Towards an Integrated Vehicle Management System in DriveOS

SOHAM SINHA and RICHARD WEST, Department of Computer Science, Boston University, USA and Drako Motors Inc., USA

Modern automotive systems feature dozens of electronic control units (ECUs) for chassis, body and powertrain functions. These systems are costly and inflexible to upgrade, requiring ever increasing numbers of ECUs to support new features such as advanced driver assistance (ADAS), autonomous technologies, and infotainment. To counter these challenges, we propose DriveOS, a safe, secure, extensible, and timing-predictable system for modern vehicle management in a centralized platform. DriveOS is based on a separation kernel, where timing and safety-critical ECU functions are implemented in a real-time OS (RTOS) alongside non-critical software in Linux or Android. The system enforces the separation, or partitioning, of both software and hardware among different OSes.

DriveOS runs on a relatively low-cost embedded PC-class platform, supporting multiple cores and hardware virtualization capabilities. Instrument cluster, in-vehicle infotainment and advanced driver assistance system services are implemented in a Yocto Linux guest, which communicates with critical real-time services via secure shared memory. The RTOS manages a real-time controller area network (CAN) interface that is inaccessible to Linux services except via well-defined and legitimate communication channels. In this work, we integrate three Qt-based services written for Yocto Linux, running in parallel with a real-time longitudinal controller task and multiple CAN bus concentrators, for vehicular sensor data processing and actuation. We demonstrate the benefits and performance of DriveOS with a hardware-in-the-loop CARLA simulation using a real car dataset.

CCS Concepts: • **Computer systems organization → Real-time systems; Real-time operating systems**;

Additional Key Words and Phrases: Automotive Systems; Safety-Criticality; Partitioning Hypervisor;

ACM Reference Format:
Soham Sinha and Richard West. 2021. Towards an Integrated Vehicle Management System in DriveOS. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 82 (September 2021), 24 pages. https://doi.org/10.1145/3477013

1 INTRODUCTION

Modern vehicles each have as many as 150 electronic control units (ECUs), to manage chassis, body and powertrain functions [19, 63]. The number of ECUs in a single vehicle is expected to rise [44], as electronics play an increasing role in support of new features, such as "virtual cockpit" instrument clusters (ICs), in-vehicle infotainment (IVI), advanced driver assistance systems (ADAS), and autonomous operation. Each ECU has its own microcontroller and logic to interface to a control network such as a CAN bus [47], as part of a vehicle's automotive system. As the scale

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.

Authors' address: Soham Sinha, soham1@bu.edu; Richard West, richwest@bu.edu, Department of Computer Science, Boston University, Boston, MA, USA, Drako Motors Inc. USA.

1539-9087/2021/09-ART82 $15.00
https://doi.org/10.1145/3477013

and complexity of automotive systems increases, hardware costs, wiring and packaging become prohibitive. Short of reflashing or adding new ECUs, it is difficult, if not impossible, to upgrade the capabilities of a vehicle already in use. Moreover, simple ECU hardware lacks modern-day computing security guarantees [33], and functional errors are hard to detect and fix in a cumbersome automotive control network [18].

An alternative to current vehicular systems is to replace ECU logic with software-defined functions. These functions are easy to upgrade, replace and extend with new capabilities over time. Software components could then be deployed on a centralized computing platform, omitting the need for complex vehicular networks. Instead, simpler networks connect sensors and actuators equipped with simple transceivers to the central computer, which hosts an appropriate bus interface.

Replacing ECUs with multiple software-defined functions requires a suitable operating system. While Linux is used by automotive companies such as Tesla [60], BMW [39], and Toyota [16], it lacks the necessary timing and safety requirements for correct vehicle operation in all conditions. Real-time components must be executed according to strict timing guarantees. Safety-critical software components must be isolated from less-critical services, according to different safety integrity levels [29]. Without significant modifications and subsequent formal method verification, Linux's use in automotive systems is limited to non-critical functions. As evidence, security attacks have been observed on Tesla's Linux software stack, which remotely gain control over the vehicle's CAN network [46].

In this paper, we describe a new system that securely and predictably consolidates software-defined functions into an integrated vehicle management system. Towards this goal, we introduce DriveOS, which supports the co-existence of an RTOS and one or more legacy OSes such as Linux or Android on a single computing platform. DriveOS is based on a real-time separation kernel [54, 62], which maps guest OSes to secure sandbox domains that have direct access to partitioned machine physical resources. DriveOS behaves like a partitioning hypervisor [14, 50, 62], whereby each guest manages its own CPU cores, physical memory and I/O devices without runtime involvement of a virtual machine monitor. DriveOS is bootstrapped by our own in-house real-time OS, Quest, which establishes secure communication channels with other guests running Linux and/or Android. Timing-sensitive services are deployed as real-time tasks in a Quest sandbox. Non-critical, legacy and library-dependent software such as IVI, IC, and machine learning-based ADAS services are deployed in Linux and Android sandboxes.

Unlike general purpose operating systems, DriveOS supports real-time and predictable I/O, similar to what is available in typical microcontrollers. One or more USB-CAN interfaces [34] connect a DriveOS central computer, acting as a CAN *concentrator*, to a network of sensors and actuators. A real-time USB 3.0 protocol ensures predictable data movement between the DriveOS host and peripheral devices connected to each CAN bus [23].

Figure 1 shows a high-level overview of DriveOS on a DX1100 Workstation [11]. This is an industrial computing platform, being tested within our partner electric car company for integrated vehicle management. Although DriveOS supports multiple guests such as Ubuntu and Android, we use Yocto Linux in this paper. Our Yocto Linux sandbox features: (1) an IC application that displays a graphical speedometer, battery meter and other indicator readings, (2) an IVI application that provides heating and ventilation (HVAC) controls, navigation, and smartphone integration, and (3) ADAS services for adaptive cruise control and autonomous driving. The Quest sandbox implements a real-time CAN gateway service to facilitate sensor data processing, control, and secure communication with separate IC, IVI, and ADAS services. *Real-time service tasks* such as an ADAS longitudinal controller are also executed in Quest.

Contributions. In this paper, we: (1) describe the separation kernel that forms the basis of DriveOS, (2) introduce a low-overhead and secure inter-sandbox communication framework, named `shmcomm`,

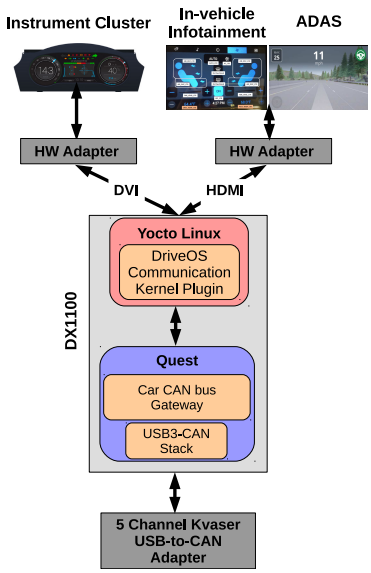


Fig. 1. Integrated Vehicle Management in DriveOS

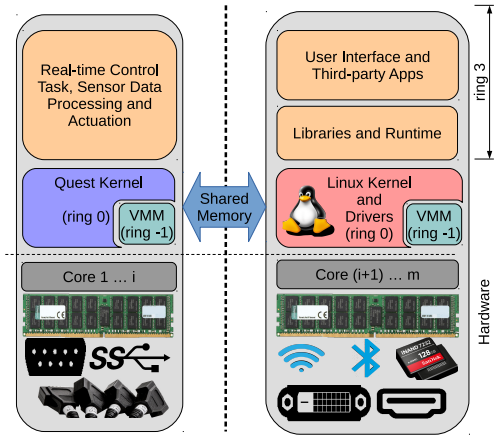


Fig. 2. Resource Partitioning in DriveOS

which allows both synchronous and asynchronous message-passing between guest sandboxes, and (3) integrate IC, IVI and ADAS services in DriveOS to show the feasibility of our approach.

We compare an implementation of DriveOS, which is actively being developed for a production-grade electric car, with an alternative Linux system that is currently used in the automotive industry. Using a hardware-in-the-loop (HIL) CARLA driving simulation [17], and real car dataset collected from Laguna Seca raceway in California, we show that an optimized Linux supporting real-time tasks is unable to achieve the end-to-end delay and throughput guarantees provided by DriveOS, when tasks process and exchange data with a CAN bus network. At the same time, DriveOS provides the added security and isolation between software components of different criticality levels.

The next section provides motivation for our approach, which is followed in Section 3 by a description of the DriveOS design. Section 4 outlines the implementation of the DriveOS partitioning hypervisor, used to support secure and predictable separation of different application and kernel components. The inter-sandbox communication framework is then explained in Section 5, followed by a description of the DriveOS applications in Section 6. A system evaluation is presented in Section 7. Related work is discussed in Section 8, with conclusions and future work in Section 9.

2 MOTIVATION

Modern vehicles support 10s to 100s of millions of lines of code [8]. To counter the growing complexity, chipmakers such as Intel, and analytics firms like McKinsey have called for a centralized vehicle management system to reimagine modern cars from a hardware-driven mechanical machine to a software-driven electronic device [9, 28].

Taking inspiration from AUTOSAR's requirements about future car operating systems [3], we envision a centralized system built on a low-cost, standardized industrial PC [47]. PC-class hardware provides multiple high-performance processing cores, gigabytes of memory, hardware virtualization, potential integration with time-triggered Ethernet or Time-Sensitive Networking (TSN) [12, 32], and support for hardware accelerators for use in machine learning. In comparison,

ECUs typically feature small flash memories and megahertz-speed microcontrollers, with limited processing capabilities for next-generation automobiles.

A centralized PC-class platform needs a suitable operating system. OSes such as Linux and Android are insufficient on their own to provide the necessary spatial separation of highly-critical services, with high consequences of failure, from those of low-criticality. Similarly, these OSes lack the real-time capabilities necessary to meet timing-critical service guarantees. For this reason, DriveOS proposes to securely co-host an RTOS with a legacy system such as Linux using a partitioning hypervisor. First-class shared memory channels between co-hosted guest OSes provide real-time, low-latency, high bandwidth, and secure inter-sandbox communication. This enables a mutually beneficial *symbiosis* between the RTOS and each legacy system: legacy systems inherit real-time capabilities without modification, while the RTOS gains access to pre-existing libraries, device drivers, and services.

Standards such as ISO 26262-3 [29] define multiple automotive software integrity levels (ASIL-A to D). Our approach enables services of different criticality, or integrity, levels to be assigned to different sandboxed guests. These sandboxes are spatially and temporally isolated as envisioned by partitioning systems for future vehicles [35]. Because of the relatively small RTOS codebase, DriveOS is amenable to formal verification to ensure functional and timing correctness [26, 41].

The potential single point of failure of a single hardware platform is addressed by introducing backup hardware, albeit with fewer replicas than ECUs found in current vehicles. Memory bit errors are addressed by replicating software functions using techniques such as Triple-Modular Redundancy [42], or N-versioning [4]. Hypervisor-based fault tolerance ensures one sandboxed guest is able to recover from failure [6]. These techniques serve to validate our approach, but are not the main focus of this paper.

3 DRIVEOS DESIGN

DriveOS uses Linux as the basis for next-generation IC, IVI applications and ADAS user-interface control, with real-time features handled by our Quest OS. 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 real-time services, the interface to configure ADAS settings will be exposed to Linux. DriveOS hosts both Linux and Quest on a single machine, supporting communication between both guest OSes via secure shared memory channels. Thus, Linux is empowered with real-time capabilities afforded by Quest, and Quest is empowered with improved user-interactivity capabilities (including graphics and touchscreen control) provided by Yocto Linux. We now describe the design of our system in further detail, beginning with the partitioning hypervisor.

3.1 DriveOS Partitioning Hypervisor

Figure 2 shows a diagram of the DriveOS partitioning hypervisor, configured for a vehicle management system. DriveOS 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 to isolate guest operating systems in distinct sandboxes.

At boot-time, DriveOS begins by executing Quest as though it were a standalone system. Quest contains the hypervisor logic to partition hardware resources among separate guest sandboxes, according to a boot-time configuration. One instance of Quest is replicated in non-overlapping physical memory for each guest sandbox, and then each guest is launched. Depending on the DriveOS configuration, one instance of Quest will act as a bootloader for Linux or Android, which

becomes the active OS in the corresponding sandbox. For a system with at least two sandboxes, it is then possible to have Quest running in a guest domain that is isolated from another running an OS such as Linux. As Quest contains the hypervisor, or virtual machine monitor (VMM) code, to bootstrap a series of guests, each sandbox subsequently contains "root mode" (ring -1) hypervisor code. This is a heightened privilege level address space over traditional kernel protection domains that run at "ring 0" on x86 systems. Similarly, each guest actively runs in "non-root mode" but is granted direct access to hardware resources that it can access without invoking the VMM.

Each guest's VMM code is only needed to execute a minimal set of privileged machine instructions that cause *VMExits* (from guest to hypervisor control-flow), and to establish new inter-sandbox shared memory channels between guests. The number of guest sandboxes and the mapping of hardware resources (CPU cores, physical memory ranges, and subsets of I/O devices) to guests is established by static system configuration information. As resources are partitioned rather than shared as in traditional hypervisors such as Xen [5], there is no duplication of potentially conflicting resource management policies between a guest OS and hypervisor. The lack of runtime resource management functionality in the DriveOS hypervisor means that ring -1 code has a text size of less than 4KB.

Each guest kernel in DriveOS operates in (non-root) ring 0. With the exception of Quest, all other guest OSes are paravirtualized to operate correctly within their virtualized domains. We have paravirtualized Yocto Linux, Ubuntu, and Android for DriveOS with less than 150 lines of code changes in the Linux kernel. These changes are mostly to handle direct memory access (DMA) requests, where guest and machine physical memory addresses differ, and the processor does not provide IO address translation capabilities (e.g., IOMMU support such as VT-d on certain Intel x86 processors). For the purpose of this work, we use only the Yocto Linux distribution for guests that complement Quest.

In this paper, DriveOS hosts Quest and Linux on two separate cores in a DX1100 machine [11]. USB, USB-CAN and serial ports are exclusively allocated to Quest, and the remaining I/O devices are allocated to Linux. For Linux to receive information via USB, it must communicate with Quest through an explicit shared memory channel. Details of the inter-sandbox communication mechanism are provided in Section 5.

3.2 Real-time Task as a Service

DriveOS temporally and spatially isolates Quest and Linux in the same physical machine. This means a guest kernel is unable to interfere with the runtime progress or memory state of another guest. Quest schedules its tasks with its own real-time scheduling policy, independent of the co-existent Linux guest. In DriveOS design, Quest provides real-time task services to other sandboxes via inter-sandbox communication channels. Applications in other sandboxes subscribe to a real-time service via a specific channel. For example, a feed-forward PI controller for a car's adaptive cruise control is implemented in DriveOS as a real-time service in Quest. Linux-based ADAS functions use the real-time PI controller service via a synchronous shared-memory channel.

A real-time service task has the following properties: maximum runtime (`C`), frequency of execution (or period `T`), and a number of communication channels. Quest schedules the real-time tasks with the rate-monotonic scheduling (RMS) algorithm [40]. The service task is given at least `C` time-units in every `T` time-units.

In addition, critical I/O tasks are also implemented in Quest as real-time service tasks. For different classes of I/O devices and I/O-waiting threads, unique `C` and `T` values are computed at runtime to handle interrupts at the correct priority. This is covered further in Section 4.1. Other sandboxes use this type of critical I/O device-handling task to implement a *real-time virtual device interface*. For example, DriveOS has a real-time CAN Gateway Service to handle USB-CAN devices

in a fast and predictable way. In the future, we plan to consolidate dozens of car ECUs into real-time services in multiple Quest sandboxes and make them available to applications in other sandboxes.

3.3 Advantages of the DriveOS Design

The DriveOS architecture brings unique advantages in vehicle management, which are crucial to building a secure, predictable and extensible automotive system.

3.3.1 ***Real-time Task and I/O Services for Linux.*** In spite of being a non real-time OS, Linux 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 Linux applications without the need to use traditional socket-based interfaces such as Ethernet. Moreover, `SCHED_DEADLINE` scheduling of Linux enables data exchanges with Quest to perform in a sufficiently predictable way. This approach removes interference from device interrupts that are managed in real-time by Quest.

3.3.2 ***Isolated I/O 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 [33], dictating the need for secure access to this network. Although malicious packet insertions must be prevented, the vehicle management 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, and engine controller.

The isolated sandboxes in DriveOS prevent unauthorized access to critical I/O devices by guests such as Linux. In our vehicle management system, the USB-CAN device is assigned to Quest and is inaccessible to Linux. CAN data is accessible to Linux only via secure shared memory channels from Quest. Quest additionally filters requests to ensure any malfunction or vulnerability in Linux will be contained within its sandbox.

3.3.3 ***Flexibility in Automotive Software Development.*** If Linux is used to interface directly with an automotive system's critical functionality, 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 Linux.

It is potentially easier for automotive engineers to develop and maintain a simpler codebase such as Quest, which provides timing and safety-critical services to a vehicle. Quest is able to consolidate the real-time functional requirements traditionally managed by separate ECUs within different process address spaces. We have also started the development of a set of toolbox functions for Matlab that is heavily used by the automotive industry, to produce target code for Quest. These functions not only gain the benefits of a real-time OS but are guaranteed isolation from a Linux address space, which is potentially open to third-party, less secure software. Thus, it is beneficial for manufacturers to develop in a separate OS in which they have the flexibility to apply their own safety and security policies. The only Linux development that is needed is the inclusion of a Linux kernel module to facilitate inter-sandbox communication to and from Quest.

4 DRIVEOS IMPLEMENTATION

As stated earlier, DriveOS uses Quest to bootstrap a series of sandboxed guests. Once the first instance of Quest is loaded into memory, it replicates itself for each sandbox that is specified in a static configuration. Hypervisor code extensions to standalone Quest enable each OS replica to be launched into non-root mode, using a `VMLAUNCH` machine instruction. Depending on the system configuration, a guest may continue to run Quest or may choose to bootstrap an alternate system such as Linux.

Each sandbox replicates the VMM logic to establish separate guest domains, but as previously observed this code is only required at runtime to handle instructions that cause VMExits and to establish secure communication channels between guests. Secure communication channels require mappings of guest to machine physical memory using extended page tables (EPTs). Although a VMM's text segment that stores instructions fits within a 4KB page of memory, additional data space is needed for EPTs. For example, assuming a 4GB address space for a single guest requires 24KB memory for the corresponding EPT. DriveOS currently uses Intel VT-x technology to allow a sandbox monitor to create one of more *virtual machine control structures* (VMCSs) per sandbox. One VMCS is created for each CPU core assigned to the sandbox, and stores guest and host state information, virtual machine execution, exit and entry control information, as well as the causes of VMExits. `VMLAUNCH` instructions refer to the active VMCS for the corresponding processor and initialize the corresponding guest state. If the guest is to replace Quest with another OS, configuration parameters required for paravirtualization are sent to the respective kernel at boot time.

Memory Partitioning. In the configuration parameters of DriveOS, a tuple containing the base and limit of host physical memory (HPM) must be specified for every sandbox. Each sandbox monitor relocates its guest in HPM according to the specified base address. EPT entries grant guests exclusive access to specific memory regions, while safeguarding the monitor logic. VMMs also manage the guest physical to host physical memory mapping of shared memory regions for the inter-sandbox communication, as described in Section 5.

Device Partitioning. 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 block their guest's access to a device or IRQ if they are not assigned to that guest. Identity-mapped MMIO regions are used by guest kernels to manage their assigned I/O devices.

The Yocto Linux 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.

Physical Address Extension. Physical address extensions (PAE) are also supported in Yocto Linux guests of DriveOS. Although DriveOS's VMM code is 32-bit, PAE allows Linux to occupy more RAM (currently a 52-bit address space), which is beneficial for its memory-consuming interactive and machine learning applications. 64-bit guest support is intended in future implementations but this incurs additional overheads due to increased levels of address translation when resolving guest virtual to host physical addresses.

AVX Extensions for OpenGL Applications. DriveOS attempts to eliminate as many invocations of the hypervisor as possible. However, a graphics-accelerated OpenGL Instrument Cluster application uses advanced vector extensions (AVX) instructions which cause a VMExit into the monitor. We update the VMCS execution control bitmap to avoid exiting into the monitor, whenever an allowed guest uses such hardware features.

4.1 Real-time I/O in Quest

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. Using 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 [15, 64]. 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.

Each interrupt handler is divided into two parts: a top- and bottom-half. In Quest, the top-half handler simply acknowledges an interrupt, and determines 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. As with the Quest RTOS [15], the budget of the BH thread is derived from the I/O device class and the waiting task's period. Each device class has a utilization percentage value, which is multiplied by the waiting task's period, to derive the budget of the BH thread. For example, the utilization percentage of USB-CAN devices in DriveOS is 10%. If a task with a period of 1 ms waits for a USB-CAN read, then the USB-CAN BH will receive 0.1 ms budget in every 1 ms to read messages from the device. All I/O handling occurs in the context of the BH thread, not in the I/O-waiting task's context. This ensures BH processing is time-budgeted separately from task execution.

5 INTER-SANDBOX COMMUNICATION

A key component of DriveOS is the secure and predictable communication between different guest domains. Address spaces in two different guests use the `shmcomm` inter-sandbox communication mechanism to interact with each other. Inter-sandbox channels are used to create communicating task pipelines [45, 49]. Figure 3 shows the shmcomm control- and data-flow in DriveOS.

A kernel (shmcomm) module mediates requests to map and unmap shmcomm communication channels in an 8MB region shared between the guests. A kernel module within each guest sends requests to the shmcomm manager in its local VMM to configure the channels. The shmcomm manager does not expose the host physical addresses (HPAs) of shmcomm channels to a guest. Instead, it establishes EPT mappings of guest physical addresses (GPAs) to HPAs, for the memory pages used for communication channels. The manager uses a secure Info Page to store all the metadata information of the channels in DriveOS. The Info Page is not mapped to any guest kernels but is instead accessed via a lock held by the VMMs of each sandbox.

Each VMM shmcomm manager resides in ring -1 in DriveOS, and requires less than 500 lines of code, thereby keeping the trusted codebase of the most privileged protection domain small. In addition, the shmcomm manager handles four specific hypercalls to service requests from guest kernels: (1) creating a channel, (2) connecting a channel, (3) getting channel metadata, and (4) destroying a channel. User-level (ring 3) address spaces cannot directly interact with the VMM, unless granted permission by guest kernels.

Once a channel is created by the shmcomm manager, applications in different sandboxes communicate without invoking system calls or VMExits. Applications use a POSIX I/O-like API provided by `libshmcomm`, to read from and write to these channels for asynchronous and synchronous communication. The channel communication protocols are entirely implemented in userspace. It is possible to extend the library with new communication protocols without modifying the kernel modules or the VMM code. Figure 4 shows the APIs provided by `libshmcomm`.

5.1 `shmcomm` Operations

5.1.1 Creating and Connecting a Channel. Channels are created and connected for sending and receiving messages using `shmcomm_open_send` and `shmcomm_open_receive` functions, respectively.

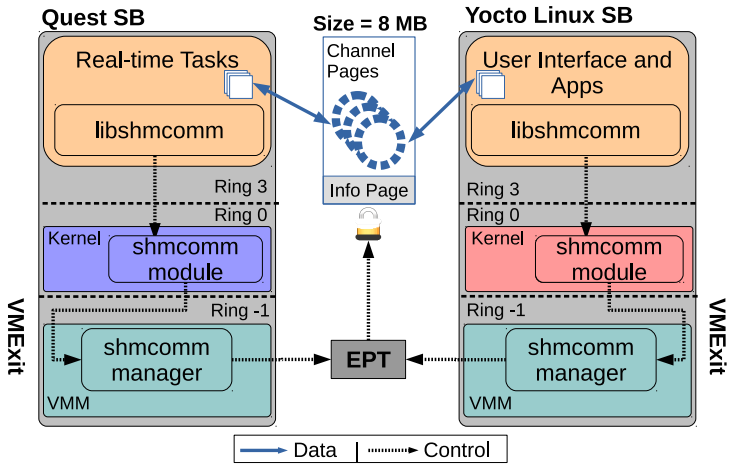


Fig. 3. shmcomm Control and Data Flow in DriveOS

```
int shmcomm_open_send (unsigned int vshm_key, unsigned int flags,
marshall_func_t marshall_function, size_t packet_size, int buffer_len);

int shmcomm_open_receive (unsigned int vshm_key, unsigned int flags,
unmarshall_func_t unmarshall_function, size_t packet_size, int buffer_len);

int shmcomm_write (int fd, void* buf, size_t nbytes);

int shmcomm_read (int fd, void* buf, size_t nbytes);

unsigned int get_vshm_key (int fd);

channel_transfer_type_t get_channel_transfer_type (int fd);

int shmcomm_destroy (int fd);

int shmcomm_close (int fd);
```

Fig. 4. The `libshmcomm` Library APIs

A unique channel key, `vshm_key`, is used to create a channel between separate guest address spaces. The `flags` argument supports the creation of a new channel (`SHMCOMM_ CREATE_CH`) or connection to an existing channel (`SHMCOMM_CONNECT_CH`). The channel type, which is either synchronous (`SHMCOMM_SYNC_CH`) or asynchronous (`SHMCOMM_ASYNC_CH`), is also specified in `flags`.

The shmcomm protocol supports marshalling and unmarshalling before sending and after receiving messages, using specific callback functions. Data marshalling is provided as a convenience because CAN messages often need to be encoded and decoded (e.g., using DBC files). Finally, the size of each message (or packet) and the length of the shared memory buffer are needed to create a channel. All this information, except the marshalling callbacks, are sent to the shmcomm manager via a system call to the guest kernel and a hypercall to the VMM. The shmcomm manager stores the channel information in the secure Info Page. Connecting to an existing channel does not need any specific information such as `packet_size` or `buffer_len`, as shmcomm supplies this information to the applications from its Info Page. The shmcomm kernel module in a guest sandbox sends a GPA to the shmcomm manager, which is mapped to a private channel HPA. Both `shmcomm_open_send`

and shmcomm_open_receive return an integer file descriptor that is used to further interact with the channel.

5.1.2 Closing and Destroying a Channel. Closing a channel with shmcomm_close frees the userspace data-structures for the channel. shmcomm_destroy frees the channel memory from the shared memory region and also removes the channel entry from the Info Page. shmcomm_destroy implicitly closes a channel.

5.1.3 Querying Channel Metadata. A channel's vshm_key and type of data-transfer (synchronous or asynchronous) are queried using get_vshm_ key and get_channel_transfer_type, respectively. An address space could query channel metadata before exchanging messages with a channel.

5.1.4 Reading from and Writing to a Channel. Reading from and writing to a channel occur entirely in guest userspace without any system call or VMExit overheads. The channel memory is mapped to userspace in the local (guest process) page table by the shmcomm kernel modules in Linux and Quest. This reduces communication overheads in DriveOS once channels are established. The shmcomm interface uses a FIFO ring buffer for synchronous communication and Simpson's four-slot algorithm [55] for wait-free asynchronous message passing. The latter is useful when loss tolerant data transfers between guests are acceptable, as long as the most recent data is exchanged (e.g., for sensor data).

5.2 Real-time Virtual Device I/O for Linux

Communication pipelines created with our libshmcomm library extend real-time I/O in Quest to address spaces in Linux. This enables Linux tasks to perform time-bounded functions such as obstacle detection and avoidance (useful for ADAS), using real-time sensor data processing and actuation tasks in Quest. Our communication APIs have bindings for C, C++, Java and Python in Linux and Android. DriveOS grants permission for IC and IVI tasks in C++, and OpenPilot ADAS tasks in C++ and Python, to interact with USB-CAN I/O services in Quest.

5.3 Example

Program 1 shows the C code of a sender in Quest that creates a synchronous shmcomm channel to send messages of struct packet type. The channel's ID is 101 and ring buffer length is 10. Program 2 shows the receiver-side code in Linux, which connects to channel ID=101 and reads from the channel into local variable p_rec.

Program 1. Sender in Quest

```
struct packet {int X; int Y};

int write_fd = shmcomm_open_send(101,
    SHMCOMM_CREATE_CH | SHMCOMM_SYNC_CH, NULL,
    sizeof(struct packet), 10);

struct packet p_snd = {.X = 10, .Y = 20};

shmcomm_write(write_fd, &p_snd,
    sizeof(struct packet));

shmcomm_close(write_fd);
```

Program 2. Receiver in Linux

```
struct packet {int X; int Y};

int read_fd = shmcomm_open_receive(101,
    SHMCOMM_CONNECT_CH, NULL,
    sizeof(struct packet), 10);

struct packet p_rec;

shmcomm_read(read_fd, &p_rec,
    sizeof(struct packet));

shmcomm_close(read_fd);
```

6 DRIVEOS APPLICATIONS

In this section, we describe the integration of three applications in DriveOS: Instrument Cluster, In-vehicle Infotainment and Advanced Driver Assistance. We also explain how these applications utilize the Real-time Task as a Service model in DriveOS, to guarantee end-to-end throughput and delay requirements for data processing. Since these applications are tested in our hardware-in-the-loop (HIL) simulation infrastructure, we first explain the HIL simulation setup to help describe the integration procedure. The HIL setup is also used in Section 7 to evaluate DriveOS against standalone Linux.

6.1 Hardware-in-the-loop Simulation Infrastructure

DriveOS is prototyped and tested on a Cincoze DX1100 machine [11], featuring an Intel Coffee Lake i7-8700T processor. This is a low-power industrial PC-class machine that is being used in an electric vehicle under development by our partner company. It has ample processing capacity to support many traditional ECU functions as software threads. Multiple I/O interfaces are capable of interfacing with different USB-CAN networks, and three display ports serve different user interfaces. Table 1 lists the machine features.

Table 1. DX1100 Specifications

Processor	Intel Coffee Lake i7-8700T ($\leq 2.4GHz$)
RAM	32 GB
eMMC Storage	64 GB
Display and UI	HDMI, DVI and DisplayPort
CAN Connector	8 USB3.x ports
Serial I/O	4 RS-232 Serial ports
Network	2 GbE Ethernet ports (x2) and mPCIe-USB Bluetooth
Power	24V, 5A
Dimension	242 mm × 174 mm × 77 mm

Figure 5 shows the HIL setup and data-flow via our car's computing hardware. A Kvaser USBcan Pro 5x HS industrial USB-CAN adapter [34] is attached to the DX1100. The CAN-Hi and CAN-Lo signals from the adapter are suitably terminated with 120Ω resistive loads. These signals feed into a USBcan Light 2x HS USB-CAN adapter attached to an Ubuntu 16.04 Linux machine, which runs the CARLA [17] driving simulator. The simulator feeds a CAN bus data trace of our partner company's electric car to test onboard IC and IVI application services. To test the ADAS services, the CARLA simulator is updated via OpenPilot running in DriveOS. Before we deploy our system fully on the road, it is necessary to rigorously test and study time-critical metrics using a HIL simulation.

6.2 Instrument Cluster (IC) and In-vehicle Infotainment (IVI)

The IC and IVI are third-party Qt applications being developed by a partner company, with sample screenshots shown in Figure 1. The IC and IVI rely on sophisticated UI libraries such as Qt, that are only supported for a selected few OSes. Therefore, it is not feasible to implement these applications in an RTOS like Quest, even though they have some critical timing requirements. DriveOS refactors these applications, so that timing-critical components are ported to Quest and the remaining parts run in Linux.

For IC and IVI, sending and receiving CAN messages needs to be fast and predictable. Slow and unpredictable CAN messaging would mean inaccurate and discrepant data in IC and IVI. DriveOS therefore features a CAN Gateway real-time service in Quest that delivers CAN packets to the IC and IVI applications via shmcomm channels. An Infotainment CAN Mapper is split between Quest (IMS) and Linux (IML) to exchange data (see Figure 5). In Linux, FIFO pipes are used to deliver

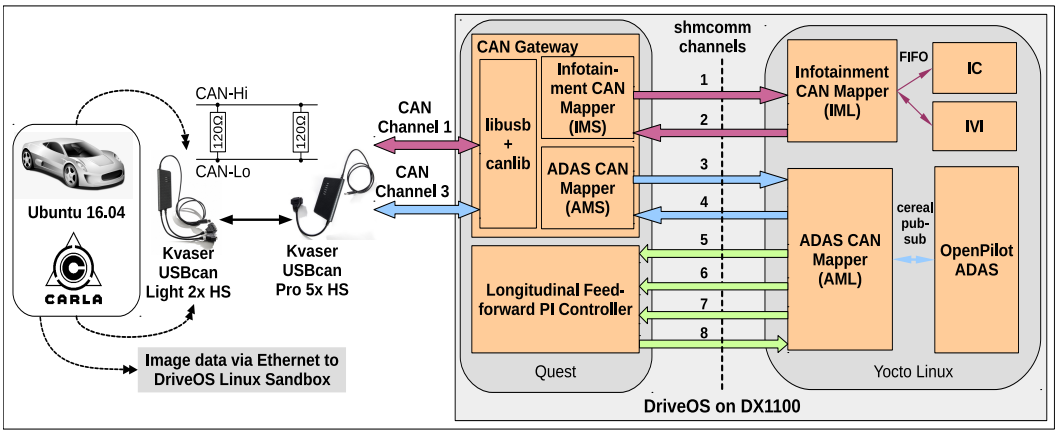


Fig. 5. Hardware-in-the-loop Simulation Infrastructure for DriveOS

messages between IC, IVI and Infotainment CAN Mapper threads. More details about the CAN Gateway real-time service task are explained later in this section.

6.3 OpenPilot Advanced Driver Assistance Systems (ADAS)

DriveOS also incorporates an open-source ADAS, OpenPilot [13], which is used daily in thousands of cars on the road. OpenPilot is originally developed for Ubuntu and Android. It supports Adaptive Cruise Control, Automated Lane Centering, Forward Collision and Lane Departure Warning. OpenPilot receives radar, gyro and other sensor data via CAN. Live images are collected via local cameras, while simulated CARLA images are received over an Ethernet link managed by Linux. OpenPilot then generates throttle, brake and steering control adjustments based on a Longitudinal PI Controller, and machine-learning (ML)-based Object Detection and Path Planning algorithms. It uses Tensorflow for ML algorithms, and Qt for a UI display as shown in Figure 1.

After OpenPilot decides an intended path by processing an image stream with the Object Detection and Path Planning algorithms, its Longitudinal Feed-forward PI Controller is responsible for generating the final throttle and brake control values. Such a longitudinal controller is central to an automotive system's safety, and is susceptible to timing violations. In general, the automotive industry expects an end-to-end (sensing-processing-actuation) delay in the order of `10ms` for such controllers [52].

OpenPilot currently runs the longitudinal controller as a `SCHED_FIFO` task and maintains a 10ms rate via the Linux `clock_gettime` API. Current OpenPilot implementations run on a dedicated Linux machine where no other applications are allowed to run, and all tasks are hand-tuned to meet their timing requirements. As the automotive industry moves towards an *integrated and extensible* vehicle management solution, arbitrary third-party applications in Linux have the potential to interfere with the timing requirements of a controller such as the one in OpenPilot [1]. Our experiments in Section 7 reveal this issue.

In an effort to port OpenPilot to DriveOS, the Longitudinal Controller is implemented in a Quest sandbox as a real-time service task. In addition, the CAN Gateway real-time service task is also utilized by OpenPilot for radar data inputs and brake, throttle and steering outputs. The rest of the OpenPilot module relies on Tensorflow, which is only available on systems such as Linux, Windows, and Mac OS, and on Qt which is also limited to a few OSes. As porting this part of OpenPilot to an RTOS would require a significant effort, it is instead deployed unchanged in the paravirtualized

Yocto Linux sandbox of DriveOS. These modules receive a stream of simulated CARLA images over Ethernet directly in Linux. Interactions between the Longitudinal Controller and the rest of the ADAS software are facilitated by inter-sandbox shmcomm channels. Thus, the DriveOS design enables a cross-sandbox implementation of OpenPilot with real-time components running in an RTOS and legacy, library-dependent components running in Linux.

6.4 Real-time Service Tasks

The above three DriveOS applications use two real-time service tasks, explained below. These tasks are based on the Real-time Task as a Service model explained in Section 3.2. Table 2 shows all the shmcomm channels between the real-time service tasks in Quest and applications in Linux.

Table 2. Shared Memory Channels in DriveOS

ID	Description	Data-flow (tasks)	Data-flow (sandboxes)	Type	Buffer size
1	IC, IVI Sensor Reading	IMS → IML	Quest → Linux	Sync	10
2	HVAC (IVI) Control Actuation	IML → IMS	Linux → Quest	Sync	10
3	CARLA Sensor Reading	AMS → AML	Quest → Linux	Async	-
4	CARLA Vehicle Control Actuation	AML → AMS	Linux → Quest	Sync	10
5	LongController Control Command	AML → LongControl.	Linux → Quest	Sync	10
6	LongController Control Data	AML → LongControl.	Linux → Quest	Sync	10
7	LongController Update Input	AML → LongControl.	Linux → Quest	Sync	10
8	LongController Update Output	LongControl. → AML	Quest → Linux	Sync	10

6.4.1 CAN Gateway Service. A CAN Gateway Service in Quest mediates real-time CAN messages for Linux applications, enabling a *real-time virtual CAN device interface*. We start describing this service from the right-hand side of Figure 5. A Linux application in DriveOS interfaces with a car CAN bus network using a CAN mapper process in Linux. Every CAN mapper process has one thread each for reading and writing CAN messages via shmcomm read and write channels. Each CAN mapper thread in the Linux sandbox interacts with a counterpart in the Quest sandbox via a specific shmcomm channel.

The CAN mapper threads in Quest (IMS and AMS in Figure 5) are part of the CAN Gateway Service. Program 3 shows how the CAN mapper threads (CAN Readers and Writers) are spawned in the Quest sandbox for `NUM_CAN` number of CAN Channels. Acting as a CAN concentrator, the CAN Gateway Service reads CAN messages from different CAN Channels of the Kvaser USBcan Pro 5x HS. We use a real-time USB xHCI (3.0) bus scheduling algorithm [23] in the interrupt bottom-half handler in Quest for fast and predictable CAN I/O. The Gateway then forwards CAN messages to appropriate shmcomm channels. Program 4 shows the C code of a CAN reader thread that reads CAN messages from a CAN Channel and forwards to Linux via a shmcomm channel. Upon receiving messages from a shmcomm channel, the CAN mapper threads in Linux (IML and AML in Figure 5) forward them to appropriate applications based on CAN message IDs. Similarly, Linux applications write to CAN channels in the reverse direction. A CAN writer thread in the CAN Gateway is shown in Program 5.

We have two CAN mappers in the Gateway service for Infotainment (IC and IVI) and ADAS. Figure 5 has color-coded CAN and shmcomm channels to show the CAN data-flow for Infotainment (violet) and ADAS (blue).

The IC and IVI use the synchronous shmcomm channel 1 to read sensor inputs such as speed, engine control type (all-wheel-drive or rear-wheel-drive), temperature inside a car, distance traveled, and so forth. The IVI application sends HVAC control commands via another synchronous shmcomm channel 2 to CAN channel 1. The IC is a read-only application and does not send any CAN messages.

OpenPilot uses the asynchronous shmcomm channel 3 to read the most recent CARLA simulator sensor readings (vehicle speed and angle) and car cruise button status (1 = initialize cruise control,

Program 3. Spawning CAN Reader and Writer Threads in the CAN Gateway Service in Quest

```
#define NUM_CAN 2
#define CAN2LINUX_VSHM_KEY0 101
#define LINUX2CAN_VSHM_KEY0 201

int can2linux_channel[NUM_CAN], linux2can_channel[NUM_CAN];

// Structure to pass arguments to the Reader and Writer Threads
typedef struct gate_th_args {
    int can_ch; int can2linux_fd; int linux2can_fd; int can_open_flags; int can_freq;
    int can_hnd; int budget_us; int period_us;
} gate_th_args_t;
gate_th_args_t* thr_args = malloc(sizeof(gate_th_args_t) * NUM_CAN);


// Structure to represent a CAN message
typedef struct can_packet {
    uint32_t id; int32_t can_msg_id; unsigned char can_msg[8]; unsigned int can_dlc;
    unsigned long can_timestamp;
} can_packet_t;


pthread_t can2linux_th[NUM_CAN], linux2can_th[NUM_CAN];

for (i = 0; i < NUM_CAN; i++) {
    // Set the arguments for a CAN Reader Thread
    thr_args[i].can_ch = i;
    thr_args[i].can_open_flags = canOPEN_EXCLUSIVE;
    thr_args[i].can_freq = canBITRATE_500K;
    thr_args[i].can2linux_fd = can2linux_channel[i];
    thr_args[i].budget_us = 100;
    thr_args[i].period_us = 2000;

    // Set up a shmcomm channel from Quest to Linux for a CAN Reader Thread
    int shm_key = CAN2LINUX_VSHM_KEY0 + i;
    can2linux_channel[i] = shmcomm_open_send(shm_key, SHMCOMM_CREATE_CH | SHMCOMM_SYNC_CH, NULL,
        sizeof(can_packet_t), 10);

    // Spawn a CAN Reader Thread to read data from CAN Channel i and send to Linux via
    // the above shmcomm channel
    pthread_create(&can2linux_th[i], NULL, can2linux_task, &thr_args[i]);

    // Set the arguments for a CAN Writer Thread, by only changing the shmcomm channel and
    // keeping everything else same from the Reader Thread
    thr_args[i].linux2can_fd = linux2can_channel[i];

    // Set up a shmcomm channel from Linux to Quest for a CAN Writer Thread
    shm_key = LINUX2CAN_VSHM_KEY0 + i;
    linux2can_channel[i] = shmcomm_open_receive(shm_key, SHMCOMM_CREATE_CH | SHMCOMM_SYNC_CH,
        NULL, sizeof(can_packet_t), 10);

    // Spawn a CAN Writer Thread to write data from Linux to CAN Channel i
    pthread_create(&linux2can_th[i], NULL, linux2can_task, &thr_args[i]);
}
```

2 = increase acceleration, 3 = decrease acceleration, 4 = cancel cruise control). OpenPilot computes throttle and brake values by a longitudinal PI controller and applies to CARLA via synchronous shmcomm channel 4 and USB-CAN.

6.4.2 ADAS Longitudinal Controller Service. As stated in Section 6.3, OpenPilot uses a feed-forward PI longitudinal controller for adaptive cruise control. The refactored implementation of OpenPilot in DriveOS runs the controller in Quest as a synchronous real-time service task. It has a 50 μs budget and 1 ms period, which is the same as in the stock OpenPilot.

For test purposes, CARLA simulator sensor data is delivered to the Longitudinal Controller from the CAN Gateway service via ADAS CAN Mapper threads in Quest and Linux. Section 7.2 explains the ADAS Controller pipeline in details. The ADAS Mapper thread in Linux (AML) is responsible for filtering sensor data and feeding it to the Longitudinal Controller in Quest via shmcomm channels (ID 5, 6, 7). AML also receives throttle, brake and other control values from the controller, and sends CAN Control commands to CARLA via shmcomm channel 4. The Linux-side implementation

Program 4. CAN Reader Thread in CAN Gateway

```
void* can2linux_task (void* thread_args) {
    gate_th_args* args = (gate_th_args*)
        thread_args;

    // Create a sporadic server in Quest
    // corresponding to thread
    struct sched_param s_params = {
        .type = MAIN_VCPU, .C =args->budget_us,
        .T = args->period_us
    };
    int new_vcpu = vcpu_create(&s_params);
    vcpu_bind_task(new_vcpu);

    // Open a CAN Channel
    args->can_hnd = canOpenChannel (
        args->can_ch, args->can_open_flags);
    canStatus can_stat = canSetBusParams (
        args->can_hnd, args->can_freq, 0, 0, 0,
        0, 0);
    can_stat = canBusOn(args->can_hnd);

    can_packet_t msg;
    unsigned int can_flag, can_dlc;
    unsigned long can_time; long can_id;
    unsigned char can_msg[CAN_MAX_DLEN];

    while(1) {
        // Read from CAN Channel
        can_stat = canRead (args->can_ch,
            &can_id, &can_msg, &can_dlc,
            &can_flag, &can_time);

        //Send to Linux via shmcomm channel
        msg.id = ++cnt;
        msg.can_msg_id = can_id;
        msg.can_dlc = can_dlc;
        msg.can_timestamp = can_time;
        memcpy(msg.can_msg, &can_msg, 8);
        while(shmcomm_write(args->can2linux_fd,
            &msg, sizeof(can_packet_t)) <= 0);
    }
}
```

Program 5. CAN Writer Thread in CAN Gateway

```
void* linux2can_task (void* thread_args){
    gate_th_args* args = (gate_th_args*)
        thread_args;

    struct sched_param s_params = {
        .type = MAIN_VCPU, .C =args->budget_us,
        .T = args->period_us
    };
    int new_vcpu = vcpu_create(&s_params);
    vcpu_bind_task(new_vcpu);

    args->can_hnd = canOpenChannel (
        args->can_ch, args->can_open_flags);
    canStatus can_stat = canSetBusParams (
        args->can_hnd, args->can_freq, 0, 0, 0,
        0, 0);
    can_stat = canBusOn(args->can_hnd);

    can_packet_t msg;
    unsigned int can_flag, can_dlc;
    unsigned long can_time; long can_id;
    unsigned char can_msg[CAN_MAX_DLEN];

    while(1){
        // Read from shmcomm channel
        while(shmcomm_read(args->linux2can_fd,
            &msg, sizeof(can_packet_t)) <=0);

        // Check if this is a valid message
        // and write to the CAN Channel
        canWrite(args->can_ch, msg.can_msg_id,
            msg.can_msg, msg.can_dlc,
            msg.can_timestamp);
    }
}
```

of OpenPilot uses its own cereal publisher-subscriber messaging framework to obtain controller values from AML, which are used for vehicle path and control planning.

Our DriveOS Longitudinal Controller depends on the control commands that it receives via the shmcomm channel ID 5. It supports three control commands: (1) `INIT` for initializing the Longitudinal Controller values such as the proportional, integral, and feed-forward constants, (2) `RESET` to reinitialize the PI loop, and (3) `UPDATE` to compute the controller throttle and brake output by the PI loop. The controller command data for `INIT` and `RESET` is sent via shmcomm channel 6. Both channels 5 and 6 are synchronous channels because missing a control command is forbidden. The `UPDATE` data is separately exchanged via shmcomm channels 7 and 8.

7 EVALUATION

Linux is commonly used in infotainment systems by popular automotive companies such as BMW [39] and Toyota [16], and as the basis of Ubuntu and Android distributions used with the OpenPilot ADAS software [13]. DriveOS is therefore compared against a standalone `PREEMPT_RT` [51] patched Yocto Linux for use in integrated vehicle management systems.

In the standalone Yocto Linux vehicle management system, software threads and interrupts are not assigned to any specific cores. For comparison, an *optimized version* of the same standalone Yocto Linux is tested. This version pins xHCI interrupts to Core 0, resulting in USB bottom-half

processing taking place on the same core, while all other threads execute on Core 1. In summary, experiments test both an unoptimized and domain-optimized standalone Linux against DriveOS.

In our experiments, DriveOS uses two cores although the system is capable of activating more - one is dedicated to Quest and the other is given to a paravirtualized Yocto Linux with the `PREEMPT_RT` patch enabled. DriveOS and the standalone Linux versions are both tested in the HIL simulation infrastructure described in Section 6.1, using a real car dataset collected from the Laguna Seca raceway in California (for IC and IVI) and CARLA (for ADAS). For fair comparisons with DriveOS, all standalone Linux systems run on a DX1100 and implement the equivalent CAN Gateway, Infotainment, OpenPilot ADAS, IC and IVI logic as shown for DriveOS in Figure 5.

7.1 Application Parameters

Table 3 shows the real-time task budgets and periods for the Quest real-time USB-CAN interface (USB xHCI Bottom-half handler and `mhydra` USB-CAN driver), CAN Gateway and longitudinal controller real-time service task. Linux-side timing critical tasks are run with the `SCHED_DEADLINE` scheduling policy. We also run non-timing-critical background tasks in Linux that log data in the eMMC storage and periodically send data over the network, as is common in modern cars [58]. Next, we describe our experimental results. All experiments were run and averaged over five times.

Table 3. Real-time Task Budgets and Periods

ID	Task	Budget (ms)	Period (ms)
Quest			
A	USB Bottom-half Handler (BH)	0.10	1
B	mhydra_rx	0.20	1
C	Infotainment read mapper	0.10	2
D	Infotainment write mapper	0.10	2
E	ADAS read mapper	0.10	2
F	ADAS write mapper	0.10	2
G	Longitudinal Controller	0.05	1
H	mhydra_tx	0.20	1
Linux			
I	Infotainment read mapper	0.10	1
J	Infotainment write mapper	0.10	1
K	ADAS read mapper	0.10	1
L	ADAS write mapper	0.10	1

Table 4. Delay Between Consecutive CAN Messages (CAN Channel 1)

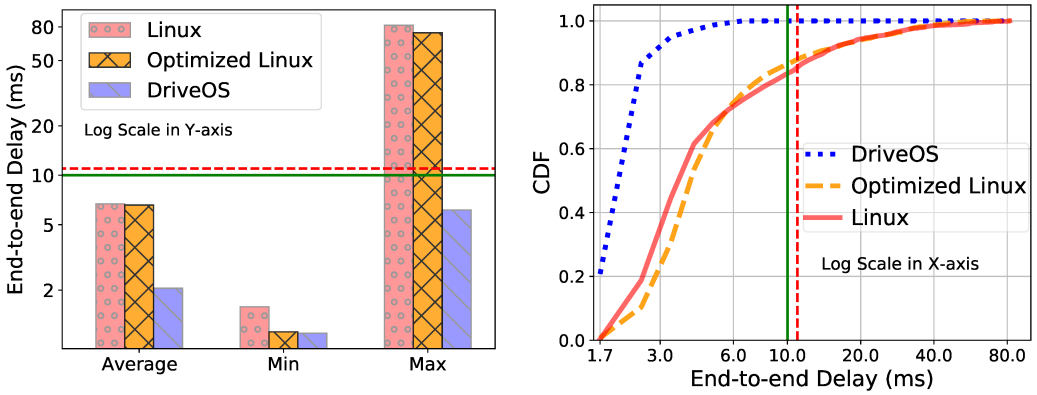
System	Average Delay
Raw CAN Frame-based	
Source (Hardware)	2.85 ms
Linux	3.73 ms
Optimized Linux	3.43 ms
DriveOS	2.86 ms
CAN ID-based	
Source (Hardware)	82.5 ms
Linux	98.97 ms
Optimized Linux	94.38 ms
DriveOS	83.25 ms

7.2 Latency Measurements

We measure two types of latency values: end-to-end delay and delay between consecutive messages. Maximum end-to-end delay gives us an upper bound on the round-trip-time of a sensor input and a corresponding actuator output. Such end-to-end latency is critical for ADAS services, which need to apply throttle and brake changes within a certain time for safety. Hence, we measure latency on the ADAS controller pipeline, while IC and IVI process CAN messages.

The task pipeline in the ADAS controller is as follows (with task IDs from Table 3 shown in parentheses): `USB BH` (A) → `mhydra_rx` (B) → `ADAS read mapper (Quest)` (E) → `ADAS read mapper (Linux)` (K) → `OpenPilot Longitudinal Controller` (G) → `ADAS write mapper (Linux)` (L) → `ADAS write mapper (Quest)` (F) → `mhydra_tx` (H) → `USB BH` (A). The theoretical worst-case end-to-end delay in a pipeline is the summation of the periods of all the tasks [24], assuming input data is available only at the beginning of a period of the pipeline's source task. Therefore, the theoretical end-to-end delay bound for the controller pipeline is: $(T(A) + T(B) + T(E) + T(K) + T(G) + T(L) + T(F) + T(H) + T(A)) = (1 + 1 + 2 + 1 + 1 + 1 + 2 + 1 + 1) = 11$ms.

This theoretical end-to-end delay bound is in the ballpark of what is expected in a working automotive system (~10ms) [52]. Empirical results show that DriveOS performs much better than both the expected and theoretical worst-case delays.



(a) Average, Minimum, Maximum

(b) Cumulative Distribution Function

Fig. 6. Controller Pipeline End-to-end Delay

Figure 6a (log scale) shows the average, minimum and maximum end-to-end delays of the controller pipeline in standalone Linux, optimized Linux and DriveOS. Figure 6b shows the corresponding cumulative distribution function (CDF) of the delays. The experiments are run with 20 non-critical background threads occupying almost 60% CPU utilization in Linux. Such background processes are representative of third-party applications (e.g., Spotify, Maps, and Data Backup). In our experiments, these tasks send data over an Ethernet network via TCP socket connections, and copy backup logs to storage. The device interrupts generated by these background tasks are intended to reveal potential interference on timing critical tasks.

In Figures 6a and 6b, the theoretical worst-case (`11ms`) and industry expected (`10ms`) delays are respectively shown with a dashed and solid line. Although Linux performs on average within bounds, the maximum delay is well above what is allowed. In Figure 6b, the CDF of delays shows that more than 15% of the end-to-end latencies are greater than 10ms in both standalone Linux versions. This could lead to an unsafe implementation of ADAS services and make the system unfavorable to regulatory authorities. DriveOS performs well within industry standards and theoretical bounds for average, minimum, and maximum end-to-end delays for a safe implementation of ADAS.

The median latencies in every 10 CAN frames in Figure 7 further reveals the unpredictable and inconsistent latency in Linux. It also shows that DriveOS has a very low end-to-end delay variation. Even though Linux does not have the additional CAN mapper threads of Quest, it performs badly because it lacks a timing-predictable interrupt handling mechanism. Optimized Linux improves the delay slightly because xHCI interrupts are pinned to Core 0. However, Linux's bottom-half processing of other interrupts on Core 1 is still able to interfere with the execution of more important `SCHED_DEADLINE` threads on that core. Quest correctly matches the scheduling priority of the I/O bottom-half handler with the thread waiting on I/O. Additionally, DriveOS refactors the longitudinal controller logic of OpenPilot to Quest, which provides temporal isolation between tasks and interrupts for time-critical tasks. Consequently, DriveOS achieves $\frac{1}{12}^{th}$ the maximum end-to-end delay observed in Linux.

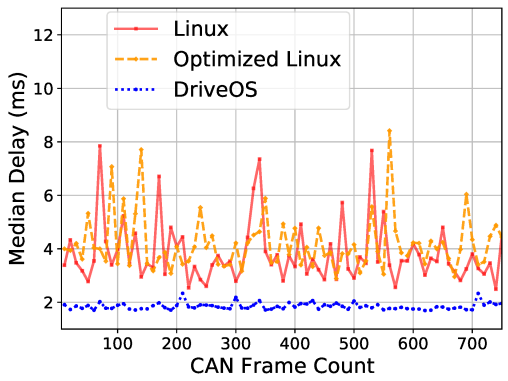


Fig. 7. Controller Pipeline Median Delay per 10 Frames

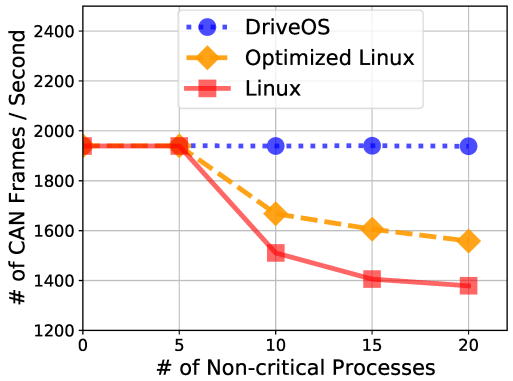


Fig. 8. Infotainment (IC and IVI) Throughput

In another set of experiments, we measure the delay between consecutive CAN frames in CAN Channel 1. Table 4 shows two types of average delay over 5000 CAN frames: (1) delay between consecutive arbitrary CAN frames and, (2) delay between consecutive frames of the same CAN ID. The source row shows the delay at the CAN message generator on the Ubuntu 16.04 simulator machine. The delay at source is representative of the delay observed at the sensor and actuator hardware. We see that DriveOS receives messages with a similar average delay. However, both versions of standalone Linux receive raw CAN messages delayed by 20-30%. Similar behavior was observed for CAN ID-based delays. This shows that DriveOS introduces negligible latency overhead on top of a CAN hardware source for a real car's CAN dataset, especially in comparison to Linux used in the automotive industry. Therefore, ECU hardware could be safely and predictably replaced by real-time software service tasks in DriveOS, where sensor readings and actuator outputs are communicated via CAN messages.

7.3 Throughput Measurements

In this experiment, we test the throughput from CAN Channel 1 to the IC and IVI applications. We measure the throughput at the end of IML in Figure 5 before forwarding the data to IC and IVI. Higher throughput means that IC and IVI tasks show more accurate and informative data on the car displays. Figure 8 shows average CAN frames per second in a period of 3 minutes, with increasing number of non-timing-critical processes in Linux. These background processes log CAN frames and make copies of data for safety. In a deployed system multiple third-party applications would actually be running as background threads.

Although Linux performs similar to DriveOS in the absence of any background threads, its performance drops as the number of such threads increases. These non-critical threads increase the number of device interrupts in Linux, and Linux fails for the same reasons stated in the earlier subsection. Optimized Linux performs a little better because xHCI interrupts are delivered to Core 0. However, DriveOS performs consistently better, and independently of the background threads because Quest's USB-CAN I/O handling is not disrupted by the background threads in the Linux sandbox. Furthermore, DriveOS's better performance is especially significant because it has a longer pipeline, traversing through a virtualized Quest sandbox and shmcomm shared-memory channels, that are absent in Linux. This shows the benefits that DriveOS's I/O handling and inter-sandbox communication mechanism provide.

Table 5 shows the average throughput and standard deviation of IC and IVI CAN message reading (at IML in Figure 5), and OpenPilot CAN message writing (at AML in Figure 5). DriveOS achieves higher throughput and better predictability with lower standard deviations. Table 5 also shows

the throughput data for our version of OpenPilot in Linux, which communicates to CARLA via Ethernet. Although the throughput is worse than DriveOS's performance with CAN, it is similar to standalone Linux's USB-CAN throughput. This shows that a timing-sensitive implementation of Ethernet could be an alternative to a CAN bus network in future automotive systems.

7.3.1 Scaling Time-Critical Processes. In the next set of experiments, we test the scalability of critical processes, which are representative of ECU functions implemented as software services. These processes read from and write to the CAN interface ($C = 20\mu s, T = 20ms$). Increasing the number of such processes should not affect the Infotainment, ADAS and other car services. Figure 9 shows the throughput of infotainment services while running 0–15 time-critical processes in the system. The throughput stays the same in DriveOS against increasing number of critical processes, as they are run as real-time services in Quest. In spite of running time-critical processes as `SCHED_DEADLINE` tasks, the drop in infotainment throughput shows that Linux does not scale against time-critical tasks that access the USB-CAN ("CAN I/O", yellow line), and disk and Ethernet devices ("Other I/O", green line). Hence, it is not a favorable choice for future ECU consolidation.

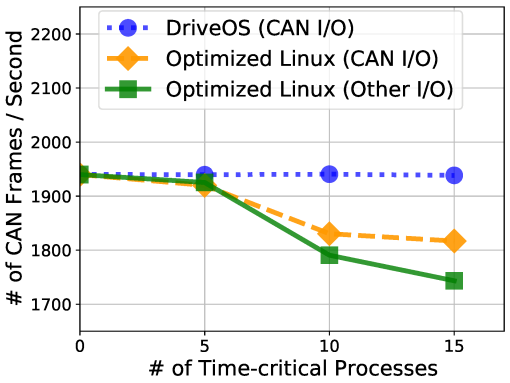


Fig. 9. Throughput against Time-critical Processes

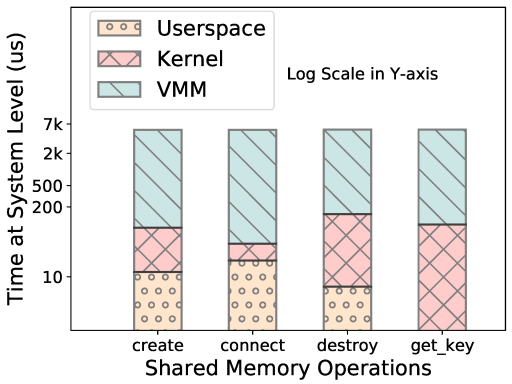


Fig. 10. Cost of `shmcomm` Operations in Quest

7.4 Startup Times

The system and application startup times are important factors for the end-users of a vehicle management system. The next set of experiments investigate whether the paravirtualization of Yocto Linux in DriveOS has any significant effect on either Linux or the applications' startup times.

The average over five cold boots is noted, where the system is initially powered down. The DriveOS paravirtualized Yocto Linux takes 18.89 seconds to boot and start the Linux shell at the serial port. In comparison, a standalone Linux takes 17.56 seconds to boot. The extra time to boot the paravirtualized Linux is the time Quest takes to boot itself before executing the boot logic of Linux. The IC and IVI application takes 674ms to start in DriveOS, whereas it is 632ms in standalone Linux. The almost negligible additional time in starting the IC and IVI in DriveOS is because of the overhead of establishing the inter-sandbox shmcomm channels. This is studied in the next section. In subsequent work, DriveOS uses ACPI power management techniques to suspend to, and resume from, RAM. A suspended system is shown to consume minimal power but is able to resume critical services in several hundred milliseconds. The details of how this works are out of the scope of this paper.

Table 5. Throughput with 20 Background Processes

System	Average (frame/sec)	Standard Deviation
IC and IVI (CAN Channel 1)		
Linux	1378.6	769.5
Optimized Linux	1558.3	632.5
DriveOS	1938.4	29.36
OpenPilot (CAN Channel 3)		
Optimized Linux (Ethernet)	17.45	0.91
Optimized Linux (CAN)	17.56	1.40
DriveOS	19.94	0.25

Table 6. Average Cost of shmcomm Channel Operations

System	Userspace (μs)	Kernel (μs)	VMM (μs)
create			
Quest	12	69	5405
Linux	30	287	
connect			
Quest	20	21	5407
Linux	40	252	
destroy			
Quest	6.5	140	5398
Linux	6.8	391	
close			
Quest	6.5	-	
Linux	6.8		
read			
Quest	0.01	-	
Linux	0.01		
write			
Quest	0.03	-	
Linux	0.03		

7.5 Inter-sandbox Communication Overhead

For all shmcomm operations, we have measured the cost at three system levels in DriveOS: (1) Guest OS userspace (ring 3 in x86), (2) Guest OS kernel (ring 0) and, (3) VMM/hypervisor (ring -1). Userspace and kernel level measurements are performed separately for Quest and Linux. VMM measurements are common for both Quest and Linux. Table 6 shows the average cost of channel operations at different system levels in Quest and in Linux. Creating, connecting and destroying a channel comes with a higher cost because we need to make an expensive hypercall (VMExit) to the VMM for these operations. The major time is thus spent in the VMM. Figure 10 shows the cost of channel operations in Quest on a log scale. It reveals how most time is spent in the VMM logic for these operations. In addition, the Linux kernel incurs more overhead in channel operations than Quest. For example, creating a channel takes 287μs in Linux, whereas it is 69μs in the Quest kernel.

Once shared memory channels are established, the costs of reading, writing and closing a channel are negligible, because the channel memory is already mapped to the userspace application. With careful time-budgeting of channel endpoints, the DriveOS inter-sandbox communication mechanism achieves fast and predictable runtime reads and writes.

7.6 Discussion

The experiments serve to show that DriveOS built as a centralized system using a partitioning hypervisor to separate real-time and legacy components is able to meet the requirements of an automotive system. While providing spatial isolation between tasks of different criticalities, it ensures legacy services in Linux work together with real-time components to meet end-to-end latency bounds and achieve higher throughput. The intent of these experiments is not to claim that DriveOS is a better partitioning hypervisor than another, but that using such a hypervisor in the context of an integrated and extensible vehicle management system is a plausible approach for future system design.

8 RELATED WORK

8.1 Vehicle Operating Systems

A number of automakers are now developing OSes for their vehicles. Tesla uses its own version of Linux [60] for its display devices. Toyota's Entune [16] for multimedia and telematics is based on Automotive Grade Linux (AGL) [61]. BMW's driving and infotainment system OS7 is built on Yocto Linux [39]. Volvo's Polestar has adopted a bare-metal Android Automotive OS [25]. An IVI system has also been implemented on a paravirtualized Android [56]. GENIVI [22] and other alliances between automotive companies are also developing AUTOSAR [3] and AGL-compliant OSes for modern vehicles. While Linux and Android provide a rich set of features, they lack real-time capabilities needed for critical tasks in modern vehicle management systems. DriveOS executes time-critical tasks in a real-time system domain, while ensuring complementary Linux services are sufficiently predictable for use in automotive applications such as IC, IVI and ADAS.

QNX is a real-time microkernel [27] used by Ford's SYNC infotainment system [20] and NVIDIA DRIVE OS [48]. The microkernel handles inter-process communication (IPC), process scheduling and interrupts. In comparison, DriveOS delegates process scheduling and interrupt handling to individual sandboxes. Only the shmcomm module in DriveOS manages the IPC between applications in different sandboxes. QNX has since become proprietary, in contrast to DriveOS's openly available design and implementation.

8.2 Partitioning Hypervisors

Quest-V [62] is a separation kernel that statically partitions hardware resources among multiple guest sandboxes. Each sandbox is responsible for task scheduling and device handling without the involvement of the Quest-V hypervisor. The Quest RTOS [15] bootstraps Quest-V and initializes other sandboxes. Other partitioning hypervisors like Jailhouse [50] and ACRN [36] rely on Linux to bootstrap the system. Bao adopts a clean-slate partitioning hypervisor implementation for ARM and RISC-V architectures, without relying on Linux [43]. Using Linux to bootstrap guests in Jailhouse and ACRN increases the security attack surface of the partitioning hypervisor. In addition, ACRN's Linux-based service OS manages the hardware resources for other safety-critical sandboxes, unlike Quest-V's policy of directly assigning devices to guest sandboxes. PikeOS [30] and Muen [7] separation kernels also do not support independent interrupt handling by the guest sandboxes.

Inspired by Quest-V, DriveOS is bootstrapped by the relatively small Quest system, with less than 4KB of its codebase remaining within the hypervisor (ring -1) privilege level. This eases the path to verification and certification by regulatory authorities. As the hypervisor occupies the most privileged protection domain, and it is not required for runtime resource management decisions by its guests, it is removed for regular control-flow operations. This heightens the security of the system.

Although Quest-V supports communication between sandboxes [37], it uses Inter-processor Interrupts for such communications. This reduces the available CPU utilization of the guest sandbox as an interrupt handling thread needs to be dedicated to communication requests. In contrast, DriveOS entirely relies on EPT hardware virtualization in x86 for predictable inter-sandbox communication, similar to what is done in Boomerang [24]. Unlike Quest-V, Jailhouse and ACRN, DriveOS shows the utility of a partitioning hypervisor in the context of a vehicle management system, where most carmakers are using flavors of Linux. DriveOS demonstrates the benefits of integrating a real-time virtual CAN interface in Linux for automotive systems.

8.3 Vehicular Software Infrastructure

Recent years have seen numerous efforts to support autonomous driving. Autoware is a self-driving infrastructure for NVIDIA PX2, which provides machine learning frameworks for object detection, and path planning [31]. Apollo is another such infrastructure project by Baidu [2]. OpenPilot is an open-source adaptive cruise controller [13], challenging Tesla's Autopilot and FSD (Full Self-Driving) [59]. These projects heavily rely on flavors of Linux and are complementary to DriveOS. DriveOS is capable of incorporating third-party applications such as Cerence [10]; it already integrates the IC and IVI applications of our partner software company, as well as OpenPilot.

In addition, DriveOS adopts a Real-time Task as a Service model, allowing ECU functions to run as software tasks in Quest. In the future, DriveOS is expected to benefit from many recent open-source ECU projects on Arduino and STM32 boards [21, 38, 53, 57], as they are integrated as real-time services in a consolidated centralized system.

9 CONCLUSIONS AND FUTURE WORK

This paper presents DriveOS, an integrated vehicle management system. To the best of our knowledge, it is the first open study[1] of a vehicle management system that is being used for a production-grade electric car. DriveOS uses a real-time separation kernel to host an in-house RTOS, Quest, and Yocto Linux as guest sandboxes. As proof of concept, a fast and predictable CAN gateway and a longitudinal controller are implemented as real-time services. These services support three vehicle management applications in Linux, namely IC, IVI and OpenPilot ADAS. A secure, predictable and low-overhead `shmcomm` module facilitates the shared memory communication between real-time services in Quest and applications in Linux.

DriveOS provides a real-time virtual CAN interface to Linux applications, via low-latency shared memory communication channels and predictable device I/O in Quest. Timing- and safety-critical tasks and devices are securely isolated from Linux. DriveOS enables application developers to use convenient libraries and APIs provided by Linux, while automotive developers focus on code deployment in a smaller, lighter-weight real-time environment such as Quest.

DriveOS is tested using timing-based metrics against a standalone Linux that is currently found in many infotainment systems in the automotive industry. Experiments show that the maximum end-to-end delay for a USB-CAN-dependent controller loop is 12 times more in Linux than in DriveOS. DriveOS also achieves 24% more throughput for a vehicle's CAN messages than a standalone Linux.

Future work will use Quest's real-time USB framework to integrate camera devices as part of an enriched autonomous vehicle framework. Additional real-time virtual device interfaces will be provided via shmcomm to allow Linux and Android to implement new ADAS services. We also aim to migrate more ECU functions to DriveOS, to support real-time torque vectoring and battery management.

10 ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF) under Grant # 2007707. 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. This work would not be possible without the financial aid, equipment and software development provided by our colleagues at Drako Motors and Celenum. Special thanks to Dean Drako, Shiv Sikand, Henry Grishashvili, Andy Cook, Mika Reinikainen and Nguyen Thi Le Chau.

[1]DriveOS is open-source. It will be available on request for anyone wishing to evaluate our system and verify our results.

REFERENCES

[1] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. 2002. A Measurement-based Analysis of the Real-time Performance of Linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. 133–142.

[2] ApolloAuto. 2021. Apollo Automotive. https://github.com/ApolloAuto/apollo.

[3] AUTOSAR. 2020. AUTomotive Open System ARchitecture. http://www.autosar.org.

[4] Algirdas Avizienis. 1985. The N-version Approach to Fault-tolerant Software. *Software Engineering, IEEE Transactions on* 12 (1985), 1491–1501.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[6] Thomas C Bressoud and Fred B Schneider. 1995. Hypervisor-based fault tolerance. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 1–11.

[7] Reto Buerki and Adrian-Ken Rueegsegger. 2013. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep* (2013).

[8] Ondrej Burkacky, Johannes Deichmann, Georg Doll, and Christian Knochenhauer. 2018. Rethinking car software and electronics architecture. *McKinsey & Company* (2018).

[9] Ondrej Burkacky, Johannes Deichmann, and Jan Paul Stein. 2019. Automotive Software and Electronics 2030: Mapping the Sector's Future Landscape. *McKinsey & Company* (2019).

[10] Cerence. 2021. Cerence. https://www.cerence.com/home.

[11] Cincoze. 2021. DX1100. https://www.cincoze.com/.

[12] Cisco. 2017. Time-Sensitive Networking: A Technical Introduction. White paper, www.cisco.com.

[13] comma.ai. 2020. OpenPilot: An Open Source Driver Assistance System. https://github.com/commaai/openpilot.

[14] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. 2010. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach.. In *2010 European Dependable Computing Conference*. 67–72.

[15] Matthew Danish, Ye Li, and Richard West. 2011. Virtual-CPU scheduling in the Quest operating system. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 169–179.

[16] Steve Dent. 2017. Toyota's latest infotainment system is powered by Linux. https://www.engadget.com/2017-05-31-toyota-automotive-grade-linux-camry.html.

[17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[18] Christof Ebert and Capers Jones. April 2009. Embedded Software: Facts, Figures, and Future. 42, 4 (April 2009), 42–52. https://doi.org/10.1109/MC.2009.118

[19] William J. Fleming. 2001. Overview of Automotive Sensors. *IEEE Sensors Journal* 1, 4 (December 2001).

[20] Ford. 2021. SYNC. https://www.ford.com/technology/sync/.

[21] FreeEMS Community. 2020. FreeEMS: Free and Open Source Engine Management System. http://freeems.org/.

[22] GENIVI. 2021. GENIVI Alliance. https://www.genivi.org/.

[23] Ahmad Golchin, Zhuoqun Cheng, and Richard West. 2018. Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 196–207.

[24] Ahmad Golchin, Soham Sinha, and Richard West. 2020. Boomerang: Real-Time I/O Meets Legacy Systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 390–402.

[25] Google. 2021. Android Automotive OS. https://source.android.com/devices/automotive.

[26] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 653–669.

[27] Dan Hildebrand. 1992. An Architectural Overview of QNX.. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. 113–126.

[28] Intel. 2020. Benefits of ECU Consolidation. (2020).

[29] ISO. 2011. ISO 26262-3: Road vehicles - Functional safety - Part 3: Concept phase .

[30] Robert Kaiser and Stephan Wagner. 2007. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, Vol. 50.

[31] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. 2018. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 287–296.

[32] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. 2005. The Time-Triggered Ethernet Design. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*.

[33] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, and Hovav Shacham et al. 2010. Experimental Security Analysis of a Modern

Automobile. In *2010 IEEE Symposium on Security and Privacy*. 447–462. https://doi.org/10.1109/SP.2010.34
[34] Kvaser. 2021. Kvaser USBcan Pro 5xHS. https://www.kvaser.com/product/kvaser-usbcan-pro-5xhs/.
[35] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. 2007. A Comparison of Partitioning Operating Systems for Integrated Systems. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 342–355.
[36] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. 2019. ACRN: a big little hypervisor for IoT development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 31–44.
[37] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. 2014. Predictable Communication and Migration in the Quest-V Separation Kernel. In *2014 IEEE Real-Time Systems Symposium*. IEEE, 272–283.
[38] LibreEMS Community. 2020. LibreEMS: Open Source Automotive EMS. https://www.libreems.org/.
[39] FOSS Linux. 2019. BMW gets closer to adopting Linux as the mainline platform. https://bit.ly/2XKD6kk.
[40] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61.
[41] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.
[42] Robert E. Lyons and Wouter Vanderkulk. 1962. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209.
[43] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. 2020. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[44] Charlie Miller and Chris Valasek. 2013. Adventures in automotive networks and control units. *Def Con* 21 (2013), 260–264.
[45] Allen Brady Montz, David Mosberger, Sean W O'Mally, Larry L Peterson, and Todd A Proebsting. 1995. Scout: A communications-oriented operating system. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE, 58–61.
[46] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking Tesla from Wireless to CAN Bus. *Briefing, Black Hat USA* (2017), 1–16.
[47] Georg Niedrist. 2016. Deterministic Architecture and Middleware for Domain Control Units and Simplified Integration Process Applied to ADAS. https://www.tttech.com/technologies/adas.
[48] NVIDIA. 2021. DRIVE OS and SDK. https://docs.nvidia.com/drive/.
[49] Roberto Pineiro, Kleoni Ioannidou, Scott A Brandt, and Carlos Maltzahn. 2011. Rad-flows: Buffering for predictable communication. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 23–33.
[50] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look mum, no VM exits!(almost). *arXiv preprint arXiv:1705.06932* (2017).
[51] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2019. The Real-time Linux Kernel: A Survey on PREEMPT-RT. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–36.
[52] RIMAC Automobili. 2016. Rimac All Wheel Torque Vectoring. *Press Release* (2016). https://www.rimac-automobili.com/media/press-releases/rimac-all-wheel-torque-vectoring/
[53] rusEFI Community. 2020. rusEFI: A GPL Open Source Engine Management System. https://rusefi.com/.
[54] John M Rushby. 1981. Design and Verification of Secure Systems. *ACM SIGOPS Operating Systems Review* 15, 5 (1981), 12–21.
[55] H.R. Simpson. 1990. Four-slot Fully Asynchronous Communication Mechanism. *IEEE Computers and Digital Techniques* 137 (January 1990), 17–30.
[56] Soham Sinha, Ahmad Golchin, Craig Einstein, and 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*. 50–55.
[57] Speeduino Community. 2020. Open-source Engine Control with Speeduino. https://speeduino.com/home/.
[58] TE Connectivity. 2018. The Car in the Age of Connectivity: Enabling Car to Cloud Connectivity. *IEEE Spectrum: Technology, Engineering, and Science News* (2018).
[59] Tesla. 2021. Autopilot. https://www.tesla.com/support/autopilot.
[60] Tesla. 2021. Linux. https://github.com/teslamotors/linux.
[61] The Linux Foundation. 2021. Automotive Grade Linux. https://www.automotivelinux.org/.
[62] Richard West, Ye Li, Eric Missimer, and Matthew Danish. 2016. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Transactions on Computer Systems* 34, 3, Article 8 (June 2016), 41 pages.
[63] Ally Winning. May 15, 2019. Number of automotive ECUs continues to rise. https://www.eenewsautomotive.com/news/number-automotive-ecus-continues-rise.

[64] Yuting Zhang and Richard West. 2006. Process-aware interrupt scheduling and accounting. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 191–201.

Received April 2021; revised June 2021; accepted July 2021; modified Apr 2022