Jumpstart: Fast Critical Service Resumption for a Partitioning Hypervisor in Embedded Systems

Ahmad Golchin
Boston University
golchin@bu.edu

Richard West
Boston University
richwest@bu.edu

***Abstract*—Complex embedded systems are now supporting the co-existence of multiple OSes to manage services once assigned to separate embedded microcontrollers. Automotive systems, for example, now use multiple OSes to consolidate electronic control unit (ECU) functions on a centralized embedded computing platform. Such platforms have the complexity of an industrial embedded PC, with multiple cores and hardware virtualization capabilities. This enables a partitioning hypervisor to spatially and temporally share the physical machine with separate guest OSes, which manage services of different criticality levels. However, PC-class hardware incurs a large latency to bootstrap an OS and associated application-level services. A firmware BIOS performs a power-on-self-test, and then loads OS images into memory from a bootable storage device. This latency is unacceptable in time-critical embedded systems, where important services must be operational within milliseconds of starting the system. In this paper, we present Jumpstart, a PC-class power management approach that minimizes the wakeup delay of a partitioning hypervisor for use in embedded systems. We show how Jumpstart resumes critical OS services and tasks from a low power suspended state in approximately 600 milliseconds, and reduces full system startup delay by a factor of 23. Additionally, Jumpstart consumes minimal power compared to approaches requiring a system boot from a previously powered-off state. By comparison, an alternative firmware-optimized bootloader, called Slim, reduces boot latency by a factor of 1.8.**

***Index Terms*—partitioning hypervisors, power management, ACPI, real-time operating systems**

I. Introduction

Embedded systems are witnessing significant advances in complexity. For example, modern automotive systems support 10s to 100s of millions of lines of code [16], and have upwards of 100 electronic control units (ECUs) for chassis, body, powertrain, infotainment and vehicle control services [21], [36], [51]. Rather than using separate microcontroller hardware to run individual tasks, embedded systems designers are looking to consolidate functionality on a centralized computing platform. This is true of automotive systems, which aim to replace the abundance of ECUs with software tasks running on a more powerful centralized machine [17], [26]. This potentially saves costs, reduces wiring, and simplifies packaging in space-limited situations.

Embedded system tasks have varying temporal and spatial constraints. For example, timing-critical tasks in automotive systems must complete within tight timing bounds for the system to remain operational, and they must be sufficiently isolated from other tasks to ensure appropriate safety integrity levels are maintained [28]. In the absence of separate hardware to isolate tasks, a centralized platform must provide other mechanisms to avoid spatial interference. One approach is to use hardware virtualization.

PC-class hardware is a low-cost approach to satisfy the requirements of a centralized embedded system. Modern PCs feature multicore CPUs with hardware virtualization capabilities (e.g., Intel VT-x, and AMD-V). An embedded PC supporting a *partitioning hypervisor* [18], [42], [50] is then able to assign tasks of different criticality (or integrity) levels to different guest domains. Unlike traditional consolidating hypervisors, which *share* a physical machine among all guests, a partitioning hypervisor *divides* machine resources among separate guests. Each guest then accesses its own processing cores, physical memory and subset of I/O devices. Tasks operating in one domain are isolated both temporally and spatially from tasks in another domain (because they cannot directly access the same cores, memory and I/O devices).

In this paper, we first introduce our real-time partitioning hypervisor, called Quest-V [50]. Our hypervisor is being used to build a centralized automotive system, DriveOS [45], for an electric vehicle being developed with a partner company. This system combines the Quest RTOS [19] with Linux to manage the functional requirements of the vehicle. Separate ECUs are replaced with a collection of simpler CAN-bus transceivers that link sensors and actuators to a central PC. Software tasks are assigned to different RTOS and Linux instances, depending on their timing and safety requirements.

The major challenge addressed in this paper is the reduction of startup latency for a partitioning hypervisor running on PC-class hardware, in the context of an automotive system. Whereas traditional embedded microcontrollers would be operational within milliseconds of receiving power, a PC takes tens of seconds to boot an OS and instantiate a task. Current vehicle users would find the added boot-time latency of a PC-class system unacceptable, given they are used to vehicles being operational within milliseconds to a few seconds of being started. Moreover, critical functions that communicate with a CAN bus must be operational with low latency to ensure the vehicle starts in a safe state. The bound on this latency is dictated by a programmed delay that determines when CAN networks are active and generating traffic.

While a PC-class system has the overhead of a firmware power-on-self-test (POST) and the loading of an OS from bootable storage into RAM, we show how to mitigate much of this latency using modern power-management techniques in the context of a partitioning hypervisor. In this paper, we describe *Jumpstart*, which enables our vehicle management system to execute its critical real-time services in several hundred milliseconds of starting the vehicle.

The contributions of this paper are: (1) a brief explanation of the partitioning hypervisor used in our vehicle management system, (2) a power management-aware version of our system, called Jumpstart, which replaces shutdown and cold boot operations of our system with suspend-to-RAM (S2R) and resume operations, in response to a vehicle's stop and start events, (3) a portable method for partitioning and safeguarding the power management features of PC-class hardware in our hypervisor, (4) a description of the challenges and techniques to orchestrate S2R and resume operations among our hypervisor, RTOS and Linux, while incurring lower overheads than a non-virtualized Linux system, and (5) an empirical demonstration of how Jumpstart prioritizes the resumption of timing-critical tasks in preference to less-critical services running in Linux.

We compare Jumpstart with a standalone Linux system and show that while both consume similar power, Jumpstart resumes critical tasks with lower latency, despite the overheads of our partitioning hypervisor. Jumpstart is able to resume a system more than 20 times faster than one requiring a cold boot. Our approach is the first to support low-latency resumption of critical services, which are isolated from less important tasks, in a partitioning hypervisor running on PC-class hardware.

The next section describes further motivation behind our system design. Section IV then describes Jumpstart. This is followed by Section V, which compares the performance of Jumpstart to standalone Linux and the Slim bootloader [4]. Slim is a firmware developed by Intel to reduce startup latency for PCs. Related work is described in Section VI, followed by conclusions and future work in Section VII.

II. Motivation

Our work is focused on the design of complex next-generation embedded systems. We envision such systems providing support for hundreds of software threads executing on multiple cores, and requiring both temporal and spatial isolation according to different criticality or integrity levels [28]. Our vehicle management system, depicted in Figure 1 and further described in Section IV-A, is designed with this philosophy. It uses relatively low-cost PC hardware to consolidate ECU functions as software threads on a centralized platform. Threads are assigned to different *sandbox* (or guest) domains according to functional, timing and safety requirements. The approach allows for automotive systems to be easily upgraded or extended with new functionality, without the need for additional ECUs or other hardware components.

Next-generation vehicle management systems are not only faced with the challenge of ensuring functional, timing and safety requirements. They are required to startup and shutdown

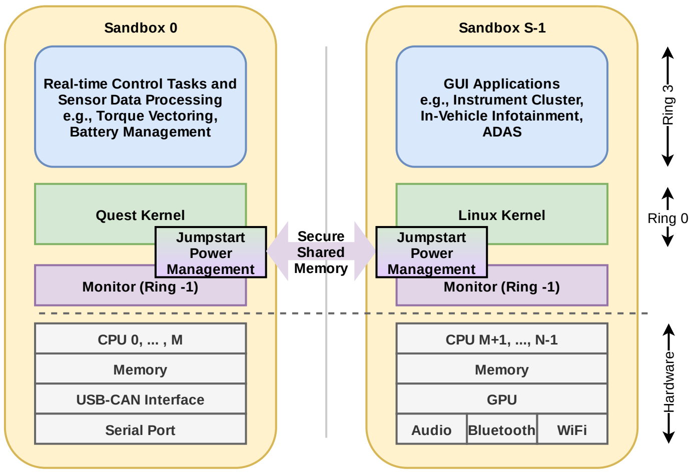

Sandbox 0
Real-time Control Tasks and Sensor Data Processing e.g., Torque Vectoring, Battery Management
Quest Kernel
Jumpstart Power Management
Monitor (Ring -1)
CPU 0, ... , M
Memory
USB-CAN Interface
Serial Port
Secure Shared Memory
Sandbox S-1
GUI Applications e.g., Instrument Cluster, In-Vehicle Infotainment, ADAS
Linux Kernel
Jumpstart Power Management
Monitor (Ring -1)
CPU M+1, ..., N-1
Memory
GPU
Audio
Bluetooth
WiFi
Ring 3
Ring 0
Hardware

Figure 1. Structure of the Jumpstart vehicle management system

when the vehicle is running or parked, respectively. The (cold) boot-time latency of a PC-class OS is far greater than what is acceptable in an automotive system. Owners of modern vehicles accustomed to minimal startup latency after pressing a dashboard start button or unlocking a door would not accept the boot delay of a PC-based OS before being able to drive. CAN buses, sensors and actuators would also need delayed activation until the system is properly initialized. In a drive-by-wire system, for example, CAN messages convey steering, throttle and braking values to actuators, to control vehicle movement. It would be potentially disastrous to allow uninitialized communication within a vehicle before the central management system is bootstrapped.

The boot delays of a PC-class vehicle management system are made worse in the context of a partitioning hypervisor, which hosts multiple guest OSes. These startup delays are avoided if the system is kept running, even when a vehicle is not in use. However, this consumes unnecessary power from the finite capacity battery within the vehicle. It is therefore preferable to shutdown or suspend a system when not in use. A parked vehicle then only needs to consume enough power to keep peripheral circuits alive, such as for an alarm.

Fortunately, modern PCs have Advanced Configuration and Power Interface (ACPI) compliant firmwares that enable the machine to switch into different global (G#), and corresponding sleep (S#) power states. While a working PC operates in global ACPI state G0 (or, equivalently, sleep state S0), it is possible to take advantage of low power sleep states when the machine is inactive. A PC-class vehicle management system is able to transition to sleep state S3, to conserve power, when not in use. The sleep state S3 suspends a system to RAM. Transition from S3 to S0 enables faster resumption of services when the vehicle is restarted.

In the context of a vehicle management system, low latency activation of timing-critical services to manage chassis and powertrain functions is essential. This is achieved by allowing real-time services to resume in parallel with legacy services provided by a system such as Linux.

Figure 2 shows the power consumption in Watts for a Cincoze DX1100 industrial PC for different ACPI sleep power states. A DX1100 running our vehicle management system is being tested in the Drako Motors GTE electric car [20]. ACPI sleep state S5 is the soft-off state, where the PC uses very little power but requires a full system reboot on restart. However, the suspend-to-RAM state, S3, uses almost identical power to S5 but allows the system to be resumed in a way that bypasses the firmware POST and bootloading stages. In Jumpstart, the partitioning hypervisor and guest OSes are suspended to RAM, enabling critical services to be subsequently resumed in hundreds of milliseconds.

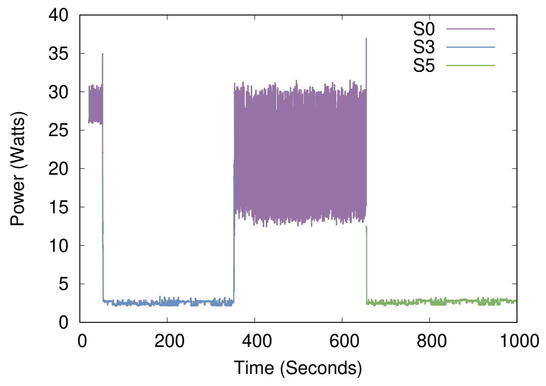


Figure 2. Cincoze DX1100 power consumption in different ACPI S-states

III. Background

A. PC Power Management

The power management capabilities of an Intel-Architecture PC (as well as modern ARM platforms) are divided between the Advanced Configuration and Power Interface (ACPI) [7] and PCI [38] standards. PCI provides the PCI-PM [39] configuration interface to observe and control power states of PCI devices and buses. PCI-PM follows a set of standard power states defined by ACPI for devices (D-States) and PCI-BUS links (B-States). ACPI also defines system-wide power states such as shutdown, suspend-to-RAM, and suspend-to-disk, as well as power states for processors. It also specifies a number of special devices such as an embedded controller (EC), power button, battery, laptop lid and so forth, and provides a uniform interface to them. Every IA-PC must implement one EC device that provides, through ACPI, a programmatic interface to the glue logic of the motherboard in order to relay electricity to CPU sockets and buses, and sense wake-up signals such as from the power button and real-time clock (RTC) alarm. Figure 3 provides an overview of the ACPI-defined CPU and system-wide power states as well as allowable transitions between them. Each allowable transition is depicted as an arrow. For example, it is possible to reach the G3 state from G0, G1, and G2, while a transition from G3 to G1 is not allowed.

The G-states are abstract system-wide power states that specify what CPU (C) and Device (D) states are available.

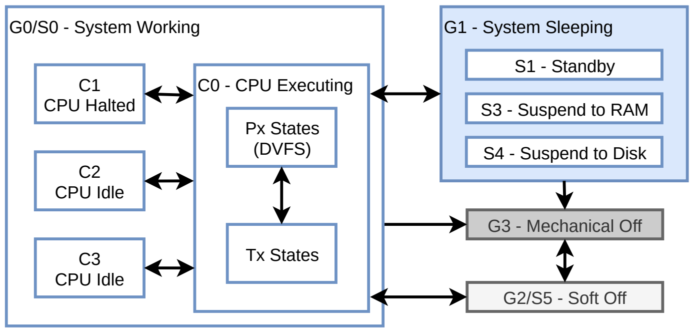


Figure 3. ACPI power states

For example, CPU states are only valid in G0 and G1/S1. CPUs are not powered in higher G-states. While in G1/S3, only some peripheral devices and buses may be on. In the G2/S5 shutdown or “soft-off” state, the system consumes a minimal amount of electricity to keep the EC and a few devices in standby mode. This enables the PC to power on in response to events such as pressing the power button, the RTC alarm expiring, or arrival of a Ethernet packet (for Wake-on-LAN operation). The G3 “mechanical off” state, on the other hand, refers to the situation where the PC is unplugged from an electrical power source.

In the G0 state, each CPU exposes several sub-states, C0 to Cn. CPUs execute in C0, and halt in C1 and deeper C states. These states differ in power saving and entry/exit latency. The deeper the C-state, the potentially greater power saving but longer delay to bring the CPU into an executing state. C1 is entered whenever the CPU executes an HLT instruction. Deeper C-states are accessible through Model-Specific Registers (MSRs). The C0 state is divided into sub-states called CPU performance states (P-States). In P-states, the CPU is working, but the CPU voltage and frequency vary. Again, system software controls P-states through MSRs while the CPU is in C0. P-states are implemented by the dynamic voltage/frequency circuitry (DVFS) of the CPU. Similar to P-states, T-states are sub-states of C0. They save power by only changing CPU frequency, and are usually used to handle thermal events. Guest kernels manage their CPU power states using MSRs granted access by the hypervisor.

As for power management of peripheral devices, ACPI specifies four D-states, D0 to D3. D0 is the working state in which the device is fully powered and operational. D3 is the power-off state. Depending on device capabilities, D3 sometimes has two sub-states: Hot-D3 and Cold-D3, with the latter being equivalent to a complete power-down of the device. In this case, there is no voltage provided to the device by the bus. Naturally, all devices support Cold-D3 as when a PCI bus link is powered down, the devices attached to the bus will lose power and transition to Cold-D3. D-states differ in power saving, device context preservation, and entry/exit latencies. Deeper D-states (D3-cold is the deepest state) lose more device context information, take longer to resume to D0 and save more power. System software controls the transition between D-states by accessing PCI-PM capability registers of PCI devices. The power states of devices are also accessible through ACPI functions written in ACPI Machine Language (AML). Firmware developers utilize AML to port such functionalities to different hardware platforms at a higher level of abstraction. Jumpstart's partitioning hypervisor uses an open source implementation of the ACPI Component Architecture (ACPICA [8]) to interpret AML functions.

The G1 sleeping state consists of three sub-states we refer to as the ACPI S1, S3 and S4 states. Deeper S-states provide more power saving at the cost of longer resumption delay. S1 is the lightweight sleep state, with only CPU caches discarded. S2 is not currently supported by the ACPI standard. S3 has all system context lost except main memory. In S3, there is electrical current running into the memory controller in addition to the EC and wake-up signal sources. All devices except wake-up sources must be in D3 before transitioning into S3. S4 saves system context to disk and then discards non-persistent memory state. The system software manages S-state transitions via special I/O ports of the EC that are enumerable through ACPI.

Power management techniques are often categorized into runtime (a.k.a. Dynamic) and system-wide (a.k.a. Static) methods. Runtime power management involves transitioning CPUs and devices into lower power states while the system is running. For example, Linux provides device usage notification APIs in the kernel as a way for drivers to help the Operating System Power Management (OSPM) make opportunistic decisions about switching D-states of the devices. CPU power and performance states (notably, C and P-states) are usually manipulated by the kernel, and based on workload and selected scheduling policies. Unlike runtime power management that only affects parts of the system, system-wide methods (G1, G2, and G3), as the name suggests, concern the system as a whole. Therefore, they cannot be employed by the OSPM as frequently as runtime methods, but they offer more power-saving properties.

B. *Challenges of Real Power Management*

In the context of virtualized environments, power methods are divided into *real* versus *virtual* [47] depending on whether or not they are passed through to the actual hardware when issued by a guest operating system.

Performing real power methods in virtualized environments leads to several challenges when more than one guest OS has access to underlying host-physical resources. Our vehicle management system's ACPI S3 power method is influenced by the following challenges: (1) The hypervisor must deny access to the EC by unauthorized guest OSes to prevent untrusted software from performing system shutdown, reboot or suspend, thus disrupting the execution of the vehicle's critical services when it is not safe to do so. Section IV-B1 describes how Jumpstart provides this capability for its hypervisor. (2) The hypervisor must orchestrate system-wide power transitions to ensure all guest OSes and their services are in the proper state before entering and after leaving the target power state. Section IV-B2 elaborates our solution in the case of the ACPI S3 power method. (3) Resuming the RTOS from RAM affects the behavior of real-time tasks involved in the reading and processing of sensors, and producing actuator values, both in terms of timing and validity of their results. Section IV-B3 explains how Jumpstart addressed this issue. (4) Upon resuming from RAM, the vehicle management system should prioritize the startup of the RTOS over Linux, while taking advantage of the parallelism offered by the underlying multicore processing hardware.

As explained in Section IV-A, the Quest RTOS shares the boot logic of our hypervisor. This design results in the RTOS being operational in milliseconds after the hypervisor's initialization.

IV. Jumpstart System Design

A. *Quest-V Partitioning Hypervisor*

Jumpstart is an extension to the DriveOS [45] centralized automotive system, which uses Quest-V to consolidate both timing-sensitive and non-critical tasks of an electric vehicle onto an embedded PC. Jumpstart adds power management capabilities to DriveOS, to ensure that critical tasks are resumed with low latency when a vehicle is started. [1] Example critical tasks include torque vectoring and battery management, while less critical tasks include those that manage the Instrument Cluster (IC) and In-Vehicle Infotainment (IVI).

The Quest-V partitioning hypervisor employs Intel's virtualization technology (VT-x) to partition hardware resources such as processor cores, physical memory, and I/O devices between two or more sandboxes (i.e., guest OS domains). Each sandbox manages its physical resources independently of other sandboxes, and without the involvement of the hypervisor.

The hypervisor's logic (a.k.a. the virtual machine monitor) is cloned for each sandbox and is only used to bootstrap the guest kernel, establish secure inter-sandbox communication channels (ISBC) via extended page tables (EPTs) [2], and on rare occasions process privileged operations such as guest faults. Unlike consolidating hypervisors, the virtual machine monitors are removed from runtime management of physical machine resources, keeping the trusted code base very small. A monitor has a text segment of less than 4KB.

Due to the reasons above and given the replication of monitors for each sandbox, the system's most privileged component is less susceptible to security attacks than a conventional OS image running directly on hardware. In the latter case, system calls must pass control to the host kernel, whereas in our partitioning hypervisor, these are restricted to the local guest.

Hypervisors such as Xen [11] and KVM [24] rely on Linux acting as a "Dom0" privileged domain, while ACRN [30] uses a similar "ServiceOS" to start a guest. In contrast, Quest-V shares much of its code with the Quest RTOS, which is paravirtualized for use as a guest. As Quest is responsible for

[1] In this paper, we refer to the Jumpstart vehicle management system as an instance of DriveOS extended with power management capabilities.

[2] Intel processors with VT-x capabilities refer to these tables as EPTs. AMD-V processors have similar nested page tables (NPTs).

a relatively small number of critical tasks and devices (e.g., a USB-CAN interface), its startup latency within Quest-V is far less than either standalone Linux, or a Linux guest running in hypervisors such as Xen, KVM and ACRN.

Quest-V combines one or more instances of the Quest RTOS with one or more instances of Linux to form a mutually beneficial *symbiotic system* [22] of S sandboxes. Each RTOS gains access to legacy functionality that would take many years to develop, while each legacy Linux system gains real-time capabilities without significant modification. At the same time, a small footprint RTOS such as Quest is more easily verified [23], [32] and certified for timing and functional correctness than a large system such as Linux adapted for use in safety-critical domains.

In our vehicle management system, legacy Linux services provide drivers for accelerators, touchscreens and graphics controllers, to display IC readings and IVI controls. Linux services are able to read and adjust vehicle settings via secure and predictable shared memory channels to the RTOS, which provides real-time control over a USB-CAN interface. This interface connects sensors and actuators to the central PC, via a series of CAN transceivers that replace more complex ECUs.

The potential 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 [33], or N-versioning [10]. Hypervisor-based fault tolerance ensures one sandboxed guest is able to recover from failure [14].

As mentioned earlier, the focus of this paper is on Jumpstart, which provides fast resumption of critical services in the context of our (automotive) partitioning hypervisor. Jumpstart uses ACPI-based power management to quickly wakeup the system from a suspended state.

B. Jumpstart Power Management

As we explained in Section II, the cold boot delay of a PC-based vehicle management system is unacceptable. Jumpstart shortens this delay by taking advantage of the low exit-latency of the ACPI S3 state. Our experiments show that jumpstarting our vehicle management system with one Quest RTOS and Linux pair of sandboxes takes about 1 second to resume both guests, whereas it takes about 25 seconds if the system undergoes a cold boot. Cold booting a system means starting it from the ACPI G2/S5 or G3 states. In contrast, a wakeup from S3 resumes the system from RAM. Figure 4 summarizes the steps involved in the cold boot and resumption from S3 of our vehicle management system with color-coded blocks. In each case, the blocks that are aligned vertically run in parallel.

Upon a cold boot, the firmware performs a Power-On-Self-Test (POST) sequence, finds the preferred bootable media and loads the first stage of the bootloader into a conventional memory location. The duration of POST has a direct correlation with the number of CPU cores and devices of the platform. More CPUs and devices require more initialization and test procedures for the firmware to follow. Once the bootloader starts executing, it bootstraps its second stage from the boot media. The second stage of the bootloader prepares the state of the bootstrap processor (BSP or CPU-0), loads the kernel and initial disk images from the disk into RAM, and calls the startup routine of the monitor.

Monitor code starts its execution on the BSP, which later will be assigned to the first sandbox, i.e., Quest in our case. It then sets up proper paging and memory protection structures in RAM, enables symmetric multiprocessing (SMP), and activates necessary processor features to partition the system hardware resources between our Quest RTOS and Linux sandboxes. Once the partitioning is done, the monitor clones itself to run on processors belonging to other sandboxes, i.e., Linux in this paper. Then each sandbox, in parallel to others, bootstraps its guest kernel. Both kernels initialize their paging structures and CPU features and then initialize their built-in modules. These are indicated by “Setup CPU” and “Initialize Built-in Modules” labels in Figure 4.

Quest only supports built-in modules and uses a ramdisk image loaded by its monitor. Once these are loaded and initialized, the RTOS starts its userspace initialization by launching an initial process. Linux, on the other hand, must also load external modules once the filesystem is operational (at the end of the “Mount Filesystem” block in the figure). This is followed by Linux loading the initial services and scripts from the root partition, and finally presenting the user with a graphical user interface.

During a resume from S3 sequence, different software components only reload hardware features that do not reside in memory (such as CPU or device registers), or re-adjust memory states (such as timers and events) that were affected by suspending the system. This usually does not include accessing disk or any media slower than the main memory, hence, reducing the startup delay. More precisely, upon resuming from the ACPI S3 state, the firmware executes a warm boot procedure (marked as “WB” in Figure 4) that transitions the system into a standard initial state, and calls the waking vector of the software entity responsible for suspending the system to RAM in the first place. We observed in our experiments that the system spends less than one-tenth of the time it would have spent in firmware for a cold boot.

The system entity whose waking vector is called by the firmware is the monitor of the first sandbox (the Quest RTOS), which we call the power master. The power master ensures orderly suspension and resumption of all sandboxes in our vehicle management system. The waking vector of the power master recovers the states of the monitors and signals them to resume their guests. Each guest, in parallel to others, restores their CPU contexts and re-adjusts their timekeeping and scheduling structures to account for the time spent in the suspended state. The guest kernels then recover the last state of their device drivers (the “Resume Modules” blocks), and enqueue tasks and pending scheduling events as part of the “Unfreeze” stage. At this time, only a context switch is required to resume the userspace programs as their process address spaces are already in main memory.

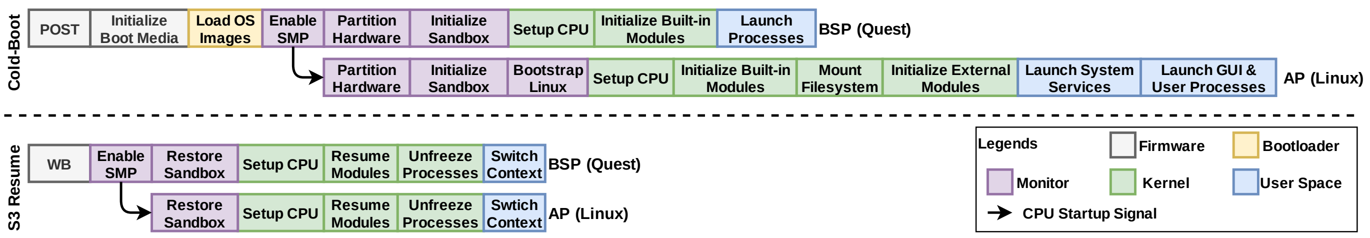


Figure 4. Sequence of events in cold boot and resuming from S3 in our vehicle management system

1) ACPI Virtualization: Unlike consolidating hypervisors that present to guests a customized ACPI interface linked to a virtual device interface model managed by the hypervisor, Jumpstart modifies the view of the guests into the host physical memory. All ACPI features (such as devices and their AML functions) that a guest is authorized to access will be identity-mapped through EPTs to the guest's physical memory space, hence removing the monitor from unnecessary ACPI-related activities at runtime. However, in the case of AML objects and functions related to system-wide power methods and runtime methods of devices allocated to a different sandbox, Jumpstart maps a modified copy of those memory pages to the guest's physical memory. This ensures that a guest cannot see/execute runtime power management of devices on behalf of another sandbox, and only the power master accesses the system-wide power management. In addition to protecting the ACPI memory-resident objects and functions, Jumpstart enumerates the I/O ports (such as those of the EC's or device PCI-PM ports) during boot time, and registers trap handlers in the monitor to prevent unauthorized guests from accessing them. Although Jumpstart's method for ACPI virtualization comes at the cost of a full ACPI and PCI enumeration during a cold boot, it is portable across new hardware platforms and minimizes hypervisor involvement at runtime.

2) Jumpstart Data and Control Flow: Jumpstart power management logic is a collection of userspace APIs, kernel modules, and a hypervisor monitor module. At the userspace level, it provides alternative system calls to perform system-wide power state transitions. Although Jumpstart does not require the guest kernel to natively support the ACPI S3 state, it requires the guest kernel to export a native kernel API to handle system calls, suspend and resume tasks, services, and device drivers. The monitor module then registers a hypercall handler in order to save and restore the state of the guest and the host, before and after power-state transitions of physical CPUs. Finally, the monitor module also adds the proper warm boot procedure to the monitor logic of the hypervisor.

Figure 5 depicts control and data flow using Jumpstart, for a full ACPI S3 suspend and resume cycle. Upon cold boot of our vehicle management system, Jumpstart selects the Quest RTOS [3] as the authorized entity capable of issuing system-wide power state transition requests. However, we would like to allow low-criticality vehicle functions running in Linux

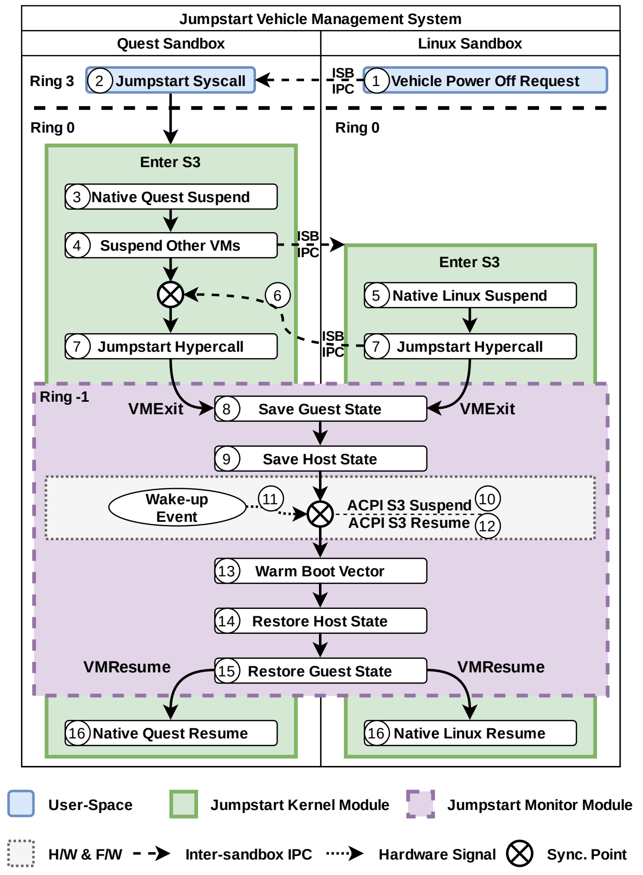


Figure 5. Jumpstart suspend and resume control and data flow

to issue requests for system-wide power state changes. In our case, such a request is required to be initiated via a user interface (e.g., a touchscreen or physical button). This request is then forwarded by Jumpstart to Quest via a secure shared memory channel set up by Quest-V. A power state change request cannot be initiated by software that runs in a sandbox accessible to outside (vehicle) access, except when communication with that sandbox is authenticated. This is to prevent unwanted system suspension attacks.

Now, let us assume that the vehicle's user interface receives a command (e.g., enter park mode) that necessitates the suspension of our vehicle management system into RAM. The Linux software sends the request to Quest using an inter-sandbox shared-memory command. This is marked by the step ① in Figure 5 and the dashed arrow labeled "ISB IPC"

[3]Or a specific RTOS when there is more than one.

(Inter-Sandbox Inter-Process Communication) originating in Linux userspace. ② Quest prepares the vehicle for a power-down and sends a suspend-to-RAM request to the Jumpstart module in the RTOS kernel. ③ Jumpstart uses the native power management of the RTOS to suspend tasks, services, and real-time device drivers. ④ It then sends inter-sandbox messages to the Jumpstart kernel module resident in Linux to perform effectively the same steps, but in Linux (i.e., the step ⑤). This is followed by step ⑥, which ensures both sandboxes have suspended their drivers and processes. ⑦ Then, all CPUs trap into their monitors using a VMCALL instruction to Jumpstart's hypercall handler. ⑧ The VMExit caused by this VMCALL instruction saves the state of each guest CPU into its respective Virtual Machine Control Structure (VMCS) in RAM. ⑨ Then the hypercall handler saves the context of each monitor instance. ⑩ The monitor of the power master (the Quest RTOS) enters S3 and halts, while other monitors just perform an HLT instruction. It is noteworthy that since our partitioning hypervisor does not need to manage devices or tasks on behalf guests, the overheads of saving the context is limited to a few CPU registers used by the monitor logic.

⑪ When the embedded controller receives a wake-up signal, it powers the system on and into the ACPI S0 state. ⑫ The bootstrap processor (BSP) starts executing the warm boot procedure of the firmware. ⑬ The firmware then yields to Jumpstart's "Warm Boot Vector", which in turn restores the context of the power master's monitor (step ⑭), sends initialization inter-processor interrupts (IPIs) to other CPUs, and ⑮ returns from the hypercall handler to the guest kernel using VMResume instructions. Upon reception of the IPIs, other CPUs perform steps ⑭ and ⑮. ⑯ At this point, all CPUs resume the execution of their Jumpstart kernel code that leads to the resumption of devices and tasks managed by the respective guest OS. Algorithm 1 provides the pseudocode of steps ⑧ to ⑮, which every CPU executes in the context of their corresponding monitor's hypercall handler. Symbols prefixed by `isb`, `arch` and `VM` access inter-sandbox objects visible to all monitor instances, perform architecture-specific functions and call VT-x specific instructions, respectively.

3) Time management: One of the challenges of resuming a partitioning hypervisor from RAM is the corruption of each guest's temporal events. When the system warm boots, the timestamp counters of the CPUs are reset to zero. If not addressed, this could lead to the freezing of the guest schedulers until the timestamp counter reaches its last value before the suspension. Normally, in a standalone OS, this is taken care of by the warm boot vector of the OS, which is now bypassed by Jumpstart. Moreover, in a symbiotic system like ours, the Quest RTOS and Linux domains require timely exchanges of data and control flow. Therefore, not only must guest timelines be restored, but they also must be re-synchronized by Jumpstart. For this reason, Jumpstart requires guest kernels to expose routines to update their timekeeping via a delta value. This delta value is derived from the last timestamp counter value before suspension, plus the time spent in the ACPI S3 state. Fortunately, both our RTOS and Linux provide such functionality in their kernel.

Specific to Quest is a real-time pipeline scheduling model [22]. Each pipeline consists of an ordered sequence of tasks with periods and budgets scheduled on sporadic servers according to Rate Monotonic Scheduling [31]. A pipeline represents a vehicle control function with tasks dedicated to reading and processing sensor values, and generating output commands. Each task is organized into two stages: an initialization stage and a processing loop. Once a task ensures that the stream of input data is stable and valid in the first stage, it enters the loop stage, and produces a predetermined number of outputs during each period.

When the system suspends to RAM, each pipeline will have a mixture of raw and processed data in different tasks, which will not correspond to the physical state of the vehicle when a subsequent warm boot (WB) operation is performed, leading to unpredictable behavior. The Jumpstart module in our Quest RTOS addresses this issue by marking pipeline tasks with a reset-on-WB flag and resetting their state (i.e., instruction and stack pointer registers) before performing S2R. Moreover, the replenishment lists of such tasks in the sporadic server of the scheduler are reinitialized during WB to avoid initial deadline misses. Non-pipelined tasks simply continue from where they were suspended.

Algorithm 1 Jumpstart Hypercall Handler
```
if event = FINISHED_COLD_BOOT then
    $Jumpstart.N_{up} \leftarrow |Jumpstart.pcpu|$ // num. of running CPUs
    return
end if
if event = SUSPEND then
    $sb \leftarrow Jumpstart.sandbox[curSB]$
    $\forall G \in sb.guest\_state : sb.saved\_vmcs[curCPU] \leftarrow$VMREAD(G)
    $\forall H \in sb.host\_state : sb.saved\_vmcs[curCPU] \leftarrow$VMREAD(H)
    if $curCPU = Jumpstart.BSP$ then
        // Notify other physical CPUs to hypercall this handler
        $\forall C \in Jumpstart.pcpu \setminus curCPU$ : sendIPI($C$, SUSPEND)
        isbWaitWhile($Jumpstart.N_{up} > 1$)
    end if
    VMXOFF($curCPU$) // Switches virtualization off on the current CPU

    archSaveCPUState($Jumpstart.lowMemBuf[curCPU]$)
    isbAtomicDec($Jumpstart.N_{up}$)
    if $curCPU = Jumpstart.BSP$ then
        acpiRegisterWakingVector($addr\_of$(wbv))
        acpiEnterS3($Jumpstart.wakup\_events$)
    end if
    archCall($HLT$)
    wbv: // Warm Boot Vector
    archRestoreCPUState($Jumpstart.lowMemBuf[curCPU]$)
    if $curCPU = Jumpstart.BSP$ then
        $\forall C \in sb.cpu \setminus curCPU$ : sendStartupIPI($C$, $addr\_of$(wbv))
    end if
    VMXON($curCPU$)
    $\forall H \in sb.host\_state$ : VMWRITE(H, $sb.saved\_vmcs[curCPU]$)
    $\forall G \in sb.guest\_state$ : VMWRITE(G, $sb.saved\_vmcs[curCPU]$)
    isbAtomicInc($Jumpstart.N_{up}$)
end if
return
```

V. Evaluation

We evaluated Jumpstart on a Cincoze DX1100 industrial embedded PC, featuring a 9th generation Intel Core-i7 hexa-core processor. Our platform features two USB3.1 Host Controllers. One is allocated to the Quest RTOS to communicate with the vehicle's sensory inputs and actuation outputs, using a USB-CAN interface. The other host controller is assigned to Linux for USB Bluetooth and infotainment services (e.g., for smartphone integration). Other physical resources were partitioned between Quest and Linux according to Figure 1.

We used four cores with hyperthreading disabled but with VT-x enabled. We also used a specialized Linux based on kernel version 4.19 to run the IC and IVI applications developed by our industrial partner. As our Advanced Driver Assistance System (ADAS) software was not yet implemented in our vehicle management system when performing these experiments, we ran the open-source `opengl-glxcontexts` benchmark to represent a high power demand for the PC. `opengl-glxcontexts` creates additional graphical contexts to fully utilize the DX1100 integrated GPU. Quest acts as the power master and exchanges CAN messages through a Kvaser USBCan Pro 2xHS CAN bus interface connected to a USB3.1 host controller. The following experiments report the power usage of our system in various ACPI S-states, as well as delays involved in switching between these states. Figure 6 summarizes those delays when booting the system using different methods investigated in this paper.

A. Jumpstart Power Consumption

For this experiment, we attached a Keithley DMM6500 digital multimeter to the power supply inputs of the DX1100. The multimeter measured both current and voltage drawn by the PC, from which we could derive power consumption. After an initial boot up, we switched the vehicle management system between running (S0) and sleep states S3 and S5 at 5 minute intervals. Table I reports the power consumption in each state of this test, over a total of 10 iterations.

Table I
Vehicle Management System Power Consumption in Watts of ACPI S0, S3 and S5 States

State/Power(Watts)	Average	Median	Standard Deviation
S0 (Running)	22.250	24.106	4.696
S3 (Suspend-to-RAM)	2.473	2.431	0.225
S5 (Shut Down)	2.643	2.690	0.234

As the results suggest, suspending the system into RAM (S3) achieves as much power-saving as placing the system into the shutdown state (S5). Moreover, we are able to save more power in S3 by disabling all unnecessary wake-up sources such as Ethernet and USB, before transitioning from S0.

Peripheral circuits that control the wakeup capabilities of the machine are either on or off in S5, depending on the profile chosen by the UEFI firmware for the EC. Normally, UEFI firmware allows system administrators to configure what peripherals stay on while the system is in S5. However, in the ACPI S3 state, it is possible for software to control which peripheral circuits maintain power for the purpose of responding to later wakeup events. In our case, the system maintains power to the memory, the EC, and the power button when in S3. However, the UEFI firmware on the DX1100 does not allow a full shutdown of the USB ports in S5, which to the best of our knowledge is why the system consumes slightly less power on average in S3. Note that S5 is in many ways worse that a mechanical off G3 state in this case, as the latter consumes no power but both G3 and S5 require a full system reboot when the system is reactivated.

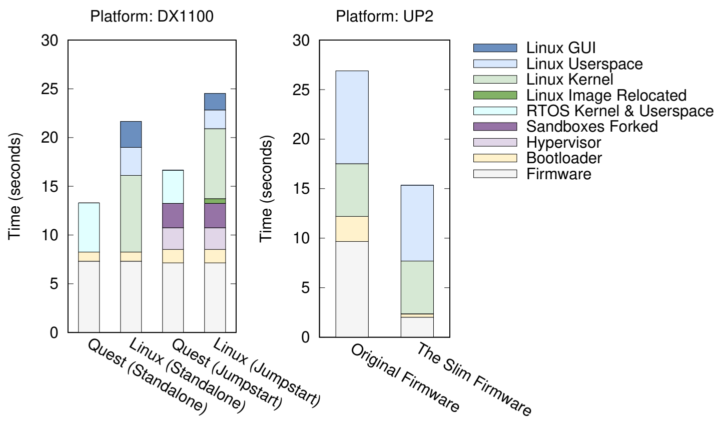
B. Cold Boot Delays

To better understand the impact of different layers - from firmware and bootloader to userspace programs - we configured the DX1100 to run Quest and Linux, both independently and as Jumpstart vehicle management system sandboxes. In each case, we measured the time it takes for the DX1100 to cold boot from the ACPI S5 (shutdown) state to the first moment that Quest or Linux userspace software becomes runnable. For Quest, userspace software is required to handle critical USB-CAN messages, while Linux is required to run IC, IVI and other non-time-critical services. Hence, we measure to the point where the RTOS initiates USB-CAN messaging, and Linux initializes the graphical user interface (GUI).

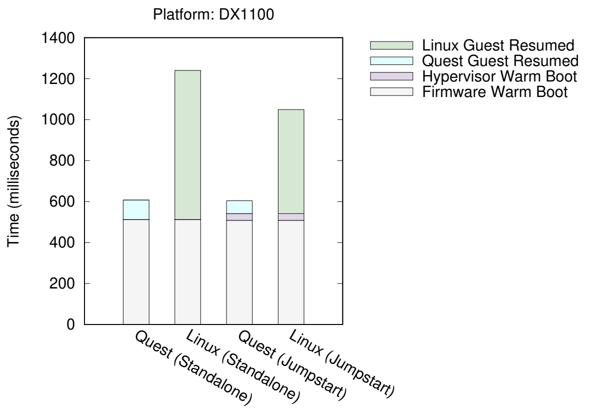
We made sure the bootloader (in our case, GRUB) automatically chooses a test system, which runs a series of scripts to collect measurement data at specific times. The UEFI firmware uses the timestamp counter of the bootstrap processor (BSP), which runs at 2.4 GHz in our case, to report the following times: (1) the start of the UEFI firmware code, (2) the time at which UEFI finishes loading the bootloader, (3) the start time of the bootloader, and (4) the launch time of the corresponding kernel code. We read these values from the ACPI Firmware Performance Data Table (FPDT).

We instrumented Quest and Quest-V to read the CPU timestamp values at various points during the initialization of the hypervisor, kernel, drivers, and userspace programs. As for Linux, we used the `systemd-analyze` tool to gather data regarding the time to initialize the kernel, drivers, userspace, and the graphical user interface.

We also investigated the cold boot delay of the Slim bootloader [4] as an alternative solution that does not rely on ACPI. Slim provides a highly optimized firmware for faster startup. It also embeds a lightweight bootloader in the system's flash memory to eliminate the need for handling a boot partition that is usually different from the partition containing the root filesystem. Unlike Jumpstart, Slim is not portable and supports only a few hardware platforms, among which, we chose UP2 as it exposes cold boot delay times similar to that of DX1100. We ran our experiments on an UP2 platform with both the original firmware and the Slim firmware. Unfortunately, Slim is not fully compliant with the Multiboot standard, which is required for our Jumpstart vehicle management system. Therefore, we only booted Ubilinux 4 on UP2.



(a) Cold boot overheads



(b) Resumption (Warm boot) overheads

Figure 6. Summary of startup delays in all scenarios

Each experimental run consists of one hundred power-on/off cycles for the system under test. We repeated the experiments until the impact of the last one hundred records on the average values fell below 0.1%. Figure 6a illustrates the average latency (in seconds) of each stage for the different configurations. Tables II and III provide the statistics of each stage of cold booting Quest and Linux, both as standalone OSes and as Jumpstart vehicle management system sandboxed guests on the DX1100, respectively. The startup latency of Quest, from the moment the kernel starts execution to the point the first userspace program is launched, is labeled "Kernel to Userspace" in the tables. For Linux, the "Kernel" overhead includes the time to initialize the Linux kernel and all system modules. The "Userspace" delay includes loading (from the root filesystem) and launching initial `systemd` services. Likewise, the "GUI" delay accounts for the time to start the `X11` graphical environment and our vehicle's user interface software. Table IV shows the startup delay of UP2 with and without the Slim firmware.

Table II
COLD BOOT DELAYS OF QUEST AND LINUX AS STANDALONE SYSTEMS (SECONDS)

Stage/Duration (s)	**Average**	**Median**	**Std. Dev.**
Common to Both Configurations			
Firmware	7.317	7.314	0.431
Bootloader	0.935	0.907	0.314
Standalone Quest			
Kernel to Userspace	5.042	5.033	0.126
Total from Power-On	**13.294**	13.278	0.379
Standalone Linux			
Kernel	7.864	7.826	0.135
Userspace	2.879	2.888	0.045
GUI Initialization	2.662	2.737	0.116
Total from Power-On	**21.658**	-	-

Comparing the load time of Quest in the two configurations reported in Tables II and III, we see the added overhead of Quest-V. The cold boot delay of Quest running as a sandboxed guest is 3.373s more than the standalone boot delay (16.667 − 13.294s). Unfortunately, Linux 4.19 does not incorporate the use of ACPI FPDT to report firmware and bootloader delays, and therefore, we could not present the exact total cold boot delay for the standalone Linux. However, it is safe to assume that the delay for the firmware and GRUB to load Linux must be at least the same as that of the RTOS, since Quest has smaller kernel and ramdisk image files. Therefore, there is an average added delay of no more than $2.865s$ (24.523 − 21.658s) when running Linux in our Jumpstart vehicle management system. The added delays

Table III
COLD BOOT DELAYS OF VARIOUS STAGES OF OUR VEHICLE MANAGEMENT SYSTEM (SECONDS)

Stage/Duration (s)	**Average**	**Median**	**Std. Dev.**
Common to Both Sandboxes			
Firmware	7.148	7.101	0.409
Bootloader	1.374	1.333	0.389
Hypervisor Initialization	2.225	2.226	0.001
Sandboxes Forked	2.505	2.506	0.002
Specific to Quest, after Fork and in Parallel to Other Sandboxes			
Kernel to Userspace	3.415	3.416	0.003
Total from Power-On	**16.667**	16.528	0.297
Specific to Linux, after Fork and in Parallel to Other Sandboxes			
Kernel Image Loaded	0.476	0.476	0.001
Kernel	7.181	7.162	0.092
Userspace	1.922	1.916	0.027
GUI Initialization	1.690	1.600	0.025
Total from Power-On	**24.523**	24.422	0.336

Table IV
COLD BOOT DELAYS USING THE UP2 WITH AND WITHOUT SLIM

Stage/Duration (s)	**Average**	**Median**	**Std. Dev.**
Original Firmware with GRUB			
Firmware	9.664	9.659	0.074
Bootloader	2.535	2.448	0.189
Kernel	5.322	5.311	0.384
Userspace	9.370	9.430	0.416
Total from Power-On	**26.891**	26.745	0.455
Slim Firmware and Bootloader			
Firmware	2.023	2.045	0.053
Bootloader	0.345	0.344	0.003
Kernel	5.323	5.261	0.277
Userspace	7.661	7.640	0.273
Total from Power-On	**15.353**	15.293	0.407

caused by Quest-V are due to loading both kernels by the bootloader, partitioning physical resources, and initializing the sandboxes. Finally, in Table IV, we observed 42.9% reduction in the system startup time using Slim on the UP2.

C. Warm Boot Resumption Delays

To show the benefits of Jumpstart, we ran another set of experiments that measure the duration of the resumption path of Quest and Linux. As before, measurements were taken up to the point where the RTOS and Linux start user-level services. These results are then compared with the cold boot delays reported earlier. Each experiment involves a hundred iterations, in which we let the OS run normally (in S0) for five seconds, suspend the system into RAM for five seconds, and collect the statistics. We performed the experiment for each OS, both in the standalone and sandboxed configurations. Each experiment was repeated until the changes made to the average values by the last run of the experiment - i.e., last one hundred power cycles - were less than 0.1%.

Figure 6b shows the overheads of different stages to resume the operation of each system configuration. Tables V and VI show the time it takes to suspend and resume the OSes to and from RAM, respectively, as independent OSes and Jumpstart guest sandboxes.

Table V
SUSPEND AND RESUME DELAYS OF QUEST AND LINUX RUNNING STANDALONE (MILLISECONDS)

Stage/Duration (ms)	Average	Median	Std. Dev.
Common to Both OSes			
Firmware Resume	512.21	512.23	4.40
Quest			
Total Quest Suspend	**5.69**	5.90	0.51
Kernel Resume	32.78	32.81	0.06
Drivers Resume	62.37	62.37	0.01
Userspace Resume	0.59	0.59	0
Total Quest Resume	**607.95**	607.96	4.45
Linux			
Total Linux Suspend	**1406.94**	1414.92	44.19
Kernel Resume	66.63	66.59	0.63
Drivers Resume	654.24	654.24	5.49
Userspace Resume	7.08	5.78	2.45
Total Linux Resume	**1240.16**	-	-

Table VI
JUMPSTART SUSPEND AND RESUME DELAYS (MILLISECONDS)

Stage/Duration (ms)	Average	Median	Std. Dev.
Common to Both Sandboxes			
Total Jumpstart Suspend	**218.80**	190.36	39.94
Firmware Resume	508.21	508.22	3.55
Hypervisor Resume	33.11	33.03	0.07
Specific to Quest: After Hypervisor Resume and in Parallel to Linux			
Quest Guest Resume	62.50	62.50	1.52
Total Quest Resume	**603.64**	603.66	3.65
Specific to Linux: After Hypervisor Resume and in Parallel to Quest			
Linux Guest Resume	507.58	507.23	3.66
Total Jumpstart/Linux Resume	**1048.90**	1048.81	5.10

Comparing Table V against Table II, we see that the startup time of Quest decreases from 13.294s to 0.607s. This is more than 95% decrease in delay and includes firmware latency. Considering only the RTOS itself, the startup time decreases from 5.042s (cold boot) to 0.096s (resume), which is a 98% reduction. This comes at a small 5.69ms cost that we pay to suspend Quest. For Linux, the startup delay decreases by 94%, i.e., from 21.658s to 1.240s at the cost of spending 1.407s to suspend the system to RAM.

To see the impact of Jumpstart, we refer to Tables III and VI. Based on this comparison, we observe that the startup time of our vehicle management system has changed from 24.523s to 1.049s, when the Linux sandbox is finally operational. In this case, Jumpstart reduces the startup time by a factor of more than 23 (conservatively, a factor of more than 20).

A closer look at Tables V and VI reveals that although adding Jumpstart capabilities to the Quest-V hypervisor imposes a negligible overhead of 33.11ms during resumption, the Quest sandbox experiences a delay similar to when it resumes from RAM as a standalone RTOS. The sum of delays to resume the kernel, drivers and the userspace of the standalone RTOS reported in Table V is 95.74ms, while the corresponding delay as a sandboxed guest is 62.50ms. The reason is that our vehicle management system uses the startup routine of Quest up to the point of switching to hypervisor mode, and the RTOS avoids duplicating this stage of initialization.

In the case of Linux, both resumption and suspension times improve when used as a Jumpstart sandbox. This due to fewer number of devices that Linux handles, and also because Jumpstart implements the ACPI S3 power transition of the whole system in a slightly more efficient manner. Our standalone Linux takes 1.407s to suspend, while the full Jumpstart system takes only 0.219s. Again, if we assume firmware resume delays for Linux that are similar to that of the standalone RTOS and Jumpstart, Linux takes about 1.240s to resume from RAM, while Jumpstart takes 1.049s to resume both sandboxes from main memory.

Summary: The take-home message is that a Jumpstart vehicle management system resumes faster than a standalone Linux system and a highly optimized firmware solution such as Slim, despite the underlying hypervisor and additional RTOS. Similarly, critical services related to tasks such as USB-CAN bus initialization complete in Jumpstart's RTOS in just over 600 milliseconds, as shown in Figure 6b. USB-CAN services become operational while less critical IC and IVI graphical software is still being initialized in Linux. As Jumpstart's hypervisor uses the same boot logic as the standalone RTOS, the boot time delay of our RTOS running as a guest and standalone is similar. There are several reasons why suspending and resuming Linux with Jumpstart takes less time than that of the standalone configuration:

1) In a Jumpstart system, some of the devices are managed by our RTOS and not Linux. Due to its lightweight nature, our Quest RTOS handles suspension and resumption more efficiently. Since the RTOS runs independently and in parallel to Linux, the time it spends to initialize such devices will not affect the total system-wide resumption latency.

2) Although the ACPI specification requires OSes to save and restore the `ACPI_NVS` memory regions for hibernation in the ACPI S4 suspend-to-disk state, Linux performs

`nvs_save` and `nvs_restore` for ACPI S3 as well. To the best of our knowledge, this is a work-around for faulty ACPI firmwares. Jumpstart can be configured to perform this extra step only for platforms that require that. The DX1100 PC has a 4MB `ACPI_NVS` region, and fortunately does not require OS help to preserve that when suspending the system to RAM.

3) Some platforms require the ACPI functions `_PTS` (prepare to sleep) and `_WAK` (wake-up) before and after any power transition. These ACPI functions help set the state of the embedded controller within the PC. However, in our experience, many modern platforms, including the DX1100, do not require the invocation of `_PTS` and `_WAK`, and we omit these steps in Jumpstart. Moreover, executing these functions on some platforms such as the DX1100 leads to ACPI runtime errors [15] that must be contained by the kernel.

VI. Related Work

In this section we describe work related to Jumpstart, which encompasses partitioning hypervisors, power management, firmware and bootloaders.

A. Partitioning Hypervisors

Most work on virtualization and power management [27], [29], [47] approaches the problem of suspending a guest while the machine maintains operation. For traditional consolidating hypervisors, which share machine resources among guests, it is often impossible to power down or suspend a shared machine unless all guests are able to suspend at the same time. However, in partitioning hypervisors such as the one used by Jumpstart, all guests cooperate to achieve one system-wide goal. Specifically, they work together to implement a vehicle management system. When a vehicle is not in use, it is possible to suspend all guests and save power.

Jumpstart uses the Quest-V [50] partitioning hypervisor, which is similar to Bao [34], Xtratum [18], ACRN [30], and Jailhouse [42]. As with Quest-V, guests are able to interact with one another through secure shared memory channels. However, Jumpstart's focus is on the use of power management for low latency suspending and resuming of guests, which can be reactivated from a low power state in parallel.

Bao [34] is a static partitioning hypervisor, aimed at mixed-criticality applications requiring spatial and temporal isolation in modern multi-core platforms. It is ported to the ARMv8 and RISC-V platforms. Bao implements ARM's Power State Coordination Interface (PSCI) [1], which is mainly used for runtime power management. As with Quest-V, Bao does not address fast resumption of critical services using power management techniques. However, it implements a relatively small footprint hypervisor and uses page coloring techniques to enable last-level cache partitioning, further isolating tasks in separate guest domains. Results with Bao show full system boot-times to be good but far in excess of the resumption delays using Jumpstart.

Xtratum [18] is another type-1 partitiong hypervisor built for timing and safety-critical aviation applications that need to be ARINC-653 [25] compliant. Onaindia et al [41] extended Xtratum with runtime power monitoring and management to dynamically reduce power consumption of peripherals and CPUs without compromising RTOS guests.

Jailhouse [42] uses Linux to bootstrap a system that provides cells for other guest OSes. Jailhouse relies on Linux runtime power management to switch power states of devices that are not shared by more than one inmate. ACRN [30] functions as both a Type-1 consolidating and a partitioning hypervisor. As a consolidating hypervisor, ACRN uses a special Linux *ServiceOS* to bootstrap other guests referred to as *UserOSes*. ACRN presents every launched UserOS with a fake ACPI firmware and a set of predefined devices that are managed by the ServiceOS. When running as a partitioning hypervisor, ACRN is able to launch a UserOS as a UEFI program, instead of requiring the ServiceOS. In this case, ACRN grants the UserOS direct access to its resources. Although ACRN supports system-wide power management, it is only available when running as a consolidating hypervisor.

For the situations where ACRN is capable of suspending-to-RAM, its ServiceOS notifies all guests via a virtual UART communication device. All guests then save their state and trap into the hypervisor, which performs an ACPI S3 invocation to suspend the system. Upon resumption, the hypervisor starts the ServiceOS, which in turn launches the guests from their previous state. The ServiceOS must start first, as other guests rely on its drivers to access physical devices using ACRN's virtual device interface.

Similar to ACRN, the Xen hypervisor [6] supports suspend-to-RAM in consolidation mode. However, the interaction with the power-management interface of the system is performed by Dom0, rather than the hypervisor itself. The resumption of unprivileged guest OSes is also carried out once Dom0 is completed. As is the case with Jumpstart and ACRN, all guests must support suspend-to-RAM and resume-from-RAM to be able to properly save and restore their state. Dom0 will wait for a specific amount of time after triggering the full system suspension for unprivileged guests to save their state. Otherwise, the system will be suspended, and the unprivileged guests should be launched again after Dom0 resumes.

The QNX micro-kernel [12] supports runtime power management of CPU and peripheral devices. The QNX hypervisor follows the same goal as our work, i.e., consolidating safety and timing critical vehicle programs into a modern PC platform with multiple CPU cores. However, it does not support power management at the hypervisor level.

The COQOS Type-1 Hypervisor [5] is designed for ARMv8 SoCs. Similar to Jumpstart's hypervisor, COQOS supports the orchestration of guest OSes for suspend-to-RAM. COQOS is not open source and lacks published documentation about its implementation or performance details. We are therefore unable to compare it to Jumpstart.

All the hypervisors mentioned above as well as those from Windriver [46] and MentorGraphics [35] are suitable for mixed-criticality application domains. However, none have focused on the problem of mitigating the long startup delays inherent with PC-class hardware.

B. Power Management

Brown and Wysocki [15] provide a detailed view of the suspend-to-RAM implementation in Linux, and discuss ways to improve the suspend and wake up latency in software. Tian et al [47] present an overview of ACPI power management and the implications of using ACPI in virtualized environments for the system software. They also discuss how system software should pass ACPI features through to the guests to implement real power management in such environments. Rush [43] also argues the benefits of using ACPI S3 to reduce the startup time of automotive software using standalone Linux.

Jiang et al [29] identified the challenges posed by virtualization on power management in large-scale server systems that employ consolidating hypervisors. In such environments, hardware resources are shared among different guests through a standard virtual device interface. Policies are then needed to coordinate the suspension of all guests; if one guest remains active and needs machine resources, then the system cannot be placed into a low power state.

Other works have investigated dynamic voltage and frequency scaling (DVFS) [40], device power management [49], and energy-aware real-time scheduling [44]. These techniques provide potential benefit to our work, but they do not address power management in a virtualized environment.

C. Systems, Firmware and Bootloaders

Closely related to Jumpstart is work that aims to mitigate system startup delays. Many such approaches resort to replacement firmware and bootloaders [2], [3], [13], [37].

Minich et al [37] argue that there are at least two and a half kernels between a Linux guest kernel and Intel-based PC hardware. These include a UEFI firmware [48], a System Management Mode (SMM) layer, and a Minix system running as part of Intel's Management Engine. Consequently, multiple layers of device drivers and initialization logic sit beneath a host OS. The authors propose reducing the UEFI ROM image of several megabytes to its bare minimum, disabling SMM or vectoring it to Linux, and then having Linux exclusively performing system initialization. This resultant project is LinuxBoot [2], which was formerly known as the Non-extensible Reduced Firmware (NERF) [3]. The main aim of LinuxBoot is to secure the system from exploitations targeting firmware, by keeping its footprint small. The precursor to LinuxBoot is the Coreboot project [13], which aimed at reducing cold boot latencies of PCs using lightweight open-source firmware. Unlike Jumpstart, these methods are tightly-coupled to the hardware platform they run on.

Cloud computing is another domain that requires fast reboot of virtual machines where short-lived and migratory serverless applications are commonplace. For example, Agache et al present Firecracker [9], a virtual machine monitor aimed at reducing the overheads of launching serverless workloads. Firecracker achieves a startup delay as low as 150ms by reducing the memory footprint of VMs and launching them in the userspace of an already booted Linux. This method is not suitable for our use cases, because it would require the hypervisor to be active at all times to start new guest sandboxes. Keeping the hypervisor running mitigates the energy saving benefits when the system is not in use.

VII. Conclusions and Future Work

This paper presents Jumpstart, a method to quickly resume a suspended PC-class vehicle management system based on a partitioning hypervisor. While it relies on ACPI power management techniques it is, to our knowledge, the first system to report the use of system-wide ACPI S3 capabilities in a partitioning hypervisor. It also does not require modification to the PC-class firmware, although this would potentially reduce the startup delay of Jumpstart further at the cost of limited portability. This paper shows how Jumpstart avoids the costly overheads of cold boot delays on a PC-class machine. These delays result from firmware-based machine initialization, bootloading guest OSes and launching application-specific services. Jumpstart is able to suspend a system comprising an RTOS and Linux, yielding an order of magnitude power reduction when a vehicle is not in use. At the same time, the system is able to restart critical USB-CAN services in the context of the RTOS within several hundred milliseconds. Less critical Linux services that manage graphical tasks are able to execute within approximately 1.0 second of system resumption. Overall, Jumpstart is more than 20 times faster at reactivating a system from a suspended state, compared to a powered off state.

With Jumpstart, a Linux guest is able to resume execution faster than an equivalent host Linux on the same physical machine. This is due in part to the efficiency of Jumpstart's resume-from-RAM logic, and the need for Linux to manage fewer devices when not running as a standalone OS. When running in the context of Jumpstart, some devices are assigned to the RTOS, which is able to wakeup in parallel with Linux. As the RTOS is relatively lightweight, it is able to resume critical services before Linux is fully operational.

Future work will study activation delays using suspend-to-disk (ACPI S4) power management for partitioning hypervisors, and fast non-volatile memory (e.g., Intel Optane) to eliminate all power usage during system suspension.

VIII. Acknowledgements

Thanks to the shepherd and reviewers for their help improving this work, which is funded in part by the National Science Foundation (NSF) 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. Special thanks also to our colleagues at Drako Motors and Celenum, without whose support this work would not be possible.

References

[1] "ARM Power State Coordination Interface," 2021, https://developer.arm.com/documentation/den0022/latest/.
[2] "LinuxBoot," 2021, https://www.linuxboot.org/.
[3] "Non-extensible Reduced Firmware," 2021, https://trmm.net/NERF/.
[4] "Slim Bootloader Project," 2021, https://slimbootloader.github.io/.

[5] "COQOS Automotive Hypervisor," 2022. [Online]. Available: https://www.opensynergy.com/automotive-hypervisor/

[6] "Xen Hypervisor," 2022, https://xenproject.org/.

[7] ACPI, "Advanced Configuration and Power Interface - Ver6.0," April 2015.

[8] ACPICA, "ACPI Component Architecture User Guide and Programmer Reference - Revision 6.2," May 2017.

[9] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[10] A. Avizienis, "The N-version Approach to Fault-tolerant Software," *Software Engineering, IEEE Transactions on*, no. 12, pp. 1491–1501, 1985.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *ACM SIGOPS OSR*, 2003.

[12] BlackBerry, "BlackBerry QNX Hypervisor," 2021, https://blackberry.qnx.com/en/.

[13] A. Borisov, "Coreboot at Your Service!" *Linux Journal*, vol. 2009, no. 186, p. 1, 2009.

[14] T. C. Bressoud and F. B. Schneider, "Hypervisor-based Fault Tolerance," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 1–11, 1995.

[15] A. L. Brown and R. J. Wysocki, "Suspend-to-RAM in Linux," in *Proceedings of the Linux Symposium*, vol. 1, 2008, pp. 39–52.

[16] O. Burkacky, J. Deichmann, G. Doll, and C. Knochenhauer, "Rethinking Car Software and Electronics Architecture," *McKinsey & Company*, 2018.

[17] O. Burkacky, J. Deichmann, and J. P. Stein, "Automotive Software and Electronics 2030: Mapping the Sector's Future Landscape," *McKinsey & Company*, 2019.

[18] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *EDCC*, 2010, pp. 67–72.

[19] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 169–179.

[20] Drako Motors, https://www.drakomotors.com/.

[21] W. J. Fleming, "Overview of Automotive Sensors," *IEEE Sensors Journal*, vol. 1, no. 4, December 2001.

[22] A. Golchin, S. Sinha, and R. West, "Boomerang: Real-Time I/O Meets Legacy Systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 390–402.

[23] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 653–669.

[24] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[25] A. R. Inc., "Avionics Application Software Standard Interface: ARINC Specification 653," 2010.

[26] Intel, "Benefits of ECU Consolidation," 2020.

[27] C. Isci, S. McIntosh, J. Kephart, R. Das, J. Hanson, S. Piper, R. Wolford, T. Brey, R. Kantner, A. Ng, J. Norris, A. Traore, and M. Frissora, "Agile, Efficient Virtualization Power Management with Low-Latency Server Power States," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 96 – 107, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485931

[28] ISO, "ISO 26262-3: Road Vehicles - Functional Safety - Part 3: Concept Phase ," 2011.

[29] C. Jiang, J. Wan, X. Xu, Y. Li, and X. You, "Power Management Challenges in Virtualization Environments," in *Systems and Virtualization Management. Standards and the Cloud*. Springer Berlin Heidelberg, 2010, pp. 1–12.

[30] H. Li, X. Xu, J. Ren, and Y. Dong, "ACRN: A Big Little Hypervisor for IoT Development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019, pp. 31–44.

[31] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[32] M. Liu, L. Rieg, Z. Shao, R. Gu, D. Costanzo, J.-E. Kim, and M.-K. Yoon, "Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation," *POPL*, vol. 4, p. 31, 2020.

[33] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[34] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "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, 2020.

[35] Mentor, "Mentor Embedded Hypervisor," 2021, https://www.mentor.com/embedded-software/hypervisor/.

[36] C. Miller and C. Valasek, "Adventures in Automotive Networks and Control Units," *Def Con*, vol. 21, pp. 260–264, 2013.

[37] R. Minnich, G. shun Lim, R. O'Leary, C. Koch, and X. Chen, "Replace Your Exploit-ridden Firmware with a Linux Kernel," 2017.

[38] PCI, "PCI Express Base Specification - Revision 2.0," December 2006.

[39] PCI-PM, "PCI Bus Power Management Interface Specification, Rev 1.2," 2004.

[40] P. Pillai and K. G. Shin, "Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 89–102.

[41] T. Poggi, P. Onaindia, M. Azkarate-askatsua, K. Grüttner, M. Fakih, S. Peiró, and P. Balbastre, "A Hypervisor Architecture for Low-Power Real-Time Embedded Systems," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 252–259.

[42] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, No VM exits! (Almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[43] S. A. Rush, "Application of Suspend Mode to Automotive ECUs," in *SAE International WCX World Congress Experience*, 2018.

[44] C. Scordino, L. Abeni, and J. Lelli, "Energy-aware Real-time Scheduling in the Linux Kernel," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 601–608.

[45] S. Sinha and R. West, "Towards an Integrated Vehicle Management System in DriveOS," in *Proceedings of the ACM SIGBED International Conference on Embedded Software (EMSOFT). Jointly published in ACM Transactions on Embedded Computing Systems (TECS), Volume 20, Issue 5s, October 2021, Article No.: 82*, October 8-15 2021.

[46] W. R. Systems, "Wind River Hypervisor," 2021, https://www.windriver.com/products/operating-systems/virtualization/.

[47] K. Tian, K. Yu, J. Nakajima, and W. Wang, "How Virtualization makes Power Management Different," in *Linux Symposium*, 2007, p. 205.

[48] UEFI, "Unified Extensible Firmware Interface Forum," 2021, https://uefi.org/specifications.

[49] A. Weissel, B. Beutel, and F. Bellosa, "Cooperative I/O: A Novel I/O Semantics for Energy-aware Applications," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 117–129, 2002.

[50] R. West, Y. Li, E. Missimer, and M. Danish, "A Virtualized Separation Kernel for Mixed-Criticality Systems," *ACM Transactions on Computer Systems*, vol. 34, no. 3, pp. 8:1–8:41, Jun. 2016.

[51] A. Winning, "Number of Automotive ECUs Continues to Rise," May 15, 2019, https://www.eenewsautomotive.com/news/number-automotive-ecus-continues-rise.