A Paravirtualized Android for Next Generation Interactive Automotive Systems

Soham Sinha, Ahmad Golchin, Craig Einstein, Richard West
Department of Computer Science, Boston University
Boston, MA
{soham1, golchin, einstein, richwest}@cs.bu.edu

ABSTRACT

Android's APIs, bluetooth support and smartphone integration provide capabilities for user interaction with In-Vehicle Infotainment (IVI) and vehicle control services. However, Android is not developed to interface with automotive subsystems accessed via CAN bus networks. This work proposes a new automotive system based on our Quest-V partitioning hypervisor, which allows Android to communicate and interact with timing and safety-critical services managed by the Quest real-time OS (RTOS). Quest is used to filter and receive messages from Android applications and to interface with a car's internal CAN bus in a timing predictable manner. Android is then used to host IVI applications and provide a user interface to real-time vehicle services. This system design allows Android to leverage the timing guarantees of Quest, while securely isolating critical hardware components and memory regions.

Quest-V hosts a paravirtualized Android 8.1 (Oreo) guest, which required modification of 126 lines of kernel code. Secure shared memory communication mechanisms between Android and a separate Quest guest provide real-time I/O to CAN bus networks.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**.

KEYWORDS

Machine Virtualization; Android; Real-Time; Automotive Systems

ACM Reference Format:
Soham Sinha, Ahmad Golchin, Craig Einstein, Richard West. 2020. A Paravirtualized Android for Next Generation Interactive Automotive Systems. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications (HotMobile '20), March 3–4, 2020, Austin, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3376897.3377861

1 INTRODUCTION

Due to its popularity in the smartphone market and its familiar user interface (UI), Android is a potentially suitable OS for interactive automotive system services in next-generation cars. These services include support for in-vehicle infotainment (IVI), and configuration of advanced driver assistance systems (ADAS). IVI provides support for audio and video playback, navigation, climate control, and smartphone integration, while ADAS services support features such as lane departure warning or active cruise control. Android's rich application ecosystem makes developing interactive automotive systems straightforward, with its well-developed communication stacks such as bluetooth allowing smartphones to interface easily with the system.

Currently, Android-based interactive automotive systems provide limited functionality such as the ability to play songs, and make phone calls through a user's smartphone. These features are useful, but emerging systems seek to host more complex services that manipulate the chassis, body and powertrain components of an automotive system. For example, an interactive climate control application requires access to a heating, ventilation, and air condition (HVAC) unit, and an ADAS service might affect the vehicle's brakes or powertrain components. Automotive systems require careful isolation of critical components that affect vehicle dynamics and safety from those that support user-interactive services. For this reason, Android-based interactive services are not trusted to directly communicate with other hardware components of the vehicle.

Traditional automotive systems assign different functional components to separate electronic control units (ECUs) connected via a CAN bus network [9]. However, as the complexity of these systems increases, hardware costs, wiring and packaging become prohibitive. It is therefore desirable to enable interactive services to co-exist on a shared hardware platform that has access to other components that influence timing and safety critical control of the vehicle. The management of a shared hardware platform requires a suitable OS to control access to resources and provide software-based functionality that replaces traditional ECUs.

Companies such as Tesla are already using systems such as Linux as the basis for IVI and auto pilot assistive vehicle control [11]. Unfortunately, a standalone Linux system does not provide sufficient isolation or real-time capabilities to guarantee correct vehicle operation without significant code modifications. Timing and security vulnerabilities limit the use of Linux in interactive automotive systems. As an example, security attacks have been observed on Tesla's Linux software stack, which remotely gain control over the vehicle's CAN network [8].

This work proposes a new design for interactive automotive systems in which two (or more) OS regimes, referred to as sandboxes, are integrated in a safe and secure system architectural design. One sandbox contains Android, suitable for user-interactive services, while another features a specialized real-time OS that interacts with timing and safety critical automotive components. The proposed

HotMobile '20, March 3–4, 2020, Austin, TX, USA

ACM ISBN 978-1-4503-7116-2/20/03…$15.00
https://doi.org/10.1145/3376897.3377861

next generation system is based on the design of a partitioning hypervisor called Quest-V [13], in which each sandbox hosts its own guest OS and is assigned exclusive access to a subset of the hardware resources of a single computational platform. Each guest OS independently manages the resources assigned to it and has the ability to communicate with other guests running on the platform if needed. The two OSs used in this architecture are the Android OS and Quest, a real-time OS (RTOS) [3].

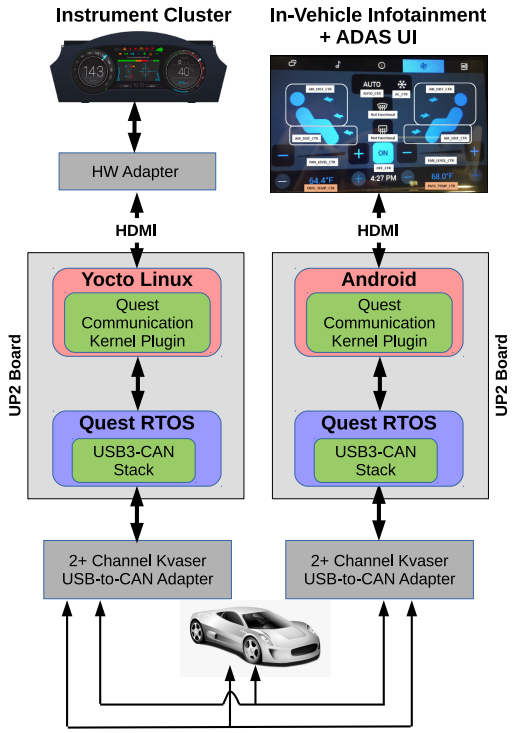


Figure 1: High-level Interactive Automotive System Design

The right-hand side of Figure 1 shows the use of Android and Quest in an interactive automotive system currently under development. The left-hand side of this figure also shows a Quest-V system running on a separate machine, hosting both Quest and Yocto Linux for use with instrument cluster (IC) features, including a speedometer, battery meter and other indicator readings. For this paper, we focus on the system development comprising Android and Quest, as the left-side of Figure 1 is largely complete.

Our current prototype system has been developed on x86-based Up Squared Apollo Lake platforms, similar to those used in Tesla's main computer units (MCUs). We have a fully operational software-based instrument cluster, developed as a Qt application for Yocto Linux. We are now in the process of developing IVI and ADAS support for Android, which has been paravirtualized to run on Quest-V. Secure and timing-predictable communication mechanisms have been implemented in Quest-V to allow Android to safely and predictably communicate with Quest.

In a multicore machine, Android is given access to one or more cores, along with hardware resources such as an HDMI display, bluetooth and wireless networks. The other cores are allocated to Quest, which accesses a serial port for logging and a USB host controller to interface with the CAN buses. Thus, Quest is given access to, and can communicate with, the hardware features of the vehicle such as the HVAC unit through a USB-CAN interface. Android communicates with Quest via a secure shared memory channel to access timing and safety critical CAN interfaces. This allows Android applications to manipulate HVAC settings and advanced driver assistance features.

In this system architecture, Quest exclusively manages and communicates with the vehicle's CAN controllers. Thus, automotive manufacturers only need to develop for and maintain the RTOS. The engineers who have exclusive knowledge of the vehicle's internal components are able to develop for their platform in an environment that is simpler than Linux and also offers real-time capabilities, which are crucial for the vehicle to operate as expected.

The IVI and ADAS applications are supported in Android. Since the critical hardware components of the vehicle will not be exposed to Android, the OS is maintainable by a collaboration between a specialized group of external developers and a few insider system developers in the vehicle manufacturing company, to develop compatible interfaces. There are already such alliances between companies like GENIVI [4], which develop standards and reference implementations of the IVI system. This work describes the design of an interactive automotive system that supports modular automotive system software development.

For this paper, Android 8.1 (Oreo) has been paravirtualized for use with the Quest-V partitioning hypervisor, with just 126 lines of changes to the system kernel. An IVI application developed by a partnering professional company runs in the paravirtualized Android. For the rest of the paper, the design of the right-hand side of Figure 1 is discussed. The integration of Android into the interactive automotive system is detailed, and promising preliminary evaluations are presented that show the timing predictable behavior of the approach.

2 DESIGN

Our interactive automotive system uses Android as the basis for next-generation IVI applications and ADAS user-interface control. Our approach supports the co-existence of the Quest RTOS with Android, to manage timing-critical components of the vehicle. For example, an ADAS torque vectoring and traction control service configured for use on wet, dry, or snow-covered roads, must manage updates to wheel torques within specific time bounds to prevent the vehicle skidding out of control. While we want real-time control to be handled by suitably predictable services, the interface to configure ADAS settings will be exposed to Android.

The Quest-V partitioning hypervisor supports the co-existence of Android and Quest, with real-time communication between each guest managed by secure shared memory channels. Thus, Android is empowered with real-time capabilities afforded by Quest, and Quest is empowered with improved user-interactivity capabilities provided by Android. We now describe the design of our system in further detail, beginning with the partitioning hypervisor.

2.1 Quest-V Partitioning Hypervisor

Figure 2 shows a diagram of the Quest-V partitioning hypervisor, configured for the IVI system. Quest-V is implemented for the x86

architecture and statically partitions the hardware resources of a physical platform amongst each guest OS. This resource assignment makes use of hardware-assisted virtualization techniques, which isolate guest operating systems into distinct sandboxes. Quest-V uses the Quest RTOS to initialize the system and allocate separate cores, memory regions, and I/O devices to the guest operating systems at boot time. Unlike traditional hypervisors such as Xen [2], there is no multiplexing of resources or device sharing. This allows each guest OS to manage its own set of hardware resources and enforce its own scheduling policies. As the hypervisor does not perform runtime resource management to share hardware amongst guests its trusted code base is minimized.

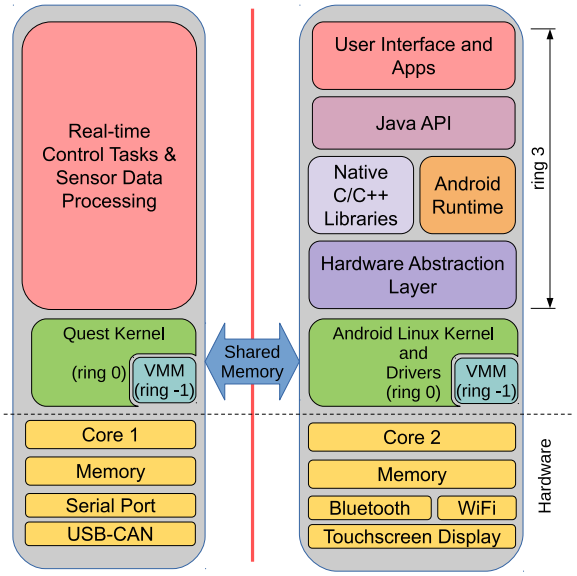


Figure 2: Design of the Quest-V Automotive System

Each sandbox in Quest-V includes a thin VMM layer which resides between the guest and the hardware. This layer effectively operates in ring -1 according to the x86 architectural definition (root, ring 0). The VMM layer is designed with a philosophy of minimal intervention with the guests. Interventions only occur when a guest OS attempts to access an out-of-range memory address, or attempts to execute some privileged instruction. In these cases, a trap is sent to, and handled by, the VMM. The guests in Quest-V operate in ring 0 (non-root, ring 0) and have been made aware that they are operating in a virtualized environment. To allow this awareness, each guest must be paravirtualized. Section 3 describes the paravirtualization of Android.

For the IVI system, Quest-V hosts the Quest RTOS and Android OS on physically separate cores. USB and serial ports are exclusively allocated to Quest, and the remaining I/O devices are allocated to Android. To receive information via USB, Android must communicate with Quest through an explicit shared memory channel. A set of remote procedure calls (RPCs) have been implemented to allow Quest to exchange information with other guests. A Linux kernel module runs in Android to facilitate these RPCs.

2.2 Advantages

The Quest-V architecture provides unique advantages to the IVI system, which are crucial to building a secure, safe, and predictable system.

2.2.1 Real-time I/O for Android Apps. In spite of being a non real-time OS, Android is able to leverage the real-time capabilities of Quest to interface with the timing critical components of a vehicle. I/O data is exchanged with Android applications without the need to use traditional socket-based interfaces such as Ethernet. Moreover, SCHED_DEADLINE scheduling within the Linux kernel of Android enables data exchanges with Quest to perform predictably. This approach removes interference from device interrupts that are managed in real-time by Quest. Details of this implementation are given in Section 4.

2.2.2 Isolated I/O memory space for sensitive devices. The USB-CAN interface is timing and safety-critical in automotive systems. The injection of a malicious packet onto the CAN bus has potentially devastating effects, dictating the need for secure access to this network. Although malicious packet insertions must be prevented, the IVI system must still be able to read from and write to this bus network to receive data and control the components of a vehicle such as the HVAC unit.

The isolated sandboxes in Quest-V prevent unauthorized access to critical I/O devices by guests such as Android. In the IVI system, the USB-CAN device is assigned to Quest and is inaccessible to Android. RPC requests from Android to Quest traverse secure shared memory channels enforced by extended page tables (EPTs) managed by the Quest-V hypervisor. Quest additionally filters requests to ensure any malfunction or vulnerability in Android will be contained within its sandbox.

2.2.3 Flexibility in System Software Development. If Android is used to interface directly with an automotive system's electronic control units (ECUs) through the CAN bus network, it ideally needs to be independently maintained by automotive manufacturers. However, these manufacturers may not have the expertise to develop and maintain a large and complex codebase like Android.

Using Quest to interface with the ECUs allows vehicle manufacturers to focus development and maintenance on a smaller, specialized RTOS that manages critical automotive subsystem components. The Quest RTOS is able to consolidate the real-time functional requirements traditionally managed by separate electronic control units (ECUs) within different process address spaces. Additionally, vehicle manufactures may not want to expose the details of their devices drivers for safety reasons, which would be required by a GPL licensed OS like Linux. Thus, it is beneficial for these manufacturers to develop in a separate OS in which they have the flexibility to apply their own safety and security policies. The only Android development that is needed is the inclusion of a Linux kernel module to handle the RPCs to and from Quest.

3 IMPLEMENTATION

The boot logic of Quest-V assigns a virtual machine monitor (VMM) to each sandbox, which is not accessible by the other sandboxes. As VMMs are not involved in runtime resource management their codebase fits within a 4KB page, although additional space is needed

for EPTs (e.g., up to 24KB for 4GB address spaces). Using Intel's VT-x features, each monitor establishes one or more virtual machine control structures (VMCSs) per sandbox. Each monitor then bootstraps its respective guest virtual machine and sends the sandbox configuration parameters required for paravirtualization to the respective guest kernels at boot time.

Through the configuration parameters of Quest-V, a tuple containing the base and limit of host physical memory (HPM) must be specified for each sandbox. Each sandbox monitor relocates its guest in HPM according to the specified base address. Extended page table (EPT) entries grant guests exclusive access to specific memory regions, while safeguarding the monitor logic. The monitors also identity-map a pool of shared memory pages for inter-sandbox communication. Section 4 provides further details on the design of the shared memory pool. Identity-mapped MMIO regions are used by the guest kernels to manage their assigned devices.

The Android kernel has been paravirtualized to compensate for the HPM base offset when a physical address is needed for DMA-enabled devices. This avoids implementing VMM drivers to support IOMMU technologies, such as Intel's VT-d for those devices. As the code size of each VMM is minimized, this helps enforce heightened security and simplifies formal verification.

Device partitioning is accomplished by interposing on ACPI configuration and PCI bus enumeration, thereby ensuring VMexits into a guest's corresponding VMM to check whether the device is blacklisted or not. Each guest's monitor will nullify their guest's access to a device or IRQ if they are not assigned to that guest.

Challenges. Quest-V attempts to eliminate as many invocations of the hypervisor as possible. However, during development there were several situations where VMexits arise through a guest's use of hardware-specific features. For example, VMexits were encountered through a guest running a graphics-accelerated OpenGL instrument cluster application. Such exits into the monitor occurred as a result of not granting the guests use of extended features related to advanced vector extensions. As a result, the VMCS control bitmap was updated to avoid exiting into the Quest-V monitor by all valid attempts by a guest to access the hardware to which it is assigned.

Physical address extensions (PAE) are also supported by Android on Quest-V. Although our monitor code is 32-bit, PAE allows Android to occupy more RAM (currently a 52-bit address space), which is beneficial for its memory-consuming Java applications.

4 REAL-TIME I/O FOR ANDROID

In Quest, every real-time task has a budget, C, determined by its worst-case execution time, and a period, T. Quest implements a static priority rate-monotonic scheduling (RMS) algorithm with a sporadic server, to guarantee a task or software thread receives at least C amount of execution time every T. Per the RMS policy, Quest assigns the highest priority level to the task with the smallest period.

Quest ensures that temporal guarantees of real-time tasks are not violated by interrupts from I/O devices. Quest handles an I/O interrupt with a *schedulable thread* at its *proper* priority level. In general, device interrupts are generated on behalf of tasks issuing I/O requests. Thus, an interrupt must be handled at the same priority level as the waiting task.

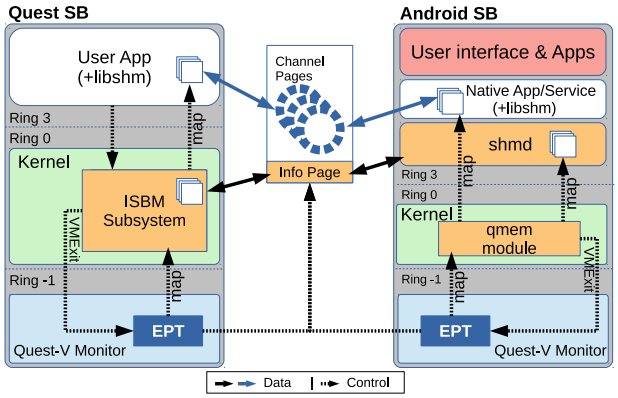


Figure 3: Inter-Sandbox Communication in Quest-V

Each interrupt handler is divided into two parts: a top-half and a bottom-half. In Quest, the top-half handler is only used to acknowledge an interrupt, and to determine which task is waiting for the I/O device. Quest then schedules the bottom-half handler as a separate thread at the same priority level as the waiting task.

As RMS determines a task's priority by its period, T, the bottom-half handler (BH) thread is assigned the same period as the task it serves. The budget of the BH thread is derived from the I/O device class and the waiting task's budget. All I/O handling occurs in the context of the BH thread.

4.1 Extending Real-time I/O to Android

A shared memory region is established between Quest and Android during the booting of the system. A Linux kernel module, `qmem`, mediates requests to map and unmap inter-sandbox communication channels in this shared region. A service, `shmd`, has been developed to manage the channels between sandboxes. In Quest, the inter-sandbox messaging (`ISBM`) subsystem implements the functionality in `qmem` and `shmd`. Once a channel is created between the two guests, applications in different sandboxes communicate without invoking system calls or VMexits. User applications use an API provided by `libshm`, to read from or write to these channels for asynchronous or synchronous communication. Figure 3 shows the tightly-coupled inter-sandbox communication mechanism. The listing below shows the API provided by `libshm` used to construct application-specific remote procedure calls spanning different sandboxes.

```
// VSHM management routines
int vshm_init (void);
void vshm_destroy (void);

// Asynchronous (4-slot) communication routines
int mk_vshm_async_ch(vshm_async_t* vac, u32 vshm_key,
    u32 elem_sz, u32 sandboxes, u32 flags);
void vshm_async_write(vshm_async_t* vac, void* item);
void vshm_async_read (vshm_async_t* vac, void* item);

// Synchronous (Ring Buffer) communication routines
int mk_vshm_sync_ch(vshm_sync_t* vcb, u32 vshm_key,
    u32 buf_sz, u32 elem_sz, u32 sandboxes, u32 flags);
int vshm_sync_insert(vshm_sync_t* vcb, void* item);
int vshm_sync_remove(vshm_sync_t* vcb, void* item);
```

Currently, the `libshm` API works for C-language applications. The API is being ported to a Java library with a Java Native Interface for the IVI and other Android applications. In addition, Android's

Binder IPC mechanism is being modified to provide real-time notifications to Android applications. The shared memory channel communication will be integrated with the Binder IPC mechanism to deliver real-time I/O from Quest to Android.

5 EVALUATION

We prototyped our system on an Up Squared Board, featuring an Intel Apollo Lake Pentium N4200 processor. A similar Apollo Lake Atom E3800 is used in Tesla's MCUs for its electric vehicles. The UP Squared's features are listed in Table 1. The board has compact size, low power, ample processing capacity and I/O capabilities befitting modern interactive automotive systems.

Table 1: Up Squared Board Specifications

Processor	Intel Pentium N4200 ($\leq 2.5GHz$)
RAM	2 GB
eMMC Storage	32 GB
Display and UI	HDMI and DisplayPort
CAN Connector	3 USB3.0 ports
Serial I/O	2 UART ports
Network	1 WiFi card and 2 Gigabyte Ethernet ports
Power	5V, 4A-6A
Dimension	85.6 mm × 90 mm

5.1 System and Apps Startup Time

The first set of experiments investigate whether the paravirtualization of Android has any significant effect on either Android or an IVI app startup time. The startup time is an important factor for the end-users of an IVI system. In our system, the IVI app automatically starts after Android is booted. Thus, the time to launch the IVI application is also measured.

The average over five boots is presented. The Quest-V paravirtualized Android took 23.7 seconds to boot. The IVI application launched in 59.2 seconds from the powering on of the platform. In comparison, a vanilla Android took 16.6 seconds to boot, and the IVI app starts in 49 seconds from power on. The extra time to boot the paravirtualized Android is the time the Quest RTOS takes to boot itself before executing the boot logic of Android. The overhead of the Android startup time is expected to be further reduced in this IVI system as debugging messages from Quest are still being sent to a serial port to ease development.

It is important to note that launching the IVI app takes roughly the same amount of time after Android is booted (35.5 seconds for the paravirtualized Android and 32.4 seconds for the vanilla Android). The minimal number of VMexits in the paravirtualized Android is the reason behind the similarity between the performance of the vanilla and paravirtualized Android for the IVI app startup. The performance of the Android application after boot time will be measured in future experiments.

5.2 Real-time I/O Performance

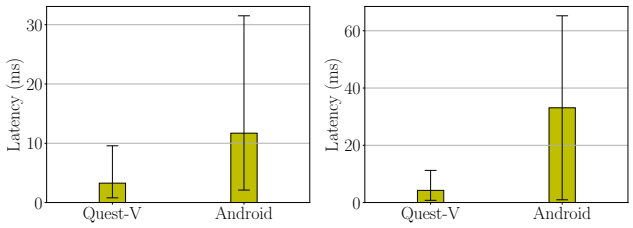
Aside from startup times, we also show how Quest-V is able to mediate real-time I/O required by ADAS services. Part of these ADAS services, including the user interface to control features such as lane departure warnings and vehicle collision avoidance, are implemented in Android. A five-channel Kvaser USBCan Pro 5xHS CAN bus interface is connected to our Up Squared via USB 3.0. A vehicle's CAN traffic, such as chassis and powertrain messaging, is simulated by connecting Woodward MotoHawk ECM5634-70 ECUs to channels 1-3. For performance measurements, channels 4-5 were replaced with Arduino UNOs, but only CAN4 measurements are reported due to space constraints.

In the experiment, the latency and throughput of CAN messages being read from CAN4, processed by an application, and then finally written back to the same channel are measured. A USB-CAN driver, called `mhydra`, facilitates the Kvaser USBCan scatter-gather functionality. Two processes, `CanRead` and `CanWrite` read from and write to I/O threads. A `ProcData` application representative of an ADAS service then processes the CAN data. Each of these tasks have specific budgets and periods. The chain of tasks along with their budgets and periods (ms, ms) shown inside parentheses are arranged as follows: `mhydra_rx` (0.2, 1) → `CanRead` (0.1, 2) → `ProcData` (0.2, 2) → `CanWrite` (0.1, 2) → `mhydra_tx (0.2, 1)`.

We compare the real-time I/O performance of a standalone Android system to a Quest-V system hosting both Android and Quest. In the latter case, an xHCI USB bottom half handler thread executes in Quest to pre- and post-process data before and after it traverses the above chain of tasks. In the standalone Android system, the xHCI handler runs within its Linux kernel. This handler processes USB operations resulting from host controller interrupts, including the wakeup of other threads waiting on completion of I/O transactions. USB bottom half processing is limited to 10% CPU utilization within a service period equivalent to that of the *mhydra* thread, according to the description in Section 4. The aim is to guarantee end-to-end I/O latencies of less than 10ms, required by modern vehicles to adapt wheel torque and maneuver quickly enough according to ADAS functionality.

For Quest-V, we focus on communication between Android's Linux kernel and Quest. This is because once an ADAS service has been activated by a user it no longer needs to communicate in real-time with the user interface. For a standalone Android system, we focus on real-time I/O between the USB-CAN interface and its Linux kernel.

For the Quest-V setup, only the `ProcData` task runs in Android, while all other tasks run in Quest. For the standalone Android system, `ProcData` is run on one core and all others are run on another core. In both cases, background tasks compete for the same CPU core used by `ProcData`.



(a) Asynchronous **(b) Synchronous**

Figure 4: Average I/O Latency (min-max as error bars)

Figure 4 shows that the average latency for the CAN messages are much lower in Quest-V than in standalone Android. With asynchronous communication, a four-slot [10] buffer is used as part of Quest-V's shared memory communication between Android and

Quest. This is based on Simpson's protocol that guarantees freshness and integrity of data but not necessarily loss-free communication. This is appropriate for sensor data processing, where recent readings are more important than stale values. In contrast, synchronous communication requires loss-free data exchanges, which are critical for control message exchanges. Quest-V uses a shared memory ring-buffer to pass messages between the `ProcData` task and Quest-based CAN tasks.

The error bars in the figure represent the min and max latency, which are also much higher in standalone Android than in Quest-V. Android is affected by the unpredictable behavior of interrupts, which increases the latency of the CAN messages. This is because its Linux kernel does not have time-budgeted, priority-aware bottom-half handlers.

In contrast, the Quest RTOS within Quest-V handles I/O by running its bottom half handler threads at the proper priority levels. Then, the data is passed to the Linux kernel in Android through a secure shared memory channel, which is mapped directly to the `ProcData` task. Therefore, communication between Quest and Android incurs minimal overhead. Thus, the latency in the Quest-V is not only smaller but also more predictable than in standalone Android for both asynchronous and synchronous I/O.

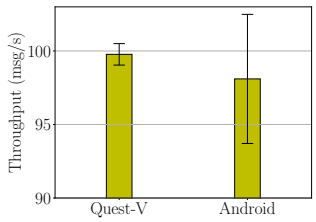


Figure 5: Synchronous I/O Throughput (stddev as error bars)

As synchronous communication potentially blocks a message until it is delivered, the throughput in Figure 5 is similar for both Quest-V and standalone Android. However, the error bars depicting the standard deviation are much smaller for Quest-V, which demonstrates that throughput is more predictable. Additional details about these experiments are available in our technical report [5].

6 RELATED WORK

Several research groups have studied the use of Android for IVI systems [6, 7]. GENIVI [4] and other alliances between automotive companies are also developing Automotive Grade Linux [12] and AUTOSAR-compliant OSs [1] for modern vehicles. However, they do not address the timing, security, and development issues of an integrated single-machine solution. Their approach requires the real-time tasks and the IVI components to run on separate machines. Android has also been redesigned to provide real-time guarantees to different software components [14, 15]. Although that work makes a number of important contributions, including making the Binder IPC priority-aware, changing most of the architecture of Android has little chance of wider adoption. Moreover, real-time I/O was not addressed in these works. In contrast, the paravirtualization of Android in our work required just 126 lines of changes, and the real-time I/O capabilities of Android are being extended through the Quest-V system design. Moreover, our design is deployed in a single-board computer while maintaining the necessary space-time partitioning requirement.

7 CONCLUSIONS AND FUTURE WORK

This paper presents a new design for interactive service integration in automotive systems based on the Quest-V partitioning hypervisor. With just 126 lines of changes, a paravirtualized Android 8.1 has been developed for Quest-V, in which a professional IVI application is run, along with prototype services for testing real-time ADAS functionality. The Quest-V system design allows Android to leverage the timing predictability of Quest to support real-time I/O for its applications. This system design isolates timing and safety-critical devices from those used by Android. Similarly, application developers leverage convenient APIs provided by Android while automotive developers focus on code deployment in a smaller, lighter-weight real-time environment such as Quest. Experiments demonstrate that paravirtualizing Android has non-intrusive startup overhead and benefits from the real-time I/O capabilities of Quest. We are exploring more real-time I/O and notification capabilities in Android for complete deployment of the interactive automotive system in a working high-performance electric vehicle.

8 ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF) under Grant # 1527050. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

REFERENCES

[1] AUTOSAR. 2019. AUTomotive Open System ARchitecture. http://www.autosar.org.
[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization. In *ACM SIGOPS OSR*.
[3] M Danish, Y Li, and R West. 2011. Virtual-CPU Scheduling in the Quest Operating System. In *2011 17th IEEE RTAS*. IEEE, 169–179.
[4] GENIVI. 2019. GENIVI Alliance. https://www.genivi.org/. Last Accessed: Oct 2019.
[5] A Golchin, S Sinha, and R West. 2019. Boomerang: Real-Time I/O Meets Legacy Systems. *arXiv preprint arXiv:1908.06807* (2019).
[6] B. Kovacevic, M. Kovacevic, T. Maruna, and D. Rapic. 2016. Android4Auto: A Proposal for Integration of Android In Vehicle Infotainment Systems. In *2016 IEEE ICCE*.
[7] G. Macario, M. Torchiano, and M. Violante. 2009. An In-vehicle Infotainment Software Architecture based on Google Android. In *2009 IEEE International Symposium on Industrial Embedded Systems*.
[8] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking Tesla from Wireless to CAN Bus. *Briefing, Black Hat USA* (2017), 1–16.
[9] Georg Niedrist. 2016. Deterministic Architecture and Middleware for Domain Control Units and Simplified Integration Process Applied to ADAS. https://www.tttech.com/technologies/adas.
[10] H.R. Simpson. 1990. Four-slot Fully Asynchronous Communication Mechanism. *IEEE Computers and Digital Techniques* 137 (January 1990), 17–30.
[11] Tesla. 2019. Linux. https://github.com/teslamotors/linux.
[12] The Linux Foundation. 2019. Automotive Grade Linux. https://www.automotivelinux.org/.
[13] R. West, Y. Li, E. Missimer, and M. Danish. 2016. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Transactions on Computer Systems (TOCS)* (2016).
[14] Y Yan, S Cosgrove, V Anand, A Kulkarni, S H Konduri, S Y Ko, and L Ziarek. 2014. Real-time Android with RTDroid. In *MobiSys*. ACM.
[15] Y. Yan, K. Dantu, S Y Ko, J Vitek, and L Ziarek. 2017. Making Android Run on Time. In *2017 IEEE RTAS*. IEEE.