JuMP2start: Time-Aware Stop-Start Technology for a Software-Defined Vehicle System

Anam Farrukh
Department of Computer Science, Boston University, MA, USA

Richard West
Department of Computer Science, Boston University, MA, USA

Abstract

Software-defined vehicle (SDV) systems replace traditional ECU architectures with software tasks running on centralized multicore processors in automotive-grade PCs. However, PC boot delays to cold-start an integrated vehicle management system (VMS) are problematic for time-critical functions, which must process sensor and actuator data within specific time bounds.

To tackle this challenge, we present JuMP2start: a time-aware multicore stop-start approach for SDVs. JuMP2start leverages PC-class suspend-to-RAM techniques to capture a system snapshot when the vehicle is stopped. Upon restart, critical services are resumed-from-RAM within order of milliseconds compared to normal cold-start times. This work showcases how JuMP2start manages global suspension and resumption mechanisms for a state-of-the-art *dual-domain* vehicle management system comprising real-time OS (RTOS) and Linux *SMP guests*. JuMP2start models automotive tasks as *continuable* or *restartable* to ensure timing- and safety-critical function pipelines are reactively resumed with low latency, while discarding stale task state. Experiments with the VMS show that critical CAN traffic processing resumes within 500 milliseconds of waking the RTOS guest, and reaches steady-state throughput in under 7ms.

2012 ACM Subject Classification Computer systems organization → Embedded systems; Computer systems organization → Real-time system architecture

Keywords and phrases Time-aware stop-start, Real-time power management, Suspend-to-RAM, Partitioning hypervisor, Vehicle management system, Vehicle-OS, Software-defined vehicles (SDV)

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.1

Funding 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.

Acknowledgements Special thanks are also given to our colleagues at Drako Motors.

1 Introduction

Automotive systems are undergoing a revolutionary move towards software-defined vehicles (SDVs). With SDVs, hardware functions are replaced with software tasks that are consolidated onto a centralized, zonal or domain-based architecture [52]. This approach replaces hundreds of single-core electronic control units (ECUs) with a smaller set of relatively low-cost, multicore processors.

For SDVs to be properly realized there needs to be an appropriate vehicle operating system that supports certifiable functional safety [23], cyber-security [24] and timing predictability. Such a system must provide temporal and spatial isolation between tasks ranging from relatively low criticality instrument cluster (IC) and infotainment services (IVI), right up to highly safety-critical powertrain tasks that control "drive-by-wire" steering, throttle and braking operations.


© Anam Farrukh and Richard West;
licensed under Creative Commons License CC-BY 4.0

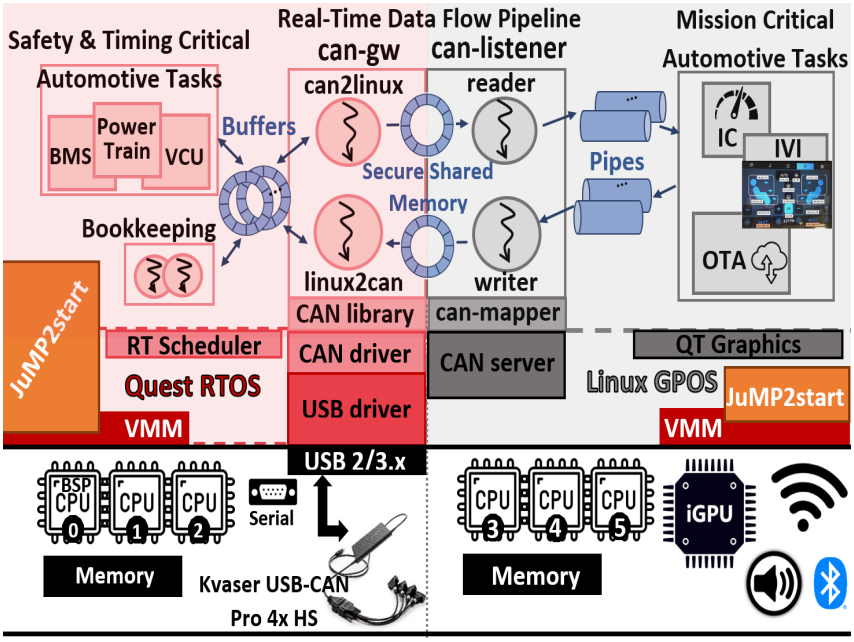


Figure 1 DriveOS™ reference architecture.

One approach proposed by the research community is to adopt isolation via hardware-assisted virtualization, whereby multiple virtual machines (VMs), or OS domains, coexist in a symbiotic relationship on top of a powerful PC-class compute platform. Inter-VM communication is achieved only through explicit and secure shared memory channels. This architecture allows automotive tasks of mixed-criticalities to be mapped between a lightweight, more easily verifiable safety-critical real-time operating system (RTOS) and a mission-critical legacy OS [21,22,63]. The legacy OS provides libraries and services that would take many person years to develop, while a properly isolated RTOS ensures predictable execution of timing-critical tasks.

One such vehicle management system (VMS), depicted in Fig. 1 and further described in Section 2, is designed with this philosophy. A VMS of this complexity requires an equally sophisticated and powerful multicore hardware platform. However, automotive-grade multicore ECUs or PCs incur long boot time delays from firmware [48,68], the bootloader [11,64], OS and device initialization. Optimizations are able to reduce startup latency [6], but the time to boot a PC-class system is nonetheless problematic for repeated power-cycles.

To save energy, a vehicle management system such as the one envisioned above must shut down all but the most basic of services when the vehicle is parked for any length of time. Keeping a PC-class VMS fully operational when a vehicle is not in use may quickly drain battery power [6]. This, in turn, requires batteries to be recharged more often than necessary to ensure a vehicle is able to restart or, in the case of an electric vehicle, have maximum range. Contrarily, shutting down all non-essential VMS services and subsequently cold-booting those system services when the vehicle is restarted incurs too high a delay before vehicle buses (e.g., CAN) are able to distribute control messages, and user interfaces (e.g., the instrument cluster) become operational.

More critically, time-sensitive automotive functions require timely activation upon startup to ensure vehicle safety. In particular, control functions have end-to-end delay constraints to process sensor data needed to make actuation decisions, rendering slow startup times unacceptable.

In light of these challenges, this work envisions a low-latency stop-start approach to suspend the VMS when not in use. A parked vehicle would then only need to consume enough power to keep peripheral circuits alive, such as for a car theft alarm. We note that in contrast to the stop-start system of traditional vehicles, which selectively shuts down vehicle services to save fuel at traffic stops, this work is aimed at addressing system-wide service shutdown and startup in a centralized SDV.

Earlier research work on Jumpstart [6] shows how suspend-to-RAM (S2R) power management techniques, enabled by the Advanced Configuration and Power Interface (ACPI) of modern PCs, can potentially be utilized to mitigate some of the cold boot delays whilst incurring minimal power cost. The first time a vehicle is started (e.g., after a new system installation or update) there is an unavoidable slow "cold boot" delay until the system transitions into a normal working state, S0 [4]. Thereafter, Jumpstart enabled subsequent restarts of the system to complete with low latency by taking advantage of the low power ACPI S3 (S2R) state. The novelty with Jumpstart was how to suspend and quickly resume system-level components comprising a partitioning hypervisor hosting both an RTOS and a single core legacy guest, as opposed to a bare-metal OS.

This work proposes a system called JuMP2start, which addresses several issues not resolved by Jumpstart. First, it handles the most critical issue of resuming software functions for vehicle services that must process timing-sensitive controller area network (CAN) bus-related data. Second, if a safety-critical bus is actively generating traffic while the vehicle management system is resuming, it is paramount that valid data is not lost during the system restart. It is possible to mitigate this problem to some extent by delaying the delivery of bus traffic to a vehicle management system until it has fully resumed operation. However, there is still a third problem: upon resumption, the system must be careful of handling sensitive bus traffic that was buffered at the time of system suspension. The dire consequences of processing stale data are not hard to imagine. For example, an obsolete powertrain CAN message to adjust vehicle speed should be discarded on system resumption.

The contributions of this paper are summarized as follows. First, we present JuMP2start, which addresses the above problems. Second, we introduce the notion of *continuable* and *restartable* tasks, to differentiate between stateful tasks that are able to semantically continue where they left off at the time of system suspension, versus those that must be restarted rather than allowed to process potentially sensitive, stale data. Third, we show the mechanisms by which JuMP2start is able to extend Jumpstart by supporting coordinated suspension and resumption of software-defined automotive functions for a dual-domain vehicle management system featuring an RTOS and legacy Linux guest. Each OS domain within the system manages multiple CPU cores under the symmetric multi-processing (SMP) model. Finally, we provide an empirical study of JuMP2start, integrated with the VMS of an electric car built by Drako Motors [19].

Experiments show that our test VMS is able to resume system-critical CAN traffic processing on a 3-core RTOS within 500 milliseconds, after invoking firmware power management services. In parallel, less critical services running on a 3-core Linux OS are able to receive CAN-related data, to update IVI and IC display readings, in about 2 seconds. Our results show 52.5% improvement when compared to the typical worstcase restoration time of 4 seconds to the last active application screen for an infotainment ECU as reported in [60].

In the next section, we present the design of JuMP2start in the context of a working VMS. This is followed by Section 3, which evaluates JuMP2start for realistic CAN traffic workloads, comprising pipelines of both *continuable* and *restartable* tasks. Related work is described in Section 4, followed by conclusions and future work in Section 5.

2 JuMP2start: A Power Management Framework

JuMP2start is a system-wide stop-start solution for vehicle operating systems running on multicore automotive-grade hardware platforms. It provides a collection of power management (PM) tasks that bind with the vehicle management system, to leverage the underlying scheduling infrastructure and orchestrate system-wide power state transitions.

Table 1 List of DriveOS™ tasks. Class of tasks specified as (R)*estartable* and (C)*ontinuable*.

Quest RTOS Sandbox					
Tasks	**Mode**	**Budget (μs)**	**Freq (Hz)**	**Util (%)**	**Description**
CAN-GW:{CAN2LINUX, LINUX2CAN}	R	200	1000	{20,20}	Read/Write CAN data in user-space
MHYDRA {RX,TX}	R	200	1000	{20,20}	CAN driver scatter/gather
USB XHCI BH	R	10	1000	1	Interrupt handler
SUSPEND	C	200	250	5	JuMP2start main user-space task
APP:PM-THREAD	C	200	1000	20	JuMP2start per application task
KERNEL:PM-THREAD x3	C	100	500	{5,5,5}	JuMP2start per core kernel task
AUTOMOTIVE	R	500	200	10	FORE-GND task(s)
BOOKKEEPING	C	5000	20	10	BACK-GND task(s)
Linux GPOS Sandbox					
PM-MODULE	C	-	-	-	JuMP2start kernel module
CAN-LISTENER	C	200	1000	20	Read/Write CAN data
MISSION APPS	C	SCHED_OTHER		-	(IC), (IVI)

To showcase JuMP2start, we rely on a state-of-the-art VMS called DriveOS™ [63]. DriveOS™ is being developed by Drako Motors, as a centralized system that consolidates automotive functions on a PC-class multicore platform. It currently features a separation-kernel [61] in a dual-sandbox configuration (Refer to Fig. 1) hosted upon the Quest-V [39,72] partitioning hypervisor.

Quest-V leverages hardware-virtualization capabilities of the automotive platform to partition hardware resources, including CPU, memory and I/O devices, between a safety-critical Quest RTOS [18] and Yocto Linux [74]. Each guest OS is assigned a mutually exclusive set of cores within the machine. Hardware-managed extended page tables securely map guest images to separate, non-overlapping regions of host physical memory. Similarly, each guest OS is given direct access to mutually exclusive subsets of I/O devices. Interrupts are delivered to the target OS, bypassing the virtual machine monitor (VMM). Both Quest and Linux manage their partitioned hardware resources without runtime intervention of the local VMM, except for system power management.

A performance monitoring subsystem employing hardware counters provides DriveOS™ with the ability to predict last-level shared cache occupancy [70,71]. Such estimates are then used by static page coloring techniques to partition shared caches between sandboxes [40,73]. Consequently, a guest kernel is isolated from any temporal and spatial interference in its execution by another guest. Additionally, caches are flushed and reloaded at stop-start transitions of the entire VMS.

DriveOS™ guests work in unison to support mixed-criticality automotive tasks according to their timing, safety, and mission criticality requirements. Default load-balancing policies of each OS are used to assign tasks to their respective CPU cores. Table 1 shows a relevant subset of the tasks and their parameters. JuMP2start tasks are distributed between the two guests across all cores as depicted in Fig. 2.

To achieve end-to-end (E2E) drive-by-wire objectives, DriveOS™ features low latency and high bandwidth shared memory communication channels [50,54,63] that enable data-flow task pipelines (Fig. 1), comprising the CAN I/O gateway (CAN-GW task) in Quest and a CAN listener server (CAN-LISTENER task) in Linux, to span the two OS domains [26,27]. The use of more than one OS domain imposes added complexity on the multicore JuMP2start system, which must coordinate power state transitions of each SMP guest OS in a timely and integrated manner.

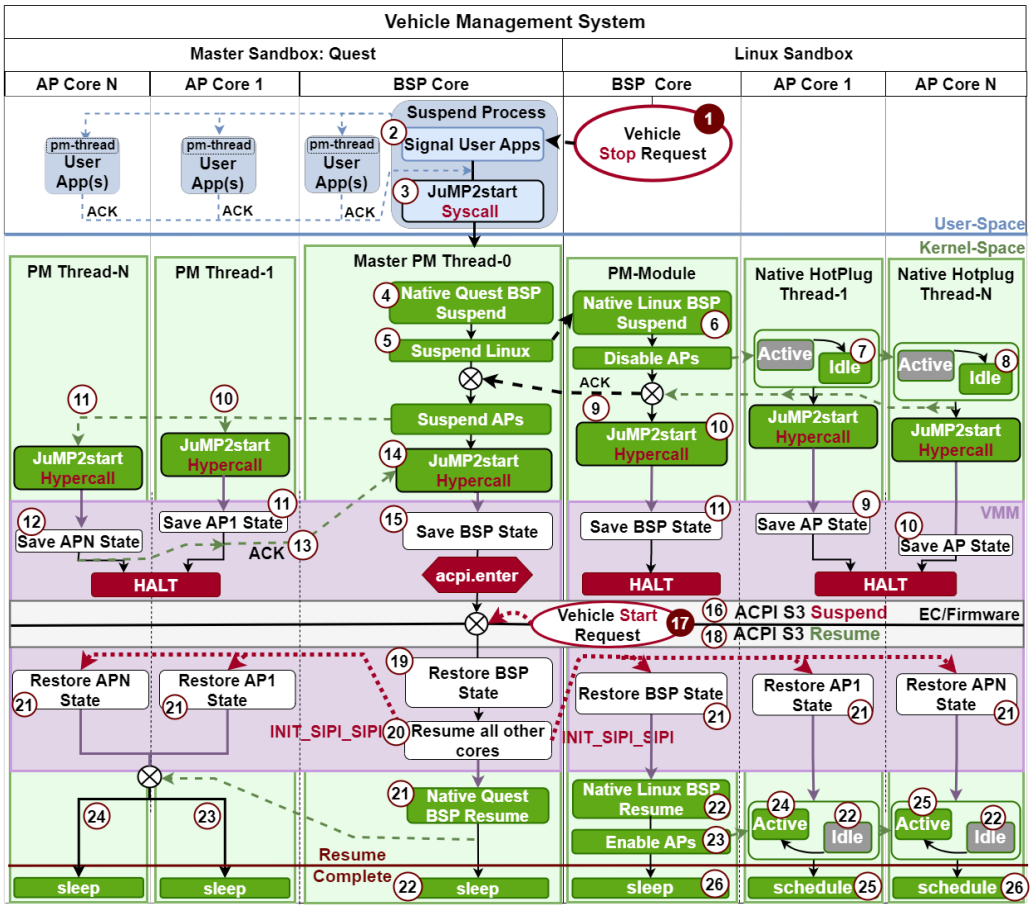


Figure 2 JuMP2start: E2E control flow.

Fig. 2 shows our design of JuMP2start that operates in two orthogonal dimensions of control for the DriveOS™ VMS system stack: (1) *within* each individual guest's *vertical* software stack comprising user-space, kernel and monitor abstraction layers, and (2) *across* the *horizontal* boundaries of the containerized guests, which employs a master-slave model for communicating suspend-resume commands and receiving corresponding acknowledgments to and from each guest. Further details are presented in the next section.

Additionally, JuMP2start appends the notion of *continuable* and *restartable* tasks to the time- and safety-critical task model(s) natively supported within a guest OS. By default, temporal reservation based task models within Quest and Linux incorporate task budgets, periods and deadlines in accordance with a periodic scheduling policy (e.g. VCPU scheduling in Quest [18]) or class (SCHED_DEADLINE [44] in Linux). A periodically dispatched job of a well-behaved task completes its budget execution within the bounds of its period (=

deadline) before the release of the next job. According to this model, all tasks within a guest are implicitly defined as *continuable* across short durations of their respective runtime preemptions. Tasks therefore maintain their semantic and temporal state across context switches. With a properly tuned system, end-to-end task pipelines of sensing, processing and actuation can therefore ensure a continuous unblocked flow of control and data packets with guaranteed throughput and delay bounds. This model works well if the system maintains availability 100% of the time. Once bootstrapped after a coldboot, system resources are therefore guaranteed uptime and accessibility to tasks until shutdown.

Power management events such as ACPI suspend-resume (S2R) are a source of major disruptions to the steady-state execution of the system. Maintaining real-time guarantees as well as functional integrity across such system-level events thus becomes critical to ensuring vehicle safety. Low latency resumption and activation of safety and time critical services is therefore essential. To mitigate safety hazards due to staleness of data and to avoid last remnants of old execution states upon system resumption as well as to ensure end-to-end timing correctness of task pipelines, a flexible task model is required. Such a model would adapt the real-time task behavior according to the system execution environment.

JuMP2start therefore introduces the *restartable* task model, which allows a task to be directly resumed from different checkpoint locations in its execution path. These checkpoints enable such class of tasks to be semantically and temporally reset upon system resumption. Resumption points are configurable and strategically inserted at any point in the sequential execution flow of each task. The restartable model therefore allows programmable control over the task's memory resident state during PM events and facilitates in reducing processing delays of stale data while maximizing data freshness and throughput within the data-flow pipeline. When handling task states during stop-start PM events, it should be noted that JuMP2start flushes and reloads hardware caches. This avoids dealing with stale cached data and eliminates variability in resumption time.

As an example, consider a task that reads steering and yaw angle sensors, as part of a torque vectoring algorithm. If a car is parked and subsequently restarted with stale steering and yaw angle data, unequal torque may be applied to the drive wheels when the vehicle begins moving. This could cause the vehicle to abruptly turn or slide, depending on the road surface traction, leading to a potentially unsafe situation.

A temporal reset of a task discards the outdated temporal context and replenishes it with an updated execution budget and period. This allows the resumed job to maintain real-time schedulability as it completes its execution cycle within the new deadline. As a consequence, throughput characteristics i.e. number of fresh data packets processed and forwarded through the pipeline, are also quickly restored to their steady-state values. We evaluate the impact on CAN I/O throughput for the VMS system in Section 3.1. Under this model, critical automotive functions can therefore adapt to changes in power state and the system runtime environment in an efficient and timely manner. Further details of JuMP2start's task & execution model are presented in Section 2.2.

2.1 System Design

Suspend-to-RAM (top half of Fig. 2 from step (1) to (16)) proceeds in a top-down fashion with the RTOS sandbox acting as a *master* power manager for a time-sensitive suspend-resume operation on behalf of all other sandboxes in the system. Since Quest acts as the I/O signal gateway for the VMS, all stop-start input requests are directly captured by the RTOS. The embedded controller (EC) [20] device for registering subsequent wake-up or resume events is also partitioned to Quest by the Quest-V partitioning hypervisor. The master power manager

therefore incurs minimal temporal cost to trigger system-wide suspension and resumption. In DriveOS™'s dual-sandbox configuration, Linux guest components of JuMP2start, listen for and receive the suspend trigger from the Quest master sandbox (5). Consequently, Linux guest suspension is acknowledged back to Quest (9) before JuMP2start proceeds to save Linux's host machine context and halt its CPU cores.

Quest receives the vehicle "stop" request (1) either via a CAN frame generated on the hardware bus or as is the case for a repeatable lab-test setup, via an echo message generated from the bash terminal. The signal is directly captured in user-space by the suspend process (2), which forwards it to other user-space processes running on each CPU core within Quest. JuMP2start spawns a pm-thread within each process's address space that binds with the underlying real-time VCPU scheduling infrastructure and executes with a distinct budget and period. The pm-threads are dispatched to poll for the suspend signal on behalf of the user application and initiate task cleanup upon receipt. For the CAN-GW example depicted in Fig. 1, this amounts to flushing all internal buffers within the CAN library and completing all pending I/O transactions before freeing up the memory to indicate a CAN bus off state. This freezes any further traffic from being sampled from or sent to the CAN bus.

Once user-space applications are frozen, the suspend task transfers control to the Quest kernel via the JuMP2start system call (3). From then on, suspension proceeds in a largely synchronized manner as indicated by the sequence of enumerated steps in Fig. 2. The green shaded regions belong to kernel level functional components while light purple indicate VM exits to the guest monitor space.

JuMP2start leverages the core PM subsystem [6,12] in each guest to issue native suspend requests to various kernel subsystems. It further extends the native system to enable suspension for multiple application processor (AP) cores for both Quest and Linux sandboxes. Each processing core: X={0,1,...,N}, allocated to Quest, hosts a kernel PM-thread-X, blocked on a wait-queue. PM-thread-0 on the boot-strap processor (BSP) core is the first to wake-up and triggered for execution by the JuMP2start system call. Thereafter, this master PM thread synchronizes fan out of the suspend signal to other guest OSes (5) as well as to other PM threads (PM-thread-{1,2,...,N}) on Quest's own AP cores (10 & 11). Serialized control flow managed by the master PM thread enables a graceful shutdown of the Linux guest and its respective cores. Mission-critical tasks in Linux as well as any active shared memory communication interfaces between Linux and Quest are therefore shutdown before the Quest master PM thread initiates its own low-level core suspension sequence. Quest RTOS features a kernel-wide system lock. JuMP2start leverages this synchronization primitive to disable interrupts and preserve each core's scheduling and task context thereby avoiding corruption of shared kernel state. This also avoids costly overheads of Linux-like kernel thread migrations to the BSP core thus ensuring JuMP2start's operational scalability across AP cores. The simplified design allows each JuMP2start thread to complete its suspension in a predetermined and well-defined order.

The Quest-V partitioning hypervisor allocates a unique per-core virtual machine control (VMCS) context for both Quest and Linux guests within their respective monitor space. The VMCS is a data structure managed by the guest-wide virtual machine monitor or VMM to save and restore core-local context on VM-entry and exit respectively. As part of the suspension sequence, all PM threads within Quest eventually transition from the kernel to the VMM by issuing a JuMP2start hypercall. Upon VM-exit, each thread saves the host state referenced by the VMCS, before issuing a halt instruction.

Similarly in the Linux guest, JuMP2start registers a kernel level module: PM-MODULE, that sets up a worker thread on Linux's BSP core. The module receives the suspend request from Quest in step 5 via intersandbox shared memory communication. At this point,

Linux's native PM-core is signaled to flush all pending transactions to the storage device, synchronize the filesystem state and freeze all user-space and kernel tasks. The PM-MODULE then triggers Linux devices to transition into their respective device low-power states. This is achieved by calling the suspend functions of individual device drivers. The suspension event is then forwarded to Linux's AP cores. The native PM-core heavily relies on the CPU hot-plugging subsystem [41] within the kernel to transition the cores from *active* to *idle* states via a series of sequential intermediate state transitions. We patched the main hot-plugging thread, hosted on each AP core, to issue a VM exit via JuMP2start's hypercall upon reaching the idle state (⑦&⑧). This allows JuMP2start to save the host context of each CPU in Linux in a manner similar to the RTOS guest. The PM-MODULE on Linux's BSP core awaits an acknowledgment from each application processor and then informs the Quest sandbox (⑨) before proceeding to save its own host and processor state via a hypercall to the Linux VMM (⑩&⑪).

We note here that Linux's native PM-core combines multiple ACPI system-sleep states: standby (S1), suspend-to-RAM (S3) and hibernate-to-disk (S4), in a largely unified and standardized implementation. As such, power management in Linux incurs high latency due to needless control flow checks and branch operations that attempt to differentiate different types of power state transitions. Some noteworthy examples include: (1) preserving ACPI non-volatile storage (NVS) memory for disk hibernation, and (2) invoking deprecated ACPI functions such as `prepare-to-sleep` and `wake-up`. Additionally Linux includes a generic set of hardware platform or host drivers for low-level suspend and resume operations. To improve Linux side suspend-resume latency, we therefore optimized JuMP2start's kernel module to bypass any nonessential functionality. JuMP2start primarily focuses on ACPI S3 suspend and resume operations for x86 based platforms, the architecture of choice for the DriveOS™ VMS. Further implementation details are discussed in Section 3 in the context of our experimental setup. Notwithstanding, PM-MODULE's code design empowers system integrators with the capability to (de-)activate different functional hooks that call down into the standard Linux PM-core. Thus JuMP2start enables and maintains reconfiguration of Linux power management for extensions to other ACPI compliant firmwares and platform specific features.

Overall, JuMP2start operates in a hierarchical manner, whereby the master PM sandbox sends a fan-out signal to each guest's respective BSP core, which in-turn handles suspension on behalf of the entire guest including its allocated AP cores, devices and tasks. This enables JuMP2start to independently control and configure PM behavior for each guest OS as well as maintain a tight ordering between the coupled sandboxes.

The master sandbox's BSP core is the last to suspend. It relies on the platform-specific ACPI specification [4] to access special I/O ports of the embedded controller device in-order to switch the machine into a global sleep (S3) state. At this point within the suspension sequence, the DriveOS™ vehicle management system enters an inactive state to conserve power. The only peripherals that retain power are the platform's memory controller, to maintain the stored system context, and the EC, which continuously monitor's the wake-up signal sources for a resume signal.

A subsequent vehicle start event (⑰) initiates system resumption. Resumption proceeds in a bottom-up manner within the system stack from the monitor all the way up to user-space within each guest OS. These steps are enumerated accordingly from ⑱ to ㉖ in Fig. 2. Upon resuming the machine firmware, control is first handed over to the master sandbox's VMM using the warm-boot vector that is pre-registered with the ACPI. This represents the entry point on the resumption pathway within JuMP2start's software stack. Host and

BSP state is restored (19) before PM-thread-0 sends out initialization signal sequences (Intel: INIT-SIPI-SIPI [30]) to all other machine cores (20), including the ones belonging to Linux. Since each guest and its respective subset of CPUs have core-local VMCS context, resumption at the VMM level proceeds in parallel. Thereafter Quest's PM-thread-0 and Linux's PM-MODULE execute a VM-entry into their respective guest kernels to resume kernel context including devices, tasks and the scheduling subsystems (21 & 22). The AP cores for each guest, after restoring their local contexts also exit the monitor space and enter an idle loop to await the resumption signal from the BSPs. In Quest, each PM thread acquires the kernel lock from the master thread and blocks to trigger the VCPU scheduler (23 & 24). For Linux, further steps are required to prime the AP cores for task execution by a series of state transitions of the linear hotplug state-space in reverse order to suspension. PM-MODULE thus calls down into Linux's PM-core (23) to trigger a series of startup callback function calls on each application processor. This also triggers task migration from the BSP core to corresponding AP cores defined in each task's cpuset. Upon reaching the active state, the AP calls the scheduler to begin task execution. PM-MODULE on the BSP cleans up the worker thread and blocks on a sleep loop to await the next suspend trigger. This marks the completion of system resumption for the DriveOS™ VMS. The next section details task suspension and resumption based on the *restartable* task model.

2.2 Task & Execution Model

JuMP2start presents a comprehensive strategy to achieve low exit and startup latencies for SDVs by integrating a novel task execution model with time-sensitive multicore power management. To the best of our knowledge, JuMP2start is the first power management solution to ensure real-time correctness of timing- and safety-critical functions across an ACPI suspend-resume system transition in the context of a vehicle OS. To this effect, JuMP2start introduces a task parameter specifying two different *modes* of operation: *continuable* (C) and *restartable* (R), to the existing real-time task models.

A real-time task, τ_i is characterized by a tuple: $\{C_i, T_i, D_i, p_i, M_i^{resume}\}$, where C_i represents the task's runtime or budget for execution within a window of time period T_i. D_i is the corresponding deadline (assumed to be at the end of the period) by which a job or task instance must complete before the release of the next job. Additionally, a job can be assigned a static priority p_i according to its rate of execution [14] as adopted by Quest's VCU scheduling algorithm or a dynamic priority under Linux's SCHED_DEADLINE. JuMP2start introduces the additional parameter: $M_i^{resume} = \{Restartable\ (R),\ Continuable\ (C)\}$, which represents the behavior and mode of the task upon system resumption. A static mode classification allows JuMP2start to adapt the semantic and temporal behavior for tasks with different state characteristics in a time-aware manner across system power transitions (see Table 1 for such a classification).

According to this model, *stateless* tasks that rely on the most recent copy of input data and current execution state for correctness are *restarted* upon system resumption without loss of functional integrity. Most sensor data sampling, processing and actuation tasks fall into this category. This implies that partially executed context of such tasks just prior to the suspension event stands null and void upon resumption. Outdated execution state would therefore be discarded. Upon resumption, such tasks would necessitate a replenishment of their computation budgets and periods to timely begin a fresh execution cycle. Contrarily, other real-time *stateful* tasks that require preservation of state for correct operation are classified as *continuable*. These tasks must continue on from their last point of preemption

when the system was suspended to avoid data corruption or partial states. Such tasks retain their execution time budgets thus preserving their computation bandwidth across a suspend-resume cycle.

In order to maintain hard real-time schedulability for both classes of tasks after a suspend-resume cycle, the scheduler's temporal reference is reset upon resumption to the warm-boot time of the system. This acts as the new synchronization reference for both restartable and continuable tasks. Task activation times, deadlines and replenishment periods are therefore renewed with respect to the updated temporal reference. Consequently, restartable tasks: τ_r, replenish their execution budgets to the maximum value (C_r^{max}), and update their periods (T_r). Continuable tasks: τ_c, only replenish their periods (T_c) while maintaining their (partially) used budgets: C_c^{remain} from before suspension. Since for any task τ_i, $C_i^{remain} \leq C_i^{max}$, system utilization upon resumption ($U_{sys}^{resume} = \sum_{r=1}^{n} \frac{C_r^{max}}{T_r} + \sum_{c=1}^{m} \frac{C_c^{remain}}{T_c}$), remains within the original schedulability bound of maximum utilization ($U_{sys}^{max} = \sum_{r=1}^{n} \frac{C_r^{max}}{T_r} + \sum_{c=1}^{m} \frac{C_c^{max}}{T_c}$). Despite the inherent discontinuities in system execution states across a suspend-resume cycle, JuMP2start's task model ensures semantic correctness and temporal integrity of individual tasks in a end-to-end manner for an automotive function pipeline.

In our prototype implementation for the DriveOS™ VMS, we integrate JuMP2start's task modes within Quest's timing and safety critical tasks. For the Linux guest's mission critical tasks however, we retain the default continuable behavior scheduled under the SCHED_DEADLINE real-time policy. This enables us to draw a sharp contrast between the two task models across the two guests and showcase benefits of one over the other in the evaluation of our test setup, presented in Section 3.3. We therefore note that all subsequent discussion of the restartable model is presented in the context of the VCPU scheduling subsystem of the Quest sandbox [18,72].

The VCPU abstraction for main periodic real-time tasks in Quest is based on the sporadic server model by Stanovich et. al. [65,66] and scheduled using a rate-monotonic scheduling (RMS) [14,36] policy. A hard real-time task is mapped to a VCPU (sporadic server), which in turn is mapped to a physical CPU (PCPU). Sporadic servers require budget replenishment lists that track each server's consumption of CPU time (replenishment budget: rep_b^i) and when it is eligible to be re-applied to the corresponding server (replenishment time: rep_t^i). Thus each entry in the finite list for task τ_i forms a tuple: (rep_b^i, rep_t^i) thereby allowing fine-grained adjustments to be made to the budget and time of replenishment upon system resumption.

Fig. 3 depicts the control-flow graph for a task as well as the changes to its temporal context under the continuable (blue) and restartable (green) behavior modes. Black arrows in the figure indicate behavior common to both types of tasks. A task's semantic context is generally structured as a sequence of basic blocks in a dual-phase execution. The first stage begins with task setup and runtime context initialization and then segways into the periodic loop stage, which incorporates the main execution workload.

JuMP2start appends a mode check after the setup stage to generate a snapshot of the task's state within its process address space. This aids functional reset in the event of a future resumption and is depicted as "Save Context" in the figure for restartable tasks. The task then proceeds normally with its loop execution until the arrival of the suspend event.

During the suspend sequence (discussed in Section 2.1), JuMP2start's PM threads in Quest suspend the per-core VCPU scheduler. This requires the threads to iterate over the scheduler runqueue to check each task's mode flag: M_i^{resume}. For restartable tasks, the replenishment lists are then purged: $(rep_b^i = 0, rep_t^i = 0)$, while for continuable tasks the

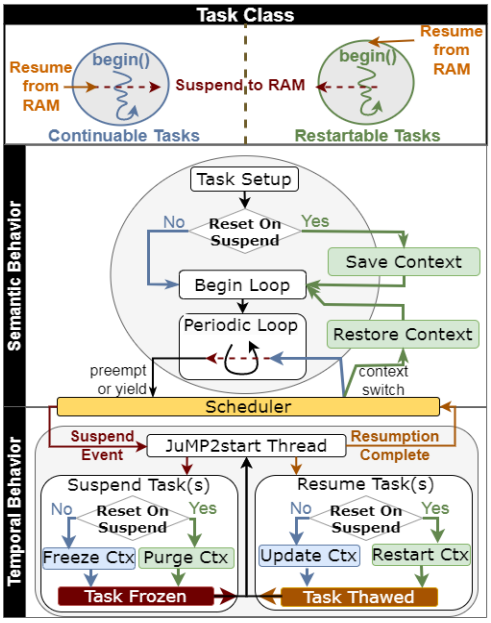


Figure 3 Semantic and temporal aspects of the *restartable* (green) and *continuable* (blue) task model.

lists are frozen in time. Purging the context during suspension saves the costly overhead of zeroing out memory resident lists upon resumption. This ensures minimal resumption latencies for time-critical restartable tasks.

Upon system resumption at time t_{sys}^{resume}, temporal context is renewed for all tasks within the scheduler runqueue. The purged lists for restartable tasks are *rebuilt* with new entries ("Restart Ctx" in Fig. 3). A single replenishment entry is added for each restartable task that allocates maximum execution budget at the time of system resumption: $(rep_b^i = C_i^{max}, rep_t^i = t_{sys}^{resume})$. In effect, the task becomes eligible for a fresh execution with an updated deadline: $D_i = T_i + t_{sys}^{resume}$. The task can begin execution based on its priority order within the runqueue. For continuable tasks, the frozen replenishment lists are thawed by simply *updating* ("Update Ctx" in Fig. 3) the stale replenishment times in already existing entries in the corresponding lists to $rep_t^i = rep_t^i + t_{sys}^{resume}$. This enables the temporal context of the task to be fast-forwarded in time with respect to the warm-boot reference. The task then resumes with a renewed deadline: $D_i = T_i + rep_t^i$, corresponding to the updated replenishment time.

Replenishment budgets (rep_b^i) for continuable tasks are retained across supend-resume boundaries, thereby enabling *continued* execution from their respective preempted semantics when scheduled. In contrast, the functional semantics for restartable tasks are reset to the initial saved context by swapping out the memory resident state from the time of suspension. This resumes the task directly at the beginning of a fresh iteration of its execution loop. Restartable tasks are therefore able to discard old data and timely acquire new inputs for processing. This in-turn improves the automotive function's response time to react to a fresh set of input commands and timely achieve steady-state stable execution after ACPI S2R transitions.

Time is a first class resource for execution correctness of various JuMP2start components spanning the entire VMS stack. Discontinuities in system time as a result of suspension and resumption of hardware clocks and timer peripheral devices can therefore result in

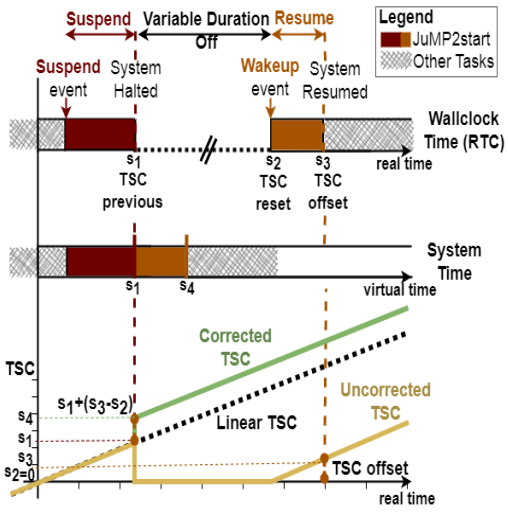


Figure 4 Time-stamp Counter (TSC) correction upon ACPI S3 resumption.

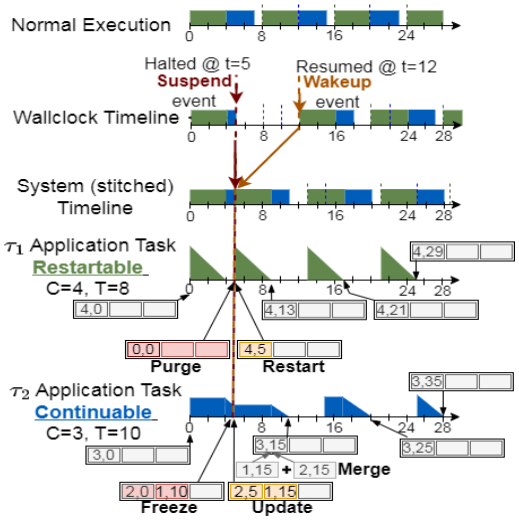


Figure 5 Example schedule for *continuable* and *restarable* tasks in the Quest sandbox.

unpredictable delays and mismatched notion of resumption times between CPU cores of a guest sandbox. In the example of the Quest OS, the warm-boot reference time would potentially diverge for the per-core VCPU scheduler, negatively impacting task replenishments. JuMP2start ensures time-sensitive stop-start of the VMS by maintaining temporal consistency across a suspend-resume cycle with a unified notion of monotonically increasing system up-time in relation to the wall-clock time (RTC clock).

In the event of an ACPI suspend, timekeeping peripherals of the system are powered down and lose all device context. Such is the case for per-core time-stamp counters (TSCs) on x86 based platforms. TSCs are reset and subsequently restarted from zero, effectively making the system go backwards in time upon resumption. This is shown as the uncorrected TSC line graph (yellow) in Fig. 4. For comparison, the system execution timeline according to the uninterrupted wall-clock time (RTC clock) is shown in the top gantt chart of Fig. 4. JuMP2start corrects the TSC (system) time by stitching together values captured before and immediately after suspension as shown in the bottom gantt chart and the green line graph. The correction removes the gaps in time due to system downtime and updates the TSC to reflect a continuous system uptime. JuMP2start therefore ensures accurate accounting of time for various task- and system-level abstractions (e.g. sleep/wakeup schedules, interrupt and signaling subsystem etc.) within the VMS.

An Example Schedule

Fig. 5 depicts the execution timeline of a restartable (τ_1) and continuable task (τ_2) in Quest. Both tasks are bound to distinct VCPUs and scheduled using RMS policy on the same core. The resulting interleaved execution without any system-wide suspend-resume events is shown in the 1^{st} gantt chart: "Normal Execution". The 2^{nd} chart: "Wallclock Timeline", shows ACPI suspend-resume events according to wallclock or RTC time. The system suspends for 7 time units shown as a wide gap in the chart. Finally, suspend and wakeup events are stitched together for corrected TSC values as shown in the 3^{rd} graph: "System (stitched) Timeline)". The graph also shows the corresponding updated execution context of each

task on the corrected system timeline. Dotted vertical lines on each graph represent task deadlines. The bottom two graphs show runtime execution budget: C_i, and the state of the budget replenishment list at different times during execution.

Each list entry is shown as a tuple (rep_b^i, rep_t^i) and is originally set to full capacity at $t = 0$. τ_1 being higher priority, begins execution at $t = 0$ and consumes its entire budget of 4 time units in one stretch. A single replenishment is then posted for τ_1 at $t = 8$, one time period after the task started using its budget. τ_2 then begins execution only to be preempted at $t = 5$ due to arrival of the suspend event. According to the mode of each task, JuMP2start purges all replenishment entries for τ_1 while freezing those of τ_2. As τ_2 blocks, the single entry from $t = 0$ is split, according to Quest's VCPU scheduling policy, into two entries of unused budget of 2 time units and a used budget of 1 time unit. In a normal execution with the system being available 100% of the time, the used budget of 1 time unit would have been replenished in the future at $t = 0 + T_2 = 10$. However, due to the arrival of the suspend event, further execution of τ_2 blocks while the system suspends.

Upon resumption, the high priority restartable task, τ_1, is immediately eligible for execution at the maximum renewed budget. τ_2, being a continuable task, retains its old used and unused budgets as before in its replenishment list. However, the replenishment times are updated for τ_2 to reflect the stitched resumption time ($t = 5$) as the new temporal reference. The updates are shown in orange boxes for both tasks in Fig. 5. τ_2, according to its priority, resumes execution at $t = 9$ and uses up its remaining unused budget of 2 time units by $t = 11$. This marks the completion of τ_2's *continued* job instance. The split entries in the list are therefore merged to reflect total used up budget of 3 time units and a new replenishment time is posted for $t = 5 + T_2 = 15$. This is when τ_2's next job would be released. Thereafter the schedule continues normally according to the VCPU scheduling policy until another suspension event arrives.

The example shows that both types of tasks maintain schedulability across a suspend-resume cycle by completing their resumed instances within the updated deadlines.

3 Evaluation

We evaluate JuMP2start's performance and its impact on throughput and delay characteristics for the multicore vehicle management system as it undergoes multiple stop-start cycles. Our testbed setup is shown in Fig. 6. DriveOS™ VMS is hosted on a Cincoze DX1100 industrial embedded PC that features an Intel Core-i7 hexa-core processor operating at 2.4GHz. We built a hardware-in-the-loop labcar setup with the DX workstation running the latest dual-sandbox configuration. The CPU cores of the DX are equally partitioned to each guest-OS and the task set in Table 1 is load-balanced across the cores. The Quest sandbox acts as the real-time I/O interface for the VMS. For our I/O network, we connect a Kvaser USB-CAN Pro 4xHS transceiver to Quest, exposing two 500kbps CAN channels: CHAN-1 and CHAN-2. Additionally, a remote Linux PC replays CAN traffic over the two channels and receives output from the VMS. The lab setup emulates the behavior of a real-world automotive system, by simulating CAN frames from various peripheral devices such as sensors and actuators connected to the DX host via the two CAN channels. The data captured by the CAN-GW in Quest is tunneled through shared memory to the Yocto Linux (kernel v4.19.80) sandbox and displayed on Instrument Cluster (IC) and In-Vehicle Infotainment (IVI) application screens. For our experiments, JuMP2start leverages ACPI-enabled UEFI firmware, flashed on the DX host, for platform suspension and firmware resumption.

3.1 Throughput Performance

Experimental Setup. To measure the impact on I/O throughput across multiple suspend-resume cycles, we generate a burst of throttle input CAN traffic on the two channels. Data for each channel is sampled by the CAN-GW stack in Quest and then transferred via two shared memory channels to the CAN-LISTENER threads in Linux for processing and display. For a complete round-trip, throttle frames are also echoed back on the output data path from Linux to Quest via a separate set of shared memory channels. Quest then forwards the output data to hardware CAN buses. A CANDUMP shell utility on the remote Linux PC receives and displays run-time logs of the data on each channel.

The CAN-GW stack defines two threads: {CAN2LINUX, LINUX2CAN} per channel for input and output CAN traffic respectively. These are modeled as *restartable* tasks to ensure timely resynchronization with the freshest data on each bus after every suspend-resume cycle. We map CAN2LINUX and LINUX2CAN to different cores within Quest to decouple input and output paths from interleaved executions on the same core. The CAN-LISTENER tasks in Linux are scheduled using SCHED_DEADLINE and rate-matched with the corresponding CAN-GW threads thus ensuring synchronous and non-blocking end-to-end (E2E) communication for the data-flow pipeline.

We set up three observation points within the stack that spans the two guests: (1) CAN input: CAN2LINUX, (2) Processing: CAN-LISTENER, and (3) CAN output: LINUX2CAN. For each task, we measure and record the throughput as number of CAN frames sampled every 1ms over four system execution windows, each lasting for one second. The system is subjected to three ACPI suspend-resume cycles of 5, 10 and 15 second durations. Thus after cold-booting at $t = 0$, a suspend signal is triggered at $t = \{1s, 7s, 18s\}$ with corresponding resume signals sent at $t = \{6s, 17s, 33s\}$ respectively. For an accurate comparison, the total number of CAN frames sampled by DriveOS™ were kept consistent across the two guests for each execution window. Thus all messages sampled by Quest from a CAN channel before the arrival of a suspend trigger would be processed to completion in Linux and sent back as output before the system could be suspended. Any further input data arriving on the bus after the suspend trigger is marked stale by the restartable CAN2LINUX task.

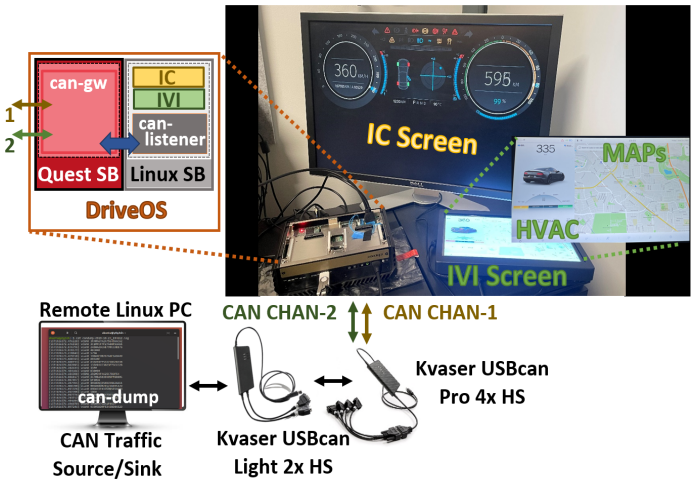


Figure 6 JuMP2start bench-test setup.

Results. Fig. 7 shows transient and steady-state throughput profiles (blue impulses) for each system execution window along the input data path: CAN2LINUX → CAN-LISTENER for CHAN-1. Temporal isolation between the data-flow pipelines of the two channels ensures similar results are achieved for CHAN-2 data. We therefore omit those profile graphs due to space constraints. Tables 2 and 3 summarize the results for both CHAN-1 and CHAN-2 across all four windows. The cumulative total CAN frames sampled by the system before each suspend trigger is overlaid on the Quest throughput profile (Fig. 7), as a discontinuous ramp graph in green. We note that reading from left to right of Fig. 7, all CAN frames sampled in Quest per execution window are transferred to Linux with lossless non-blocking communication. Results show that both Quest and Linux exhibit throughput variations

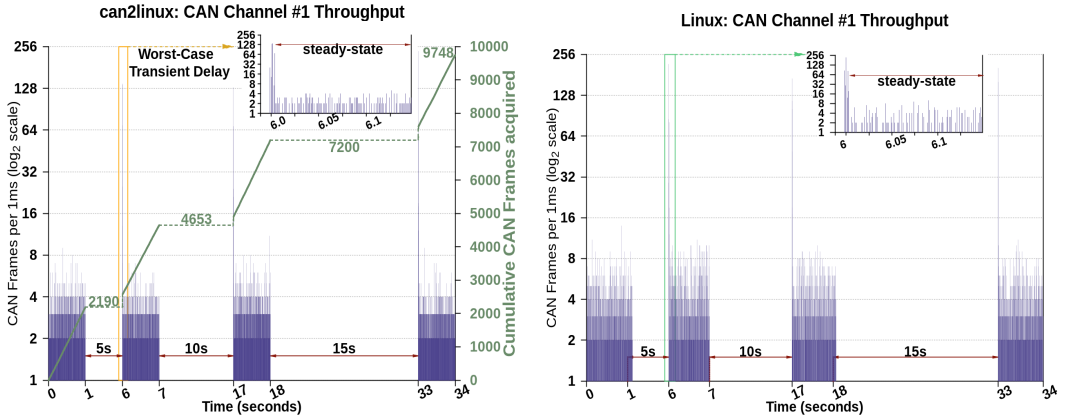


Figure 7 CHAN-1 throughput for 1^{st} observation point: Quest (*left*) and 2^{nd} observation point: Linux (*right*) sandboxes.

Table 2 CAN2LINUX task in Quest.

Quest		Execution Windows			
Steady-State Throughput (frames/ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1	**Min**	0			
	Avg	2			
	Max	9	7	11	9
CHAN-2	**Min**	0			
	Avg	2			
	Max	8	7	10	9
Transient Duration (ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1 | CHAN-2 (**Avg:** 5.33 ms)		-	6	4	6

Table 3 CAN-LISTENER task in Linux.

Linux		Execution Windows			
Steady-State Throughput (frames/ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1	**Min**	0			
	Avg	1	2	1	1
	Max	14	11	10	11
CHAN-2	**Min**	0			
	Avg	2	2	1	2
	Max	15	13	13	13
Transient Duration (ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1 (**Avg:** 5.33 ms)		-	6	7	3
CHAN-2 (**Avg:** 4.67 ms)		-	4	5	5

between min and max frame rates at millisecond granularity However, Linux exhibits a higher frequency of such variations than Quest. Therefore a minimum of 0 frame/s throughput is abound in Linux leading to a sparse impulse profile. This indicates a less predictable frame sampling rate over time in Linux. As a consequence, the CAN-LISTENER task experiences delays in system suspension while it tries to process all the data sent across shared memory by Quest. This is seen in the Linux graph in Fig. 7 as throughput impulses extending beyond the points of suspension. The delay in input data processing and consequent overrun in execution of the CAN-LISTENER task is due to the non-real time nature of the Linux kernel and associated overheads of unbounded system-level interference from device interrupts.

Table 4 Round-trip throughput and duration of transient phase for the two CAN channels measured in LINUX2CAN task in Quest.

Quest		**Execution Windows**			
Steady-State Throughput (frames/ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1	**Min**	0			
	Avg	3			
	Max	35	35	38	30
CHAN-2	**Min**	0			
	Avg	3			
	Max	66	35	24	48
Transient Duration (ms)		1^{st}	2^{nd}	3^{rd}	4^{th}
CHAN-1 (**Avg:** 28.33 ms)		-	27	32	26
CHAN-2 (**Avg:** 23.00 ms)		-	17	24	28

Each graph also shows a magnified view of the worst-case transient phase within an execution window just after system resumption. Before the CAN throughput reaches steady-state, a transient throughput spike is observed. This is because the host controller device and CAN device driver in the Quest kernel resumes before the CAN-GW user-space task. To avoid valid data being lost, the actively generated traffic is buffered as fresh data while the VMS resumes. Upon restart of the CAN2LINUX thread, stale data from before suspension is discarded and the newly cached data is processed in bursts yielding throughput spikes.

Due to the synchronous nature of the data pipeline between the two guests, Linux also shows a similar spike upon resumption (green box in Fig. 7). However, due to the lightweight nature of Quest, the CAN-GW resumes before CAN-LISTENER in Linux. Refer to Section 3.2 for a detailed latency analysis. This mismatch in resumption latencies causes additional backlog of freshly sampled messages sent to the inter-sandbox shared memory upon resumption. Compounded with added processing delays, Linux thus exhibits higher throughput spikes that sometimes exceed Quest's maximum frame rate. In this case, the CAN-LISTENER task tries to catch up to Quest to process all the backlogged frames in the inter-guest shared-memory.

Overshoots in the CAN throughput die down quickly however with transient durations recorded for each execution window in Tables 2 and 3 for Quest and Linux respectively. Worst-case settling time for both guests is also highlighted. Timely transition to steady-state CAN throughput is essential for system safety. JuMP2start's time-sensitive resumption ensures that steady-state CAN throughput is achieved in under 7ms for the input pipeline. In particular, the restartable task mode enables CAN2LINUX task to discard any stale data upon resumption thereby preventing needless processing of such frames along the input path.

Table 4 shows the throughput summary for the complete round-trip data path: CAN2LINUX $\rightarrow$ CAN-LISTENER $\rightarrow$ LINUX2CAN recorded at the 3^{rd} observation point. LINUX2CAN acts as a data concentrator for both the echoed back throttle traffic from Linux as well as the newly generated output traffic from the infotainment applications. Longer transient durations and higher maximum throughput values compared to the input path are observed in steady-state across all execution windows. These can be attributed to the bursty nature of traffic generation by Linux's application tasks and variability in task dispatch between different scheduling classes. This inturn impacts availability of output data in shared memory to Quest. Furthermore the default continuable nature of the CAN-LISTENER task results in Linux retaining some stale data in its buffers if generated by Linux applications during suspension. Upon resumption, Linux sends the data to shared memory as valid output CAN frames. Consequently LINUX2CAN task in Quest processes the forwarded stale data, which contributes towards longer lasting transient durations before reaching steady-state output frame rates. This further motivates the potential benefits of JuMP2start's restartable task model over the continuable mode of execution.

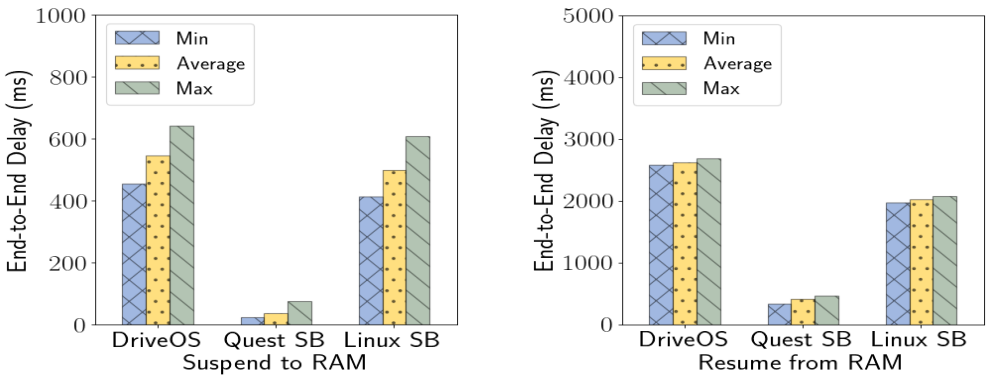


Figure 8 E2E latencies for JuMP2start.

3.2 JuMP2start: End-to-End Delay Performance

Experimental Setup. We measured overheads of suspension and resumption as JuMP2start transitions the DriveOS™ VMS between the two ACPI power states: S0 (System Working) ↔ S3 (Suspend-to-RAM). To ensure that timestamp counters (TSCs) remain consistent across all CPU cores of the DX platform, we disabled dynamic performance scaling for each guest sandbox and fixed the operating frequency per core to 2.4GHz. For our experiments, the dual-sandbox system is allowed to run in normal execution mode (ACPI S0) for 10 seconds before a suspend signal triggers the system to transition to ACPI S3 state.

The resumption signal is delivered 25s later by the platform's real-time clock (RTC), which is programmed to be a wake-up source. Once the UEFI firmware resumes, control is handed over to the Quest master sandbox, which begins to transition the system back to S0 state. Resumption is marked complete when user-space applications in both sandboxes are operational and CAN packet communication in the data-flow pipeline: CAN-GW↔CAN-LISTENER, is restored.

Latency data is collected for the end-to-end system, individual guests, and various JuMP2start components (enumerated in Fig. 2) over several consecutive system suspend-resume cycles. We note that our results, presented in the next section, not only complement the original Jumpstart results for native Quest and Linux BSPs but extend them by providing additional insights into the worst-case overhead cost associated with suspending and resuming the AP cores in addition to BSPs for each guest within our VMS.

Results. Fig. 8 shows the end-to-end delays incurred by JuMP2start threads in each sandbox as well as collectively for the entire DriveOS™ system. The bars correspond to the *critical path* that incurs the longest delay in each case. A detailed breakdown across the different abstraction layers of each guest's PM software stack is presented in Table 5 and Table 6.

For Linux guest suspension, we present the cumulative delay for the full vertical guest stack since user-space applications are frozen along with the kernel-level threads within JuMP2start's kernel module. We observe that the delay far exceeds that of Quest's guest local delays and therefore contributes a major percentage (94.7% in the worst-case) to the total E2E suspension latency of DriveOS™. The difference arises because the RTOS kernel hosts a minimal number of optimised services to support timing- and safety-critical tasks compared to Linux that hosts complex mission-critical and user-interactive graphical applications. We also note that for our DX platform, the default UEFI firmware only reports performance

Table 5 Suspend delays for JuMP2start in DriveOS™ (milliseconds).

Stage/Duration (ms)	Minimum	Average	Maximum
E2E DriveOS™ Suspend	454.306	544.983	640.963
E2E Quest Suspend	22.198	36.216	74.864
Quest Userspace Suspend	6.483	20.306	58.48
Quest Kernel + Hypervisor Suspend	14.999	15.909	16.677
E2E Linux Suspend (Full Guest Stack)	411.573	497.032	607.585

Table 6 Resume delays for JuMP2start in DriveOS™ (milliseconds).

Stage/Duration (ms)	Minimum	Average	Maximum
E2E JuMP2start Resume (System Critical Path)	2,578.83	2,622.00	2,677.14
Common to **Both** Sandboxes			
Firmware	536.47	540.74	546.33
Quest Guest: Parallel to Linux			
Monitor	53.23	53.44	53.52
Kernel	104.29	104.32	104.33
Userspace application: CAN-GW	158.58	236.56	297.98
E2E Quest Guest Resume (Warmboot→can-gw)	328.32	406.11	467.67
Linux Guest: Parallel to Quest			
Monitor	20.99	21.08	21.11
Kernel + Userspace	1,954.02	1,997.65	2,050.25
Core Kernel	23.18	23.27	23.31
Devices	1,926.32	1,965.98	2,018.23
Userspace application: CAN-LISTENER	0.67	6.35	44.67
E2E Linux Guest Resume (Warmboot→can-listener)	1,975.10	2,018.72	2,071.33

data for ACPI S3 resume. We therefore record the latency of firmware resumption common to both guest VMs in Table 6. For the rest of system resumption, we distinguish between each system management layer within Quest and Linux.

The pipeline comprising the CAN-GW and CAN-LISTENER tasks is the last to resume. The critical path in each guest therefore accounts for the delay measured from the point of warmboot (WB)/wake-up of the local guest monitor to the time the first input CAN message is successfully sampled in Quest and Linux respectively. This is represented as Warmboot→{CAN-GW, CAN-LISTENER} in Table 6.

Quest's entire stack undergoes suspension and resumption in well under 550ms in the worst-case. Linux on the other hand takes 2.7s to complete the full cycle. Due to the lightweight nature of the RTOS, Quest handles a suspend-resume cycle more efficiently while Linux incurs a larger latency. For Linux, JuMP2start reuses builtin functionality of the native PM-core subsystem, whose kernel-level routines are by default scheduled under non real-time scheduling classes.

Although we simplify JuMP2start's design by leveraging predefined power-management methods within Linux, it comes at the cost of added delay. The native PM subsystem utilizes Linux's hotplugging infrastructure for suspending and resuming its application processor cores. As a consequence, threads running on those cores are first migrated to and from Linux's BSP core before being frozen or thawed respectively in their execution. This incurs thread migration delays. Additionally, multi-state transitions of hotplug states on each AP core, filesystem synchronization and power management of devices comprising a complex hierarchy of buses and device classes, lead to further delays.

In our design, we discovered that these latencies are unavoidable in Linux due to a tight interdependence between various kernel subsystems that require a deterministic order of power-management events. However, as stated in Section 2.1, JuMP2start's design optimizes

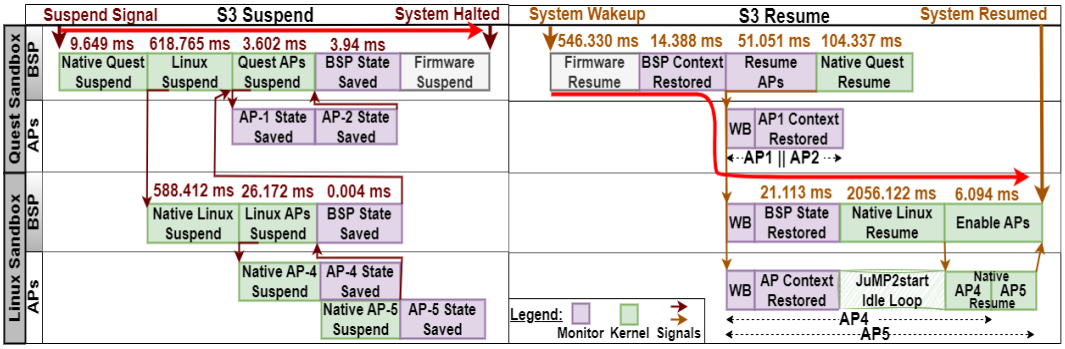


Figure 9 JuMP2start critical path for suspension (*left*) and Resumption (*right*) with functional component latencies.

out needless saving and restoring of ACPI non-volatile memory regions, which are otherwise auto preserved by the platform's firmware. We also avoid forced invocations of various deprecated methods that are retained by Linux for backward compatibility. Nevertheless, compared to Quest, Linux retains a larger and more complex code base to host multiple kernel level services, which necessitate synchronization before and after every power transition. The RTOS guest on the other hand only supports a single VCPU scheduling class and a smaller subset of I/O devices. Quest also operates under a kernel-wide locking mechanism, which makes thread migration completely unnecessary.

The critical path for the E2E DriveOS™ system comprising both guests differs between suspension and resumption. Fig. 9 marks the path in red arrows for each case with the worst-case delays annotated on respective functional components of JuMP2start (Refer to Fig. 2). Since suspension is serially orchestrated by the master power manager, DriveOS™'s suspension latency therefore includes the time between the arrival of a system-wide suspend event and when the master Quest sandbox issues ACPI suspend-to-RAM method for the final transition to S3 state.

Upon receiving the wake-up event, the embedded controller hardware resumes the ACPI firmware and then begins the warmboot procedure for Quest's BSP. Once startup inter-processor interrupt sequences are sent to all AP cores, resumption for each guest proceeds in parallel. Since Linux takes longer to resume, the majority of the resumption critical path latency is contributed by the Linux guest. The E2E suspend-resume path then becomes a simple sum of all critical latencies as summarized in Fig. 8. Compared to cold-boot latencies of 16.7s (Quest) and 24.52s (Linux) for the single core version of DriveOS™ [6], JuMP2start massively cuts down on the startup delays for a multicore VMS to just under 2.7s.

Furthermore, in a series of preliminary experiments, not shown for brevity, we observed a minimal delta increase of 6μs to save an AP core's state when scaling up from a single BSP core to 3-cores in Quest. The majority of the Quest AP suspension overhead results from the kernel-wide locking mechanism, a necessary built-in feature of the Quest kernel. JuMP2start leverages this synchronization primitive to maintain execution correctness during suspension. For the Linux guest however, increasing CPU count from 1 to 3 incurs a much higher overhead. This is measured in the PM module on the Linux BSP to be 17.387ms for a single core and 26.172ms for 3 cores. A linear projection would indicate an $\approx 9ms$ increase in cost per AP core added thereafter. This delay originates within the native Linux PM-core as each AP traverses the state-space of the hotplugging subsystem.

3.3 JuMP2start: Restartable v/s Continuable Task Performance

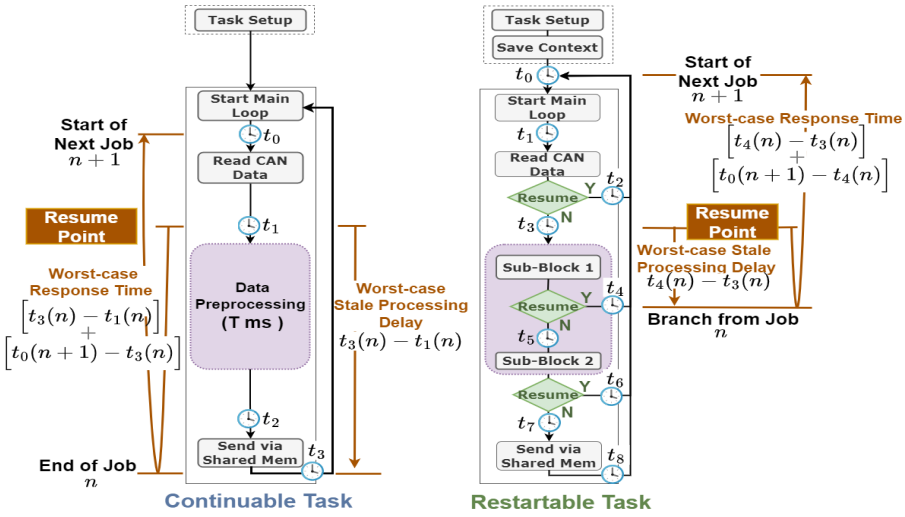


Figure 10 CAN2LINUX modeled as a *restartable* and *continuable* task.

Experimental Setup. To showcase the benefits of JuMP2start's task model, we compared worst-case response time performance of *restartable* tasks against their *continuable* behavior across multiple suspend-resume cycles. For our setup, we toggled the task mode ($M_i^{resume} = \{\text{R}, \text{C}\}$) for the CAN2LINUX thread in Quest's CAN-GW task. Fig. 10 shows the thread's control flow graph under the two task modes. CAN2LINUX periodically acquires and pre-processes raw CAN data from the hardware bus and communicates it to Linux's CAN-LISTENER task across shared memory for further processing.

During a suspend-resume cycle, the task would freeze within its main loop and resume from this last point of preemption. For the continuable case, it would then continue processing the old CAN data until it commits to shared memory at the end of the loop. For the restartable case, checkpoints after each basic block of straight-line code allow the task to discard any stale data within the sampling pipeline that was acquired before suspension.

To highlight the differences between the two models, we vary the pre-processing duration within the body of the main loop in the range $T = 1 : 1000$ ms. These values signify data processing delays corresponding to safety-critical automotive control functions within the real vehicle. We measured the timestamps (t_x) at each functional stage to determine the worst-case duration for processing stale data and the corresponding response time of the task to reset its execution upon resumption. The experiment was repeated over 50 suspend-resume cycles and the results are presented in Fig. 11.

We also determined the number of stale messages processed by the continuable CAN-LISTENER task in Linux. For these measurements, CAN2LINUX was fixed to restartable task semantics to ensure that Quest does not forward any stale data to Linux after resumption. This allowed us to isolate any unprocessed data committed to Linux during suspension. The experiment aims to highlight the differences between the two models across suspend-resume boundaries. To track data flow from Quest to Linux, we tagged each CAN frame with a unique sample-id and counted the number of messages committed to shared memory during suspension. Fig. 12 shows the results averaged over consecutive ACPI state transitions.

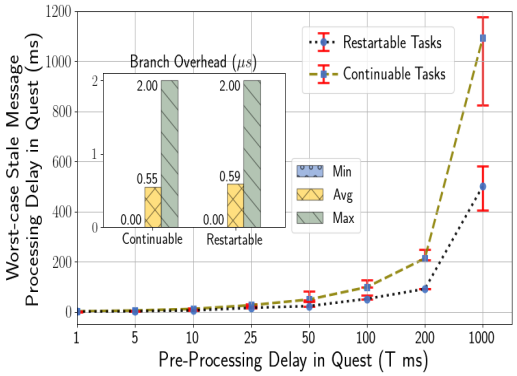
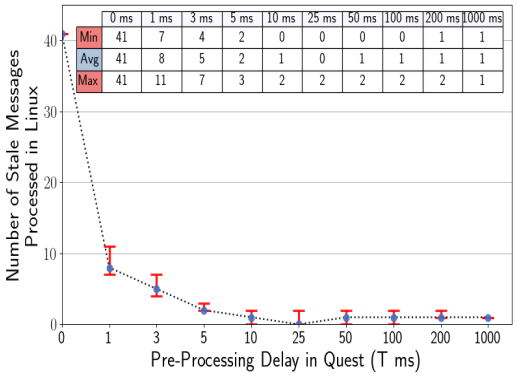


Figure 11 Stale message processing in Quest across system suspend-resume.



	0 ms	1 ms	3 ms	5 ms	10 ms	25 ms	50 ms	100 ms	200 ms	1000 ms
Min	41	7	4	2	0	0	0	0	1	1
Avg	41	8	5	2	1	0	1	1	1	1
Max	41	11	7	3	2	2	2	2	2	1

Figure 12 Number of stale messages in Linux across system suspend-resume.

Results. The line graphs in Fig. 11 represent the average worst-case stale data processing delay incurred by the CAN2LINUX task. An increase in pre-processing delays results in the two graphs diverging. Even with the course-grained user-space implementation of resumption checks, the restartable mode performs 50% better on average (40% in the worst-case) than the continuable case. We note that stale message processing delays can be made negligible by implementing the checkpoints within the kernel, allowing the task to resume directly at the beginning of the loop. However for proof-of-concept and ease of measurement, only user-space checkpoints are used.

The inset bar chart shows the negligible branching overhead to start the next iteration of the main loop indicating little to no additional overhead for a semantic reset of restartable tasks upon resumption. The restartable mode thus avoids propagating stale data within the system and timely samples fresh input data.

In Fig. 12 the number of stale messages committed to shared memory while each guest suspends, show a decreasing trend with increasing Quest pre-processing delay. At the zero point, the shared memory buffer fills to capacity (41 messages) almost immediately. However, for delays beyond 10 ms, the number drops to 2 stale messages processed in the worst-case. For mission-critical tasks in Linux, this means the IC display would show outdated instrument data from before suspension. To ensure end-to-end data consistency and synchronicity as well as safety and timing correctness across vehicle stop-start cycles, our results motivate the shift from continuable to a restartable task model for Linux's time-sensitive tasks.

4 Related Work

JuMP2start is motivated by prior work on fast, critical service resumption [6] for partitioning hypervisors [38,39,47,56]. Leveraging previously proposed techniques for ACPI power management [4], JuMP2start is applied to an SDV system [7,25,55] comprising an RTOS and a legacy OS. Our approach differs from that in Jumpstart [6] by extending all its guests to support multicore suspension and resumption. The use of multiple cores across all guests is important for consolidating many vehicle functions as software tasks on the same platform.

System-level restarts and cold-boot optimisations have been extensively explored in the context of fault-tolerance for simplex-based controller architectures in real-time safety-critical systems [1,2,8,9]. Fast recovery times are ensured by either reactively [3,32] or proactively [15,17] reinitializing system states. However, such approaches often modify the

firmware and bootloaders to reduce restart time for the RTOS and applications running on single-core processors or microcontroller platforms [3,29]. Although JuMP2start's suspend-to-RAM approach does not preclude firmware optimisations [6], this work primarily focuses on a multicore stop-start technique that spans the hypervisor and virtualized OS domains of a complex software-defined VMS. Additionally, JuMP2start differentiates between continuable and restartable tasks, performing appropriate time management and selective reinitialization to avoid processing stale data on system resumption. This contrasts with the cold restart approach, which always loads a fresh image of the software stack from an initial clean state.

Unlike runtime rejuvenation strategies that periodically rollback software components to protect against faults [58,59], JuMP2start asynchronously suspends and resumes an entire VMS whilst ensuring state coherency, functional safety and real-time schedulability of stateful and stateless tasks. Notwithstanding, the JuMP2start model offers an opportunity to protect safety- and timing-critical automotive systems against faults with fast service deactivation and recovery. We leave further exploration of this feature for our future work.

Rush from General Motors Corp. [60] explores Linux's suspend-to-RAM mechanism for meeting startup performance requirements of an Infortainment ECU. According to the author, rising software complexity leads to long cold startup times between 10-45 seconds thereby exhibiting sluggish performance. This significantly degrades end-user experience and dampens expectations. With the availability of suspend-capable automotive-grade microprocessors, high robustness and quality can be achieved by initializing operating systems from low-power states as quickly as 4 seconds in the worst case. Unlike JuMP2start, the work focuses on improving reactivation latency through ACPI supend-resume for less critical automotive functions that have no timing or safety requirements. The proposed PM model is also restricted to a standalone ECU managed entirely by Linux, as opposed to a centralized multicore hardware platform under the control of a multi-domain vehicle OS. In contrast, JuMP2start coordinates power-management across multiple guest OSes. It builds on techniques for Quest and Linux power management [12,42,43,45,49,51] to reduce the latency of multi-OS suspension and resumption.

Virtualized power management [16,31,33] reduces the overheads of launching and migrating serverless workloads in the cloud [5,34] and data-centers [10,46]. However, this is restricted to virtual machines or guest user-space applications and has no effect on real power states of the host system. In contrast, JuMP2start supports physical machine power management [67] through full-stack suspension of each guest OS and its locally replicated VMM.

Other research efforts have investigated multicore power management techniques that range from dynamic voltage and frequency scaling [28,37,57] to energy-aware scheduling [62,75], and device power management [35,69] in the general computing domains. The increasing complexity of SDVs [13,52,53] requires more careful consideration of how to minimize power usage during the operation of a vehicle management system.

5 Conclusion & Future Work

JuMP2start is a time-aware stop-start solution that leverages ACPI power state transitions to reduce the startup latency of mixed-criticality automotive functions for SDV systems. This work describes JuMP2start's implementation in the context of the DriveOS™ VMS, which combines an RTOS with a legacy OS. DriveOS™ replaces separate ECUs with software functions supported on a multicore industrial PC. JuMP2start mitigates cold-boot delays of PC-based systems by transitioning the VMS into and out of suspend-to-RAM sleep states when a vehicle is stopped and restarted.

JuMP2start introduces a novel restartable task model to ensure semantic and temporal correctness for the VMS tasks across suspend-resume transitions. *Restartable* tasks avoid stale data processing on resumption, and are correctly re-initialized with appropriate time budgets. *Continuable* tasks, in contrast, are able to resume execution from where they are suspended without re-initializing their budgets. JuMP2start is shown to resume critical CAN traffic within 500ms after invoking firmware power management services, while low-criticality IC readings are updated within a further 2s.

The resumption times of JuMP2start are far closer to traditional ECU startup delays, compared to PC-class cold boot latencies. JuMP2start is operational in a road-legal electric vehicle, which has been configured to activate CAN buses only after the VMS has fully resumed critical services. Delaying the activation of CAN traffic until approximately 500ms after invoking firmware power management services, ensures that no stray messages are on any vehicle buses while the system is still resuming operation.

Future work will investigate dynamic power management in multi-domain vehicle management systems. The aim is to minimize total machine power while ensuring task pipelines spanning multiple guests achieve their timing and functional objectives.

References

1 Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco Caccamo. Reset-based Recovery for Real-time Cyber-physical Systems with Temporal Safety Constraints. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2016.

2 Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. Guaranteed Physical Security with Restart-Based Design for Cyber-Physical Systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 10–21, 2018.

3 Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. Application and System-Level Software Fault Tolerance through Full System Restarts. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 197–206, 2017.

4 ACPI. Advanced Configuration and Power Interface – Ver6.0, April 2015.

5 Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. URL: `https://www.usenix.org/conference/nsdi20/presentation/agache`.

6 Ahmad. Golchin and Richard. West. Jumpstart: Fast Critical Service Resumption for a Partitioning Hypervisor in Embedded Systems. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–67. IEEE, 2022.

7 ARM Automotive: Software-Defined Vehicles. Accessed May. 2024. URL: `https://www.arm.com/markets/automotive/software-defined-vehicles`.

8 Miguel A. Arroyo, Hidenori Kobayashi, Simha Sethumadhavan, and Junfeng Yang. FIRED: Frequent Inertial Resets with Diversification for Emerging Commodity Cyber-Physical Systems. *ArXiv*, abs/1702.06595, 2017. `arXiv:1702.06595`.

9 Miguel A Arroyo, M Tarek Ibn Ziad, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. YOLO: Frequently Resetting Cyber-physical Systems for Security. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, volume 11009, pages 166–183. SPIE, 2019.

10 Mathieu Bacou, Grégoire Todeschi, Alain Tchana, Daniel Hagimont, Baptiste Lepers, and Willy Zwaenepoel. Drowsy-dc: Data center power management system. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 825–834, 2019. `doi:10.1109/IPDPS.2019.00091`.

11 Anton Borisov. Coreboot at Your Service! *Linux Journal*, 2009(186):1, 2009.

12 A. Leonard. Brown and Rafael. J. Wysocki. Suspend-to-RAM in Linux®. In *Proceedings of the Linux Symposium*, volume 1, 2008.

13 Burkacky, Ondrej and Deichmann, Johannes and Doll, Georg and Knochenhauer, Christian. Rethinking Car Software and Electronics Architecture. *McKinsey & Company*, 2018.

14 C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. In *Journal of the ACM*, volume 20(1), pages 46–61, 1973.

15 George Candea, James Cutler, and Armando Fox. Improving Availability with Recursive Microreboots: A Soft-state System Case Study. *Perform. Eval.*, 56(1–4):213–248, March 2004. doi:10.1016/j.peva.2003.07.007.

16 Huacai Chen, Hai Jin, Zhiyuan Shao, Kan Hu, Ke Yu, and Kun Tian. Clientvisor: Leverage cots os functionalities for power management in virtualized desktop environment. *SIGOPS Oper. Syst. Rev.*, 43(3):6271, July 2009. doi:10.1145/1618525.1618532.

17 Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A Survey of Software Aging and Rejuvenation studies. *J. Emerg. Technol. Comput. Syst.*, 10(1), January 2014. doi:10.1145/2539117.

18 M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–179, 2011.

19 Quad-motor GTE Electric Supercar. Accessed May. 2024. URL: https://www.drakomotors.com/.

20 Embedded Controller Interface Description. Accessed May. 2024. URL: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/12_ACPI_Embedded_Controller_Interface_Specification/embedded-controller-interface-description.html.

21 Anam Farrukh and Richard West. Flyos: Integrated modular avionics for autonomous multicopters. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 68–81, 2022. doi:10.1109/RTAS54340.2022.00014.

22 Anam Farrukh and Richard West. Flyos: rethinking integrated modular avionics for autonomous multicopters. In *Real-Time Systems Journal*, volume 59, pages 256–301, 2023. doi:10.1007/s11241-023-09399-w.

23 International Organization for Standardization. ISO 26262-1-12:2018 Road vehicles – Functional safety, December 2018.

24 International Organization for Standardization. ISO/SAE 21434:2021 Road vehicles – Cybersecurity engineering, August 2021.

25 Francis Chow. The new standard: Red Hat In-Vehicle Operating System in modern and future vehicles. Accessed May. 2024, 2022. URL: https://www.redhat.com/en/blog/new-standard-red-hat-vehicle-operating-system-modern-and-future-vehicles.

26 A. Golchin, Z. Cheng, and R. West. Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices. In *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

27 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)*, pages 390–402. IEEE, 2020.

28 Jawad Haj-Yahya, Mohammed Alser, Jeremie Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu. Sysscale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 227–240, 2020. doi:10.1109/ISCA45697.2020.00029.

29 Sena Hounsinou, Vijay Banerjee, Chunhao Peng, Monowar Hasan, and Gedare Bloom. Work-in-progress: Enabling secure boot for real-time restart-based cyber-physical systems. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 524–527, 2021.

30 Intel®64 and IA-32 Architecture Software Developer Manuals. Accessed May. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

31 Canturk Isci, Suzanne McIntosh, Jeffrey Kephart, Rajarshi Das, James Hanson, Scott Piper, Robert Wolford, Thomas Brey, Robert Kantner, Allen Ng, James Norris, Abdoulaye Traore, and Michael Frissora. Agile, Efficient Virtualization Power Management with Low-Latency Server Power States. *SIGARCH Comput. Archit. News*, 41(3):96–107, June 2013. doi:10.1145/2508148.2485931.

32 Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. Software Fault Tolerance for Cyber-Physical Systems via Full System Restart. *ACM Trans. Cyber-Phys. Syst.*, 4(4), August 2020. doi:10.1145/3407183.

33 Congfeng Jiang, Jian Wan, Xianghua Xu, Yunfa Li, and Xindong You. Power Management Challenges in Virtualization Environments. In *Systems and Virtualization Management. Standards and the Cloud*, pages 1–12. Springer Berlin Heidelberg, 2010.

34 Congfeng Jiang and Zhao Ying-Hui. Patpro: Power aware thin provisioning of resources in virtualized servers. *Applied Mathematics & Information Sciences*, 7:201–208, February 2013. doi:10.12785/amis/071L28.

35 Amit Kumar and Rakesh Kumar. Preferred device early availability for faster user response. In *2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence*, pages 331–336, 2017. doi:10.1109/CONFLUENCE.2017.7943171.

36 J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 1989.

37 Matthew Lentz, James Litton, and Bobby Bhattacharjee. Drowsy power management. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 230244, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815414.

38 Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. ACRN: A Big Little Hypervisor for IoT Development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–44, 2019.

39 Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable Communication and Migration in the Quest-V Separation Kernel. In *2014 IEEE Real-Time Systems Symposium*, pages 272–283. IEEE, 2014.

40 Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

41 CPU hotplug in the Kernel. Accessed May. 2024. URL: https://docs.kernel.org/core-api/cpu_hotplug.html.

42 Device Power Management Basics. Accessed May. 2024. URL: https://docs.kernel.org/driver-api/pm/devices.html#interfaces-for-entering-system-sleep-states.

43 Power Management. Accessed May. 2024. URL: https://docs.kernel.org/power/index.html.

44 Linux SCHED_DEADLINE Policy. Accessed May. 2024. URL: https://docs.kernel.org/scheduler/sched-deadline.html#:~:text=Overview,of%20tasks%20between%20each%20other.

45 LWN.net:Linux Power Management. Accessed May. 2024. URL: https://lwn.net/Kernel/Index/#Power_management.

46 M. Marcu and D. Tudor. Power consumption measurements of virtual machines. In *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 445–449, 2011. doi:10.1109/SACI.2011.5873044.

47 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro 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. doi:10.4230/OASIcs.NG-RES.2020.3.

48 Ron Minnich, Gan shun Lim, Ryan O'Leary, Chris Koch, and Xuan Chen. Replace Your Exploit-ridden Firmware with a Linux Kernel, 2017.

49 Patrick Mochel. Linux Kernel Power Management. In *Proceedings of the Linux Symposium*, 2003.

50 A. B Montz, D. Mosberger, S. W. O'Mally, L. L. Peterson, and T. A. Proebsting. Scout: A Communications-Oriented Operating System. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61. IEEE, 1995.

51 Neil Brown. Linux Power Management: The documentation I wanted to read. Accessed May. 2024, 2012. URL: `https://lwn.net/Articles/505683/`.

52 Ondrej Burkacky, Fabian Steiner,Martin Kellner, Johannes Deichmann and Julia Werra. Getting Ready for Next-generation E/E Architecture with Zonal Compute. *McKinsey & Company*, 2023.

53 Ondrej Burkacky, Martin Kellner, Johannes Deichmann, Patrick Keuntje and Julia Werra. Rewiring car electronics and software architecture for the 'Roaring 2020s'. *McKinsey & Company*, 2021.

54 R. Pineiro, K. Ioannidou, S. A. Brandt, and C. Maltzahn. Rad-flows: Buffering for Predictable Communication. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–33. IEEE, 2011.

55 BlackBerry | QNX: Software-Defined Vehicles. Accessed May. 2024. URL: `https://blackberry.qnx.com/en/ultimate-guides/software-defined-vehicle`.

56 Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look Mum, No VM exits! (Almost). *arXiv preprint*, 2017. `arXiv:1705.06932`.

57 Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 302313, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1555754.1555793`.

58 Raffaele Romagnoli, Bruce H. Krogh, Dionisio de Niz, Anton D. Hristozov, and Bruno Sinopoli. Runtime System Support for CPS Software Rejuvenation. *IEEE Transactions on Emerging Topics in Computing*, 11(3):594–604, 2023.

59 Raffaele Romagnoli, Bruce H. Krogh, Dionisio de Niz, Anton D. Hristozov, and Bruno Sinopoli. Software Rejuvenation for Safe Operation of Cyber–Physical Systems in the Presence of Run-Time Cyberattacks. *IEEE Transactions on Control Systems Technology*, 31(4):1565–1580, 2023.

60 Scott A. Rush. Application of Suspend Mode to Automotive ECUs. In *SAE International WCX World Congress Experience*, 2018. `doi:10.4271/2018-01-0021`.

61 J. M. Rushby. Design and Verification of Secure Systems. In *8th ACM Symposium on Operating Systems Principles*, pages 12–21, 1981.

62 Claudio Scordino, Luca Abeni, and Juri Lelli. Energy-aware Real-time Scheduling in the Linux Kernel. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 601–608, 2018.

63 Soham Sinha and Richard 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.

64 Slim Bootloader Project, 2021. https://slimbootloader.github.io/.

65 Brinkley Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

66 Mark Stanovich, Theodore P. Baker, An I Wang, and Michael Gonzalez Harbour. Defects of the POSIX sporadic server and how to correct them. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

67 Kevin Tian, Ke Yu, Jun Nakajima, and Winston Wang. How Virtualization makes Power Management Different. In *Linux Symposium*, page 205, 2007.

68 UEFI. Unified Extensible Firmware Interface Forum, 2021. URL: `https://uefi.org/specifications`.

69 Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative I/O: A Novel I/O Semantics for Energy-aware Applications. *ACM SIGOPS Operating Systems Review*, 36(SI):117–129, 2002.

70 R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online Cache Modeling for Commodity Multicore Processors. In *SIGOPS Oper. Syst. Rev.*, volume 44, pages 19–29, 2010.

71 R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. CAFÉ: Cache-Aware Fair and Efficient Scheduling for CMPs. In *Multicore Technology: Architecture, Reconfiguration and Modeling, CRC Press*, pages 221–253, 2013.

72 Richard West, Ye Li, Eric Missimer, and Matthew Danish. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Transactions on Computer Systems*, 34(3):8:1–8:41, June 2016.

73 Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: A Dynamic Cache Partitioning System using Page Coloring. In *23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

74 The Yocto Project. Accessed May. 2024. URL: `https://www.yoctoproject.org/`.

75 Yi-Wen Zhang, Rong-Kun Chen, and Zonghua Gu. Energy-aware partitioned scheduling of imprecise mixed-criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):3733–3742, 2023. `doi:10.1109/TCAD.2023.3246926`.