes data to an external block device, it first communicates with the file system, which then notifies the disk device driver to write data to the block device. All communication is carried out through IPC. Therefore, IPC performance is a crucial technical metric for microkernels [36].

2.4 Linux Container for In-Car Virtualization

One way to address the challenges discussed in §2.3 is to combine a consumer-grade SoC chip with an automotive-grade MCU chip. High-computational tasks like deep learning, computer vision, video encoding/decoding, and 3D graphics run on the ordinary SoC chip, while critical tasks run on the MCU chip. This approach not only alleviates the shortages of automotive-grade SoC chips but also reduces costs. Furthermore, this combined chip structure does not require the use of a complicated microhypervisor. Instead, the Linux container technology, together with an independent MCU solution, can better meet the "one-core, multiple-screen" requirements of intelligent cockpits [37]. The entire solution involves isolating non-critical subsystems using the container technology to ensure software service quality. Simultaneously, it employs a dedicated MCU chip to host safety-critical tasks.

2.5 Android Container Framework

The Linux container technique used by AutoVP is derived from the Cells project [26, 27], a lightweight Android virtualization framework. It enables multiple containerized Android

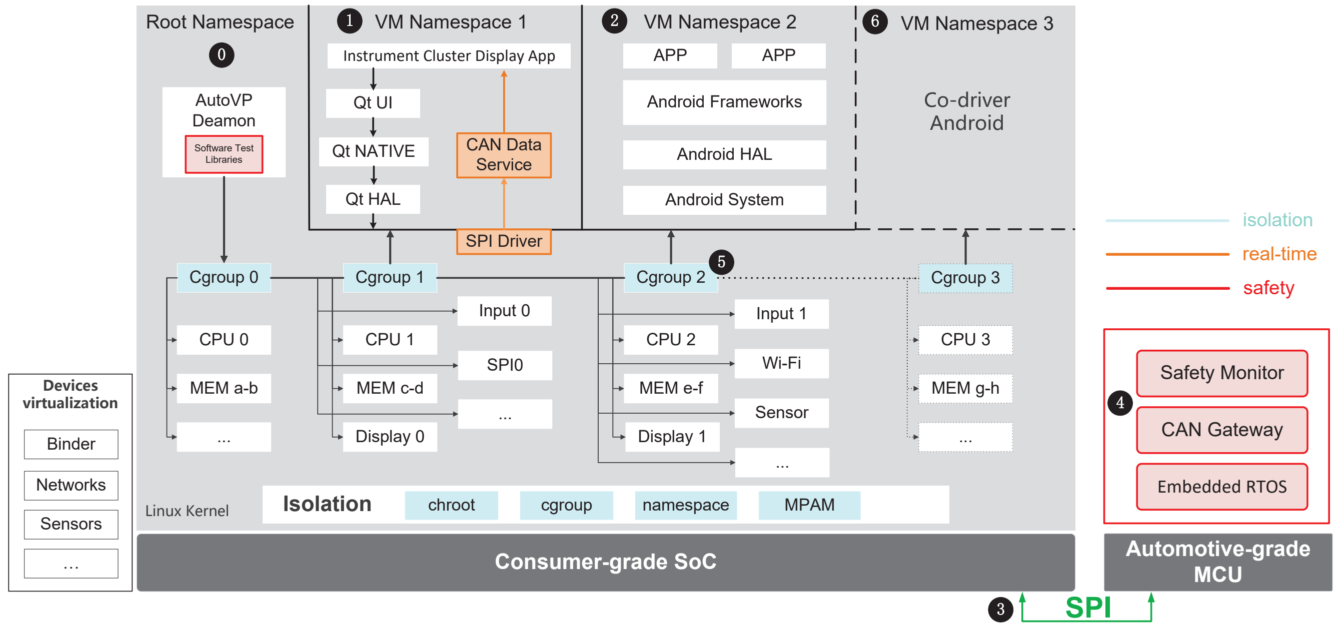


Figure 4: Overview of AutoVP's architecture. AutoVP supports multiple Android containers. For example, both driver and co-driver can have their own in-vehicle infotainment systems running on isolated Android containers.

instances to run simultaneously on the same mobile device in an isolated manner. As shown in Figure 3, Cells introduces the concept of foreground and background container systems, where only one container system is displayed in the foreground while others operate in the background. Cells implements a new device namespace mechanism and user-level proxy method, which, in conjunction with Linux namespaces, allows for the multiplexing of hardware resources across multiple containerized Android instances while providing nearly lossless performance.

However, the foreground-background design used by Cells is not applicable to the automotive application scenario, in which, each virtual system needs to directly provide services to users simultaneously; in other words, all virtual systems must be running in the foreground. Additionally, Cells primarily focuses on how to share hardware resources among multiple virtual Android systems through software virtualization methods. In contrast, in the automotive application scenario, the emphasis is on isolating and decoupling multiple virtual systems and reducing interference between them.

Therefore, the virtualization methods of Cells, such as user-level proxy and mutual-exclusion use of hardware modules, may not be applicable in automotive cockpits. Furthermore, in Cells, all virtual phones share hardware resources such as CPU and memory. When multiple virtual phones run simultaneously, they contend for these resources, leading to mutual interference. In automotive systems, hardware resources also need to be forcibly isolated.

3 AutoVP Overview

Figure 4 illustrates the architecture of AutoVP, including the mixed-criticality decoupled design using two low-cost chips. The consumer-grade SoC chip runs a root namespace and two container spaces. The root namespace runs AutoVP's control plane (⓿ in Figure 4), which has a small software stack primarily responsible for managing the startup, shutdown, resource allocation, and isolation functions of container systems. The other two container spaces are dedicated to instrument cluster (IC) display and in-vehicle infotainment (IVI) functions, respectively. These container systems have undergone software-based resource isolation for CPU, memory, and peripheral resources. Additionally, the separate MCU chip hosts safety-critical operations, such as safety monitoring and Controller Area Network (CAN bus). Interconnection between the SoC chip and MCU chip is facilitated through inter-chip communication technologies such as Serial Peripheral Interface (SPI). AutoVP employs software virtualization techniques for various devices such as binder, WiFi network, and sensors, allowing multiple virtual systems to simultaneously utilize these devices.

The container hosting the instrument cluster display (❶ in Figure 4) consists of the Qt framework abstraction layer [38], Qt native API, Qt UI, CAN message parsing service, and instrument display app. The vehicle data required for the digital instrumentation comes from the vehicle standard communication service running on the MCU chip. The instrument display app on the SoC chip exchanges data with the MCU chip through the SPI interface (❸ in Figure 4). This con-

tainer needs to occupy GPU and display hardware resources to visualize instrument graphical interfaces.

The second container (❷ in Figure 4) runs a complete Android system. It is responsible for hosting complex in-vehicle infotainment functions such as high-definition navigation map, voice broadcasting, Bluetooth headset, and WiFi network services. It also provides smooth interactive functions to respond promptly to user command operations. This container occupies most of the SoC chip's hardware resources, such as GPU, display, WiFi, camera, Bluetooth, and USB.

Both the two containers are configured with independent touchscreens for user operations. The display and input subsystems of the two containers are completely independent. Please note that AutoVP supports multiple Android containers. For example, two isolated Android containers (❷ vs. ❻ in Figure 4) can host separate in-vehicle infotainment systems for the driver and co-driver, respectively.

For safety-critical functions, AutoVP utilizes a hardware-based isolation by employing an external MCU chip to host them (❹ in Figure 4). This MCU chip is responsible for running safety-critical functions such as automotive regulation communication on an embedded real-time OS, serving as a guardian system to the main SoC chip.

4 In-Vehicle Container Implementation

The development of an in-vehicle container presents a plethora of challenges, ranging from intricate hardware resource multiplexing to the need for a fine-grained isolation mechanism, and to interactions with safety-critical tasks. In this regard, AutoVP has made significant advancements over the Cells solution [26, 27][†] to fulfill the requirements of an intelligent cockpit. Unlike Cells, AutoVP does not rely on the foreground-background container system design. Instead, all container systems operate in the foreground. Therefore, our approach's crux is to ensure that these virtual systems operate independently without interfering with each other.

Compared to Cells, AutoVP's container mainly differs in three perspectives: device virtualization methods, isolation, and monitoring mechanisms. AutoVP's monitoring mechanism allows for the early detection of potential failures in complex systems running on the SoC chip. We will introduce them in the follow-up subsections.

4.1 Device Virtualization Methods

AutoVP's device virtualization involves a significant workload, encompassing various board-level and peripheral devices. In particular, the device virtualization methods can be summarized as follows.

[†] Cells's virtualization methods to many hardware devices (e.g., filesystem, network, display, and power) have been obsolete since Android 6.0.

Table 2: The list of virtualized devices and services. Virtualization method ID: (1) multiple identical devices; (2) kernel-level device virtualization; (3) user-level device virtualization.

Virtualization Method ID	Virtualized Devices and Services
(1)	Display, Input, Audio, Bluetooth
(2)	Binder, Power Management, Network Sensors, GPS, SELinux
(3)	WiFi, Adb

1. Multiple Identical Devices: In this approach, the hardware base supports multiple instances of the same type of device, each serving different virtual systems. For instance, in an automotive cockpit, there may be multiple touchscreen displays. These displays can be allocated to different virtual systems by modifying the system configuration.

2. Kernel-level Device Virtualization: In cases where the hardware base is equipped with a single device, such as power management, we made modifications to the kernel power driver to provide data isolation and multiplexing capabilities for that hardware module, allowing the hardware module to respond to power control requests from multiple virtual systems simultaneously.

3. User-level Device Virtualization: In scenarios where virtualization at the kernel level for complex devices is challenging, like the WiFi module, modifications are made to the user-level WiFi service process. Then, we implement a system-level IPC mechanism to enable multiple virtual systems to share the WiFi functionality provided by the customized WiFi service process.

AutoVP employs the above methods to conduct specific device virtualization work, resulting in the list as shown in Table 2. Simultaneously, in order to meet the requirements of smart cockpit applications, adjustments have been made to the virtualization methods of certain modules.

One-Core, Multiple-Screen To fulfill the demand for a "one-core, multiple-screen" smart cockpit, the SoC chip supports multiple display interfaces, such as Camera Serial Interface and DisplayPort, allowing the system to connect to multiple independent touchscreens. We configure separate touchscreens for both the IC display container (❶ in Figure 4) and the IVI container (❷ in Figure 4) to accommodate user interactions. The display and input subsystems of these two containers are entirely independent.

Inter-system Communication Within the automotive cockpit, multiple touchscreens are positioned in close proximity. This configuration necessitates frequent collaboration among these screens, such as sharing map services from the driver's in-vehicle infotainment (IVI) system to the instrument display screen or sharing a video between the driver's IVI and the

co-driver's IVI systems. Therefore, the inter-system communication mechanism becomes pivotal.

AutoVP achieves this by virtualizing the *binder* IPC mechanism [39], ensuring that each virtual machine (VM) has its own virtual binder node. The binder driver is a pseudo device in the kernel and does not correspond to any actual hardware. We modify the binder driver so that each VM has its own independent set of data structures within the binder driver; these binder data structures for different VMs do not interfere with each other. Furthermore, we introduce a routing and forwarding mechanism, adding "bridges" between virtual binder nodes, making it easier for virtual machines to access each other's functional services. The virtual binder routing and forwarding mechanism effectively meets the requirements for inter-VM screen casting and data transmission. For instance, AutoVP provides compositor image stacking services in the IC display container, allowing the IVI container to access the compositor service to project navigation maps onto the instrument cluster display screen, thereby enhancing the user experience.

4.2 Isolation Mechanisms

As an intelligent cockpit system, ensuring an isolation among different functions is imperative to prevent interference between them. AutoVP achieves the isolation of various business subsystems through several mechanisms within the host system. First, it employs the *chroot* mechanism to isolate the file system environment, thereby separating the software stacks of each business subsystem. Second, it utilizes the *cgroups* mechanism to allocate system hardware resources such as CPU, memory, and peripherals, ensuring that high-priority tasks have independent resources to execute critical functions. Third, AutoVP employs the *namespace* mechanism to segregate the kernel resources of different systems, effectively hiding them from each other, thereby reducing inter-system coupling and ensuring minimal interference among the various business subsystems. Additionally, it leverages the new MPAM (Memory System Resource Partitioning and Monitoring) feature [40] to dynamically isolate resources like cache and memory bandwidth, thus mitigating performance interference between different workloads at the hardware level and ensuring stable performance for high-priority tasks.

Cgroups AutoVP employs the *cgroup* mechanism [41] to isolate the hardware resources utilized by each business subsystem, thus reducing coupling and interference between these subsystems (❺ in Figure 4). For instance, the *cpuset* subsystem is used to allocate independent CPU cores and memory nodes to task groups; the *memory* subsystem is employed to allocate memory usage to task groups, and the *devices* subsystem is utilized to allocate peripheral resources to task groups. AutoVP, taking into account the characteristics of each business subsystem, engages in precise management of the hardware resources consumed by these subsystems. This

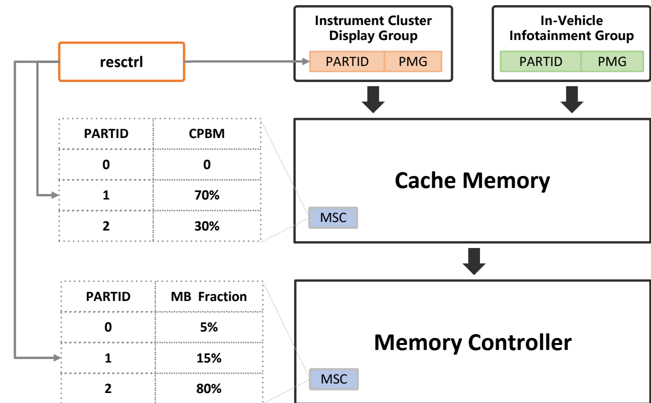


Figure 5: MPAM utilizes PARTID (Propagation of a Partition ID) and PMG (Performance Monitoring Group) to label business groups, and it employs MSC (Memory-System Component) configuration list for various hardware modules.

Table 3: Cache and memory bandwidth allocation strategies for various business subsystems.

Business Subsystem	Cache	Memory Bandwidth
Instrument Cluster Display (❶ in Figure 4)	70%	15%
In-Vehicle Infotainment (❷ in Figure 4)	30%	80%
Root Namespace (⓿ in Figure 4)	0	5%

prevents resource contention among the business subsystems during their operation, thereby avoiding mutual interference. For example, in certain scenarios where the IVI container (❷ in Figure 4) plays multiple videos, consuming a significant amount of memory resources, this might lead to memory starvation in the IC display functions running in ❶ of Figure 4. By implementing resource isolation through the cgroup mechanism, AutoVP effectively ensures the long-term stability and robust operation of each business subsystem.

Namespace AutoVP utilizes the *namespace* mechanism [42] to isolate the kernel resources of each system, such as processes, networking, file systems, and driver data. This separation allows multiple business subsystems to hide from each other, reducing coupling between subsystems and striving to ensure the independent operation of each business subsystem without mutual interference. For instance, it becomes possible to halt a specific business subsystem without affecting the normal operation of other business subsystems.

MPAM MPAM (Memory System Resource Partitioning and Monitoring) is a new feature introduced in ARM v8 [43], enabling the allocation and monitoring of resources such as cache, memory bandwidth, and SMMU (System Memory Management Unit). This feature, operating at the hardware

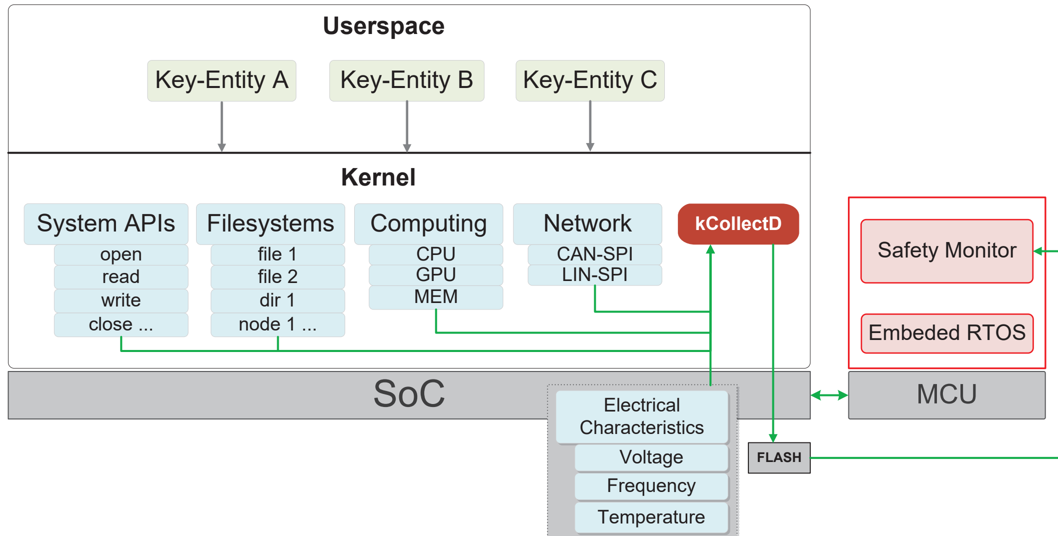


Figure 6: The kernel kCollectD module periodically collects data related to critical entities' interactions with kernel resources and hardware characteristics. This data is then stored in an external flash memory. The Safety Monitor module running on the MCU chip analyzes the data stored in the flash memory and provides a determination or assessment.

level, reduces the interference between different workloads, ensuring the stability of high-priority task performance.

AutoVP leverages the hardware feature of MPAM to dynamically isolate resources at runtime for various business subsystems. For instance, cache and memory bandwidth for the IC display subsystem (➊ in Figure 4) and the IVI subsystem (➋ in Figure 4) are isolated using MPAM. As shown in Figure 5, different configurations of cache and memory bandwidth access policies are tagged with PARTID (Propagation of a Partition ID). These PARTIDs are then bound to the IC display subsystem and the IVI subsystem, respectively. Consequently, every time cache and memory resources are accessed, the resource queries the resource usage policy of the subsystem associated with the bound PARTID. This control of resource utilization boundaries reduces resource conflicts and competition when the IC display subsystem and the IVI subsystem access cache and memory resources.

We empirically configure cache and memory bandwidth allocation strategies for business subsystems based on their specific characteristics, as outlined in Table 3. The IC display subsystem, although much smaller compared to the IVI subsystem, demands higher real-time performance. Therefore, it is allocated a greater share of cache space and a smaller portion of memory bandwidth. This resource allocation strategy allows the instrument cluster subsystem to dominate more than half of the cache resources, significantly enhancing cache hit rates. Besides, it is also assigned 15% of the memory bandwidth, reducing interference from the complex central control Android container. By employing this configuration strategy to elevate the priority of dynamic resource utilization for critical instrument cluster tasks, we are able to reduce IC display function latency and improve stability.

4.3 Monitoring Mechanisms

In Figure 4, the subsystems built on top of the main SoC chip primarily host non-safety-critical tasks. However, these subsystems include certain critical functions that have to interact with safety-critical data. For example, the IC display subsystem is responsible for displaying real-time vehicle information such as speed, fuel consumption, warning indicators, and other vehicle status data while the vehicle is in operation. These pieces of information are particularly crucial for the driver during the driving process. As a result, AutoVP needs to implement mechanisms for real-time monitoring of such functions. These mechanisms are designed to determine whether these critical functions are operating according to the prescribed procedures. In the event of any irregularities or malfunctions detected in these critical functions, the system must promptly alert the driver and employ redundancy mechanisms to rectify the abnormal functions.

kCollectD Kernel Module AutoVP establishes a monitoring module within the system kernel. As shown in Figure 6, this module, called *kCollectD*, periodically collects behavioral data from critical entities within the system, analyzing and assessing whether these entities are operating in accordance with predefined procedures. The critical entities under scrutiny include processes and kernel modules. The behavioral data encompass various aspects such as which system interfaces were invoked, which files were manipulated, the utilization of heap and stack memory, the CPU time slices consumed, and the number of frames rendered, among others. Given that collecting behavioral data from these entities can incur performance overhead, it is imperative to target the acquisition of key data specific to the relevant business processes and analyze essential behavioral characteristics. For example, in

the case of the instrument cluster display subsystem where the displayed content is crucial for the driver, it is essential to monitor the number of frames of graphics rendered by the IC display subsystem per second. If, during the periodic analysis phase, it is determined that a critical entity has deviated from the prescribed actions, an audit alarm is triggered immediately. This leads to the termination of the relevant entity's operations, followed by the initiation of redundancy mechanisms to rectify the error. For instance, if it is observed over a period of time that the number of frames rendered per second by the IC display subsystem is lower than expected, indicating a freeze in the display program, the system will sound an alarm to alert the driver and activate an emergency instrument cluster program to take over the display screen.

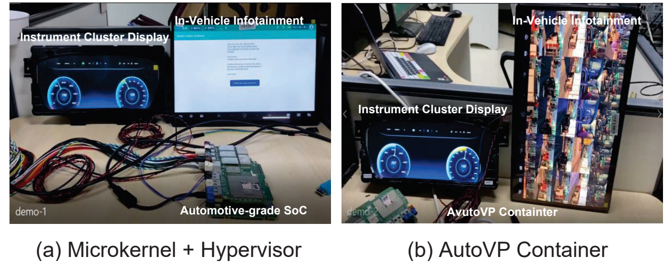
Monitoring Hardware Data Simultaneously, AutoVP's monitoring module, kCollectD, also collects hardware information from the SoC chip in real-time, such as power, clock, reset, and temperature. Both collected kernel resources (e.g., APIs, files, and networks.) and hardware characteristics (e.g., voltage, frequency and temperature) are then stored in an external flash memory. The Safety Monitor module, which runs on the MCU chip, analyzes the data stored in the flash memory and assesses whether the critical functions running on the SoC are operating normally. For example, if the Safety Monitor module detects an excessively high CPU voltage, changes in clock frequency, or excessive temperature on the SoC chip, it indicates that the software running on the SoC may not be functioning correctly or could potentially result in abnormal behavior. In such cases, the Safety Monitor triggers an alarm sound, notifying the driver of the system's abnormal operation and the need for manual intervention.

5 Evaluation

Our experiments focus on measuring performance metrics from seven aspects: startup time, memory usage, battery consumption, CPU, GPU, network performance, and real-world workloads. We compare AutoVP with a microhypervisor-based in-vehicle virtualization product. Although they are designed to run on different hardware bases, we deploy them on the same automotive-grade SoC chip to preserve the same test conditions. The commercial product used in our experiments has terms of use that disallow publication of tool performance and tool output. Therefore, we anonymize the product name in this paper. Furthermore, as MPAM is a new feature introduced in ARM v8 [43], we also want to evaluate the effect of MPAM on dynamically isolating cache and memory bandwidth.

5.1 Experiment Setup

We conduct performance experiments on top of an automotive-grade SoC chip (configuration: 8-core ARM Cortex-A710 at 2.15 GHz, ARM Mali G78 GPU, 8GB DDR, 128GB UFS) to run the microhypervisor-based in-vehicle virtualization system (commercially licensed) and AutoVP. Please note that here we also run AutoVP on an automotive-grade SoC instead of a consumer-grade SoC. The primary reason is that Microhypervisor solutions can only run on top of automotive-grade SoCs; otherwise, we cannot perform the comparative evaluation. Due to safety considerations, we are unable to conduct road testing on an actual vehicle; instead, we use the simulated data provided by the automotive manufacture as vehicle-related data. Figure 7 shows the effect of running both in-vehicle virtualization solutions with simulated vehicle-related data. The specific subsystems and their boot sequence for both solutions are listed in Table 4. Next, we present detailed software & hardware configurations for each solution. *Please note that with regards to the static allocation of specific hardware resources for each subsystem, we adhere to the reference configuration information provided by the automotive manufacturer*.



(a) Microkernel + Hypervisor (b) AutoVP Container

Figure 7: Two in-vehicle virtualization solutions are running.

Table 4: The subsystems of two in-vehicle virtualization solutions. The numbers in Row 2∼4 represent the boot sequence of each subsystem. "IC" is short for instrument cluster.

Boot Sequence	Microhypervisor	AutoVP
(1)	Application Domain (Instrument Cluster)	Root Namespace
(2)	System Domain	VM Namespace 1 (IC Display)
(3)	Android VM (In-Vehicle Infotainment)	VM Namespace 2 (In-Vehicle Infotainment)

Microhypervisor Configurations The instrument cluster (IC) subsystem, deployed within the microkernel's application domain, consists of a set of graphical display programs. Hardware resources, including CPU physical core 1 and 2, 500MB of memory, Display 1, and 50% of GPU partition resources are allocated to the IC subsystem. This group of programs also includes driver modules that can directly utilize the corresponding hardware devices. The microkernel's system domain comprises system components, hardware modules, and some device drivers, such as network device driver. The system domain is allocated with CPU physical core 3 and 4, 2GB of memory, storage, and network devices. The in-vehicle infotainment (IVI) subsystem is deployed in the form

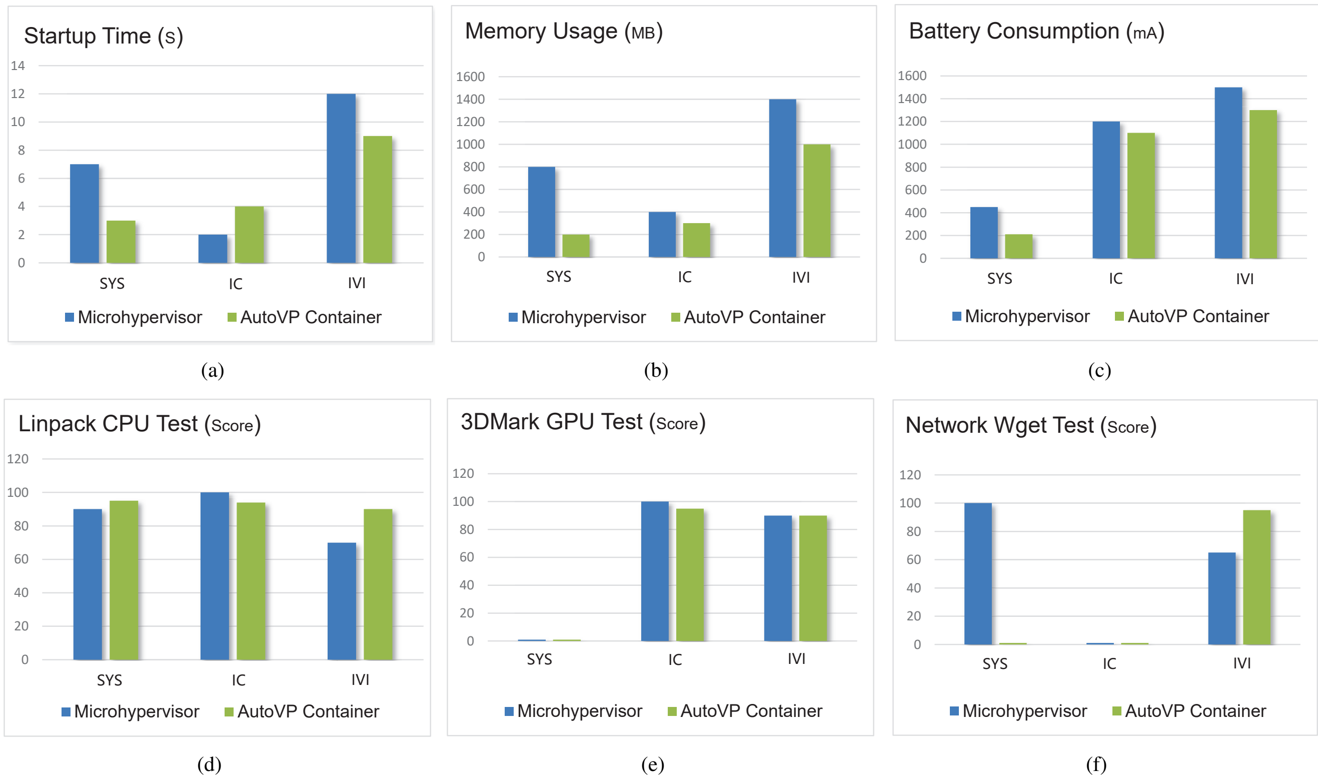


Figure 8: Standard benchmark evaluation results. “SYS” represents the system domain for the microhypervisor product and the root namespace for AutoVP, respectively. “IC” is short for instrument cluster, and “IVI” means in-vehicle infotainment.

of an Android virtual machine, which runs the Android version of 12.0. We allocate all remaining hardware resources to the virtual machine, such as CPU physical cores 5∼8, 5.5GB of memory, 50% of GPU partition resources, and Display 2.

AutoVP Configurations The Linux kernel version used by AutoVP is 5.10.221. The root namespace require low computing resources to run management panel software, and thus AutoVP employs cgroup technology to allocate CPU physical core 1 and 300MB of memory to the root namespace. The instrument cluster display subsystem runs the Qt graphical display framework (version 5.12.); CPU physical core 2 and 3, 500MB of memory, Display 1, and 50% of GPU partition resources are allocated to this subsystem. The container running Android 12.0 hosts the in-vehicle infotainment subsystem, which occupies all remaining hardware resources of the chip, including CPU physical cores 4∼8, 7.2GB of memory, 50% of GPU partition resources, and Display 2.

5.2 Methodology

According to the boot sequence of each subsystem (see Table 4), the method for collecting startup time data involved sequentially starting the application domain, system domain, and Android virtual machine for the microhypervisor solution and recording the startup time. This process was repeated 20 times, and the average startup time was calculated in seconds. Measuring AutoVP’s startup time followed a similar procedure, but with a boot sequence of the root namespace, IC display, and Android container. We collect real-time memory usage data for each subsystem after it has been running stably for a period of two hours. Similarly, battery consumption data are recorded within the same two-hour time window.

To collect CPU performance data, we installed the Linpack benchmark in the application domain, system domain, and Android virtual machine for the microhypervisor solution, respectively. We ran the Linpack benchmark for 1 hour and recorded normalized resource utilization data—we repeated this process 10 times and calculated the average value. The CPU measurement of AutoVP followed a similar procedure, but with performance data collected in the root namespace, IC display, and Android container. A higher Linpack score indicates more efficient CPU utilization, resulting in lower performance overhead. It should be noted that the Linpack benchmark evaluates the single-core CPU performance.

The method for collecting GPU performance data was similar to that for CPU performance testing, but we used the 3DMark benchmark. A higher 3DMark score indicates superior GPU utilization, leading to reduced performance overhead. In a similar vein, the method for collecting network performance data leverages the wget application. A high wget score is indicative of an efficient utilization of network bandwidth, thus resulting in lower bandwidth consumption.

5.3 Performance Measurements

Startup Time Figure 8(a) shows the results of startup time. The microhypervisor solution, due to its microkernel structure, can minimize the startup time of the instrument cluster (IC) subsystem. The IC operates as a microkernel application running on top of the microkernel OS, enabling a swift initiation of the IC display. In contrast, AutoVP employs a monolithic kernel approach, with many drivers deployed in the kernel mode. During the system startup phase, it is necessary to initialize all driver modules as a priority, even if certain driver modules will not be immediately used. This results in a longer initialization time for the Linux kernel itself in AutoVP. Additionally, AutoVP requires the container management program to be started before the IC display can be initiated. Experimental data indicates that the startup time for the IC display in AutoVP is slower compared to the microhypervisor solution.

However, the microhypervisor needs to first run the system domain and then initiate complex software virtualization frameworks, such as the virtual machine manager, before launching the virtualized Linux kernel and subsequently the Android VM that hosts in-vehicle infotainment functions. AutoVP, by contrast, requires initializing the Linux kernel only once and then running the container management program to start the Android container. As a result, AutoVP starts the IVI subsystem faster than the microhypervisor solution.

Memory Usage Figure 8(b) shows the comparison of memory usage. Both of these two solutions are allocated 500MB of memory for the IC subsystem. However, in the microhypervisor, the IC subsystem also includes display driver and GPU driver, whereas in AutoVP, the required display and GPU drivers for the IC run within the Linux kernel. Therefore, the memory usage of the IC subsystem in the microhypervisor is slightly higher. Furthermore, the microhypervisor requires the prior execution of complex software virtualization framework, such as the virtual machine manager, before running the virtualized Linux kernel and subsequently the Android VM. In contrast, AutoVP can directly run the Android container on the Linux kernel. Experimental data demonstrates that the memory usage of the container in AutoVP is lower than the that of Android VM in the microhypervisor solution.

Battery Consumption Figure 8(c) shows the measurement of battery consumption. The IC subsystem in the microhypervisor requires running display driver and GPU driver, as well as directly managing hardware resources for IC display peripherals and GPU components. In contrast, the required display and GPU drivers in AutoVP run within the Linux kernel. This can explain why the IC subsystem in the microhypervisor consumes more power. The microhypervisor solution necessitates the prior execution of complex software virtualization framework. Additionally, some hardware devices are directly allocated to and managed by the virtualized Android system. Conversely, AutoVP can directly run the Android container on the kernel, with the kernel managing all peripheral hardware. Therefore, experimental data demonstrates that the power consumption of the Android container in AutoVP is lower than that of the virtualized Android system in the microhypervisor solution.

CPU Performance Figure 8(d) shows the scores of Linpack benchmark, which is the most popular benchmark for ranking of high performance systems. Since both the microhypervisor and AutoVP allocate two CPU physical cores and 500MB of memory to the IC subsystem, the CPU performance data for the IC subsystem are comparable between the two solutions. However, the virtualization framework and virtualized Linux kernel running in the microhypervisor solution consume a significant amount of CPU and memory resources. As a result, experimental results indicate that the CPU performance data for AutoVP's container is superior to that of the virtualized Android system in the microhypervisor.

GPU Performance Figure 8(e) shows the scores of 3DMark benchmark that tests the system's GPU performance. Both the microhypervisor and AutoVP employ GPU partitioning technology, a hardware resource slicing technique that effectively addresses the isolation of GPU resources between the IC subsystem and the Android VM. Neither the system domain of the microhypervisor nor AutoVP's root namespace utilizes GPU resources. Therefore, the GPU performance data obtained from tests conducted on the IC subsystems and the Android VMs are comparable between these two solutions.

Network Performance The network performance data measured by wget is depicted in Figure 8(f). While the microhypervisor's WiFi module is deployed in the system domain, enabling it to benefit from superior network performance, AutoVP's root namespace does not utilize the network in its normal functioning. Furthermore, neither of the IC subsystems in these two solutions use the network. However, the Android VM in the microhypervisor necessitates software virtualization techniques to time-share the WiFi functionality in the system domain. On the other hand, the Android container in AutoVP requires only Linux kernel features to access external networks through the WiFi module. Experimental results show that the network performance of AutoVP's container is significantly better than that of the virtual Android system in the microhypervisor solution.

Real-world Workloads We run common in-vehicle applications such as the dashboard, navigation, music, movie, climate control, vehicle settings, infotainment, Bluetooth connectivity, voice navigation, and reverse camera in both solutions. We use Android *gfxinfo* tool to measure the frame rate (frames drawn per second) of each application. The frame rate metric reflects the performance of the application interface and can assess whether user interactions with the application are smooth. The evaluation results are depicted in Figure 9, wherein a higher frame rate correlates with a better quality of the video or animation. If an application is relatively smooth with a high frame rate, it indicates good performance in the current

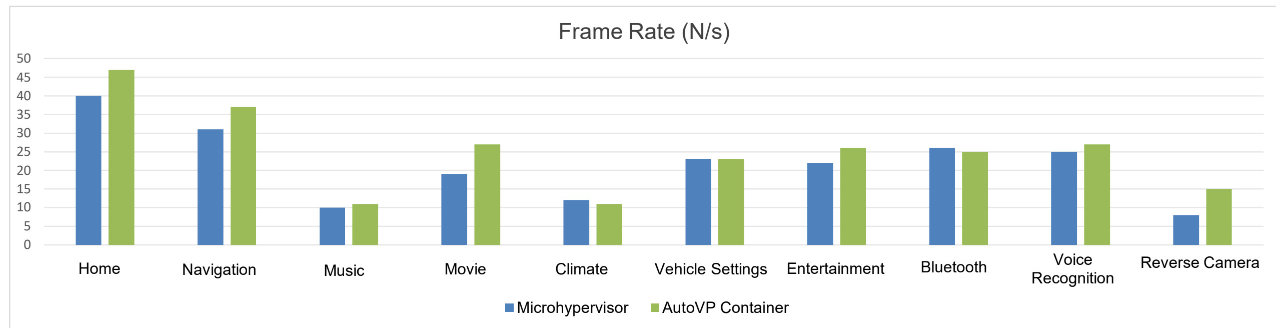


Figure 9: Comparative performance evaluation results with common in-vehicle applications. The higher the frame rate, the better the quality of the video or animation will be.

system environment, including CPU, GPU, and file I/O. Experimental results indicate that the frame rates per second for common in-vehicle applications are higher when running in AutoVP compared to the Hypervisor solution.

Summary The microhypervisor solution, owing to its microkernel structure, enables a quick launch of its IC subsystem. In contrast, AutoVP runs the IC display on the Linux monolithic kernel, which fails to ensure a rapid startup of the IC display. The distinct advantage of AutoVP lies in its ability to ensure efficient utilization of system resources by the Android container. In this configuration, the Android container does not need to run complex virtualization frameworks, and thus can make better use of CPU, network, memory, and battery, resulting in a smoother user experience when running common in-vehicle applications. Due to the same GPU resource allocation, the microhypervisor's VM achieves GPU resource utilization efficiency that is on par with AutoVP's container.

5.4 MPAM Measurements

Due to the noticeable isolation effect of MPAM on mixed-criticality systems [40, 43], we conduced a separate experiment for MPAM resource isolation scenarios in AutoVP. The experiment involved disabling/enabling the MPAM mechanism for IC display and IVI subsystems. For the initial 14 minutes, IC display and IVI shared CPU cache and memory bandwidth resources. At the 15th minute, MPAM was activated. We follow the specific cache and memory bandwidth allocation strategies for various business subsystems, as outlined in Table 3.

As show in Appendix Figure A1(a), it is evident that without enabling MPAM, there is intense competition between IC and IVI services for cache resources, leading to significant mutual interference. However, upon enabling MPAM, the usage of cache resources by IC and IVI services stabilized. Notably, IVI's usage of cache resources was significantly suppressed, with IC's cache usage percentage increasing to about 70%. This drastic improvement notably enhanced cache hit rates and consequently improved IC's performance. However, Appendix Figure A1(b) reveals that the impact of MPAM mechanism on memory bandwidth control was minimal. One contributing factor is that MPAM is a new mechanism introduced by ARM v8, and the current testbed does not offer comprehensive support for the new MPAM mechanism.

6 Discussion & Conclusion

The development of smart automobile cockpits for civilian vehicles needs to balance multiple factors such as safety, reliability, and production costs. The advantages of AutoVP are evident: cost-effectiveness and an almost entirely open-source system. Thanks to the lightweight virtualization features of containers, it allows non-safety-critical tasks to efficiently utilize hardware resources. AutoVP is poised to have enduring significance, persisting beyond the resolution of the automotive chip crisis [23–25]. At present, our approach involves a static resource allocation strategy, as outlined in Table 3. Future endeavors will delve into enabling dynamic resource allocation. Furthermore, we will also explore how to provide real-time capabilities for safety-critical tasks on a Linux system without the need for an external MCU chip.

Conclusion In this paper, we present a new intelligent cockpit virtualization architecture. We segregate safety-critical functions from other non-critical functions into two low-cost chips, respectively. We also run an Android container on the main SoC chip to host non-safety-critical tasks. Our Android container solution features a rich software ecosystem, excellent performance, and inherent cost-effectiveness. By incorporating automotive-grade MCUs to handle safety-critical tasks, the entire system complies with automotive standards. Our comparative evaluation results with a commercial in-vehicle microhypervisor product are exciting.

Acknowledgments

We sincerely thank NSDI anonymous reviewers and our shepherd Amit Levy for their insightful comments. Jiang Ming was supported by the National Science Foundation under grant CNS-2312185 and Carol Lavin Bernick Faculty Grant.

References

[1] Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. Making the Case for Centralized Automotive E/E Architectures. *IEEE Transactions on Vehicular Technology*, 70(2), 2021.

[2] Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. Cannon: Reliable and Stealthy Remote Shutdown Attacks via Unaltered Automotive Microcontrollers. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*, 2021.

[3] S. Kanajan, C. Pinello, Haibo Zeng, and A. Sangiovanni-Vincentelli. Exploring Trade-off's Between Centralized versus Decentralized Automotive Architectures Using a Virtual Integration Environment. In *Proceedings of the 2006 Design Automation & Test in Europe Conference (DATE '06)*, 2006.

[4] Jan Pelzl, Marko Wolf, and Thomas Wollinger. Virtualization Technologies for Cars: Solutions to Increase Safety and Security of Vehicular ECUs. In *Proceedings of the 2009 Embedded World Conference*, 2009.

[5] Marius Strobl, Markus Kucera, Andrei Foeldi, Thomas Waas, Norbert Balbierer, and Carolin Hilbert. Towards Automotive Virtualization. In *Proceedings of the 2013 International Conference on Applied Electronics*, 2013.

[6] Thomas Gaska, Brian Werner, and David Flagg. Applying Virtualization to Avionics Systems — The Integration Challenges. In *Proceedings of the 29th Digital Avionics Systems Conference*, 2010.

[7] Victor Bandur, Vera Pantelic, Matthew Dawson, Alexander Schaap, Bryon Wasacz, and Mark Lawford. A Domain-Centralized Automotive Powertrain E/E Architecture. Technical report, SAE Technical Paper 2021-01-0786, 2021.

[8] Ashraf Gaffar and Shokoufe Monjezi. Using Artificial Intelligence to Automatically Customize Modern Car Infotainment Systems . In *Proceedings of the International Conference on Artificial Intelligence (ICAI '16)*, 2016.

[9] Arm. The Evolution of Car Cockpit and IVI Systems. https://www.arm.com/markets/automotive/digital-cockpit, 2023.

[10] Michele Paolino, Walt Miner, Daniel Bernal, Artem Mygaiev, Tiejun Chen, and Rich Persaud et al. The Automotive Grade Linux Software Defined Connected Car Architecture. Technical report, The Linux Foundation, June 2018.

[11] Srdjan Usorac, Dejan Bogdanovic, Dario Peric, and Zeljko Lukac. Adding Android Capabilities in Automotive Linux Infotainment: Available Virtualization Technologies. In *Proceedings of the 29th Telecommunications Forum*, 2021.

[12] BlackBerry QNX. QNX Hypervisor Virtualization Software. https://blackberry.qnx.com/en/products/foundation-software/qnx-hypervisor, 2023.

[13] Green Hills Software. INTEGRITY Multivisor. https://www.ghs.com/products/rtos/integrity_virtualization.html, 2023.

[14] SYSGO GMBH. PikeOS: Certifiable RTOS & Hypervisor. https://www.sysgo.com/pikeos, 2023.

[15] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2009.

[16] Everton de Matos and Markku Ahvenjärvi. seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM. *Electronics*, 11(24), 2022.

[17] Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Björn Döbel, and Alexander Böttcher. L4 – Virtualization and Beyond. *Korean Information Science Society Review*, 2, 2008.

[18] Dong-Guen Kim, Sang-Min Lee, and Dong-Ryeol Shin. Design of the Operating System Virtualization on L4 Microkernel. In *Proceedings of the 4th International Conference on Networked Computing and Advanced Information Management*, 2008.

[19] Jörn Schneider and Tillmann Nett. Safety Issues of Integrating IVI and ADAS functionality via Running Linux and AUTOSAR in Parallel on a Dual-Core-System. *Automotive - Safety & Security*, 2015.

[20] Jason Belt, John Hatcliff, Robby, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind, Ihor Kuz, and Kent Mcleod. Model-Driven Development for the SeL4 Microkernel Using the HAMR Framework. *Journal of Systems Architecture*, 134, 2023.

[21] Aswin Sampath Kumar and Tuğrul Daim. Study on Consumer Requirements for Automotive Infotainment Systems. *R&D Management in the Knowledge Era: Challenges of Emerging Technologies*, 2019.

[22] Alessandra Nardi and Uyen Tran. Key Requirements for Automotive SoC Design. Technical report, Synopsys, April 2023.

[23] Keith Naughton. Automotive Chip-Shortage Cost Estimate Surges to $110 Billion. Bloomberg Technology, May 2021.

[24] Xiling Wu, Caihua Zhang, and Wei Du. An analysis on the crisis of "chips shortage" in automobile industry-Based on the double influence of COVID-19 and trade friction. In *Journal of Physics: Conference Series*, volume 1971. IOP Publishing, 2021.

[25] Kristin Dziczek et al. Why the Automotive Chip Crisis Isn't Over (Yet). *Chicago Fed Letter*, 2022.

[26] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

[27] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture. *ACM Transactions on Computer Systems*, 30(3), August 2012.

[28] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, 2007.

[29] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[30] Michael Eder. Hypervisor- vs. Container-based Virtualization. In *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications*, 2016.

[31] International Organization for Standardization. ISO 26262: Road Vehicles — Functional Safety. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en, 2018.

[32] J Langheim, Bruno Guegan, Laurent Maillet-Contoz, Kamel Maaziz, Gilles Zeppa, S Boutin, H Aboutaleb, F Philippot, and Pierre David. System architecture, tools and modelling for safety critical automotive applications–the R&D project SASHA. In *Proceedings of the 2010 Embedded Real Time Software and Systems Conference*, 2010.

[33] Pierre Lucas, Kevin Chappuis, Michele Paolino, Nicolas Dagieu, and Daniel Raho. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS '17)*, 2017.

[34] Google. Automotive OS. https://developers.google.com/cars/design/automotive-os, 2023.

[35] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, 2010.

[36] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys '19)*, 2019.

[37] Business Wire. Automotive Cockpit Multi & Dual Display Trends Report. https://www.researchandmarkets.com/r/61epwn, 2020.

[38] Qt Group. Tools for Each Stage of Software Development Lifecycle. https://www.qt.io/, 2023.

[39] Android Open Source Project. Using Binder IPC. https://source.android.com/devices/architecture/hidl/binder-ipc, [2022].

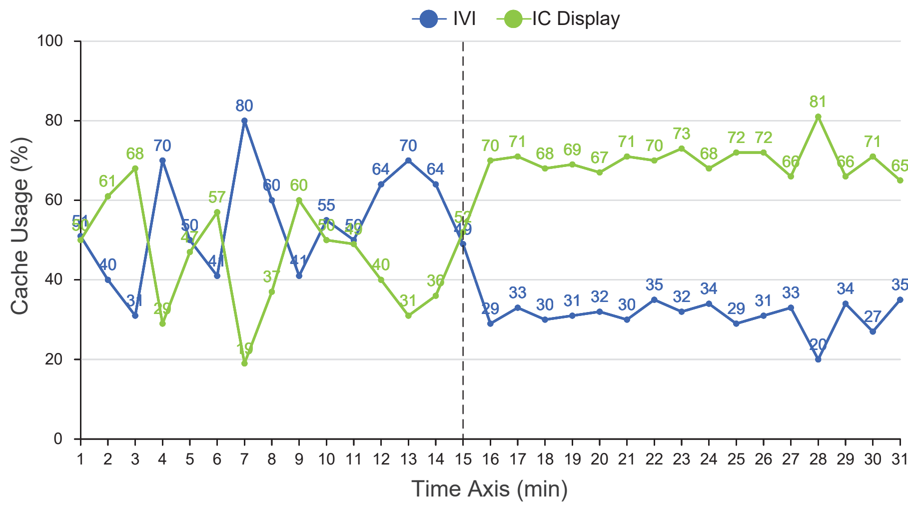
[40] Arm. Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture. Arm Architecture Reference Manual Supplement, 2022.

[41] Chris Down. 5 Years of Cgroup v2: The Future of Linux Resource Control. In *Proceedings of the 34th Large Installation System Administration Conference (LISA '21)*, 2021.

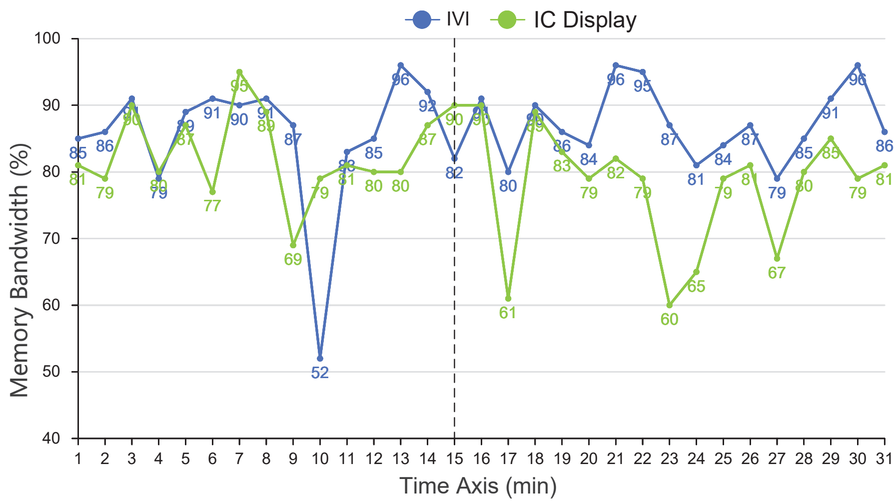
[42] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[43] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Transactions on Computers*, 72(1), 2023.

Appendix



(a) Real-time Cache Usage



(b) Real-time Memory Bandwidth

Figure A1: The measurement of MPAM effects on dynamically isolating cache and memory bandwidth.